New resource allocation algorithms for emerging computer systems

Pan, Lai

2017

Pan, L. (2017). New resource allocation algorithms for emerging computer systems.
Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/70623>

<https://doi.org/10.32657/10356/70623>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

New Resource Allocation Algorithms for Emerging Computer Systems

PhD Thesis

By

Pan Lai

Advisor: Dr. Rui Fan

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of Doctor of Philosophy

March, 2016

Abstract

Computer systems today are moving towards both greater *distribution* and *consolidation*, ranging from multicore systems to clouds. For example, in a multicore system, tasks are distributed and executed by multiple concurrent threads, and meanwhile the threads are consolidated to run on the system. As another example, in Amazon EC2 service which provides its computing resource to clients in terms of virtual machine instance, clients' tasks are distributed and executed in thousands of virtual machines, and meanwhile the virtual machines run on a set of physical machines, leading to consolidation of the virtual machines. The systems consolidating threads contain a number of limited resources (e. g. cache, memory) to make their hosted threads run, and therefore their resource should be shared among these threads. Properly allocating the shared resource among threads is crucial to achieve high overall performance. A realistic method to allocate resource effectively is to use utility function to track each thread's resource demand and direct the resource allocation, where utility function describes each thread's performance based on the amount of resource it gets. In the thesis, we study how to allocate a shared resource in the emerging computer systems, such as multicore systems, virtual machine systems to achieve high overall performance. In addition, there are a number of distributed resource in distributed systems. For example, in a web hosting center, a number of web service threads run on multiple servers and compete for resources such as processing or memory. We also investigated distributed resource allocation problem. In the thesis, we proposed efficient and effective resource allocation algorithms, guaranteeing good performance bounds.

- (i) We first studied how to allocate a resource among multiple threads to minimize *makespan*, which measures the amount of time needed to complete a set of tasks. We assume that a thread's performance increases with its resource allocation, and that the performance function is concave. Our first contribution is to give a mathematical characterization of the properties satisfied by an optimal resource allocation. Second, we present an algorithm that finds a $1 + \epsilon$ approximation to the optimal allocation in $O(n \log \frac{n}{\epsilon} \log \frac{n}{\epsilon p})$ time, where n is the number of tasks and p is a value that depends on the optimal solution. Simulations show that our algorithm achieves better makespan compared to several allocation schemes used in practice or proposed in the literature.

- (ii) In the aforementioned work, it is assumed that the performance function of each thread is concave. However, many realistic performance functions are not concave. We study how to allocate a shared resource among multiple threads with nonconcave performance functions to maximize the total utility, which can represent the total running speed of all threads. The problem is in general NP-complete, and past works have mostly focused on studying concave performance curves, applying heuristics to nonconcave curves, or finding optimal solutions using slow dynamic programming methods. In our work, we observe that realistic performance curves are often not concave, but rather can be broken into a small number of concave or convex segments. We present efficient algorithms for optimal and approximately optimal resource allocation leveraging this observation. We also introduce several algorithmic techniques that may be of independent interest. Our optimal algorithm runs in $O(sn\alpha(m)m(\log m)^2)$ time, and our approximation algorithm finds a $1 - \epsilon$ optimal allocation for any $\epsilon > 0$ in $O(\frac{s}{\epsilon}\alpha(\frac{n}{\epsilon})n^2 \log \frac{n}{\epsilon} \log m)$ time; here, s is the number of segments, n the number of processes, m the amount of shared resource, and α is the inverse Ackermann function that is ≤ 4 in practice. Existing exact and approximation algorithms have $O(nm^2)$ and $O(\frac{n^2m}{\epsilon})$ running times, respectively, so our algorithms are much faster in the practical case where $n \ll m$. Simulations show that our algorithm performs better than other algorithms.
- (iii) Besides centralized resource, there are also lots of distributed resource in many distributed systems. There is also a need to allocate distributed resource to maximize the total utility. The distributed resource allocation combines thread assignment and resource allocation together. Both the assignment and allocation component of this problem have been studied extensively, though separately in the literature. However, achieving high performance in distributed systems requires to finding good assignments of threads to servers as well as effectively allocating each server's resource to its assigned threads. In our work, we introduce the distributed resource allocation problem, which seeks to simultaneously find an assignment and allocations that maximize the total utility of the threads. We first show that this problem is NP-hard, even when there are only two servers. We then present a $2(\sqrt{2} - 1) > 0.828$ factor approximation algorithm, which runs in $O(mn^2 + n(\log mC)^2)$ time for n threads and m servers with C amount of resources each. We also present a faster algorithm with the same approximation ratio and $O(n(\log mC)^2)$ running time. Simulations show that our algorithm achieves better performance than other algorithms.

Acknowledgments

First of all, I would like to devote my thanks to my supervisor Prof. Rui Fan sincerely and gratefully for his valuable advice and effective and patient guidance during my PhD study. I learned a lot from him in these years. With his help, I learned how to do a survey on an interesting field, how to find interesting research topics, how to write academic articles and how to give clear presentations. From him, I also learned to be positive and active, and pursue perfect results in research as well as in life. What I learned from him would benefit a lot in my future career and life. Moreover, I am thankful for his valuable advice and information on my job hunting.

I also would like to thank Prof. Xueyan Tang, Prof. Bingsheng He, Prof. Wee Peng Tay for serving as committee members of my PhD qualification examination. Their valuable advice helps me a lot in my research work.

Moreover, I am grateful to Wei Zhang, Fang Liu, Shanjiang Tang for their comments and discussion on my research topic and papers.

I am also thankful to many colleagues and friends, such as Jianlong Zhong, Tianyi Zhou, Yuseng Li, Liudong Zuo, etc.

Finally, I would like to thank my parents and my elder sister who persistently afford support to me when my life did not go on smoothly. Without their support, it is hardly for me to fulfill the PhD study.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Makespan Optimal Resource Allocation	5
1.2 Fast Optimal Nonconcave Resource Allocation	6
1.3 Utility Maximizing Distributed Resource Allocation	8
1.4 Summary of Our Contributions	9
1.5 Outline of the Thesis	10
2 Background	12
2.1 Overview of Resource Sharing Applications	13
2.1.1 Shared Cache	13
2.1.2 Shared Memory	15
2.1.3 Shared Distributed Resource	15
2.1.4 Other Shared Resource	17
2.2 Resource Sharing Techniques	17
2.2.1 Cache Sharing Techniques	18
2.2.2 Memory Sharing Techniques	25
2.2.3 Distributed Resource Sharing Techniques	27
2.2.4 Makespan and Total Utility	28
2.2.5 Resource Allocation Algorithmic Techniques	29
3 Makespan Optimal Resource Allocation	32
3.1 Introduction	32
3.2 Prior and Related Work	33
3.3 Model	34

3.3.1	Resource behavior	34
3.3.2	Problem	36
3.4	Properties of the Optimal Schedule	36
3.5	Scheduling algorithm	40
3.5.1	Algorithm analysis	41
3.6	Evaluation and comparisons	49
3.6.1	Performance compared to uniform allocation	50
3.6.2	Performance compared to utility-based partitioning	52
3.6.3	Performance compared to fair partitioning	53
3.7	Summary	54
4	Fast Optimal Nonconcave Resource Allocation	55
4.1	Introduction	55
4.2	Prior and Related Work	57
4.3	Model	58
4.4	Exact Algorithm	59
4.4.1	Dynamic programming algorithm	59
4.4.2	A faster algorithm	60
4.4.3	Algorithm analysis	63
4.5	Approximation algorithm	65
4.5.1	Scaling the problem	66
4.5.2	Estimating F^*	67
4.5.3	Reducing the problem	68
4.5.4	Algorithm analysis	70
4.5.5	Lemma for estimating F^*	72
4.6	Experimental Evaluation	75
4.6.1	Performance comparison with dynamic programming	75
4.6.2	Quality comparison with greedy	76
4.7	Summary	77
5	Utility Maximization Distributed Resource Allocation	78
5.1	Introduction	78
5.2	Related Works	80
5.3	Model	81
5.4	Hardness of the Problem	82
5.5	Approximation Algorithm	84

5.5.1	Linearization	84
5.5.2	Solving the linearized problem	85
5.5.3	Solving the concave problem	92
5.6	A Faster Algorithm	93
5.6.1	Algorithm Analysis	93
5.7	Experimental Evaluation	95
5.7.1	Uniform and normal distributions	96
5.7.2	Power law distribution	97
5.7.3	Discrete distribution	98
5.8	Summary	99
6	Conclusions and Future Work	100
6.1	Conclusions	100
6.2	Future works	101
6.2.1	Dynamic Resource Allocation	102
6.2.2	Makespan Minimization Distributed Resource Allocation	102
6.2.3	Heterogeneous Distributed Resource Allocation	103
6.2.4	Multiple Type Resource Allocation	104
6.2.5	Online Resource Allocation	105

List of Figures

1.1	IPC graphs for the SPEC CPU benchmarks using data adapted from [55]. The x -axis shows the number of ways an application is allocated from a 16 way 1MB cache, the y axis shows the IPC.	3
2.1	Simplified views of a traditional single-core (unicore) processor and a multiple- core (multicore) processor system.	14
2.2	Illustration of page coloring	22
2.3	Cache miss rate of <i>art</i> as a function of cache blocks	23
2.4	Illustration of Mattson's algorithm	24
2.5	Page miss ratio vs. memory size curve (MRC) for <i>bzip</i> [73]	27
2.6	Utility functions	30
3.1	Comparison to uniform allocation for varying job lengths and miss rate exponents	51
3.2	Comparison to uniform allocation as a function of thread count	51
3.3	Comparison to utility-based partitioning for varying job lengths and miss rate exponents	52
3.4	Comparison to utility-based partitioning as a function of thread count	53
3.5	Comparison to fair partitioning for varying job lengths and miss rate expo- nents	54
3.6	Comparison to fair partitioning as a function of thread count	54
4.1	Illustration of Algorithm 4.1	61
4.2	Running time of ALLOC versus DP for varying m	76
4.3	Running time ALLOC for varying s	76
5.1	Performance of Algorithm 1 versus SO,UU,RU,UR, and RR as a function of β under the uniform and normal distributions.	96
5.2	Performance of Algorithm 1 versus SO,UU,RU,UR, and RR as a function of β and α under the power law distribution.	97

5.3 Performance of Algorithm 1 versus SO,UU,RU,UR, and RR as a function
of β, γ and θ under the discrete distribution. 98

6.1 Gcc’s cache miss rate over time [95] 103

List of Tables

2.1	Commercial multicore processors configurations [93]	14
2.2	Several types of EC2 instance configurations	16
6.1	Server configurations in one of Google’s clusters [89, 90]. CPU and memory are normalized.	104

Chapter 1

Introduction

Two opposing trends *distribution* and *consolidation* are occurring today in computer systems, ranging from multicore to clouds. For example, in multithread applications, the tasks are distributed and executed by multiple concurrent threads. Meanwhile, multithread applications often run on chip multiprocessor (CMP) system, making the threads executing the applications consolidated on the system. CMP systems are ubiquitous nowadays. Most new processors released recently by major vendors, such as AMD, Fujitsu, IBM, Intel and Sun Microsystems, are CMP. As another example, in cloud computing, Amazon EC2 provides its computing resource to clients in terms of virtual machine instance. Clients' tasks are distributed and executed in thousands of Amazon EC2 virtual machines. Meanwhile, the virtual machines run on a set of physical machines, leading to consolidation of the virtual machines. Cloud computing is significant today. For instance, it is claimed Amazon's cloud generates 1% of all Internet traffic and access from 1/3 of daily Internet users in North America in 2012 [1].

The systems consolidating users (threads, virtual machines) today contain a number of limited resources (e. g. cache, memory) to make their hosted threads run. Therefore, their resource should be shared among these users. For example, a modern CMP often contains multiple cores and several layers of caches, including cache shared by all the cores on a chip. When a multithread application runs on the system, the cache is shared by threads running on different cores. The shared on-chip cache is a crucial hardware resource to achieve high performance because on-chip cache memory access is much faster than off-chip main memory access. For instance, on the IBM POWER5 processor, on-chip shared

L2 cache access takes 14 processor cycles, whereas main memory access needs 280 cycles. Shared on-chip caches are a scarce resource, since the on-chip cache is often small due to high cost while the cache demands of many emerging applications are relatively very large, leading to severe contention for shared cache space. For example, in practice, the shared on-chip cache in CMP systems is at most 8MB, while most programs nowadays need surprisingly large amount of cache. In Amazon EC2 service, many virtual machines run on physical machines, each has resources such as processing, memory. Each physical machine should share its processing and memory resource to different virtual machines running on it. Main memory is a significant resource because memory access is about 200 times faster than disk access. Memory is a scarce resource in clouds. For example, many servers (e.g. HP ProLiant ML350 Gen9 Server) have only several hundreds of GB memory whereas two types of Amazon EC2 virtual machine instances, i.e. i2.8xlarge and d2.8xlarge, which may run on these servers need 262GB memory. In a web hosting center, a number of web service threads run on multiple servers. Each server should share its resources such as processing or memory to the web services hosted on it [56, 57]. Besides threads and virtual machines, resource sharing occurs for many different kinds of users (e.g. process, program, application).

In resource sharing, allocating shared resource effectively requires computer system to be aware of each thread's resource demand. A flexible method to capture the resource demand of a thread is to use utility function, which describes the performance based on the amount of resource it is allocated. For example, Figure 1.1 shows the utility for 15 applications from the SPEC benchmark as a function of their cache allocation, where the utility represents IPC (Instructions per Cycle) rates. Usually, a utility function is increasing since a thread often performs better when given more resource. This method has the advantage of providing each thread with multiple possible resource options and therefore has been widely used to direct shared resource allocation among threads [40, 73, 55, 54]. For example, Qureshi et al. [55] used utility functions to determine cache allocations among a set of threads to maximize the total utility.

In resource sharing, different resource allocations may lead to distinct overall system performance, since each thread may have disparate resource requirement. For example,

Figure 1.1: IPC graphs for the SPEC CPU benchmarks using data adapted from [55]. The x -axis shows the number of ways an application is allocated from a 16 way 1MB cache, the y axis shows the IPC.

consider a situation with two threads sharing a 16-way cache, one is “galgel” and performs better when given more cache, and the other is “mcf” and does not perform better when given more cache. Consider two allocations, one is to give each thread half of the cache: 8-way of the cache and the other is give “galgel” 15-way of the cache and “mcf” 1-way cache. The total IPC under the two allocation is 0.7536 and 1.2357, respectively according to the IPC functions of “galgel” and “mcf” shown in Figure 1.1. Clearly, we achieve better performance under the second allocation than the first one. Therefore, it is crucial to determine how to allocate the shared resource among different threads to achieve high overall performance.

In resource allocation, there are several methods to evaluate the quality of allocation. In parallel computing, one important type of parallelism is task parallelism. For example, in an image processing application *alpha blending*, separate images from different sources are processed independently and then combined to a single image. The source images that are combined are different, and different image processing operations are performed on each of them before they are combined. The application is designed in parallel to save the run-time, where the operation for processing each image is performed by different concurrent

threads. In task parallelism, each thread may have different work length, since each type of work is disparate. The completion time of the task is determined by the slowest thread. Naturally, we want the task to finish as soon as possible. In other words, we want to minimize the completion time of the task, which is defined as *makespan*. However, some tasks are persistent, which means they run all the time. For example, many web and antivirus services run persistently because web service and antivirus requests may arise at any time. Makespan cannot be used in these settings. Naturally, we want the threads to finish as much work as possible at every time interval. This characteristic can be evaluated by *total utility*, which represents the sum of all threads' running speed measured by IPC. Makespan and total utility are two different performance metrics, since makespan is determined by length of each thread while utility is not affected by thread's length.

In the thesis, we investigated how to allocate a shared resource¹ among threads to achieve high overall performance. In addition, there are also lots of distributed resource in distributed systems, where there are many threads running on multiple servers, each with certain resource capacity. While our terminology generically refers to threads and servers, distributed resource is used in a range of distributed systems. For example, in a web hosting center, a number of web service threads run on multiple servers and compete for resources such as processing or memory [56, 57]. For another example, on multi-chip multicore processors such as 4-core Intel Core 2, there are multiple chips, each hosts multiple cores with shared cache. There are many multithread applications running on the multi-chip multicore processors. Each chip corresponds to a server offering its shared cache as a resource to threads executing on different cores on the chip. Distributed resource allocation contains two steps: determine on which server each thread should be assigned and then how much resource it should be allocated on that server. Compared to centralized resource allocation, distributed resource allocation needs to additionally consider how to assign the threads on different servers. In the thesis, we also studied how to allocate distributed shared resource to achieve high overall performance.

¹In the thesis, we use the terms a shared resource and centralized resource interchangeably

1.1 Makespan Optimal Resource Allocation

We first studied how to allocate a resource among multiple concurrent threads to minimize the makespan. There have been some works on allocating a shared resource to multiple simultaneous threads to minimize makespan [26, 23]. However, in these works only greedy heuristic algorithms are proposed and no performance bounds are guaranteed. This motivates us to design an efficient algorithm to guarantee certain performance bounds for allocating a shared resource among multiple simultaneous threads to minimize makespan in Chapter 3.

We consider a setting in which a set of concurrent threads, each executes a task of a certain length using a shared resource. Our model is quite general: we only assume that a thread's performance increases with its resource allocation, and that the performance function is concave. This model is a generalization of the widely used "square root rule" on cache miss rates [2, 3, 4, 5, 53]. This makes sense because it models a diminishing return property, which says the gain of a thread from each unit of resource diminishes with the increasing amount of resource given to it and the property is frequently observed in practice [55]. While it does not apply in all settings, concavity is a common assumption used in many utility models, especially on cache and memory performance [5, 53].

With this model, we show that there always exists an optimal allocation with a simple form: it is *static* and *balanced*, meaning that each thread is allocated the same amount of resource throughout its execution, and the amounts ensure that all the threads finish at the same time. In general, this optimal allocation might consist of arbitrary real numbers, and so we cannot compute it exactly. However, we present an algorithm based on the characterization that efficiently computes for any $\epsilon > 0$ an allocation whose makespan is at most $1 + \epsilon$ times the optimal. The time complexity of the algorithm is $O(n \log \frac{n}{\epsilon} \log \frac{n}{\epsilon p})$, where n is the number of threads and p is a value that depends on the optimal solution.

We then apply our algorithm to the cache partitioning problem. We compared our algorithm with several existing cache partitioning algorithms. We found that our algorithm shows the greatest improvement over existing algorithms when the number of threads is relatively large, the lengths of the threads are unequal but similar and the cache miss rate

functions of the threads are quite dissimilar. In particular, our algorithm performs up to 59% better than uniform partitioning, and 22% better than a utility-based partitioning algorithm [54, 15]. Finally, our algorithm works quickly in practice. Using 8 threads and $\epsilon = 10^{-3}$, an unoptimized Matlab implementation on a desktop PC runs in only 0.01 seconds, and with 16 threads it takes less than 0.03 seconds.

1.2 Fast Optimal Nonconcave Resource Allocation

In the aforementioned work and most existing works on resource allocation problems, the performance function of each thread is assumed to be concave. However, many realistic performance functions are not concave. For example, Figure 1.1 shows several IPC functions (i. e. *swim*, *wupsize*, *ammp*, *apsi*, *galgel*, *perlbnk*) are nonconcave. One reason is that sometimes a highly accessed portion of the working set fits entirely within cache once the cache allocation exceeds a threshold. Likewise, nonconcavities can occur whenever there is a “qualitative” change in performance once the resource allocation meets certain criteria.

Some works have been done on how to allocate a shared resource among multiple threads with nonconcave performance functions to maximize the total utility [37, 43, 46]. Unfortunately, the resource allocation problem with nonconcave utility functions is NP-complete [37]. In practice, one may use the greedy algorithms designed for concave functions and hope they also work well for nonconcave functions [73, 54]. However, there are cases where greedy algorithms produce solutions that are $\Omega(\frac{m}{n})$ times away from optimal, where n is the number of processes and m the amount of shared resource. Alternatively, there are dynamic programming based methods [43, 46] to find the optimal solution in $O(nm^2)$ time. However, m may be quite large in practice; for example, when partitioning memory in the virtual machine setting, m may be on the order of the number of memory pages. Thus, the m^2 term in the running time of these algorithms makes them quite inefficient. This motivates us to design an algorithm to solve centralized resource allocation problem with nonconcave utility functions for the sake of maximizing total utility in Chapter 4.

In our work, we begin by observing that while realistic throughput functions are not necessarily concave, they are also not arbitrary. In particular, the functions often consist of

a small number of *segments*, where within each segment the function is concave or convex. For example, the IPC curve of “swim” in Figure 1.1 consists of two concave segments, and the curve for “galgel” consists of one concave and one convex segment. Using the assumption, we present an efficient algorithm to compute the optimal allocation. Our algorithm runs in $O(sn\alpha(m)m(\log m)^2)$ time, where s is the maximum number of concave or convex segments in the utility function for any thread and $\alpha(m)$ is the inverse Ackermann function. In most settings, s and $\alpha(m)$ are both small constants. Thus, compared to the $O(mn^2)$ running time of existing optimal algorithms, our algorithm is almost $O(m)$ times faster. This speed is especially important in settings where the optimal allocation needs to be computed multiple times and on the fly, as occurs when the utility functions can change over time [54].

In some situations it suffices to find an approximately optimal allocation. We also present an algorithm that finds an allocation with total utility at least $1 - \epsilon$ times the optimal, and runs in time $O(\frac{s}{\epsilon}\alpha(\frac{n}{\epsilon})n^2 \log \frac{n}{\epsilon} \log m)$, for any $\epsilon > 0$. A $1 - \epsilon$ approximation can be found using existing methods by reducing resource allocation to the multiple choice knapsack problem [46]. This algorithm takes $O(\frac{n^2m}{\epsilon})$ time, which is almost $O(m)$ times slower than our algorithm.

A novel aspect of our algorithms is that, while most current exact and $1 - \epsilon$ approximate algorithms use dynamic programming, we relate the resource allocation problem to an *upper envelope* computation and make use of efficient algorithms from computational geometry [45]. In addition to our improved exact and approximate allocation algorithms, we also introduce an efficient algorithm to find a constant factor estimate of the maximum total throughput. This task is an important step in many approximation algorithms for resource allocation and related problems [68, 69]. Existing algorithms typically have running times proportional to m , while our algorithm runs in $O(n^2 \log n)$ time. Hence, our algorithm is much faster for the typical case of $n \ll m$. We show that instead of considering all possible allocation values when maximizing the throughput, it suffices to consider a much smaller set of “rounded” values. This result may be of independent interest.

We conducted experiments to test the performance of our algorithms. For $m = 1M$, our exact algorithm was over 215 times faster than the dynamic programming method. Also,

for randomly created throughput functions, our algorithm achieved 33% higher throughput than the greedy algorithm.

1.3 Utility Maximizing Distributed Resource Allocation

The aforementioned works studied how to allocate a centralized resource among many threads. Besides centralized resource, there are also many distributed resource in computer systems. In these systems, there are many threads running on multiple servers, each with certain resource capacity. We investigated distributed resource allocation, which contains two steps: thread assignment and resource allocation. In the first step, each thread should be assigned on a server. In the second step, the thread should be allocated some resource on the assigned server. To our knowledge, the threads assignment problem and resource allocation problem have not been studied together in the unified context considered in our work. Existing works on resource allocation [63, 83, 55, 54, 62] largely deal with dividing the resources on a single server among a given set of threads. It is not clear how to apply these algorithms when there are multiple servers, since there are many possible ways to initially assign the threads to servers, and certain assignments result in low overall performance regardless of how resources are subsequently allocated. For example, consider 10 homogeneous memory intensive threads are running on two homogeneous servers, each with limited amount of memory. A possible way to allocate resource is to assign all threads on one server and then allocate equal amount of the memory on the server to all the threads. Clearly, better performance can be achieved if half of the threads are assigned on each server, respectively and each thread is allocated the equal amount of memory on the server. Likewise, existing works on thread assignment [59, 60, 57, 58] often overlook the resource allocation aspect. Threads typically request a certain amount of resources², and once assigned they are allocated precisely the resources they request. without any adjustments based on the requests of other threads assigned to the same server. This can also lead to suboptimal performance. This motivates us to design efficient algorithm to determine distributed resource allocation to optimize total utility in Chapter 5.

²The request may be a scalar when there is a single type of resource, or a vector for multiple resource types.

In the distributed resource allocation problem, we assume the utility functions are non-decreasing and concave. The concavity property makes sense because it models a diminishing return property and have been observed in many realistic utility functions. The goal is to simultaneously find an assignment and allocations for each thread that maximize the total utility of all the threads. We first show that the problem is NP-hard, even when there are only two servers. In contrast, the problem is efficiently solvable when there is a single server. Next, we present an approximation algorithm that finds a solution with utility $\alpha = 2(\sqrt{2} - 1) > 0.828$ times the optimal. The algorithm relates the optimal solution to a single server problem to an approximately optimal solution to the multiple server problem. The algorithm runs in time $O(mn^2 + n(\log mC)^2)$, where n and m are the number of threads and servers, resp., and C is the amount of resource on each server. We then present a faster algorithm with $O(n(\log mC)^2)$ running time and the same approximation ratio. Finally, we conduct experiments to test the performance of our algorithm. We used several types of synthetic threads, and show that our algorithm obtains at least 99% of the maximum utility on average for all the thread types. We also compared our algorithm with several simple but practical assignment and allocation heuristics, and show that our algorithm performs on average 3.9 to 5.7 times better.

1.4 Summary of Our Contributions

In this section, we summarize the motivations and contributions we made in the thesis. Two opposing trends distribution and consolidation are occurring in today's computer systems, ranging from multicore to clouds. The systems where threads are consolidated contain numerous limited resources (e. g. cache, memory) to run their hosted threads, leading to the fact that their resource are shared among threads. In resource sharing, allocating shared resource effectively requires computer system to be aware of each thread's resource demand, and a flexible method to capture the resource demand of a thread is by utility function. In resource sharing, properly allocating shared resource is crucial to achieve high overall performance. In resource allocation, makespan and total utility are two important metrics to evaluate the quality of allocation.

In the thesis, we investigated how to allocate a shared resource among threads with utility functions to achieve high overall performance in terms of makespan and total utility. Many threads have concave utility functions from observations and this property has certain theoretical justifications. We first studied how to allocate a shared resource to multiple threads with concave utility functions to minimize the makespan in Chapter 3. We show that the characteristics of optimal schedule and present an efficient approximation algorithm to find an approximately optimal schedule based on the characterization. Also, we observe many realistic utility functions are not concave. We present efficient algorithms to allocate a shared resource among multiple threads with nonconcave utility functions to maximize the total utility in Chapter 4. A key observation is that though the utility functions are nonconcave, they can be divided into a small number of concave or convex segments. We present both efficient optimal algorithm and approximate algorithm for the problem leveraging the observation. In addition, there are lots of distributed resource in distributed systems, where there are many threads running on multiple servers, each with certain resource capacity. In Chapter 5, we introduced the distributed resource allocation problem. We want to simultaneously find an assignment and allocations for each thread that maximize the total utility of all the threads. We show that the problem is NP-hard and present two approximation algorithms that find a solution with utility $\alpha = 2(\sqrt{2} - 1) > 0.828$ times the optimal.

1.5 Outline of the Thesis

We now describe the organization of the thesis. In Chapter 2, we describe the background of various resource sharing applications and techniques in computer systems. In Chapter 3, we study the problem of determining the allocations among multiple threads in a shared resource environment to minimize the makespan and propose an algorithm to find the approximation optimal allocation and analysis for the algorithm. In Chapter 4, we investigated how to find the allocations to maximize the overall throughput for threads with nonconcave throughput functions in a shared resource environment, and proposed both fast optimal and approximate algorithms to solve the problem and analysis for the two algorithms. In Chapter 5, we introduce the problem of determining the allocations for threads in a distributed

shared resource environment to maximize the total utility and proposed two approximation algorithms and analyzed the two algorithms. Finally, we conclude the thesis and discuss several directions on future work in Chapter 6.

Chapter 2

Background

The aim of this chapter is to provide a perspective on how this thesis fits into the context of resource management in computer systems, surveying relevant knowledge in order to understand where our research problems came from, what state our works are located at in the research area of resource management of computer systems, how our works can be used in real computer systems, and where we are potentially proceeding towards in the future.

In the chapter, we first describe various resource shared applications, (e. g. cache sharing, memory sharing, distributed resource sharing). Following this, we show how cache, memory, and distributed resource are shared in computer systems. In particular, we describe existing models and algorithms, which determine how much resource each user can be given, as well as technologies which can be used to implement the models and algorithms in real resource sharing applications. Our research focuses on the model and algorithms and therefore contributes to computer system resource management.

Users in computer systems represent many kinds of parities (e.g. thread, process). Both processes and threads are independent sequences of execution widely used in computer systems. The typical differences of them are that thread is often within a process, and threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. Though they have differences, both of them can run with a private portion of resource in a resource sharing environment with limited resource. For example, several threads may share a cache in CMP systems and each is allocated a private section of cache. Also, several processes may share a memory and each is allocated some private memory pages. Our model studied how to share a limited resource among many users by allocating

each user a private portion of resource and our model fits for both thread and process in computer systems. Thus, we use the term “thread” and “process” interchangeably in the thesis.

2.1 Overview of Resource Sharing Applications

In computer systems, there are lots of resource, such as cache, memory and many distributed resources that are shared among many users. In this section, we overview various resource sharing applications, e.g. cache sharing, memory sharing, distributed resource sharing.

2.1.1 Shared Cache

There is a large speed gap between processor and main memory. Fast cache memory is proposed to bridge the gap taking advantage of program locality. Cache is used to store the recently accessed data of a program. When the program wants to access the data again due to program locality, it accesses the data directly from fast cache instead of long latency memory, and therefore the memory access latency is reduced.

In the past decade, Chip Multiprocessors (CMP) have been widely adopted for computer systems. Figure 2.1a illustrates a simplified view of a traditional uncore processor, while Figure 2.1b illustrates a simplified view of a multicore processor. In CMP architecture, multiple processing cores are added on a chip to provide high performance computing capability. As the number of processor cores increases in CMP system, the speed gap between processor and main memory becomes larger. Caching is still the most widely used technique to bridge the gap. In most CMP systems, the multiple cores are designed to be served by multi-level on-chip caches. Instead of designing a private last level on-chip cache for each core, CMP designers built shared last level cache for the multiple cores for the purpose of saving space on the chip. Table 2.1.1 shows many commercial multicore processors. In many multicore architecture, (e. g. 2-core IBM POWER4, 2-core IBM POWER5, 2-core Intel Core 2 Duo), the L2 cache is shared, and L3 cache is shared in several other multicore architectures (e. g. 4-core AMD Opteron, 8-core IBM POWER7).

Figure 2.1: Simplified views of a traditional single-core (unicore) processor and a multiple-core (multicore) processor system.

Multicore Processor	On-chip caches	Shared caches
2-core IBM POWER4	L1,L2	L2
2-core IBM POWER5	L1,L2	L2
3-core IBM-Microsoft PowerPC Xenon	L1,L2	L2
8-core Sun UltraSPARC T1	L1,L2	L2
2-core Intel Core 2 Duo	L1, L2	L2
4-core Intel Core 2 Quad	L1, L2	L2
4-core Intel Xeon & Core 2 Extreme	L1,L2	L2
2-core Sun-Fujitsu SPARC64 VI	L1,L2	L2
4-core AMD Opteron	L1,L2,L3	L3
8-core Sun UltraSPARC T2	L1,L2	L2
6-core Intel Xeon	L1,L2,L3	L3
4-core Sun-Fujitsu SPARC64 VII	L1,L2	L2
6-core AMD Opteron	L1,L2,L3	L3
8-core IBM POWER7	L1,L2,L3	L3
8-core Intel Xeon	L1,L2,L3	L3

Table 2.1: Commercial multicore processors configurations [93]

2.1.2 Shared Memory

Main memory is designed to store instructions and data of a running program. In computer systems, there are often multiple programs that need to run together and share the memory.

Currently, memory sharing is still significant in many situations. Memory sharing is especially significant in virtual machine systems. A virtual machine (VM) is an emulation of a particular computer system and executes programs like the computer system. Virtualization technologies, whose roots originated from IBM VM/370 used for multitasking in 1960s are drawing a great deal of new attention from a variety of applications, such as data centers, web hosting by providing benefits in terms of security, energy-saving, and resource scalability.

Virtual machines run on top of the virtual machine monitor (VMM) or hypervisor¹. VMM is a host program which manages either multiple operating systems, each is corresponding to a virtual machine or multiple instances of the same operating system on a single computer system. It is to manage the system's processor, memory and other resources to satisfy the virtual machines' resource needs. VMM multiplexes processor between virtual machines in a manner quite similar to an operating system switching between running processes, where the operating system saves the state of the running process, and restores the state of the to-be run process and runs it. VMM virtualizes memory by controlling memory page mapping of each virtual machine.

Memory is a scarce resource in virtual machine systems. A single computer/server usually boots several virtual machines. These virtual machines usually need lots of memory and eventually compete for the memory in the computer/server. For example, Amazon EC2 service runs multiple virtual machine instances on a server. Figure 2.1.2 shows the configurations of several types of EC2 virtual machine instances. We see both i2.8xlarge and d2.8xlarge instance take 244GiB memory.

2.1.3 Shared Distributed Resource

Currently, distributed systems have been widely adopted. In distributed systems, there are many threads running on multiple servers, each with certain resource capacity. This envi-

¹hypervisor can be regarded as a VMM

Type of EC2 Instance	Number of virtual CPUs	Memory (GiB)
i2.xlarge	4	30.5
i2.2xlarge	8	61
i2.4xlarge	16	122
i2.8xlarge	32	244
d2.xlarge	4	30.5
d2.2xlarge	8	61
d2.4xlarge	16	122
d2.8xlarge	36	244

Table 2.2: Several types of EC2 instance configurations

ronment is named distributed resource sharing environment in this thesis.

There are many distributed resource sharing scenarios in practice. While our terminology generically refers to threads and servers, distributed resource sharing is used in a range of distributed systems. For example, currently web hosting service is widely commercially adopted since it provides great convenience to companies who do not want to spend time managing the web hosting hardware but still want to make their goods and service accessible on the Internet. There are many types of web hosting service. One important type is shared web hosting service, which has been widely used by many web hosting companies, such as HostRocket, Namecheap, Liquidweb. In shared web hosting service, there are many web servers and each server hosts many websites, ranging from a few to hundreds or thousands. Each website consumes some resource (e. g. cache, memory) on the hosted server to run. Different website has different resource demands. It needs to determine to assign each website on which server, and allocate how much resource to it. Our model in Chapter 5 can be applied in this problem by using utility function to capture each website’s resource demand .

For another example, in a multi-chip multicore processor, there are multiple chips, each hosts multiple cores with shared cache. For example, in 4-core Intel Core 2 Quad architecture, there are two chips, each hosts two cores which share a L2 cache. There are many threads running on the multi-chip multicore processor. Each chip corresponds to a server offering its shared cache as a resource to threads executing on different cores on the chip. Each thread runs on a chip and takes some L2 cache on the chip. Different threads may have different cache requirement. It is essential to decide to assign each thread on which

chip and partition how much L2 cache on the chip to it. Our model in Chapter 5 can be also applied in this problem by using utility function to capture each thread's cache demand.

2.1.4 Other Shared Resource

The purpose of the subsection is to show that resource sharing is widely used in many applications, besides the three application scenarios described earlier. For example, shared resource is also encountered in effort distribution, software reliability, core sharing. In effort distribution application, it needs to find an object which has several possible positions. The total effort to search the object is shared among the positions. Each position has a probability function, which describes the probability to find the object, based on the amount of effort allocated to the position. Different positions have different probability functions to find the object. The key problem is how to allocate the search efforts to each position to maximize the total probability of finding the object [86]. In software reliability, a software system is composed of multiple independent modules. There are a limited amount of testing resource used to test the software system. The number of software errors remaining in each module is a function of the amount of testing resource allocated to it. Different modules have different functions of remaining errors since they have different resource demands. The manager has to determine the optimal resource allocation to each module to minimize the number of software errors remaining in the system [88]. Moreover, in a manycore system, there are several multithread applications running on hundreds of cores. Each application has a performance function, which describes its performance based on the number of cores given to it. Different applications have different performance functions since they have different computing resource requirement. It is essential to determine how to allocate the cores among these applications to maximize the total performance [94].

2.2 Resource Sharing Techniques

In this section, we show various techniques to share cache, memory, distributed resource. These techniques include existing sharing models, algorithms as well as technologies.

2.2.1 Cache Sharing Techniques

As described before, the last level cache is shared by several cores in CMP systems. Several techniques have been proposed to share cache among multiple threads running on the different cores in CMP systems. In this section, we describe these techniques.

Least Recently Used Replacement Policy

A common technique devised to manage cache for single thread applications is least recently used (LRU) replacement policy. It works by finding the least recently used cache block to replace when a data access encounters a cache miss. For multithread applications running in CMP settings, it is still the common technique used for cache sharing. However, it has been shown to perform poorly [54]. Some researchers observed that a process can slow down by hundreds of percent if it shares last level cache with processes running on neighboring cores relative to running alone [66, 9]. This is because each cache block is shared by all threads and one thread's data cached in a cache block may be evicted by other threads. When the former thread accesses the data accessed before, it encounters cache miss. This is called *inter-thread interference* problem.

The problem is especially serious when there are two applications, one with low data locality and the other with high data locality. This is because the first application benefits little from the cache and the second application benefits a lot from the cache if their data is well cached. However, in some case, the first application may evict all of the second application's data stored in the cache. The first application still does not benefit from the cache while it undermines the benefit of the second application from the cache. Thus, the total benefit from the cache is undermined seriously. For example, assume there are two applications: one is streaming application (e. g. video playing) T_1 which has low data locality and the other is an application T_2 of high data locality. From the view of a thread, the execution of a program is a sequence of data access. Assume T_1 has data access sequence ABCDEF and application T_2 has data access sequence XYZXYZ, and the two applications share a four unit cache managed by LRU. When the two applications runs concurrently, their data access sequence interleaves from the perspective of the shared L2 cache. Their interleaving sequence may be XYZABCXYZDEF. Then T_1 encounters

6 cache misses, and T_2 gets 6 cache misses. For T_2 , its first three accessed data XYZ encounters cache misses because they are the first time to be accessed and its last three accessed data XYZ encounters cache misses since its cached data XYZ are evicted by T_1 's data ABC.

The cache interference problem becomes more severe as the number of threads increases, due to the greater disparity between cache size and memory demand. Programmers always would like very large enough cache. However, it is not practical since fast cache memory are built with registers and very expensive. Thus, the shared cache in practice is often small. For example, the size of the last level cache in most CMP systems is at most 8MB. However, many applications are emerging with very large scale of instructions and data sets and aggravate the cache demand. The fact that cache demands of many applications exceed the realistic last level cache size in most CMP systems makes threads contend for the cache fiercely.

Co-scheduling

To address the inter-thread interference problem, a great deal of strategies are proposed. One important technique is co-scheduling, which divides a set of threads into multiple subsets of threads and executes each subset concurrently on one chip of multicore processor [64]. When a thread co-runs with other threads, its performance is often degraded. [64] defines the performance degradation of a thread as the performance gap between its performance when co-running with other threads and its performance when running alone. The goal is to divide the threads into several co-running subsets such that the total performance degradation of all threads is minimal. The technique can alleviate inter-thread interference problem in CMP systems to some extent. However, it cannot resolve the problem absolutely. This is because each subset of co-run threads still share the last level cache using LRU policy and one thread may evict the data of other threads in the same subset. Besides, the overhead of the method is very high since each thread needs to run many times in all possible co-run subsets to gain the performance degradation information to decide how to coschedule the threads.

Cache Partitioning

A better technique to address the inter-thread interference problem is *cache partitioning*. It works by giving each thread a private portion of cache. In this way, each thread only uses the cache partition given to it. When it encounters a cache miss, it only evicts the data cached in its portion of cache, which prevents the possibility of interfering other threads. For example, recall the example for inter-thread interference problem aforementioned. If we give T_1 one unit of cache and T_2 three units of cache by cache partitioning, then T_1 still gets 6 cache misses while T_2 gets only three cache misses. For T_2 , its first three accessed data XYZ encounter cache misses again because they are the first time to be accessed, and its last three accessed data XYZ encounter cache hits since its first three accessed data XYZ are cached in its private portion of cache and are not evicted by T_1 's data ABC.

There has been a great deal of work on cache partitioning [54, 15, 36]. It has been shown to be an effective method to resolve the inter-thread interference problem. For example, cache partitioning improves the total IPC of two CPU benchmarks *art* and *mcf* by 100% over LRU when they share a 4MB L2 cache, and improves the total IPC of *mcf* and *applu* by 32% when they share a 16MB L2 cache [54]. Partitioning cache is especially effective when the cache is not large enough to hold the entire working set, but not too small so that it can hold some critical portion of the working set.

Two key issues arise with the advent of cache partitioning. One issue is how to actually allocate an amount of cache to a thread, and the other is how to determine the right amount of cache to each thread.

Various partitioning techniques have been proposed to allocate cache to threads, including modifying the LRU replacement policy [54], *page coloring* [79, 93], *column caching* [78].

- **Modified LRU replacement policy:** It works by restricting cache block a thread can replace. Suppose a thread desires to be allocated d blocks. When the thread encounters a cache miss, the replacement decision is determined by the number of cache blocks a that actually belongs to the thread. If $a > d$, the thread's least recently used cache blocks will be chosen to be replaced. Otherwise, the least recently used cache block of other over-allocated threads is chosen to be replaced. It has the advantage

of controlling the cache allocation at granularity of cache block. The disadvantage is that the implementation cost is expensive since each block needs to record the thread number that occupies it.

- **Page coloring:** It works by allocating memory pages mapped to a section of cache that a process desires to use. Figure 2.2 illustrates the page coloring cache partitioning mechanism. In a physically shared L2 cache, every physical memory page is mapped to a fixed contiguous group of cache lines and the group is mapped from several fixed physical memory pages. The figure shows that a contiguous group of cache lines (Color A) is mapped from several fixed physical memory pages (Color A). To allocate the cache lines (Color A) to process A, it suffices to allocate physical pages (Color A) to process A. Operating system can achieve the allocation by restricting mapping from the virtual memory pages to physical memory pages (Color A). The advantage of the mechanism is relatively low implementation cost whereas the disadvantage is that it operates at page granularity, which is relatively coarse compared to the first partitioning technique.
- **Column caching:** It partitions the cache space by restricting cache ways that each process can replace. On a miss, the least recently used cache blocks from a subset of cache blocks belonging to the process in a set is chosen to be replaced. Similarly with page coloring, the advantage of the mechanism is relatively low implementation cost whereas the disadvantage is that it operates at way granularity, which is also relatively coarse compared to the first partitioning technique.

The second main issue is how much cache each thread should be allocated at best. To do this, it is crucial to track the cache demand of each thread. A common method is to use cache miss rate curve (MRC), which describes the cache miss rate of a thread given different amount of cache. For example, Figure 2.3 shows the cache miss rate function of a CPU benchmark *art*. It shows that the MRC of *art* drops initially but eventually plateaus after a certain point (3×10^4). It is indicated that even if more than 3×10^4 cache blocks are provided to this application during that interval, it does not benefit in decreasing its

Figure 2.2: Illustration of page coloring

cache miss ratio obviously. In other words, it is best to allocate only 3×10^4 cache blocks to deal with most of accesses made by this application during that interval. [5] explored the dependence of cache size on cache miss rates and found that it satisfies a so-called “square root rule”. They found that if the workload is large, the cache miss rate $M(x)$ for a thread using x amount of cache behaves as $M(x) = M_0 x^{-\alpha}$, for some constant M_0 and α is approximately 0.5.

Before using MRC, a key problem is to gain a thread’s MRC. To do this, a simple method is to run the thread many times under different cache allocation. The disadvantage of the method is the overhead is very high because the thread needs to run many times, which limits the use of the method. In practice, a widely used method to compute MRC is *Mattson’s Stack Algorithm* proposed by Mattson et al. [8], which can track MRC of a thread running under LRU policy. The algorithm maintains a LRU stack which consists of memory addresses generated from the sequence of memory accesses. The top element of the stack is the most recently accessed memory address and the bottom stores the least recently accessed memory address. On each memory access, the algorithm first finds the location of the memory access in the stack and then determines the distance of the current location of the accessed address from the top of the LRU stack, which is the memory access’s *stack distance*. If the memory element is in the stack, the stack distance of the access is the distance of the current location of the accessed address from the top of the LRU stack.

Figure 2.3: Cache miss rate of *art* as a function of cache blocks

Otherwise, the stack distance is regarded ∞ . After the access, the memory element of the access will be moved to the top of the stack, since it is the most recently used at that time. The stack distance of the memory access can be used to determine whether the access would result in a miss or a hit given a certain cache size. That is, for any cache size larger than the stack distance, the access would be regarded as a hit since at the time of the memory access, the corresponding memory element in the cache has not yet been replaced. On the other hand, for any cache size smaller than the stack distance the memory access would be considered as a miss. In order to calculate the MRC, a histogram, H , is computed where $H(d)$ shows the total number of memory accesses with stack distance of d . To calculate each $H(d)$, $H(d)$ is set to be zero at the beginning and increased by one on each memory access with stack distance d . Therefore, the number of misses $M(s)$ for the thread using cache of size s , can be calculated as follows:

$$M(s) = \sum_{d=s+1}^{\infty} H(d)$$

The miss rate $m(s)$ of the thread using s amount of cache is computed as follows:

$$m(s) = M(s) / \left(\sum_{d=1}^s H(d) + M(s) \right)$$

Figure 2.4 illustrates how Mattson's algorithm is used to calculate the MRC of a thread with reference sequence 14114. By Mattson's Stack algorithm, we can calculate a histogram, H , where $H(1) = 1$, $H(2) = 2$, $H(3) = 0$, $H(\infty) = 2$. The number of miss of

Access sequence	1	4	1	1	4																									
Stack	<table border="1"> <tr><td>1</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table>	1					<table border="1"> <tr><td>4</td></tr> <tr><td>1</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table>	4	1				<table border="1"> <tr><td>1</td></tr> <tr><td>4</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table>	1	4				<table border="1"> <tr><td>1</td></tr> <tr><td>4</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table>	1	4				<table border="1"> <tr><td>4</td></tr> <tr><td>1</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table>	4	1			
1																														
4																														
1																														
1																														
4																														
1																														
4																														
4																														
1																														
Distance	∞	∞	2	1	2																									
Histogram	H(1)=0 H(2)=0 H(3)=0 H(∞)=1	H(1)=0 H(2)=0 H(3)=0 H(∞)=2	H(1)=0 H(2)=1 H(3)=0 H(∞)=2	H(1)=1 H(2)=1 H(3)=0 H(∞)=2	H(1)=1 H(2)=2 H(3)=0 H(∞)=2																									

Figure 2.4: Illustration of Mattson's algorithm

the thread using 1, 2, 3 amount of cache is 4, 2, 2, respectively. Thus, the miss rate of the thread using 1, 2, 3 amount of cache is 0.8, 0.4, 0.4, respectively.

A number of partitioning schemes have been proposed to determine the right amount of cache based on each thread's MRC. One scheme is proposed by Kim et al. [36] who balanced the cache miss rates of the threads to optimize the fairness of the system. Fairness is a critical issue in computer systems, since otherwise some threads may starve and run very slowly, which undermines the satisfaction of the thread's owner. They define a metric $M_{ij} = |X_i - X_j|$ to measure the degree of fairness of threads i and j , where X_i, X_j is cache miss rate of thread i and j . To optimize fairness of all threads, it is equivalent to minimize the metric $\sum_{i=1}^n \sum_{j=1}^n M_{ij}$, where n is the number of simultaneous threads. They captured the MRC information of each thread and used them to compute the cache allocation for each thread. They proposed both static and dynamic partitioning schemes. Simulation results show that they improved fairness by up to 47% over LRU. Another common scheme is to minimize the total misses of the threads proposed [54], or maximize the total utility of the cache allocation [15], where the total utility represents the total cache miss reduction of all threads. [54] used a software-based method to measure the MRC information of each thread and applied a greedy algorithm to compute the cache allocation based the MRC information. Since maximizing the total cache miss reduction in [15] is essentially minimizing the total cache misses [54], the optimization goal in [15] is essentially the same with that in [54]. However, [15] proposed a hardware-based method to measure MRC information, which is different from [54].

Besides CMP system, cache partitioning can be also used in single processor system to share cache among multiple processes. For instance, Stone et al. [33] considered to share cache to two processes running in a single processor system using cache partitioning. Suh et al. [33] again applied their partitioning work to improve disk cache hit-ratios. The model is extended to be applicable for more than two processes. [13] proposed a cache partitioning scheme to allocate cache to multiple threads in simultaneous multithreading systems. Trace-driven simulation results show a relative improvement in the L2 hit-rate of up to 40.5% over those generated by the standard LRU, and IPC improvements of up to 17%.

2.2.2 Memory Sharing Techniques

As said before, memory sharing is very significant in virtual machine systems. In virtual machine systems, memory is usually shared by partitioning the memory among multiple virtual machines.

In a typical virtualized system, there is a virtual machine monitor (VMM) or hypervisor, which actually allocates memory to virtual machines. A VMM usually runs multiple virtual machines. Its job is to manage the system's processor, memory and other resources to allocate what each operating system requires. VMM actually allocates memory to a virtual machine by restricting the real physical system memory pages a virtual machines can map to. When a process is running on a virtual machine, the operating system on the virtual machine maps its virtual address to physical address via the process' page table on the operating system, and then the VMM maps the resulting physical address to the real system memory address via the operating system's page tables on the VMM.

With respect to memory allocation in virtual machine systems, a key aspect is how to allocate memory to each virtual machine at best. Most memory allocation is static at the beginning. That is, each virtual machine is given a fixed amount of memory initially. After that, the memory is allocated dynamically to adapt to the dynamic memory requirement of virtual machines. There is much work on allocating memory resource to virtual machines dynamically [81, 92, 40]. Two widely used VMMs, i.e. Xen and VMware, provide a ballooning driver to dynamically allocate host memory. Waldspurger et al. [81] proposed a

scheme to dynamically allocate memory to virtual machines on VMware ESX server. The allocation of a virtual machine is decided by its memory utilization. A sampling method is used to calculate memory utilization. The system periodically samples a random set of pages and the utilization is the ratio of the active pages. A virtual machine with low memory utilization has small allocation and therefore is more likely to reclaim its memory. The drawback of the method is that it can only predict the memory requirement of a virtual machine when there is free memory, and the potential performance loss is not known if some of its memory is reclaimed.

A better method is to use the page miss ratio curve (MRC), which describes the miss ratio based on allocated memory size, to capture memory needs of applications. MRC provides useful information about an application's performance under different amount of memory. For example, Figure 2.5 shows the MRC of the CPU benchmark *bzip* during a particular time interval. It can tell the potential performance loss if some of the application's memory is reclaimed. This method has been widely used to configure the memory hierarchy for applications. For example, Zhou et al. [73] used MRC to dynamically allocate memory among multiple applications to maximize the overall system performance in a multiprogramming environment. Similarly, Zhao et al. [40] used MRC to direct the memory configuration of virtual machines. They proposed Memory Balancer (MEB) to dynamically monitor the MRC of each virtual machine, accurately predicts its memory demands, and periodically reallocates memory to achieve high performance. They used each virtual machine's MRC to determine the memory allocation to minimize the total memory page misses.

Before using memory MRC, a key problem is how to track each thread's memory MRC. Though Mattson's stack algorithm is originally developed to track a thread's cache miss rate under different cache size under LRU replacement policy, it can be also used to determine memory MRC under LRU replacement policy. This can be done by maintaining a LRU stack in which each element stores a memory page number instead a memory access address accessed by the thread. Therefore, this method is used in many works to track memory MRC [73, 40].

Figure 2.5: Page miss ratio vs. memory size curve (MRC) for *bzip* [73]

2.2.3 Distributed Resource Sharing Techniques

As described before, in the distributed resource sharing environment, the resources on the servers need to be allocated among the threads. The resource allocation in this environment is called distributed resource allocation in the thesis. It contains two steps. In the first step, each thread is assigned to a server. Subsequently, the resource at each server are allocated among the threads assigned to it. While our terminology generically refers to threads and servers, distributed resource allocation is used in a range of distributed systems. For example, in a multichip multicore processor, each chip corresponds to a server offering its shared cache as a resource to concurrently executing threads. Each thread is first bound to a chip, after which cache partitioning [55, 54] can enforce an allocation of the chip's shared cache among the assigned threads. As each thread's cache requirement differs, achieving high overall utility requires both a proper mapping of threads to chips and properly partitioning each chip's cache. In a web hosting center, a number of web service threads run on multiple servers and compete for resources such as processing or memory [56, 57]. The host seeks to maximize system utility to run a larger number of services. In [56, 57], they use all or nothing method to allocate resource. Each web service requires certain amount of resource and the system determines to assign it on some server and allocate the amount of resource it wants or does not assign it on any server and give no resource to it. Our work differs in that we use utility functions to track each thread's resource demand.

2.2.4 Makespan and Total Utility

There are lots of multithread programs in computer systems today. In a multi-thread program, different threads may have different length. When a programmer runs his/her multi-thread program, naturally he/she often wants the program to finish as soon as possible. The finishing time of the program is determined by the thread with the largest finishing time, which is defined as makespan.

In computer systems, there are many persistent applications. For example, many web or antivirus services run all the time. Makespan cannot be used to evaluate the performance in these settings since these applications do not finish at all. An effective metric to evaluate the performance in these settings is total utility. Utility is a concept borrowed from economics and it measures the benefit a thread gets from some amount of resource. Total utility is the sum of all threads' utilities and represent the total benefits from all the resources. In computer systems, utility can be used to represent many metrics, such as throughput measured by Instruction per Cycle (IPC), cache miss reduction, cache hit rate. For example, Qureshi et al. [15] considered to share cache to threads using cache partitioning to maximize the total utility, where the utility of each thread represents cache miss reduction given some amount of cache. Zhao et al. [40] studied how to share memory among processes to maximize total utility, where the utility represents hit rate of each process.

Makespan and total utility are two different performance metrics. This is because makespan is determined by threads' job length while total utilities are not affected by threads' job length. For example, consider there are two concurrent threads T_1 and T_2 , which share 8 amount of resource. Their job lengths are 1 and 10 respectively, and their running speed $S(x)$ using x amount of resource behaves $S(x) = x^{0.5}$. To maximize the total utility, both thread T_1 and T_2 should be allocated 4 amount of resource. Under this allocation, threads T_1, T_2 both have finishing time $1/2$ and $10/2 = 5$, respectively and the makespan is 5. This allocation does not guarantee the optimal makespan, since clearly we achieve smaller makespan if we increase thread T_2 's resource and decrease thread T_1 's resource.

2.2.5 Resource Allocation Algorithmic Techniques

The resource allocation problem was first introduced by Koopman [86] in dealing with the optimal distribution of efforts to search for an object whose position is a random variable. Since then, a great deal of work has been done on various resource allocation problems, depending on the form of objective functions (e. g. separate or non-separate) and constraints (e.g. whether the constraint is the sum of all threads' allocation is not larger than the total resource size) or on the type of variables (e.g., continuous or integer). A simple form among all these resource allocation problems is: consider n threads share N amount of resource. Each thread has a utility function f_i , which describes the performance based on the amount of resource it gets. Each thread is allocated x_i (x_i is non-negative integer) amount of resource such that $\sum_{i=1}^n x_i \leq N$. Thread i has utility $f_i(x_i)$ given x_i amount of resource. The total utility of an allocation $(x_1, x_2, \dots, x_n)$ is $\sum_{i=1}^n f_i(x_i)$. The goal is to find an allocation to maximize the total utility.

A common assumption in most of the existing works for the simple resource allocation problem is that the objective function is concave. This makes sense in some settings because it models a diminishing return property, which says the gain of a thread from each unit of resource diminishes with the increasing amount of resource given to it. A number of researchers have proposed efficient polynomial algorithms to find the optimal allocation [82, 83, 85]. The fastest one among them is proposed by Frederickson and Johnson [82] and runs in $O(\max\{n, n \log(N/n)\})$ time. Besides these algorithms, a simple greedy algorithm was developed by Gross [84], and the same algorithm has been rediscovered by many researchers (e.g. Fox [63] and Shih [87]). It allocates resource iteratively and give each resource unit to the user who gains the most. For example, consider there are two threads and 10 units of resource. Each thread has a utility function, which describes its performance function based on the amount of resource it gets. Figure 2.6 shows the utility functions of the two threads. The problem is to allocate the 10 units of resource to thread 1 and 2 to maximize the total utility. The greedy algorithm first gives the first unit to thread 1 since thread 1 increases its utility by 2 while thread 2 increases its utility by 1 from the unit. The second unit will be given to thread 2 since thread 2 increases 1 amount of utility

Figure 2.6: Utility functions

while thread 1 increases 0 amount of utility from the unit. Similarly, the last 8 units are also given to thread 2. Finally, thread 1 is given 1 unit of resource and thread 2 is given 9 units. The running time of this algorithm is $O(Nn)$. However, this time complexity is not polynomial but pseudo-polynomial in the input size. Though its running time is higher than the efficient polynomial time optimal algorithm [82, 83, 85], it is still widely used in practice because it is very easy to be implemented.

For a more complex situation where each thread has a non-concave utility function, the problem is proved to be NP-hard [38]. The optimal algorithms for this problem often uses dynamic programming. Currently, the fastest dynamic programming algorithm is proposed by Ibaraki and Katoh [43]. Let f_i be the utility function of thread i and let $F(i, j)$ be the optimal total utility for the first i threads given j amount of resource. We have the recursive function:

$$F(i, j) = \max_{k=0, \dots, j} (F(i-1, k) + f_i(j-k)) \quad (2.1)$$

The right term in Equation 2.1 represents the maximum total utility for the first i threads in all the ways of splitting the j amount of resource to the first $i-1$ threads and thread i . Thus, Equation 2.1 holds since $F(i-1, k)$ represents the optimal total utility for the first $i-1$ threads using k amount of resource. Using the recursive function, we can compute $F(n, N)$ finally, which is the optimal total utility that is aimed to compute. This dynamic programming algorithm runs in $O(nN^2)$ which is pseudo-polynomial time in the input size.

In practice, the greedy algorithm [63] is often used directly to solve the problem. However, it does not guarantee to get the optimal solution.

Besides the algorithms described in this section, there are a wide range of optimal and approximation algorithms proposed for various resource allocation problems. We defer the detailed description of these works later in the section of related works in Chapter 3, 4 and 5.

Chapter 3

Makespan Optimal Resource Allocation

3.1 Introduction

In this chapter, we consider a multicore setting in which a set of concurrent threads, one running on each core, each executes a task of a certain length using a shared resource. We introduce the problem of determining the resource allocation that minimizes the *makespan*, or the amount of time needed for all the threads to finish their tasks. We conduct a theoretical study of this problem, and present provably optimal results. Previous allocation algorithms have only been evaluated experimentally, and were not shown to achieve optimal makespan. Our model is quite general: we only assume that a thread's performance increases with its resource allocation, and that the performance function is concave. This model is a generalization of the widely used "square root rule" on cache miss rates [2, 3, 4, 53]. With this model, we show that there always exists an optimal allocation with a simple form: it is *static* and *balanced*, meaning that each thread is allocated the same amount of resource throughout its execution, and the amounts ensure that all the threads finish at the same time. In general, this optimal allocation might consist of arbitrary real numbers, and so we cannot compute it exactly. However, we present an algorithm based on the characterization that efficiently computes for any $\epsilon > 0$ an allocation whose makespan is at most $1 + \epsilon$ times the optimal. The time complexity of the algorithm is $O(n \log \frac{n}{\epsilon} \log \frac{n}{\epsilon p})$, where n is the number of threads and p is a value that depends on the optimal solution. We apply our algorithm in cache partitioning to test the performance. We compared our algorithm with several existing partitioning algorithms. We found that our algorithm shows the greatest improvement over existing algorithms when the number of threads is relatively large, the

lengths of the jobs are unequal but similar and the cache miss rate functions of the threads are quite dissimilar. In particular, our algorithm performs up to 59% better than uniform partitioning, and 22% better than a utility-based partitioning algorithm [54, 15]. Finally, our algorithm works quickly in practice. Using 8 threads and $\epsilon = 10^{-3}$, an unoptimized Matlab implementation on a desktop PC runs in only 0.01 seconds, and with 16 threads it takes less than 0.03 seconds.

The rest of the chapter is organized as follows. In Section 3.2, we describe related works of the chapter. Section 3.3 formally defines our model and the problem we solve. Section 3.4 gives a mathematical characterization of the optimal schedule, and Section 3.5 presents an algorithm to compute a nearly optimal schedule and analysis for the algorithm. Section 3.6 applies our algorithm in cache partitioning and compares it to other cache partitioning algorithms. Finally, Section 3.7 concludes the chapter.

3.2 Prior and Related Work

Our work is close to cache partitioning. Stone et al. [33] are the first to study optimal cache partitioning between two competing processes with the goal of minimizing the total cache miss rate. Their paper shows the optimal allocation occurs when the miss rate derivatives of the processes are equal. Suh et al. [12] considered a multiprogramming model in which threads are scheduled with arbitrary quanta, and also proposed an algorithm that minimizes the total miss rate. Both these works are aimed at improving multithreaded performance in a single core processor, whereas we consider multicore processors where each core runs a single thread.

Suh et al. [54] in a different paper investigated dynamic cache partitioning on a CMP. They again minimize the total miss rate, by allocating cache to threads that derive the most utility from the cache. However, this may not result in the minimum makespan, since the makespan is determined by the thread with the maximum number of cache misses instead of the total cache misses of all the threads. For example, our algorithm produces a different allocation when some of the threads' tasks are longer than others. We show in section 3.6 that our algorithm can finish tasks up to 22% faster than the algorithm in [54].

In recent years with the proliferation of CMPs, a number of methods for implementing cache partitioning or finding effective partitions have been proposed. Qureshi et al. [15] introduced utility-based cache partitioning, a low overhead runtime mechanism to partition the cache based on the utility threads obtain from their allocation. Chang et al. [16] proposed a partitioning that optimizes throughput, fairness, and QoS simultaneously. Moreto et al. [20] focused on the distribution of cache misses. They gave each miss a cost according to its impact on the final performance and minimized the total cost using cache partitioning. Lin et al. [21] proposed an efficient software approach to support cache partitioning in operating systems, overcoming earlier shortcomings of evaluating cache partitioning using only simulations. While the previous cache partitioning techniques are limited to coarse-grained allocations, Sanchez and Kozyrakis [24] proposed a technique to partition the cache at cache line granularity. Muralidhara et al. [23] proposed a cache partitioning scheme that aims to minimize execution time by allocating more cache to threads with higher miss rates. Kandemir et al. [26] also proposed an algorithm with the same objective. Our work differs from [23] and [26] in that their algorithms are evaluated experimentally, while we prove the optimality of our allocation.

Our algorithm uses a thread's MRC in order to determine its cache allocation. Zhou et al. [28], Tam et al. [93] and He et al. [30] presented effective methods to collect MRCs online with high accuracy. There has been a large body of work on the effect of cache size on cache misses. Smith [4] and Srinivasan et al. [5] explored the dependence of cache size on cache miss rates and found that it satisfies a so-called "square root rule". They found that if the workload is large, the cache miss rate is observed to decrease as a power of the cache size, where the exponent is approximately $-\frac{1}{2}$. The property is used to determine the optimal L2 cache size of each core in a CMP design so that the total cost in terms of energy and delay is minimal [7].

3.3 Model

3.3.1 Resource behavior

We first present a model for the performance of a thread as a function of the amount of resource it is allocated. We consider a multicore system with n cores, in which each core

executes a single thread. The core all uses a shared resource, whose size we normalize to 1. Given a thread i , we let $S_i(x)$ be the number of instructions that thread i executes when it is allocated $x \in [0, 1]$ amount of resource. Different threads may have different S_i functions. We assume all the functions satisfy the following properties.

Assumption 3.1. *Let i be any thread. Then $S_i(x)$ satisfies the following.*

- (i) $S_i(0) = 0$.
- (ii) $S_i(x)$ is increasing in x .
- (iii) $S_i(x)$ is concave.

The first assumption basically says that a thread must be allocated some amount of resource in order to run. This assumption is motivated by a setting where threads access cache. While a thread can run by accessing only main memory without using any cache, it would operate so slowly that we can practically consider it not to be running. The second assumption says that a thread always runs faster when given more resource. Our results would still hold if we only assume $S_i(x)$ is nondecreasing in x , but this would complicate our analysis. Thus for simplicity, we assume S_i is increasing. Finally, the third assumption is motivated by the observation that increasing the amount of cache given to a thread by a factor $\lambda > 1$ usually increases its speed less than λ times. This is not always true, especially for threads that can entirely fit within cache if their cache allocation is increased. Nevertheless, it holds in many practical settings. The motivation for our assumptions comes from the empirically observed model in [4, 5], which says that the cache miss rate for a thread using cache size x behaves as $M(x) = M_0 x^{-\alpha}$, for some $M_0 > 0$, and $\alpha \in (0.3, 0.7)$ depending on properties of the thread. We show below that our assumptions generalize this widely used model.

Consider a setting where a thread runs on a modern processor. The speed of a modern processor is largely determined by the time needed to fetch data from cache or main memory. Let the latency to cache be one time unit, and let $L > 1$ be the latency to access main memory. Then the average time for a thread to execute one instruction is given

by $M(x)L + 1 - M(x) = 1 + M(x)(L - 1)$, where $M(x)$ is the cache miss rate using x amount of cache. Thus, the number of instructions the thread executes per time unit is $S(x) = \frac{1}{1+M(x)(L-1)}$. Assuming $M(x) = M_0x^{-\alpha}$, we have $S(0) = 0$. Also, since $M(x)$ is decreasing in x , $S(x)$ is increasing in x . Finally, to show $S(x)$ is concave, we compute the second derivative $S''(x) = \frac{M_0(L-1)\alpha x^{\alpha-2} [(-\alpha-1)x^\alpha + M_0(L-1)(\alpha-1)]}{(x^\alpha + M_0(L-1))^3}$. Since $0 < \alpha < 1$, we have $S''(x) < 0$, and so $S(x)$ is concave. Hence, the model from [5] satisfies Assumption 3.1.

One last assumption we make is that time is discrete. Thus, only a bounded number of events occur in any finite time interval. Our results also hold for continuous time, but the analysis is more complicated. In terms of notation, we write $[j..k]$ to denote the set $\{j, j+1, \dots, k\}$, for natural numbers $j \leq k$. Finally, for notational convenience we define $S_i(x) = 0$ for any i and $x < 0$, and we let $\frac{x}{0} = \infty$ for all $x > 0$.

3.3.2 Problem

The problem consists of a set of n threads, where each thread i needs to perform a task of size ℓ_i . The threads share a resource of size 1. At any time t , we denote the amount of resource allocated to thread i by $x_i(t)$. A *schedule* consists of a resource allocation for each thread, $x_1(\cdot), \dots, x_n(\cdot)$. A schedule is *feasible* if for any t , we have $\sum_{i=1}^n x_i(t) \leq 1$. The number of instructions that thread i executes per time unit when it is given x amount of resource is $S_i(x)$, where S_i is any function that satisfies Assumption 3.1. The *finishing time* T_i of thread i is the amount of time it takes i to perform all ℓ_i instructions. That is, T_i is the smallest value such that $\sum_{t=0}^{T_i} S_i(x_i(t)) \geq \ell_i$. The makespan T of a schedule is the maximum finishing time of all the threads: $T = \max(T_1, \dots, T_n)$. An *optimal* schedule is one whose makespan is minimal. Our goal is to find an optimal schedule.

3.4 Properties of the Optimal Schedule

In this section, we present a characterization of optimal schedules. Intuitively, we show that there exists an optimal schedule in which all the threads run concurrently, using up the entire resource. Furthermore, each thread uses the same cache allocation throughout its execution, and all the threads finish at the same time. Additionally, we show that any schedule which satisfies these properties is optimal. In the next section, we will use the

characterization to compute an approximately optimal schedule. Formally, we prove the following.

Theorem 3.1. *There exists an optimal schedule $x_1(\cdot), \dots, x_n(\cdot)$ with makespan T satisfying the following properties.*

- (i) *For any t , $\sum_{i=1}^n x_i(t) = 1$.*
- (ii) *For every thread i , there exists a constant c_i such that $x_i(t) = c_i$ for all $t \in [0, T]$.*
- (iii) *For every thread i , $T_i = T$.*

Theorem 3.2. *Assume a schedule with allocation functions $x_1(\cdot), \dots, x_n(\cdot)$ and makespan T satisfies the following properties.*

- (i) *For any t , $\sum_{i=1}^n x_i(t) = 1$.*
- (ii) *For every thread i , there exists a constant c_i such that $x_i(t) = c_i$ for all $t \in [0, T]$.*
- (iii) *For every thread i , $T_i = T$.*

Then the schedule is optimal.

We say a schedule is *static* if all threads have the same resource allocation throughout the schedule, as in the second property above. We say a schedule is *balanced* if all threads finish at the same time, as in the third property above.

Theorem 3.2 follows easily from Theorem 3.1. To prove Theorem 3.1, we first prove Lemmas 3.1, 3.2 and 3.3. Lemma 3.1 simply says that an optimal allocation uses the entire resource. Lemma 3.2 says that given a thread whose resource allocation may change over time, we can use another allocation in which the thread is always given the average of its resource from the first allocation, and the thread will do at least as much work in the second allocation as in the first. While Lemma 3.2 deals with a single thread, Lemma 3.3 applies to a set of threads sharing a resource. It says that any resource allocation for the threads can be transformed to a static allocation whose makespan is no larger than that of the first.

Lemma 3.1. *There exists an optimal schedule in which the total amount of resource used by the threads at any time is 1.*

Proof. If we have a schedule that does not use all the resource at some time, we can allocate the unused resource uniformly to all the threads. Since the work function is increasing, each thread will finish all its task earlier than in the original schedule. Hence, an optimal schedule always uses up all the resource. $\square$

Definition 3.1. Given a thread with allocation function $x(\cdot)$ and work function $S(\cdot)$, we define $W(x, t_1, t_2) = \sum_{t=t_1}^{t_2} S(x(t))$ to be the amount of work the thread does during a time interval $[t_1, t_2]$ under x and S .

Lemma 3.2. Let $[0, t_1]$ be a time interval, and consider a thread with allocation function $x(\cdot)$ and work function $S(\cdot)$. Define another allocation for the thread $x'(t) = (\sum_{t=0}^{t_1} x(t))/t_2$ for all $t \in [0, t_2]$, where $t_2 \geq t_1$. Then $W(x, 0, t_1) \leq W(x', 0, t_2)$.

Proof. Define a third allocation function for the thread $x''(t) = (\sum_{t=0}^{t_1} x(t))/t_1$ for all $t \in [0, t_1]$. For brevity, let $W_0 = W(x, 0, t_1)$, $W_1 = W(x'', 0, t_1)$ and $W_2 = W(x', 0, t_2)$. We will prove $W_0 \leq W_1 \leq W_2$.

Since we assume that time is discrete, let $0 = \tau_0 < \tau_1 < \dots < \tau_b = t_1$ be the set of times at which $x(\cdot)$ changes values during $[0, t_1]$. That is, we have $x(t) = x_i$ for some x_i , and all $t \in [\tau_{i-1}, \tau_i)$ and $i \in [1..b]$. Let $\Delta\tau_i = \tau_i - \tau_{i-1}$ for $i \in [1..b]$. Then we have

$$W_0 = \sum_{i=1}^b \Delta\tau_i S(x_i), \quad W_1 = t_1 S\left(\frac{\sum_{i=1}^b \Delta\tau_i x_i}{t_1}\right)$$

Since $S(x)$ is concave, then $W_0 \leq W_1$ by Jensen's inequality. Next, let $\bar{x}_1 = (\sum_{t=0}^{t_1} x(t))/t_1$ and $\bar{x}_2 = (\sum_{t=0}^{t_2} x(t))/t_2$. Again by Jensen's inequality, we have

$$S(\bar{x}_1)t_1 + S(0)(t_2 - t_1) \leq S\left(\frac{\bar{x}_1 t_1 + 0 \cdot (t_2 - t_1)}{t_2}\right) t_2.$$

Since $S(0) = 0$ and $\bar{x}_1 t_1 = \bar{x}_2 t_2$, then we get $S(\bar{x}_1)t_1 = W_1 \leq S(\bar{x}_2)t_2 = W_2$. Hence the lemma follows. $\square$

Lemma 3.3. Consider a schedule with allocation functions $x_1(\cdot), \dots, x_n(\cdot)$, finishing times $T_1, \dots, T_n$ and makespan T . Then there exists a static schedule with allocation functions $x'_1(\cdot), \dots, x'_n(\cdot)$ and makespan $\leq T$.

Proof. For any thread $i \in [1..n]$, define $x'_i(t) = (\sum_{t=0}^{T_i} x_i(t))/T$ for all $t \in [0, T]$. Then $x'_1(\cdot), \dots, x'_n(\cdot)$ is a static schedule. We will show $x'_1(\cdot), \dots, x'_n(\cdot)$ is a feasible schedule, and then show it has makespan $\leq T$. To show it is feasible, we show for any $t \in [0, T]$ that $\sum_{i=1}^n x'_i(t) \leq 1$. We have

$$\begin{aligned} \sum_{i=1}^n x'_i(t) &= \sum_{i=1}^n \frac{\sum_{t=0}^{T_i} x_i(t)}{T} \\ &\leq \frac{1}{T} \sum_{i=1}^n \sum_{t=0}^T x_i(t) \\ &= \frac{1}{T} \sum_{t=0}^T \sum_{i=1}^n x_i(t) \\ &\leq \frac{1}{T} \sum_{t=0}^T 1 = 1 \end{aligned}$$

The first inequality follows because $T = \max(T_1, \dots, T_n)$, and the second inequality follows because $x_1(\cdot), \dots, x_n(\cdot)$ is a feasible schedule, and so $\sum_{i=1}^n x_i(t) \leq 1$ for all t . Hence $x'_1(\cdot), \dots, x'_n(\cdot)$ is a feasible schedule.

Next, to show the makespan is $\leq T$, we observe that for any i , $W(x_i, 0, T_i) \leq W(x'_i, 0, T)$ by Lemma 3.2. Thus, since thread i finishes its task by time T_i under allocation x_i , it also finishes its task by time T under x'_i . Since this holds for every thread, the makespan of schedule $x'_1(\cdot), \dots, x'_n(\cdot)$ is at most T . Hence, the lemma follows. $\square$

Putting together the above, we can now prove Theorem 3.1.

Proof of Theorem 3.1. Let X be an optimal schedule. We can assume that X is static, since otherwise by Lemma 3.3 there exists a static schedule whose makespan is no larger than X 's. By Lemma 3.1, X always uses the entire resource. We now show X is balanced. Suppose not, and let T be the makespan of X , and let thread i have finishing time $T_i < T$. i must have been allocated a positive amount of resource x_i up to time T_i , since if i was allocated no resource then $S_i(0) = 0$ and i can never finish. When i finishes at T_i , we can allocate i 's resource x_i uniformly to the other threads. This causes every other thread to finish earlier than they do in X , since $S_j(x)$ is increasing in x for every j . However, this leads to a schedule whose makespan is smaller than X , which is a contradiction. Hence X is balanced, and the theorem is proved. $\square$

Using Theorem 3.1, we can prove Theorem 3.2.

Proof of Theorem 3.2. Suppose for contradiction the schedule is not optimal. Then by Theorem 3.1 there exists an optimal schedule X' with allocation functions $x'_1(\cdot), \dots, x'_n(\cdot)$ and makespan $T' < T$ satisfying the following properties.

- (i) For any t , $\sum_{i=1}^n x'_i(t) = 1$.
- (ii) For every thread i , there exists a constant c'_i such that $x'_i(t) = c'_i$ for all $t \in [0, T]$.
- (iii) For every thread i , $T'_i = T'$.

Since for every i , $T_i = T$ and $T'_i = T'$, then $T'_i < T_i$. Since both schedules are static, we have $T_i = \frac{\ell_i}{S_i(c_i)}$ and $T'_i = \frac{\ell_i}{S_i(c'_i)}$, and so $S_i(c'_i) > S_i(c_i)$. So since $S_i(x)$ is increasing in x , then $c'_i > c_i$ for every i . But then $\sum_{i=1}^n c'_i > \sum_{i=1}^n c_i = 1$, which is a contradiction. Thus, the theorem follows. $\square$

3.5 Scheduling algorithm

In this section, we describe our scheduling algorithm and present its analysis. Pseudocode for the algorithm is given in Algorithm 3.1. Given an $\epsilon \leq 1$, the algorithm calls the function $M(\epsilon')$, where $\epsilon' = \frac{1}{7}\epsilon \in (0, \frac{1}{7}]$, and the function returns a schedule with makespan $\leq 1 + 7\epsilon' = 1 + \epsilon$ times the optimal.

The algorithm only considers static schedules in which each thread is allocated a constant amount of resource. Given a thread $i \in 1, \dots, n$ that is allocated resource $x \in (0, 1)$, we let $f_i(x) = \frac{\ell_i}{S_i(x)}$ denote i 's finishing time. f_i is a decreasing convex function with $f_i(0) = \infty$. Our algorithm tries to find values $x_1^*, \dots, x_n^*$ such that $f_1(x_1^*) = \dots = f_n(x_n^*) = M^*$, for some M^* . This corresponds to a static and balanced schedule that is optimal by Theorem 3.2.

The algorithm starts with an arbitrary value $x_1 \in [0, 1]$, then finds values $x_2, \dots, x_{n-1}$ such that $f_i(x_i) = f_1(x_1)$, for $i \in [2..n-1]$. After this, it sets $x_n = 1 - \sum_{i=1}^{n-1} x_i$. The goal is to make $x_i = x_i^*$ for all i . This is done by comparing $t_1 = f_1(x_1)$ and $t_2 = f_n(x_n)$. If $t_1 < t_2$, then $x_1 > x_1^*$ and the algorithm decreases x_1 . If $t_1 > t_2$, then $x_1 < x_1^*$ and x_1 is

increased. The adjustments to x_1 are done using binary search. In the pseudocode, x_1^L and x_1^U denote the lower and upper bounds for the search.

In order to find x_i with $f_i(x_i) = f_1(x_1) = t_1$ for $i \in [2..n-1]$, we call $\text{BALANCE}(t_1, x_1, \epsilon', c)$. Here $c = 1 - x_1$ is the amount of resource available to threads $2, \dots, n$. BALANCE calls $\text{BALANCE-1}(i, t_1, x_1, \epsilon')$ for each thread $i \in [2..n-1]$. We denote the return value of the call by x'_i , and we want $x'_i = x_i$. However, since x_i is an arbitrary real value, there may be some error in x'_i . BALANCE-1 works by binary searching for a x'_i satisfying $f_i(x'_i) = f_1(x_1)$ until the error in x'_i is $\leq \frac{1}{n}\epsilon' \min(x_1, x_1^L)$ in line 51. In the pseudocode, x_i^L and x_i^U denote the lower and upper bounds in the search for x_i .

After BALANCE finds the allocation for a thread i in line 34, it subtracts this from the remaining resource amount c . After threads $2, \dots, n-1$ have all been allocated, it gives the remaining resource to thread n on line 38. That is, it sets $x'_n = 1 - x_1 - \sum_{i=2}^{n-1} x'_i$. Then it compares thread n 's finishing time to thread 1's, in order to determine how to adjust x_1 . BALANCE returns the adjustment direction and the allocations for threads $2, \dots, n$ back to procedure M in line 44.

The main problem we face is that each $x'_i, i \in [2..n-1]$, is an approximation for x_i . Thus, errors in the x'_i values can cause an error in x'_n , the approximation for x_n . This can then cause the binary search for x_1 to go in the wrong direction. For example, even if $t_1 = f_1(x_1) < t_2 = f_n(x_n)$, we may have $t_1 > f_n(x'_n)$ because $x_n < x'_n$. This would make us increase x_1 whereas we should decrease its value. We deal with this by adjusting the error tolerances in the algorithm so that x_1 's search direction can only be wrong when x_1 is already very close to its correct value x_1^* , as shown in Lemma 3.6. Using this approximately correct value for x_1^* , we can then find approximately correct values for $x_2^*, \dots, x_n^*$.

3.5.1 Algorithm analysis

We now show that Algorithm 3.1 efficiently computes a $1 + \epsilon$ optimal allocation. We first show in Lemma 3.7 that thread 1 finishes in time $T_1 \in [M^*, (1 + 5\epsilon')M^*]$. The proof of Lemma 3.7 requires Lemmas 3.4 and 3.6. We then show in Claim 3.8 that every thread $i = 2, \dots, n-1$ finishes in time $T_i \leq (1 + 7\epsilon')M^*$, and also show that thread n finishes in time $T_n \leq M^*$. The accuracy of the algorithm then follows by setting $\epsilon' = \frac{1}{7}\epsilon$. To show the

Algorithm 3.1 $M(\epsilon')$ computes a $1 + 7\epsilon'$ optimal schedule

```

1: procedure  $M(\epsilon')$ 
2:   LABEL()
3:    $x_1^L \leftarrow 0, x_1^U \leftarrow 1$ 
4:   repeat
5:      $x_1 \leftarrow \frac{1}{2}(x_1^L + x_1^U), t_1 \leftarrow \ell_1/S_1(x_1)$ 
6:      $\triangleright$  compute search direction  $b$ , and allocations  $A$ 
7:      $[b, A] \leftarrow \text{BALANCE}(t_1, x_1, \epsilon', 1 - x_1)$ 
8:     if  $b = \text{INC}$  then
9:        $x_1^L \leftarrow x_1$ 
10:    else
11:       $x_1^U \leftarrow x_1$ 
12:    end if
13:  until  $(x_1^U - x_1^L < \epsilon' x_1)$ 
14:   $\triangleright$  compute thread 1's final allocation, finishing time
15:   $x_1 \leftarrow x_1^L - \epsilon' x_1^L, A[1] \leftarrow x_1, t_1 \leftarrow \ell_1/S_1(x_1)$ 
16:   $\triangleright$  compute the final allocation for threads  $2, \dots, n$ 
17:   $[b, A] \leftarrow \text{BALANCE}(t_1, x_1, \epsilon', 1 - x_1)$ 
18:  for  $i = 2$  to  $n$  do
19:     $t_i = \ell_i/S_i(A[i])$ 
20:  end for
21:   $t = \max(t_1, \dots, t_n)$   $\triangleright t$  is the makespan
22:  return  $[t, A]$ 
23: end procedure

24: procedure  $\text{LABEL}()$ 
25:   for  $i = 1$  to  $n$  do
26:      $t_i = \ell_i/S_i(\frac{1}{n})$ 
27:   end for
28:   find the thread  $k$  with  $t_k = \max(f_1(\frac{1}{n}), \dots, f_n(\frac{1}{n}))$ 
29:   swap the labels of threads 1 and  $k$ 
30: end procedure

31: procedure  $\text{BALANCE}(t, x, \epsilon', c)$ 
32:   for  $i = 2$  to  $n - 1$  do
33:      $\triangleright$  compute thread  $i$ 's allocation
34:      $A[i] \leftarrow \text{BALANCE-1}(i, t, x, \epsilon')$ 
35:      $c \leftarrow c - A[i]$ 
36:   end for
37:    $\triangleright$  compute thread  $n$ 's allocation and finishing time
38:    $A[n] = c, t' \leftarrow \ell_n/S_n(c)$ 
39:   if  $t' < t$  then
40:      $b = \text{INC}$   $\triangleright$  increase thread 1's allocation
41:   else if  $t' \geq t$  then
42:      $b = \text{DEC}$   $\triangleright$  decrease thread 1's allocation
43:   end if
44:   return  $[b, A]$ 
45: end procedure

```

```

46: procedure BALANCE-1( $i, t, x, \epsilon'$ )
47:    $x_i^L \leftarrow 0, x_i^U \leftarrow 1$ 
48:   if  $f_i(1) \geq t$  then
49:     return 1
50:   end if
51:   while  $x_i^U - x_i^L > \frac{1}{n} \epsilon' \min(x, x_i^L)$  do
52:      $c \leftarrow \frac{1}{2}(x_i^L + x_i^U), t' \leftarrow \ell_i/S_i(c)$ 
53:     if  $t' < t$  then
54:        $x_i^U \leftarrow c$ 
55:     else
56:        $x_i^L \leftarrow c$ 
57:     end if
58:   end while
59:   return  $x_i^L$ 
60: end procedure

```

efficiency of the algorithm, we first prove in Lemma 3.9 that thread 1 uses $\geq \frac{1}{n}$ resource in the optimal allocation. We then use this in Lemma 3.4 to show that the overall complexity of the algorithm is $O(n \log \frac{10n}{\epsilon} \log \frac{7n}{\epsilon p})$, where $p = \min(\frac{7}{2n(7+\epsilon)}, \frac{7n}{7n+\epsilon} \hat{x}_2, \dots, \frac{7n}{7n+\epsilon} \hat{x}_{n-1})$, and where $\hat{x}_i = f_i^{-1}(2(1 + \frac{1}{7}\epsilon)M^*)$ for $i = 2, \dots, n-1$. Thus, the complexity of the algorithm depends on the number of threads n and the accuracy parameter ϵ , as well as the actual value of the optimal solution, reflected through the value p .

We first show the following lemma, which we call the *diminishing returns* property.

Lemma 3.4. *For any $i = 1, \dots, n$, any $x \geq 0$ and $\lambda \geq 1$, $\frac{f_i(x)}{f_i(\lambda x)} \leq \lambda$.*

Proof. It suffices to show that for any $x \geq 0$ and $\lambda > 1$, we have $\frac{S_i(\lambda x)}{S_i(x)} \leq \lambda$. Since $S(x)$ is concave, we have $\gamma S_i(0) + (1 - \gamma)S_i(c) \leq S_i((1 - \gamma)c)$ for all $c \geq 0, \gamma \in (0, 1)$. So $\frac{S_i(c)}{S_i((1-\gamma)c)} \leq \frac{1}{1-\gamma}$, since $S_i(0) = 0$. Then, by letting $\lambda = \frac{1}{1-\gamma}$ and $x = (1 - \gamma)c$, we have $\lambda > 1, x \geq 0$, and $\frac{S_i(\lambda x)}{S_i(x)} \leq \lambda$. $\square$

Recall that for $i = 2, \dots, n-1$, x_i is defined so that $f_i(x_i) = f_1(x_1) = t_1$. However, if x_1 is too large, we may have $x_i \geq 1$. The following lemma shows that as long as this doesn't happen, BALANCE-1 computes a good approximation to x_i .

Lemma 3.5. *Let $i \in 2, \dots, n-1$, and suppose $x_i < 1$. Let x_i^L, x_i^U be the final values of these variables when a call to BALANCE-1(i, t_1, x_1, ϵ) returns, and let x_i' be the value the function returns. Then we have $x_i \in [x_i^L, x_i^U]$, and $0 \leq x_i - x_i' \leq \frac{1}{n} \epsilon' \min(x, x_i^L)$.*

Proof. Since $x_i < 1$ and x_1^L and x_1^U are initialized to 0 and 1, respectively, in line 47, we initially have $x_i \in [x_1^L, x_1^U]$. We have $f_1(x_1) = f_i(x_i) > f_i(1)$ since $x_i < 1$. Hence, BALANCE-1 will run the binary search in lines 51-58, during which $x_i \in [x_1^L, x_1^U]$ will continue to hold. Thus, the first statement in the lemma follows. From line 59, the final value returned by BALANCE-1 is x_i^L , i.e. $x'_i = x_i^L$. Therefore, we have $x_i \geq x_1^L = x'_i$, where the inequality follows because $x_i \in [x_1^L, x_1^U]$. Also, $\frac{1}{n}\epsilon' \min(x, x_i^L) \geq x_i^U - x_i^L = x_i^U - x'_i \geq x_i - x'_i$, where the first inequality holds because $x_i^U - x_i^L \leq \frac{1}{n}\epsilon' \min(x, x_i^L)$ when BALANCE-1 returns, by line 51, and the second inequality again follows because $x_i \in [x_1^L, x_1^U]$. $\square$

Lemma 3.6. *At any point in the execution of procedure M, one of the following holds.*

- (i) $x_1^L \leq x_1^* \leq x_1^U$.
- (ii) $x_1^L > x_1^*$ and $x_1^L - x_1^* \leq \epsilon' x_1^L$.
- (iii) $x_1^U < x_1^*$ and $x_1^* - x_1^U \leq \epsilon' x_1^U$.

Proof. Since x_1^L and x_1^U only change values in the repeat loop from line 4 to 13, it suffices to consider a particular iteration of this loop. Consider the value of x_1 after line 5 of this iteration. We say we move in the *right direction* if $x_1 < x_1^*$ and line 9 occurs, or $x_1 > x_1^*$ and line 11 occurs. Otherwise we say we move in the *wrong direction*. It is easy to see that if the lemma holds at the beginning of the iteration and we move in the right direction, then the lemma also holds at the end of the iteration. So, suppose we move in the wrong direction in the iteration. We will show then that

$$|x_1 - x_1^*| \leq \epsilon' x_1$$

We claim this implies the lemma. Indeed, if $x_1 < x_1^*$ and we move in the wrong direction, then line 11 occurs, which sets $x_1^L = x_1$. Then, since $|x_1 - x_1^*| \leq \epsilon' x_1$, we get $x_1^* - x_1^L \leq \epsilon' x_1^L$ at the end of the iteration. Similarly, if $x_1 > x_1^*$ and we move in the wrong direction, then line 9 occurs which sets $x_1^U = x_1$, and so we get $x_1^U - x_1^* \leq \epsilon' x_1^U$ at the end of the iteration.

To show that $|x_1 - x_1^*| \leq \epsilon' x_1$ when we move in the wrong direction, we prove the contrapositive. Thus, we show that if either $x_1 - x_1^* > \epsilon' x_1$ or $x_1 - x_1^* < -\epsilon' x_1$, then we move in the right direction. In the first case, there are two subcases. In the first subcase, there exists an $i \in \{2, \dots, n-1\}$ with $x_i \geq 1$. Then $f_i(1) \geq f_i(x_i) = f_1(x_1) = t_1$, and so the condition on line 48 holds and BALANCE-1 returns 1. Then we have $x'_n = 1 - x_1 - \sum_{i=2}^{n-1} x'_i \leq 0$, so that $t' = \frac{t_n}{S_n(x'_n)} = \infty^1$. Hence the condition on line 41 holds, and we move in the right direction.

In the second subcase, we have $x_i < 1$ for all $i \in [2..n-1]$. Thus by Lemma 3.5, we have $0 \leq x_i - x'_i \leq \frac{1}{n}\epsilon' x_1$ for all $i \in [2..n-1]$. Also, since $x_1 - x_1^* > \epsilon' x_1 > 0$, we have $f_1(x_1) < f_1(x_1^*) = M^*$, and we have $x_i > x_i^*$ for all $i \in [2..n-1]$ because $f_i(x_i) = f_1(x_1) < M^* = f_i(x_i^*)$. Then

$$\begin{aligned} x'_n &= 1 - x_1 - \sum_{i=2}^{n-1} x'_i \\ &< 1 - x_1 - \sum_{i=2}^{n-1} x_i + \epsilon' x_1 \\ &< 1 - x_1 - \sum_{i=2}^{n-1} x_i^* + \epsilon' x_1 \\ &< 1 - x_1^* - \epsilon' x_1 - \sum_{i=2}^{n-1} x_i^* + \epsilon' x_1 \\ &= 1 - \sum_{i=1}^{n-1} x_i^* = x_n^* \end{aligned}$$

The first inequality follows because $x_i - x'_i \leq \frac{1}{n}\epsilon' x_1$, the second follows because $x_i > x_i^*$, and the third follows because $x_1 - x_1^* > \epsilon' x_1$. So, we have $f_n(x'_n) > M^* = f_n(x_n^*) > f_1(x_1)$. Thus, the condition on line 41 again holds, and we move in the right direction.

Now, consider the second case where $x_1 - x_1^* < -\epsilon' x_1$. Then $f_1(x_1) > M^*$, and for all $i \in [2..n-1]$ we have $x_i < x_i^* \leq 1$, because $f_i(x_i) = f_1(x_1) > M^* = f_i(x_i^*)$. Thus by Lemma 3.5, we have $0 \leq x_i - x'_i \leq \frac{1}{n}\epsilon' x_1$, and we can use a similar derivation as above to get $x'_n > x_n^*$, so that $f_n(x'_n) < M^* < f_1(x_1)$. So line 39 holds, and we also move in the right direction.

¹Recall that we set $S(x) = 0$ for $x < 0$ and $\frac{x}{0} = \infty$ for $x > 0$.

In summary, if $|x_1 - x_1^*| > \epsilon' x_1$, then we move in the right direction. Hence the lemma follows. $\square$

We now first prove that thread 1's makespan is approximately optimal.

Lemma 3.7. *Thread 1 has finishing time $T_1 \in [1, 1 + 5\epsilon')M^*$.*

Proof. In the following, let x_1 be the final allocation for thread 1 computed on line 15. To prove the lemma, we will show

$$\frac{(2 - \epsilon')(1 - \epsilon')}{(2 + \epsilon')(1 + \epsilon')} x_1^* \leq x_1 \leq x_1^* \quad (3.1)$$

Let x_1^U, x_1^L denote the upper bound and lower bound for x_1^* when $M(\epsilon')$ returns. Then by line 15, we have

$$x_1 = (1 - \epsilon')x_1^L \quad (3.2)$$

By Lemma 3.6, we either have $x_1^* \geq x_1^L$ (cases 1 and 3) or $x_1^* < x_1^L$ and $x_1^L - x_1^* \leq \epsilon' x_1^L$ (case 2). In both cases, we get $x_1 \leq x_1^*$, and so the upper bound on x_1 in Equation 3.1 holds.

Next, since $1 - \epsilon' > 0$, to get a lower bound for x_1 , we need to lower bound x_1^L . Since $x_1^U - x_1^L \leq \frac{1}{2}\epsilon'(x_1^L + x_1^U)$ by line 13, we get $x_1^U \leq \frac{2+\epsilon'}{2-\epsilon'}x_1^L$. We again use Lemma 3.6. In case 1 of the lemma, we have $x_1^L \geq \frac{2-\epsilon'}{2+\epsilon'}x_1^U$, and so $x_1^L \geq \frac{2-\epsilon'}{2+\epsilon'}x_1^*$. In case 2, we have $x_1^L > x_1^*$. Finally, in case 3, using the fact that $x_1^* - x_1^U \leq \epsilon' x_1^U$, we get $x_1^U \geq \frac{x_1^*}{1+\epsilon'}$. Combining this with the fact that $x_1^L \geq \frac{2-\epsilon'}{2+\epsilon'}x_1^U$, we get

$$x_1^L \geq \frac{2 - \epsilon'}{(2 + \epsilon')(1 + \epsilon')} x_1^* \quad (3.3)$$

Substituting this into Equation 3.2, we get the lower bound on x_1 in Equation 3.1.

Using Equation 3.1 and the diminishing returns property, we have $\frac{f_1(x_1)}{f_1(x_1^*)} \leq \frac{x_1^*}{x_1} \leq \frac{(2+\epsilon')(1+\epsilon')}{(2-\epsilon')(1-\epsilon')}$. So, since we assumed $\epsilon' \leq \frac{1}{7}$, we have $\frac{x_1^*}{x_1} \leq \frac{(2+\epsilon')(1+\epsilon')}{(2-\epsilon')(1-\epsilon')} \leq 1 + 5\epsilon'$. Also, we have $f_1(x_1) \geq f_1(x_1^*)$, since $x_1 \leq x_1^*$ by Equation 3.1. Hence, as $f_1(x_1^*) = M^*$ and $T_1 = f_1(x_1)$, we have $T_1 < [1, 1 + 5\epsilon')M^*$. $\square$

Next, we prove that the makespans of threads 2, ..., $n - 1$ are approximately optimal.

Lemma 3.8. *Each thread $i \in [2..n-1]$ has finishing time $T_i \leq (1 + 7\epsilon')M^*$, and thread n has finishing time $T_n \leq M^*$.*

Proof. In this proof, let x_1 denote the final allocation for thread 1 computed on line 15, and for $i \in [2..n-1]$, let x_i be defined such that $f_i(x_i) = f_1(x_1)$, and let x'_i be the final allocation for thread i returned on line 17. Since $T_i = f_i(x'_i)$, we will prove $f_i(x'_i) \leq (1 + 7\epsilon')M^*$. To do this, we first prove

$$f_i(x'_i) \leq (1 + \frac{1}{n}\epsilon')T_1. \quad (3.4)$$

Here T_1 is thread 1's finishing time. Let x_i^L, x_i^U be the final lower and upper bounds for x_i when the call to BALANCE on line 17 returns. By Equation 3.1 in Lemma 3.7, $x_1 \leq x_1^*$, so $T_1 = f_1(x_1) \geq M^*$, so $f_i(x_i) = f_1(x_1) \geq M^* = f_i(x_i^*)$, and hence $x_i \leq x_i^* < 1$. So by Lemma 3.5, we have $x_i \in [x_i^L, x_i^U]$. From line 59, we have that $x'_i = x_i^L$, and so since $x_i^U - x_i^L \leq \frac{1}{n}\epsilon'x_i^L$ by line 51, we have $x_i - x'_i \leq \frac{1}{n}\epsilon'x'_i$. Thus, $\frac{x_i}{x'_i} \leq 1 + \frac{1}{n}\epsilon'$, and so by the diminishing returns property, $\frac{f_i(x'_i)}{f_i(x_i)} \leq 1 + \frac{\epsilon'}{n}$. Since $f_i(x_i) = f_1(x_1) = T_1$, Equation 3.4 follows. Combining the Equation with the fact that $T_1 \leq (1 + 5\epsilon')M^*$ by Lemma 3.6, and $\epsilon' \leq \frac{1}{7}$, we get $T_i \leq (1 + 5\epsilon')(1 + \frac{\epsilon'}{n})M^* \leq (1 + 7\epsilon')M^*$, and so the first part of the lemma follows.

To show the second part, we have $x'_n = 1 - x_1 - \sum_{i=2}^{n-1} x'_i$ and $x_n^* = 1 - \sum_{i=1}^{n-1} x_i^*$. Now, since $x_1 \leq x_1^*$ by Equation 3.1 in Lemma 3.7, and we showed $x'_i \leq x_i^*$ for $i \in [2..n-1]$, we have $x'_n \geq x_n^*$. Thus, the second part of the lemma follows. $\square$

Combining Lemmas 3.7 and 3.8, we obtain the following.

Theorem 3.3. *Let $\epsilon' \leq \frac{1}{7}$. The makespan of the schedule returned by $M(\epsilon')$ is at most $(1 + 7\epsilon')M^*$, where M^* is the optimal makespan.*

Given an $0 < \epsilon < 1$, we can set $\epsilon' = \frac{1}{7}\epsilon \leq \frac{1}{7}$, and so by Theorem 3.3, $M(\epsilon')$ returns a $1 + \epsilon$ optimal schedule.

We now bound the time complexity of Algorithm 1. To do this, we first show that thread 1's optimal resource allocation is not too small.

Lemma 3.9. $x_1^* \geq \frac{1}{n}$

Proof. We will show that $M^* \leq f_1(\frac{1}{n})$, which implies the lemma since f_1 is increasing. Indeed, if $M^* > f_1(\frac{1}{n})$, since after calling LABEL we have $f_1(\frac{1}{n}) \geq f_i(\frac{1}{n})$ for all $i \in [1..n]$, then $f_i(\frac{1}{n}) < M^*$ for all $i \in [1..n]$. So, since $f_i(x_i^*) = M^*$, we have $x_i^* < \frac{1}{n}$ for all $i \in [1..n]$, which contradicts the fact that $\sum_{i=1}^n x_i^* = 1$. $\square$

We now show the time complexity of Algorithm 3.1. Since we are using binary search, the complexity of the algorithm depends on the values we are searching for. This dependence is captured by the values p and $\hat{x}_i$ below.

Theorem 3.4. *Algorithm 3.1 has time complexity $O(n \log \frac{10n}{\epsilon} \log \frac{7n}{\epsilon p})$, where $p = \min(\frac{7}{2n(7+\epsilon)}, \frac{7n}{7n+\epsilon}\hat{x}_2, \dots, \frac{7n}{7n+\epsilon}\hat{x}_{n-1})$, and $\hat{x}_i = f_i^{-1}(2(1+\epsilon)M^*)$ for $i \in [2..n-1]$.*

Proof. We assume that all the function evaluations in M, for example evaluating $S_i(x)$, can be done in $O(1)$ time. Let K , K_1 and K_2 be upper bounds on the complexities of a single call to procedures M, BALANCE and BALANCE-1, respectively. M contains k calls to BALANCE on line 7, for some number k , followed by another call to BALANCE on line 17. All the other parts of M can be done in $O(n)$ time. BALANCE contains $n-2$ calls to BALANCE-1; the other parts of BALANCE can be done in $O(n)$ time. Thus, we have $K_1 = O((n-2)K_2 + n)$, and $K = O((k+1)K_1 + n) = O(nkK_2)$. We will focus on bounding k and K_2 .

To bound k , consider the loop from line 4 to 13. We have $x_1^U \geq x_1^L$, and so $x_1^L + x_1^U \geq 2x_1^L$. Also, from Equation 3.3 in the proof of Lemma 3.7, we have $x_1^L \geq \frac{(2-\epsilon')}{(2+\epsilon')(1+\epsilon')}x_1^*$, and so $x_1^L + x_1^U \geq \frac{2(2-\epsilon')}{(2+\epsilon')(1+\epsilon')}x_1^*$. The loop runs until $x_1^U - x_1^L < \epsilon'x_1$, where $x_1 = \frac{1}{2}(x_1^L + x_1^U)$, and so the number of iterations of the loop is at most $\log \frac{2}{\epsilon'(x_1^L - x_1^U)} \leq \log \frac{(2+\epsilon')(1-\epsilon')}{\epsilon'(2-\epsilon')x_1^*}$. So, since $x_1^* \geq \frac{1}{n}$ by Lemma 3.9, then the number of calls to BALANCE on line 7 is $k \leq \log \frac{(2-\epsilon')(1+\epsilon')n}{\epsilon'(2-\epsilon')}$.

Next, we bound the time complexity K_2 of a call to BALANCE-1(i, t, x, ϵ'). Notice that K_2 is on the order of the number of iterations of the loop from line 51 to 58; for simplicity, we let K_2 equal the number of iterations. Recall that x_i is defined so that $f_i(x_i) = f_1(x) = t_1$. There are two cases. If $x_i \geq 1$, then BALANCE-1 returns immediately from line 49. Otherwise, $x_i < 1$. Consider the final values of x_i^L, x_i^U in BALANCE-1 before the function

returns. Then by line 51, $K_2 \leq \log \frac{n}{\epsilon' \min(x_1, x_i^L)}$. Since $x_i < 1$, we have $x_i \in [x_i^L, x_i^U]$ by Lemma 3.5. Also, we have $x_i^U - x_i^L \leq \frac{1}{n} \epsilon' x_i^L$ when BALANCE-1 terminates. Thus, we get $x_i^L \geq \frac{n}{n+\epsilon'} x_i$, and so

$$K_2 \leq \log \frac{n}{\epsilon' \min(x_1, \frac{n}{n+\epsilon'} x_i)} \quad (3.5)$$

To upper bound K_2 , it suffices to lower bound x_1 and x_i . For x_1 , consider two cases, either x_1 was computed on line 5, or line 15. In the first case, consider the values of x_1^L, x_1^U in procedure M. Since $x_1 = \frac{1}{2}(x_1^L + x_1^U)$ and $x_1^L \geq 0$, we have $x_1 \geq \frac{1}{2}x_1^U$. Also, from Lemma 3.6, we have $x_1^* \leq (1 + \epsilon')x_1^U$. Hence $x_1^U \geq \frac{1}{1+\epsilon'}x_1^*$, and so $x_1 \geq \frac{1}{2(1+\epsilon')}x_1^*$. When x_1 is computed on line 15, we have by Equation 3.1 that $x_1 \geq \frac{(2-\epsilon')(1-\epsilon')}{(2+\epsilon')(1+\epsilon')}x_1^*$. Since $\epsilon' \leq \frac{1}{7}$, then $\frac{(2-\epsilon')(1-\epsilon')}{(2+\epsilon')(1+\epsilon')} > \frac{1}{2(1+\epsilon')}$, and so again $x_1 \geq \frac{1}{2(1+\epsilon')}x_1^*$. By Lemma 3.9, we have $x_1^* \geq \frac{1}{n}$, and so $x_1 \geq \frac{1}{2n(1+\epsilon')}$.

Finally, we bound x_i . We have $f_i(x_i) = f_1(x_1) = t_1$. Since $x_1 \geq \frac{1}{2(1+\epsilon')}x_1^*$, then $f_1(x_1) \leq 2(1 + \epsilon')f_1(x_1^*) = 2(1 + \epsilon')M^*$, where the first inequality follows by the diminishing returns property. Thus, since $f_i(x)$ is decreasing in x , we have $x_i = f_i^{-1}(t_1) \geq \hat{x}_i \equiv f_i^{-1}(2(1 + \epsilon')M^*)$, for $i \in [2..n-1]$. Combining this with the bound for x_1 and using Equation 3.5, we have $K_2 \leq \log \frac{n}{\epsilon' p}$, where $p = \min(\frac{1}{2n(1+\epsilon')}, \frac{n}{n+\epsilon'}\hat{x}_2, \dots, \frac{n}{n+\epsilon'}\hat{x}_{n-1})$. Plugging this into the overall complexity of the algorithm $K = O(kK_1 + n) = O(nkK_2)$, using the bound for k derived earlier and recalling $\epsilon' = \frac{1}{7}\epsilon$ where $\epsilon \leq 1$, the theorem follows. $\square$

3.6 Evaluation and comparisons

In this section, we apply our algorithm in cache partitioning to test the performance of our algorithm. We compare the makespan achieved by our algorithm with the makespan from three other cache partitioning algorithms used in practice or described in the literature. The first algorithm is uniform partitioning, which gives each thread the same amount of cache. Next, we consider Suh et al's algorithm [54], which works by iteratively allocating each block of cache to the thread that derives the most utility from the block. Finally, Kim et al's algorithm [36] works by equalizing the cache miss rates of all the threads: we will call this *fair partitioning*. In our experiments, each thread is described by its job length ℓ and its cache miss rate function. We use the model from [5], and assume that a thread using

x amount of cache has miss rate $M(x) = M_0 x^{-\alpha}$, where M_0 is a normalization constant that is the same for all the threads, and $\alpha \in (0, 1)$ is an exponent which is particular to that thread. We assume that a thread performs $S_i(x) = \frac{1}{1+(L-1)M_i(x)}$ amount of work per unit time, where L represents the latency of accessing main memory compared to cache. We set $L = 100$ as a common value from practice. We describe how the α and ℓ values for different threads are selected in our experiments later.

3.6.1 Performance compared to uniform allocation

We first compare our algorithm with the uniform partitioning. We will look at the effects of the number of threads, the job lengths and the miss rate functions of the threads on our performance. We first consider the effects of job length and miss rate. To do this, we fix the number of threads to $n = 10$, and set the miss rate exponent α of these threads to be an arithmetic progression with difference d , starting from 0.01. We then vary the value of d between 0 and 0.1. For example, using $d = 0.05$, we consider 10 threads with α values 0.01, 0.06, 0.11, $\dots$, 0.46. Also, we set the job lengths of the threads to be a geometric progression with ratio r , and vary r from 1 to 1.8. We set $\epsilon' = 10^{-6}$, so that our algorithm is guaranteed to achieve an accuracy better than 10^{-5} . Figure 3.1 shows, for each setting of d and r , the ratio I_1 of the makespan generated by uniform partitioning versus the makespan of our algorithm. Since our algorithm finds the (nearly) optimal makespan, I_1 is at least 1. We see that in the best case, for $r = 1.1$ and $d = 0.09$, our schedule finishes 1.28 times faster than uniform allocation. We also see that I_1 initially grows as r increases. That is, as the variation in job lengths grows, our algorithm performs better compared to uniform partitioning. However, beyond some point, I_1 starts to decrease with r . This is because for large r , the job lengths are so different that small jobs can finish very quickly, and then reallocate their cache to the larger jobs. In other words, it begins to look as if there is only a single large job, in which case our algorithm and uniform allocation perform the same. We also see a similar trend with the value of d , where I_1 initially increases with d , but later stops increasing. Lastly, we see that I_1 varies more as a function of d than r , indicating that differences in the miss rates of the threads has a bigger effect on performance than different job lengths.

Figure 3.1: Comparison to uniform allocation for varying job lengths and miss rate exponents

Figure 3.2: Comparison to uniform allocation as a function of thread count

Next, we consider how the number of threads affects performance. To do this, we vary the number of threads n from 2 to 16. For each value of n , we create 100,000 random problem instances, where for each instance, we create n threads whose job lengths are distributed uniformly at random in the range $[1, 3]$, and whose miss rate exponents are distributed uniformly at random in the range $[0.01, 0.99]$. Again, we set $\epsilon = 10^{-6}$. Figure 3.2 shows for n between 2 and 16 both the average and maximum values of I_1 , the relative performance of uniform partitioning compared to our algorithm, over the 100,000 trials. For example, for $n = 5$, it shows that in the 100,000 trials, our algorithm performed on average 1.26 times better than uniform, and performed at most 1.59 times better. We see that our algorithm performs better as n increases, up to about $n = 5$, at which point it plateaus. The best performance is with 11 threads, where we performed 1.59 times better, using threads with miss rate exponents and job lengths equal to

$$\vec{\alpha} = [0.03, 0.04, 0.07, 0.12, 0.13, 0.14, 0.17, 0.20, 0.21, 0.29, 0.95]$$

$$\vec{\ell} = [1.52, 1.51, 1.72, 1.62, 1.64, 1.28, 1.26, 1.72, 1.34, 1.63, 1.87]$$

Figure 3.3: Comparison to utility-based partitioning for varying job lengths and miss rate exponents

We see that in this case, the job lengths are moderately different, while the miss rate exponents consist of mostly small values, and one large value. Lastly, we note that there is some fluctuation in the value of the maximum I_1 . We believe this may be due to not running enough random trials for large values of n , since the number of possible job lengths and miss rate exponents increases exponentially in n .

3.6.2 Performance compared to utility-based partitioning

Next, we compare our makespan to the makespan generated by the utility maximizing partitioning from [54]. A similar algorithm was described in [15]. We run a similar set of experiments as above, except for we set $\epsilon = 10^{-3}$, because the complexity of utility-based partitioning is $O(\frac{n}{\epsilon})$, and our experiments involving thousands of trials would take too long with smaller ϵ values. First, we look at the effect of job lengths and miss rate exponents. Here again, we consider an arithmetic progression of miss rate exponents with difference d and a geometric progression of job lengths with ratio r . Figure 3.3 shows I_2 , the ratio of the makespan from utility partitioning versus the makespan of our algorithm. We perform up to 1.17 times better for $r = 1.35$ and $d = 0.07$. Also, as with uniform partitioning, the performance improvement initially increases with d and r , but later decreases.

Next, we consider the effect of the number of threads on our performance. As before, we run random trials, this time limited to 10,000 trials to save time. Figure 3.4 shows our average and maximum performance increase compared to utility-based partitioning. We see that our average performance increase plateaus to around 1.10, while our maximum performance increase plateaus to around 1.2. Our best performance is with 16 threads,

Figure 3.4: Comparison to utility-based partitioning as a function of thread count

where we perform 1.22 times better in a job instance where the miss rate exponents were fairly evenly distributed between 0.11 to 0.64, and the job lengths varied between 1.00 to 1.97. Lastly, we again observe fluctuations in our maximum performance increase, likely caused again by not taking enough random samples.

3.6.3 Performance compared to fair partitioning

Lastly, we compare our makespan to the one generated by the algorithm from [36], which tries to equalize the cache miss rates of all the threads. Here also, we set $\epsilon = 10^{-3}$, and run similar experiments as before. Figure 3.5 shows I_3 , the ratio of the makespan from fair partitioning versus ours as we vary r and d . We see that we perform up to 1.16 times better, for $r = 1.35$ and $d = 0.08$. Again, the performance initially increases then decreases with d and r . Finally, Figure 3.6 shows the effect of thread number on our performance. Here we run 10,000 samples, and we only consider up to 10 threads, due to the time needed to run fair partitioning. We achieve our best performance using 10 threads, with miss rate exponent values varying evenly between 0.06 to 0.66, and job lengths between 1.01 to 1.98.

In summary, the experiments show that our algorithm can perform significantly better than existing cache partitioning algorithms. Our best performance tends to come from job instances with a wide range of miss rate exponents, and similar sized jobs. We also mention that our algorithm ran quite quickly in practice. Even though speed was not our main concern and we used an unoptimized Matlab implementation on a desktop PC, it took only about 0.01 seconds to run our algorithm with $\epsilon = 10^{-3}$ and 8 threads, and about 0.05 seconds to run with $\epsilon = 10^{-6}$ and 16 threads.

Figure 3.5: Comparison to fair partitioning for varying job lengths and miss rate exponents

Figure 3.6: Comparison to fair partitioning as a function of thread count

3.7 Summary

In this chapter, we studied how to allocate a shared resource in a computer system in order to minimize the makespan of a set of threads. We showed that the optimal schedule occurs when all the threads are allocated a constant amount of resource and all threads complete at the same time. Based on the property, we also presented an efficient algorithm that finds a nearly optimal resource allocation. Simulation results show that our algorithm can produce schedules between 1.22 to 1.59 times shorter than those from existing allocation algorithms.

Chapter 4

Fast Optimal Nonconcave Resource Allocation

4.1 Introduction

The resource allocation problem for optimal total utility has been widely studied in the literature [83, 85, 63, 43, 37, 38]. A common assumption in many works is that the *throughput functions*, giving each process's performance as a function of its allocated resources, are concave. This assumption leads to efficient algorithms to compute the optimal allocation [33, 73, 61]. However, while the concavity assumption has certain theoretical justifications and is frequently observed in practice, there are also a number of settings in which it does not hold. For example, Figure 1.1 shows the IPC (instructions per clock) rates for applications from the SPEC benchmark as a function of their cache allocation. Several IPC functions are nonconcave. One reason is that sometimes a highly accessed portion of the working set fits entirely within cache once the cache allocation exceeds a threshold. Likewise, nonconcavities can occur whenever there is a "qualitative" change in performance once the resource allocation meets certain criteria.

In our work, we begin by observing that while realistic throughput functions are not necessarily concave, they are also not arbitrary. In particular, the functions often consist of a small number of *segments*, where within each segment the function is concave or convex. For example, the IPC curve of "swim" in Figure 1.1 consists of two concave segments, and the curve for "galgel" consists of one concave and one convex segment. Using this assumption, we present an algorithm to compute an optimal allocation of the shared resource that maximizes the total throughput. Our algorithm runs in $O(s\alpha(m)m(\log m)^2)$ time,

where s is the maximum number of segments in the throughput function for any thread and $\alpha(m)$ is the inverse Ackermann function. In most settings, s and $\alpha(m)$ are both small constants; for example, s is at most 3 for all the SPEC applications. Thus, compared to the $O(nm^2)$ running time of existing optimal algorithms, our algorithm is almost $O(m)$ times faster. This speed is especially important in settings where the optimal allocation needs to be computed multiple times and on the fly, as occurs when the throughput functions can change over time [54]. In some situations it suffices to find an approximately optimal allocation. We also present an algorithm that finds an allocation with total throughput at least $1 - \epsilon$ times the optimal, and runs in time $O(\frac{s}{\epsilon} \alpha(\frac{n}{\epsilon}) n^2 \log \frac{n}{\epsilon} \log m)$, for any $\epsilon > 0$. A $1 - \epsilon$ approximation can be found using existing methods by reducing resource allocation to the multiple choice knapsack problem [46]. This algorithm takes $O(\frac{n^2 m}{\epsilon})$ time, which is almost $O(m)$ times slower than our algorithm. A novel aspect of our algorithms is that, while most current exact and $1 - \epsilon$ approximate algorithms use dynamic programming, we relate the resource allocation problem to an *upper envelope* computation and make use of efficient algorithms from computational geometry [45].

In addition to our improved exact and approximate allocation algorithms, we also introduce an efficient algorithm to find a constant factor estimate of the maximum total throughput. This task is an important step in many approximation algorithms for resource allocation and related problems [68, 69]. Existing algorithms typically have running times proportional to m , while our algorithm runs in $O(n^2 \log n)$ time. Hence, our algorithm is much faster for the typical case of $n \ll m$. We show that instead of considering all possible allocation values when maximizing the throughput, it suffices to consider a much smaller set of “rounded” values. This result may be of independent interest.

We conducted experiments to test the performance of our algorithms. For $m = 1M$, our exact algorithm was over 215 times faster than the dynamic programming method. Also, for randomly created throughput functions, our algorithm achieved 33% higher throughput than the greedy algorithm.

The rest of the chapter is organized as follows. In Section 4.2, we describe related works, followed by Section 4.3 which formally defines our model. Section 4.4 presents an algorithm to find an optimal allocation and give analysis for the algorithm. Section 4.5

proposes a fully polynomial approximation algorithm and its analysis. In Section 4.6, we show our experiments and results. Finally, Section 4.7 summarizes our work in this chapter.

4.2 Prior and Related Work

Resource allocation techniques have been proposed for a wide range of computer systems. Qureshi and Patt [55] introduced a low overhead utility based cache partitioning mechanism for multicore systems to maximize total processor performance. Zhou et al. [73] proposed a method to dynamically track memory page misses and reallocate memory to minimize the total number of misses. A key problem for virtualization in datacenters and clouds is how to maximize resource utilization [32]. Minarolli et al. [31] proposed a CPU allocation scheme for virtual machines in the cloud. Zhao et al. [40] proposed a memory monitoring method for virtual machines and a dynamic memory allocation mechanism. Chase et al. [61] considered utility functions of customers in hosting centers and dynamically allocated resources to save energy. Yehuda et al. [52] allocated memory in cloud systems to maximize vendor profit. Lee et al. [50] proposed a utility based algorithm to allocate power in a CDMA wireless network.

There are a number of algorithmic techniques used to determine an optimal allocation. Fox et al. [63] studied the concave utility¹ optimization problem and presented a greedy algorithm which iteratively allocates resources to the process that derives the highest increased utility. The algorithm produces optimal solutions only for concave utility functions. Rajkumar et al. [38] showed the general resource allocation problem is NP-hard and also presented a nonoptimal greedy allocation algorithm. Stone et al. [33] and Suh et al. [54] studied optimal cache allocation for competing threads in a multicore processor with the goal of minimizing the total cache miss rate. They apply Fox's algorithm for concave miss rate functions and a nonoptimal heuristic for nonconcave functions. Ibaraki et al. [43] gave a dynamic programming algorithm to find the optimal allocation in $O(nm^2)$ time, which may be prohibitive for large m 's. Maximizing the total system throughput can be considered as a type of multiple choice knapsack (MCKP) problem. Several exact and

¹In this chapter we use the terms utility and throughput interchangeably.

approximate algorithms have been proposed for MCKP [51, 46], including a $1 - \epsilon$ approximate algorithm by Lawler [69] that runs in $O(\frac{nm}{\epsilon})$ time, and a $\frac{4}{5}$ approximate algorithm by Gens and Levner [68] that runs in $O(nm \log n)$ time.

Our resource allocation algorithm translates the throughput maximization problem to a set of upper envelope computations. Upper envelopes have been studied extensively in computational geometry. Hart et al. [45] proposed a divide and conquer algorithm to compute the envelope of n line segments in $O(n\alpha(n) \log n)$ time. Hershberg et al. [44] improved the running time to $O(n \log n)$ and extended the algorithm to compute the envelope of curved segments.

Our algorithm uses a thread's utility function to determine its resource allocation. The function can be determined by running the thread multiple times using different allocations. To decrease the overhead of this method, Qureshi et al. [55] have proposed efficient hardware based methods to measure the utility functions.

4.3 Model

We now describe the model used in the chapter of the thesis. All values are assumed to be natural numbers unless otherwise specified. We consider n processes $p_1, \dots, p_n$ sharing a common resource of total size m . Each process p_i is allocated x_i units of resources, where x_i must be in p_i 's *allocation range* $[b_i, b'_i]$, i.e. $0 < b_i \leq x_i \leq b'_i \leq m$. p_i is called *active* if $x_i \geq b_i$, otherwise it is called *inactive*. We assume $\sum_{i=1}^n b'_i \geq m$, as otherwise there are redundant resources. For any $r \leq m$, an allocation $(x_1, \dots, x_n)$ is *r-feasible* if $\sum_{i=1}^n x_i \leq r$. Each p_i has an associated *throughput function* f_i giving p_i 's performance as a function of its resource allocation. We assume each f_i has the following properties.

- (i) $f_i(x)$ is nonnegative and nondecreasing in x .
- (ii) $f_i(x)$ consists of at most $s \geq 1$ *segments*, each of which are either concave or convex.

That is, there exist values $s+1$ values $b_i^1, \dots, b_i^{s+1}$, with $b_i = b_i^1 \leq \dots \leq b_i^{s+1} = b'_i + 1$, such that within each interval $[b_i^j, b_i^{j+1} - 1]$, $f_i(x)$ is either *concave* or *convex*².

² $f_i(x)$ is concave in an interval if for any x where $x, x+1$ and $x+2$ all belong to the interval, we have $f_i(x+2) - f_i(x+1) \leq f_i(x+1) - f_i(x)$; it is convex if $f_i(x+2) - f_i(x+1) \geq f_i(x+1) - f_i(x)$ whenever $x, x+1$ and $x+2$ all belong to the interval.

The *total throughput* of an allocation $(x_1, \dots, x_n)$ is $\sum_{i=1}^n f_i(x_i)$. Our goal is to find a m -feasible allocation with maximum throughput, which we call an *optimal allocation*.

Given a set of functions $h_1, \dots, h_t$, we define the *upper envelope* of the functions to be a function H that is the pointwise maximum of the h_i 's. That is, $H(x) = \max\{h_1(x), \dots, h_t(x)\}$ for all x .

4.4 Exact Algorithm

In this section we present an efficient algorithm to find an optimal allocation. We first describe the existing dynamic programming based method for solving the problem [43]. Our algorithm uses some of the same notation, though our techniques are different.

4.4.1 Dynamic programming algorithm

The dynamic program iteratively finds optimal allocations for increasing sets of processes. Given $r \in [1, m]$ and $k \in [1, n]$, let $F_k(r)$ be the maximum throughput of any r -feasible allocation in which only processes $p_1, \dots, p_k$ may be active. That is,

$$F_k(r) = \max \left\{ \sum_{i=1}^n f_i(x_i) \mid (\forall i \in [1, k] : x_i \in \{0\} \cup [b_i, b'_i]) \wedge \left(\sum_{i=1}^k x_i \leq r \right) \wedge (x_{k+1} = \dots = x_n = 0) \right\} \quad (4.1)$$

For any $r \in [0, m]$ we have

$$F_k(r) = \max_{x \in \{0\} \cup [b_k, b'_k]} \{f_k(x) + F_{k-1}(r - x)\} \quad (4.2)$$

This holds because to maximize the throughput for the first k processes, process p_k is either inactive or is allocated a positive amount of resources x in its allocation range. After this, the remaining $r - x$ resources must be allocated optimally to $p_1, \dots, p_{k-1}$. Using this equation, $F_k(r)$ can be computed in $O(m)$ time. We can compute the maximum total throughput $F_n(m)$ in $O(nm^2)$ time by finding all $F_k(r)$ in increasing order of r and k .

4.4.2 A faster algorithm

We now describe how to compute the optimal allocation faster by transforming the problem into a series of upper envelope computations. In the rest of this section, fix a k , and assume we know $F_{k-1}(r)$ for all $r \in [0, m]$. For $j \in [1, s]$, let the function $F_k^j(r)$ to be the maximum throughput for the first k processes using total resource r , where p_k 's allocation is in the j 'th interval $[b_k^j, b_k^{j+1} - 1]$ of its allocation range. Similar to Equation 4.2, we have

$$F_k^j(r) = \max_{x \in [b_k^j, b_k^{j+1} - 1]} \{f_k(x) + F_{k-1}(r - x)\} \quad (4.3)$$

Also, let $F_k^0(r) = F_{k-1}(r)$ be the maximum throughput for $p_1, \dots, p_k$ using r resource, where p_k is inactive. Then we have $F_k(r) = \max\{F_k^0(r), \dots, F_k^s(r)\}$.

We know $F_k^0(r)$, since we already have $F_{k-1}(r)$. Fix a $j \geq 1$ for the rest of this section. We show how to compute $F_k^j(r)$ for all $r \in [0, m]$ in $O(\alpha(m)m(\log m)^2)$ time. For this, define the following functions, for each $r \in [0, m]$.

$$h_r^j(x) = f_k(x - r) + F_{k-1}(r), \text{ for } x \in [r + b_k^j, r + b_k^{j+1} - 1] \quad (4.4)$$

Outside its domain $[r + b_k^j, r + b_k^{j+1} - 1]$, h_r^j is undefined. $h_r^j(x)$ represents the maximum throughput for the first k processes using x resources, where the first $k - 1$ processes are given r resources and the remaining $x - r$ resources are given to p_k , and where in addition p_k 's allocation lies in the j 'th interval $[b_k^j, b_k^{j+1} - 1]$. These functions are useful because as we show in Lemma 4.1, the upper envelope of $h_0^j, \dots, h_m^j$ is exactly the function F_k^j . Furthermore, we can compute the upper envelope quickly because, as we show in Lemma 4.2, each pair of the h_r^j functions cross each other at most once, where crossing is defined as follows.

Definition 4.1. Given two functions f, g and a value x , we say f and g cross each other at x if $f(x) \leq g(x)$ and $f(x + 1) > g(x + 1)$.

Figure 4.1 illustrates our algorithm ALLOC. Function f_k consists of two concave segments. Functions $h_0^1, \dots, h_m^1$ consist of copies of f_k in the range $[b_k^1, b_k^2)$, shifted to lie on top of F_{k-1} . Similarly $h_0^2, \dots, h_m^2$ are copies of f_k in the range $[b_k^2, b_k^3)$ lying on top of a

(a) Functions F_{k-1} and f_k , with f_k consisting of two concave segments.

(b) Each function h_r^1 is a shifted copy of the first segment of f_k . Intuitively speaking, these functions are laid on top of F_{k-1} to form the upper envelope H_k^1 , part of which is shown in blue.

(c) The h_r^2 functions are shifted copies of the second segment of f_k . These functions are laid on top of a shifted copy of F_{k-1} to form the upper envelope H_k^2 , part of which is shown in blue.

Figure 4.1: Illustration of Algorithm 4.1

shifted copy of F_{k-1} . H_k^1 and H_k^2 , shown in blue, are the upper envelopes of $h_0^1, \dots, h_m^1$ and $h_0^2, \dots, h_m^2$, resp. We show later that $H_k^1 = F_k^1$ and $H_k^2 = F_k^2$.

The code for ALLOC is shown in Algorithm 4.1. The outer loop of ALLOC loops through the processes. Lines 4-7 compute $H_k^1, \dots, H_k^s$ for process p_k by calling the upper envelope procedure UPPER. Here, each H_k^j is the upper envelope of functions $h_0^j, \dots, h_m^j$. Line 10 sets H_k to the pointwise maximum of $H_k^1, \dots, H_k^s$. We describe line 9 in more detail when we describe procedure UPPER. Lastly, ALLOC returns the maximum m -feasible throughput $H_n(m)$.

Algorithm 4.1 Algorithm 4.1 for finding an optimal allocation

```

1: procedure ALLOC( $f_1, \dots, f_n$ )
2:    $H_1 \leftarrow f_1$ 
3:   for  $k \leftarrow 2$  to  $n$  do
4:     for  $j \leftarrow 1$  to  $s$  do
5:        $S \leftarrow \{h_r^j \mid r = 0, \dots, m\}$ 
6:        $H_k^j \leftarrow \text{UPPER}(S)$ 
7:     end for
8:      $H_k^0 \leftarrow H_{k-1}$ 
9:     convert  $H_k^j$  to arrays for  $j \in [1, s]$ 
10:     $H_k \leftarrow \text{pointwise max of } H_k^0, \dots, H_k^s$ 
11:  end for
12:  return  $H_n(m)$ 
13: end procedure

14: procedure UPPER
15:   Input: A list of functions  $S$ 
16:   Output: The upper envelope of the functions
17:   if  $|S| = 1$  return  $S$ 
18:   partition  $S$  into sets  $S_1$  and  $S_2$  of equal size
19:    $H_1 \leftarrow \text{UPPER}(S_1)$ ;  $H_2 \leftarrow \text{UPPER}(S_2)$ 
20:    $H \leftarrow \text{merge } H_1 \text{ and } H_2$ 
21:   return  $H$ 
22: end procedure

```

We now describe the procedure UPPER. It follows the divide and conquer approach proposed in [45]. We outline the approach and describe some terminology relevant in later parts of the paper. For more details, we refer the reader to [45, 44]. UPPER breaks the functions $h_0^j, \dots, h_m^j$ into two equal size sets (to within one) and recursively computes the upper envelope of each set. Then it merges the two envelopes to produce the final output. An

upper envelope is represented by a list of labeled intervals. A label r for an interval means function h_r^j lies on the upper envelope during the interval. The intervals are contiguous and listed from left to right. They are also maximal in the sense that consecutive intervals have different labels.

The merging process requires finding the crossing point of two functions $h_{r_1}^j$ and $h_{r_2}^j$. We describe how to do this using binary search. This is the main difference between our algorithm and [45]. Both $h_{r_1}^j$ and $h_{r_2}^j$ are defined using the j 'th segment of function f_k . The search interval is initially set to the intersection of $h_{r_1}^j$ and $h_{r_2}^j$'s domains. Assume WLOG that $h_{r_1}^j \geq h_{r_2}^j$ at the right endpoint of the search interval. If $h_{r_1}^j \geq h_{r_2}^j$ at the left endpoint as well, then there is no crossover point. Otherwise, we compare $h_{r_1}^j$ and $h_{r_2}^j$ at the midpoint x of the search interval. If $h_{r_1}^j$ is larger, then we move the right endpoint of the search interval to x . Otherwise, we move the left endpoint to x . The search terminates at a crossover point. Since the initial search domain has size $O(m)$, this procedure takes $O(\log m)$ time.

The output of UPPER is a function H_k^j , represented as a list of labeled intervals. In line 10, we need to take the pointwise max of $H_k^0, \dots, H_k^s$. To do this, we convert H_k^j to an array representation $[H_k^j(0), \dots, H_k^j(m)]$, by iterating through the intervals and evaluating h_r^j functions corresponding to the label of the intervals.

4.4.3 Algorithm analysis

We now show the correctness of our algorithm and analyze its complexity. Following the proofs in [45, 44], procedure UPPER correctly computes the upper envelope of set S in line 6. Next, we prove the output H_k^j of UPPER equals the function F_k^j , for all j and k .

Lemma 4.1. *For any $j \in [0, s]$, $k \in [0, n]$ and $x \in [0, m]$, we have $H_k^j(x) = F_k^j(x)$.*

Proof. Fix an $x \in [0, m]$. Since H_k^j is the upper envelope of functions $h_0^j, \dots, h_m^j$, we have

$$\begin{aligned} H_k^j(x) &= \max\{h_0^j(x), \dots, h_m^j(x)\} \\ &= \max\{h_{x-b_k^{j+1}-1}^j(x), \dots, h_{x-b_k^j}^j(x)\} \end{aligned}$$

The second equality holds because by definition $h_r^j(x)$ is only defined if $x \in [r + b_k^j, r + b_k^{j+1} - 1]$, and so $h_r^j(x)$ is undefined for $r < x - b_k^{j+1} + 1$ or $r > x - b_k^j$. Expanding each

term in the last expression using the definition of h_k^j , we have

$$H_k^j(x) = \max\{f_k(b_k^{j+1} - 1) + F_{k-1}(x - b_k^{j+1} + 1), \dots, f_k(b_k^j) + F_{k-1}(x - b_k^j)\}$$

The last expression is equal to Equation 4.3, and so $H_k^j(x) = F_k^j(x)$. $\square$

The following theorem states that ALLOC works correctly.

Theorem 4.1. *Algorithm 4.1 outputs the maximum total throughput $F_n(m)$.*

Proof. Since $H_k^j = F_k^j$ for all j and k by Lemma 4.1, then $H_k = \max\{H_k^0, \dots, H_k^s\} = \max\{F_k^0, \dots, F_k^s\} = F_k$, where the first equality follows by line 10 of ALLOC. Thus, the output $H_n(m)$ equals $F_n(m)$. $\square$

We now analyze the complexity of the algorithm. Fix a $k \in [0, n]$. The next lemma follows from the fact that each of f_k 's segments is either concave or convex.

Lemma 4.2. *Let $j \in [0, s]$ and $r_1, r_2 \in [0, m]$. Then the functions $h_{r_1}^j$ and $h_{r_2}^j$ cross at most once.*

Proof. Assume WLOG that $r_1 < r_2$. Let $t(x) = h_{r_1}^j(x) - h_{r_2}^j(x)$, where x is in the domain of both $h_{r_1}^j$ and $h_{r_2}^j$. Thus, since $h_{r_1}^j(x) = F_{k-1}(r_1) + f_k(x - r_1)$ and $h_{r_2}^j(x) = F_{k-1}(r_2) + f_k(x - r_2)$ by Equation 4.4, we have $t(x) = F_{k-1}(r_1) - F_{k-1}(r_2) + f_k(x - r_1) - f_k(x - r_2)$. To prove the lemma, we will prove the monotonic property of function $t(x)$. Since x is in the domain of both $h_{r_1}^j$ and $h_{r_2}^j$, we have $x - r_1, x - r_2 \in [b_{k-1}, b_k - 1]$ and then we have $(x - r_1, f_k(x - r_1)), (x - r_2, f_k(x - r_2))$ are both on the k th concave or convex segment of f_k . There are two cases. If the k th segment of f_k is concave, we have $f_k(x - r_1) - f_k(x - r_2)$ is non-increasing with x . Thus, $t(x)$ is non-increasing with x . Therefore, $h_{r_1}^j$ and $h_{r_2}^j$ cross at most once. Otherwise, the k th segment on f_k is convex, we have $f_k(x - r_1) - f_k(x - r_2)$ is non-decreasing with x . Thus, $t(x)$ is non-decreasing with x . Therefore, $h_{r_1}^j$ and $h_{r_2}^j$ have cross at most once. Thus, the lemma follows. $\square$

We show the time complexity of Procedure UPPER in the following lemma. The proof of the lemma largely follows [45]. The extra $\log m$ factor come from the time to compute the crossing point of two h_r^j functions by binary search, as described earlier.

Lemma 4.3. *Procedure UPPER runs in $O(\alpha(t)t \log t \log m)$ time when the input consists of t functions.*

Proof. Hart et al. [45] proposed an algorithm to compute the upper envelope of t' continuous segments in $O(\alpha(t')t' \log t')$ time. A key assumption in Hart's algorithm is that any two segments intersects at most one point. Thus, since any two segments in Procedure UPPER cross at most once by Lemma 4.2, we can use Hart's algorithm to compute the upper envelope of the t functions in Procedure UPPER. Thus, since we need extra $\log m$ time to compute the crossing point of two h_r^j functions by binary search, the lemma follows. $\square$

Finally, we state the complexity of the entire algorithm.

Theorem 4.2. *Algorithm 4.1 runs in $O(sn\alpha(m)m(\log m)^2)$ time.*

Proof. For each $k \in [2, n]$ and $j \in [1, s]$, we call UPPER with $m + 1$ functions $h_0^j, \dots, h_m^j$, which takes $O(\alpha(m)m(\log m)^2)$ time by Lemma 4.3. To merge $H_k^0, \dots, H_k^s$ we first convert each of them to arrays in $O(m)$ time, then take their maximum in $O(sm)$ time. Thus, the total complexity is $O(sn\alpha(m)m(\log m)^2)$. $\square$

4.5 Approximation algorithm

ALLOC finds an optimal allocation much faster than the existing methods for large m . However, for practical purposes we may want an even faster algorithm, while settling for an approximately optimal solution. In this section, we present a fully polynomial time approximation scheme (FPTAS) that finds an allocation with throughput at least $1 - \epsilon$ times the optimal in $O(\frac{s}{\epsilon}\alpha(\frac{n}{\epsilon})n^2 \log \frac{n}{\epsilon} \log m)$ time, for any $\epsilon > 0$. The existing dynamic programming based FPTAS [46] takes $O(\frac{n^2 m}{\epsilon})$ time, and is much slower for large m .

Our FPTAS consists of two main parts, which we call *scaling* and *reducing*. Scaling divides each throughput function by a large constant. It is used in FPTAS's for a number of resource allocation and knapsack related problems [46], and typically requires finding a good estimate for the maximum throughput of the unscaled problem. The estimation is often done using a greedy method with running time linear in m . However, we want our

FPTAS to run in time proportional to $\log m$, and therefore we require a much faster estimation algorithm. One novelty of our FPTAS is an algorithm to approximate the maximum throughput in $O(n^2 \log n)$ time, independent of m . After scaling, we form a set of functions and compute their upper envelopes, similar to in ALLOC. However, whereas previously each upper envelope involved $m + 1$ functions, we show that we can reduce the number of functions to $O(\frac{n}{\epsilon})$ in our FPTAS. This allows us to replace the m terms in Theorem 4.2 by $O(\frac{n}{\epsilon})$ (except for one of the $\log m$ terms) and obtain the running time stated above. We now describe the algorithm APPROXALLOC in detail. Pseudocode for the algorithm is shown in Algorithm 4.2.

Algorithm 4.2 Algorithm 4.2 for finding a $1 - \epsilon$ approximate allocation

```

1: procedure APPROXALLOC( $f_1, \dots, f_n, \epsilon$ )
2:   compute  $D$  as in Section 4.5.2
3:    $g_i \leftarrow \lfloor \frac{f_i}{D} \rfloor$  for  $i = 1, \dots, n$ 
4:    $H_1 \leftarrow g_1$ 
5:   for  $k \leftarrow 2$  to  $n$  do
6:     for  $j \leftarrow 1$  to  $s$  do
7:       compute  $C_k$  and  $\hat{h}_r^j$  as in Section 4.5.3
8:        $S \leftarrow \{\hat{h}_r^j \mid r \in C_k\}$ 
9:        $H_k^j \leftarrow \text{UPPER}(S)$ 
10:    end for
11:     $\overline{H}_k^0 \leftarrow \overline{H}_{k-1}$ 
12:     $\overline{H}_k^j \leftarrow \text{MINRES2}(H_k^j)$  for  $j = 1, \dots, s$ 
13:     $\overline{H}_k \leftarrow \text{MINRES}(\overline{H}_k^0, \dots, \overline{H}_k^s)$ 
14:  end for
15:  return  $\max\{v \mid \overline{H}_n(v) \neq \perp\}$ 
16: end procedure

17: procedure MINRES
18:   Input: Arrays  $\overline{H}_k^0, \dots, \overline{H}_k^s$ 
19:   Output: Pointwise min of the input arrays
20:    $\overline{H}_k \leftarrow \min\{\overline{H}_k^0, \dots, \overline{H}_k^s\}$ 
21: end procedure

```

4.5.1 Scaling the problem

We first replace each function f_i by $g_i = \lfloor \frac{f_i}{D} \rfloor$, for a constant D defined later. Let G^* be the maximum throughput of an m -feasible allocation when each process p_i 's throughput function is g_i . We call the problem with the g_i functions the *scaled* throughput problem,

```

22: procedure MINRES2
23:   Input: List of labeled intervals  $H_k^j$ 
24:   Output: “Dual” of  $H_k^j$ 
25:   for  $v \leftarrow 0$  to  $\frac{n}{\epsilon}$  do
26:     find  $([a, b], r) \in H_k^j$  s.t.  $\hat{h}_r^j(a) \leq v \leq \hat{h}_r^j(b)$ 
27:     if no such  $([a, b], r)$  then
28:        $\overline{H}_k^j(v) \leftarrow \perp$ 
29:     else
30:       find  $x = \min\{t \mid \hat{h}_r^j(t) \geq v, t \in [a, b]\}$ 
31:        $\overline{H}_k^j(v) \leftarrow x$ 
32:     end if
33:   end for
34:   return  $\overline{H}_k^j$ 
35: end procedure

```

and call G^* the optimal *scaled* throughput. We will show in Lemma 4.4 that G^*D is a $1 - \epsilon$ approximation to the optimal unscaled throughput F^* .

To make the FPTAS fast, we want to make D large enough so that $G^* = O(\frac{n}{\epsilon})$. Since the maximum total throughput with the f_i functions is F^* , we have $G^* = O(\frac{F^*}{D})$ and $D = O(\frac{\epsilon F^*}{n})$. Thus, to compute D we need to estimate F^* to within a constant factor, and we also want to compute the estimate in $o(m)$ time. This algorithm is described next.

4.5.2 Estimating F^*

The estimation algorithm uses the *multiple choice knapsack problem (MCKP)* [68]. In MCKP, there is a set of knapsacks, and the goal is to select at most one item from each knapsack³ to maximize the total value of the selected items subject to a weight restriction.

We form an MCKP problem with n knapsacks, where each knapsack has $n + 1$ items. For $i \in [1, n]$, let $r_i = \frac{b'_i - b_i}{n}$. Assume for simplicity r_i is an integer: it is straightforward to adapt the algorithm to noninteger r_i 's. The j 'th item in the i 'th knapsack has weight $b_i + (j - 1)r_i$ and value $f_i(b_i + (j - 1)r_i)$. The total weight restriction is m . Let $\hat{F}^*$ be the maximum value of the MCKP problem. Then we claim there exists a corresponding m -feasible allocation with total throughput $\hat{F}^*$. Indeed, if in the i 'th knapsack the MCKP solution selected the j 'th item, then we set $x_i = b_i + (j - 1)r_i$, and if MCKP did not select

³MCKP usually requires selecting exactly one item per knapsack, but it is easy to modify MCKP algorithms to allow selecting no items.

any item we set $x_i = 0$. In this way, the total throughput equals the total MCKP value $\hat{F}^*$, and the total resource use is within the weight restriction m .

We will show in Lemma 4.6 that $\frac{\hat{F}^*}{F^*} \geq 0.128$, so that $\hat{F}^*$ is a constant factor approximation for F^* . However, it takes too much time to find $\hat{F}^*$ directly. Instead, we use Gens and Levner's MCKP approximation algorithm [68] to find a value $\tilde{F}$ with $\hat{F}^* \geq \tilde{F} \geq \frac{4}{5}\hat{F}^*$ in $O(n^2 \log n)$ time. Then, since $\frac{\hat{F}^*}{F^*} \geq 0.128$, we have $\tilde{F} \geq 0.1F^*$. We then set $D = \frac{10\epsilon\tilde{F}}{n}$, so that we have $G^* \leq \frac{F^*}{D} = \frac{nF^*}{10\epsilon\tilde{F}} \leq \frac{n}{\epsilon}$, as desired. It remains to state and prove Lemma 4.6. However, the statement and proof of the lemma are somewhat involved, so for continuity of presentation we defer them till Section 4.5.5.

4.5.3 Reducing the problem

For any $k \in [1, n]$, $j \in [0, s]$, and $r \in [0, m]$, define functions G_k and G_k^j as in Equations 4.1 and 4.3, but with the g_i functions in place of the f_i functions. The meaning of these functions is similar to before. For example, $G_k(r)$ is the maximum r -feasible scaled throughput for $p_1, \dots, p_k$. Also, we (re)define

$$h_r^j(x) = g_k(x - r) + G_{k-1}(r), \text{ for } x \in [r + b_k^j, r + b_k^{j+1} - 1]$$

For any $v \in [0, \frac{n}{\epsilon}]$, let $c_k(v) = \min\{r \mid G_{k-1}(r) \geq v\}$ be the minimum amount of resource $r \leq m$ such that there exists an r -feasible allocation for $p_1, \dots, p_k$ with scaled throughput $\geq v$; if there is no such $r \leq m$, $c_k(v)$ is undefined. Since G_{k-1} is a nondecreasing function, we can find $c_k(v)$ using binary search once we know function G_{k-1} . Let $C_k = \{c_k(0), \dots, c_k(\frac{n}{\epsilon})\}$. Given $r \in C_k$, let $d(r) = r' - r$, where r' is the minimum value in C_k larger than r , or $d(r) = 0$ if r is the largest element in C_k .

Consider any $j \in [0, s]$, $k \in [1, n]$ and $r \in C_k$. We now define a function $\hat{h}_r^j$ that is similar to h_r^j , except its domain extends $d(r)$ further to the right, where it has constant value $g_k(b_k^{j+1} - 1) + G_{k-1}(r)$. Let $a = r - b_k^j$ and $b = r + b_k^{j+1} - 1$.

$$\hat{h}_r^j(x) = \begin{cases} g_k(x - r) + G_{k-1}(r) & \text{if } x \in [a, b] \\ g_k(b - r) + G_{k-1}(r) & \text{if } x \in [b, b + d(r)] \end{cases}$$

From the analysis in Section 4.4.3, we know that if we call UPPER (as defined in Algorithm 1) with the m functions $h_0^j, \dots, h_m^j$, the output is G_k^j , as it should be. The main observation

in the reduction step is that if we call UPPER with the $\frac{n}{\epsilon} + 1$ functions $\{\hat{h}_r^j \mid r \in C_k\}$, then the upper envelope is also G_k^j . We prove this in Lemma 4.4. Basically, this is because any h_r^j for $r \notin C_k$ is dominated by some $\hat{h}_{r'}^j$ for an $r' \in C_k$, meaning that the former function is less than or equal to the latter at all points.

We now describe the main procedure APPROXALLOC, as shown in Algorithm 5.2. It loops over the processes and throughput function segments. On line 11, we call UPPER on the functions $\{\hat{h}_r^j \mid r \in C_k\}$ and get back a result H_k^j . We want to form the max of $H_k^0, \dots, H_k^s$. In ALLOC, this was done by converting each H_k^j to an array then maximizing over the arrays, which took $O(sm)$ time. But we want the running time of APPROXALLOC to be proportional to $O(\log m)$. Procedure MINRES2 gives a more efficient way to combine the H_k^j functions, as follows.

Recall that each upper envelope H_k^j is represented as a list of labeled intervals, where the intervals are contiguous and ordered left to right, and where each label r corresponds to a function $\hat{h}_r^j$ that is on the upper envelope in that interval. Since we called UPPER with $|C_k| = O(\frac{n}{\epsilon})$ functions in line 9, H_k^j contains $O(\frac{n}{\epsilon})$ intervals. We can think of H_k^j as a function that maps an amount of resources to the maximum throughput for that much resource. In MINRES2, we form the “dual” of H_k^j , denoted $\overline{H}_k^j$, which maps an amount of throughput to the minimum amount of resource needed to achieve the throughput. More precisely, $\overline{H}_k^j(v) = r \leq m$, where r is the minimum amount of resource such that there exists an r -feasible allocation for $p_1, \dots, p_k$ with scaled throughput $\geq v$, and such that p_k 's allocation is within its j 'th interval $[b_k^j, b_k^{j+1} - 1]$; if no such $r \leq m$ exists, then $\overline{H}_k^j = \perp$.

To compute $\overline{H}_k^j(v)$, MINRES2 first searches the list H_k^j for a labeled interval $([a, b], r)$ with $\hat{h}_r^j(a) \leq v \leq \hat{h}_r^j(b)$. Observe that $H_k^j(x)$ is a nondecreasing function in x . So for any two consecutive labeled intervals $([a_1, b_1], r_1)$ and $([a_2, b_2], r_2)$ in H_k^j , we have $\hat{h}_{r_1}^j(a_1) \leq \hat{h}_{r_1}^j(b_1) \leq \hat{h}_{r_2}^j(a_2) \leq \hat{h}_{r_2}^j(b_2)$. Therefore, we can binary search through the intervals in H_k^j to find $([a, b], r)$. If no such $([a, b], r)$ exists, we set $\overline{H}_k^j = \perp$. Otherwise, after finding $([a, b], r)$, we do another binary search within $[a, b]$ to find $x = \min\{t \mid (\hat{h}_r^j(t) \geq v) \wedge (t \in [a, b])\}$. Finally, we set $\overline{H}_k^j(v) = x$.

Procedure APPROXALLOC calls MINRES2 with H_k^j and gets back $\overline{H}_k^j$, for all $j \in [1, s]$. After this, MINRES combines $\overline{H}_k^0, \dots, \overline{H}_k^s$ into $\overline{H}_k$ by taking the pointwise minimum of

$\overline{H}_k^0, \dots, \overline{H}_k^s$, which are all arrays of size $O(\frac{n}{\epsilon})$. $\overline{H}_k(v)$ equals r if $r \leq m$ is the minimum resources needed for $p_1, \dots, p_k$ to achieve $\geq v$ scaled throughput, or equals $\perp$ if no such r exists. MINRES returns $\overline{H}_k$ to the main function APPROXALLOC. After APPROXALLOC computes $\overline{H}_1, \dots, \overline{H}_n$, it finds the maximum v such that $\overline{H}(v) \neq \perp$. v is the maximum total scaled throughput for processes $p_1, \dots, p_n$ in any m -feasible allocation, which is the value G^* we seek.

4.5.4 Algorithm analysis

We now show APPROXALLOC computes a $1 - \epsilon$ approximation to the maximum throughput F^* in $O(\frac{\epsilon}{\alpha} \alpha(\frac{n}{\epsilon}) n^2 \log \frac{n}{\epsilon} \log m)$ time. We first show that the algorithm computes the maximum scaled throughput G^* , then show $G^* D$ is a $1 - \epsilon$ approximation to F^* . The first lemma shows we can compute G_k^j using the reduced set of $\hat{h}_r^j$ functions.

Lemma 4.4. *Use the definitions from Section 4.5.3. Then $G_k^j = \text{UPPER}(\{h_0^j, \dots, h_m^j\}) = \text{UPPER}(\{\hat{h}_r^j \mid r \in C_k\})$.*

Proof. The first equality follows from Lemma 4.1. To prove the second equality, we will show that for any $r' \notin C_k$, there exists $r \in C_k$ such that $h_{r'}^j(x) \leq \hat{h}_r^j(x)$ for all x in $h_{r'}^j(x)$'s domain. Thus, $h_{r'}^j$ is not on the upper envelope, and so the second equality follows. Let $r \in C_k$, and consider any $r' > r$ such that $G_{k-1}(r) = G_{k-1}(r')$. Then $r' \notin C_k$, and since G_{k-1} is nondecreasing, $r' - r < d(r)$. Let x be in the domain $[r' - b_k^j, r' + b_k^{j-1} - 1]$ of $h_{r'}^j$, and suppose first $x \in [r + b_k^j, r + b_k^{j+1} - 1]$. Then $\hat{h}_r^j(x) - h_{r'}^j(x) = g_k(x - r) - g_k(x - r') \geq 0$, since $r < r'$ and g_k is nondecreasing. Otherwise, we have $x \in [r + b_k^{j+1}, r' + b_k^{j-1} - 1]$, so $\hat{h}_r^j(x) - h_{r'}^j(x) = g_k(b_k^{j+1} - 1) - g_k(x - r') \geq 0$. So, in both cases we have $h_{r'}^j(x) \leq \hat{h}_r^j(x)$. $\square$

Procedures MINRES and MINRES2 are correct by inspection. Combined with Lemma 4.4, we get the following theorem. The proof is similar to the proof of Theorem 4.1.

Theorem 4.3. *APPROXALLOC computes the maximum m -feasible total scaled throughput $G_n(m)$.*

We now show that the optimal scaled throughput computed by APPROXALLOC approximates the optimal unscaled throughput. Note that the 10ϵ term in Theorem 4.4 can be reduced to ϵ by simply calling APPROXALLOC with $\epsilon' = \frac{\epsilon}{10}$.

Theorem 4.4. *Let $(x_1, \dots, x_n)$ be an optimal allocation for the scaled problem, and let $F = \sum_{i=1}^n f_i(x_i)$. Then $\frac{F}{F^*} \geq 1 - 10\epsilon$.*

Proof. Let $(x_1^*, \dots, x_n^*)$ be an optimal allocation for the unscaled problem with throughput F^* . We have

$$F = \sum_{i=1}^n f_i(x_i) \geq D \sum_{i=1}^n g_i(x_i) \geq D \sum_{i=1}^n g_i(x_i^*) \quad (4.5)$$

The first inequality holds because $g_i = \lfloor \frac{f_i}{D} \rfloor$, and the second inequality holds because $(x_1, \dots, x_n)$ is optimal for the scaled problem. Also, since $Dg_i(x_i^*) + D \geq f_i(x_i^*)$ for every i , then $D \sum_{i=1}^n g_i(x_i^*) + nD \geq \sum_{i=1}^n f_i(x_i^*) = F^*$. Combining this with Inequality 4.5, we get $F + nD \geq F^*$, and $\frac{F}{F^*} \geq 1 - \frac{nD}{F^*}$. Now, $D = \frac{10\epsilon \tilde{F}}{n}$, and $\tilde{F} \leq \hat{F}^* \leq F^*$, so $D \leq \frac{10\epsilon F^*}{n}$. Plugging this into the expression $1 - \frac{nD}{F^*}$, the lemma follows. $\square$

Finally, we analyze the time complexity of APPROXALLOC. The following lemma is similar to Lemma 4.2, though note that the $\hat{h}_r^j$ functions are not necessarily concave or convex.

Lemma 4.5. *Let $k \in [1, n]$, $j \in [0, s]$ and $r_1, r_2 \in C_k^j$. Then the functions $\hat{h}_{r_1}^j$ and $\hat{h}_{r_2}^j$ cross at most once.*

The following theorem gives APPROXALLOC's complexity.

Theorem 4.5. *APPROXALLOC runs in $O(\frac{s}{\epsilon} \alpha(\frac{n}{\epsilon}) n^2 \log \frac{n}{\epsilon} \log m)$ time.*

Proof. Assume that $\frac{n}{\epsilon} < m$, as otherwise the exact algorithm ALLOC runs faster than APPROXALLOC. The main loops in lines 7 and 8 run a total of ns times. In line 7, we have $|C_k| = O(\frac{n}{\epsilon})$. Computing each value in C_k requires searching H_{k-1} , which is actually represented in its dual form $\overline{H}_{k-1}$ as an array of size $O(\frac{n}{\epsilon})$. Thus, computing C_k takes $O(\frac{n}{\epsilon} \log \frac{n}{\epsilon})$ time total. Since each pair of $\hat{h}_r^j$ functions cross at most once by Lemma 4.5, then by Lemma 4.3, each call to UPPER in line 11 takes $O(\alpha(\frac{n}{\epsilon}) \frac{n}{\epsilon} \log \frac{n}{\epsilon} \log m)$ time. Line

12 takes $O(\frac{sn}{\epsilon} \log m)$ time. Indeed, we call MINRES2 s times, and each call to MINRES2 loops $O(\frac{n}{\epsilon})$ times. In each loop, line 26 can be done in $O(\log \frac{n}{\epsilon}) = O(\log m)$ time using binary search, since $|G_k^j| = O(\frac{n}{\epsilon})$. Also, line 30 can be done using binary search in $O(\log m)$ time. Adding everything together, we get the required bound. $\square$

4.5.5 Lemma for estimating F^*

Finally, we give the algorithm to estimate the maximum total throughput F^* . When searching for the optimal allocation, each variable x_i can potentially take on $O(m)$ values in its allocation range $[b_i, b'_i]$. We show that we can consider a smaller set of $n+1$ possible values for each x_i and still find an approximately optimal solution. For each i , let $r_i = \frac{b'_i - b_i}{n}$ and $B_i = \{b_i, b_i + r_i, b_i + 2r_i, \dots, b_i + \frac{n-1}{n}r_i, b'_i\}$.

Lemma 4.6. *Let $\hat{F}^*$ be the maximum total throughput for an m -feasible allocation when $x_i \in B_i$ for all i . Then $\frac{\hat{F}^*}{F^*} > 0.128$.*

Proof. We first introduce some notation. Consider a value $x \in [b_i, b'_i]$ of x_i . We define $\lfloor x \rfloor_i$ to be the maximum value in B_i that is $\leq x$. We call this the *floor* of x with respect to x_i , and say we *round down* x (w.r.t x_i); we omit saying “with respect to x_i ” when x_i is clear from the context. Also, we define $\lceil x \rceil_i$ to be the minimum value in B_i that is $\geq x$; we call this the *ceiling* of x (w.r.t. x_i), and say we *round up* x (w.r.t. x_i). If $x = 0$, let $\lfloor x \rfloor_i = \lceil x \rceil_i = 0$. If $x \in B_i \cup \{0\}$, we say x is a *whole* value; otherwise we say it is a *partial* value. For example, if $[b_i, b'_i] = [0, n]$, then $\lfloor \cdot \rfloor_i$ and $\lceil \cdot \rceil_i$ are the same as the regular floor and ceiling functions, and whole and partial values correspond to integers and nonintegers between 0 and n , resp.

Consider the optimal allocation $x^* = (x_1^*, \dots, x_n^*)$. We partition the coordinates based on whether they are whole or partial values. Assume (by renumbering the indices if necessary) that $x_1^*, \dots, x_k^*$ are partial values and $x_{k+1}^*, \dots, x_n^*$ are whole; k may be 0, in which case all coordinates are whole. Also assume WLOG that $x_1^* \geq \dots \geq x_k^*$. Let $X = \sum_{i=1}^k x_i^*$ and $X' = \sum_{i=k+1}^n x_i^*$ be the total value of the partial and whole coordinates, resp. Also, let $F^- = \sum_{i=1}^k f_i(\lfloor x_i^* \rfloor_i)$ and $F^+ = \sum_{i=1}^k f_i(\lceil x_i^* \rceil_i)$ be the throughput from the partial coordinates when we round them down and up, resp. Let $F' = \sum_{i=k+1}^n f_i(x_i^*)$ be the

throughput from the whole coordinates. Let $\beta > 0$ be a parameter that we fix later. Finally, let $\Delta = F^+ - F^-$. Basically, if Δ is small, we can round down all the coordinates x^* and get a good approximation. If Δ is large, we can adjust x^* to a whole value solution whose value is a fraction of Δ , again giving a good approximation.

We first show that if $F' > \Delta$, we can find a whole solution that is a $\frac{1}{3}$ approximation. Since $\sum_{i=1}^n x_i^* = m$, then $x = (\lfloor x_1^* \rfloor, \dots, \lfloor x_n^* \rfloor)$ is an m -feasible whole solution and has value $F^- + F'$. Also, we have $F^+ \geq \sum_{i=1}^k f_i(x_i^*)$, since the f_i 's are nondecreasing, and so $F^+ + F' \geq F^*$. Lastly, $F^+ = F^- + \Delta$. Thus, solution x has approximation ratio $\alpha \geq \frac{F' + F^-}{F' + F^+} = \frac{F' + F^-}{F' + F^- + \Delta}$. If $F' > F^-$, then $\alpha \geq \frac{F'}{3F^-}$. Otherwise, $F' \leq F^-$, and so $\alpha \geq \frac{F^-}{3F^-}$. Thus in both cases $\alpha \geq \frac{1}{3}$. In the rest of the proof, it suffices to assume $F' \leq \Delta$.

Consider two cases, $\Delta \leq \beta F^-$ and $\Delta > \beta F^-$. In the first case, since $F^+ = F^- + \Delta$, then $F^+ \leq (\beta + 1)F^-$ and $(\beta + 1)(F^- + F') \geq F^+ + F' \geq F^*$. Since the throughput from solution $(\lfloor x_1^* \rfloor, \dots, \lfloor x_n^* \rfloor)$ is $F^- + F'$, this is a $\frac{1}{\beta+1}$ approximation for F^* .

If $\Delta > \beta F^-$, then consider two cases, either $x_1^* \geq \sum_{i=2}^k x_i^*$, or $x_1^* < \sum_{i=2}^k x_i^*$. We will decrease certain partial coordinates in order to increase other partial coordinates until they become whole. We decrease the coordinates for which a smaller allocation does not cause much less throughput, and increase the coordinates for which a larger allocation leads to much more throughput.

If $x_1^* \geq \sum_{i=2}^k x_i^*$, let $\Delta_1 = f_1(\lceil x_1^* \rceil) - f_1(\lfloor x_1^* \rfloor)$ and $\Delta_2 = \sum_{i=2}^k f_i(\lceil x_i^* \rceil) - f_i(\lfloor x_i^* \rfloor)$, so that $\Delta = \Delta_1 + \Delta_2$. Consider two subcases, either $\Delta_1 \geq \Delta_2$, or $\Delta_1 < \Delta_2$. In the first subcase, consider the whole solution $(\lceil x_1^* \rceil, 0, \dots, 0)$. This is feasible because $\lceil x_1^* \rceil \leq m$, since $\lceil x_1^* \rceil \leq b'_1 \leq m$. The solution has value $F = f_1(\lceil x_1^* \rceil) \geq f_1(\lceil x_1^* \rceil) - f_1(\lfloor x_1^* \rfloor) = \Delta_1 \geq \frac{\Delta}{2}$. As before, we have $F^* \leq F^+ + F'$ and $F^+ = F^- + \Delta$. Thus, since $\Delta > \beta F^-$ and since we assumed $F' \leq \Delta$, we have $F^* \leq (\frac{1}{\beta} + 1)\Delta + F' \leq (\frac{1}{\beta} + 2)\Delta$. So $\frac{F}{F^*} \geq \frac{\frac{\Delta}{2}}{(\frac{1}{\beta} + 2)\Delta} = \frac{\beta}{2(\beta + 2)}$.

In the second subcase $\Delta_1 < \Delta_2$. We can assume $\sum_{i=1}^n x_i^* = m$, since $\sum_{i=1}^n b'_i \geq m$. Since $x_1^* \geq \sum_{i=2}^k x_i^*$ and $\sum_{i=1}^k x_i^* = m$, then $x_1^* + X' \geq \frac{m}{2}$. We redistribute $x_1^* + X'$ among $x_2^*, \dots, x_k^*$ to round up some of these values to their ceilings. We see we can round up at least $\frac{n}{2}$ values, since rounding up any value x_i^* requires adding at most $r_i = \frac{b'_i - b_i}{n} \leq \frac{m}{n}$ to it. In addition, we can choose which values we round up. We choose as follows: let the *gain*

of a value x_i^* be $f_i(\lceil x_i^* \rceil)_i - f_i(\lfloor x_i^* \rfloor)_i$. Order $x_2^*, \dots, x_k^*$ in order of nonincreasing gain, and round up the first $\frac{n}{2}$ values in this order. Since there are at most $n - 1$ values in $x_2^*, \dots, x_k^*$ and the total gain from all the values is $\Delta_2 = \sum_{i=2}^k f_i(\lceil x_i^* \rceil)_i - f_i(\lfloor x_i^* \rfloor)_i$, then the gain from the first $\frac{n}{2}$ values is $\geq \frac{\Delta_2}{2} \geq \frac{\Delta}{4}$. Thus, the whole solution in which we round up the first $\frac{n}{2}$ values in the gain ordering and set all the other values to 0 has throughput $F \geq \frac{\Delta}{4}$. As before, we have $F^* \leq (\frac{1}{\beta} + 1)\Delta + F' \leq (\frac{1}{\beta} + 2)\Delta$. Thus solution F has approximation ratio $\geq \frac{\Delta/4}{(\frac{1}{\beta} + 2)\Delta} \geq \frac{\beta}{4(\beta + 2)}$.

The final case is $x_1^* < \sum_{i=2}^k x_i^*$. To get the approximation of F in this case, we first show the following claim. The claim follows from sorting the x_i^* 's and considering the prefix sums until one is $\geq \frac{X}{3}$.

Claim 4.1. *We can partition $x_1^*, \dots, x_k^*$ into two sets S_1 and S_2 such that $\min(\sum_{x \in S_1} x, \sum_{x \in S_2} x) \geq \frac{1}{3}X$.*

Let Δ_1 and Δ_2 be the total gains from all the values in S_1 and S_2 , resp. If $\Delta_1 > \Delta_2$, then we round up some values in S_1 while setting all values in S_2 and the whole coordinates to 0; we do the opposite if $\Delta_1 \leq \Delta_2$. More precisely, suppose first $\Delta_1 > \Delta_2$. Using the claim and the fact that $X + X' = m$, we have that $\sum_{x \in S_2} x + X' \geq \frac{m}{3}$. So arguing as before, we can round up $\frac{n}{3}$ values S_1 if we set the values in S_2 and the whole coordinates to 0. As before, we round up the $\frac{n}{3}$ values with the greatest gains. Since the total gain from S_1 is $\Delta_1 \geq \frac{\Delta}{2}$ and there are $\leq n - 1$ values in S_1 , the total gain from the S_1 values we round up is $\geq \frac{\Delta}{2} \cdot \frac{1}{3} = \frac{\Delta}{6}$, and so we can obtain a solution with value $\geq \frac{\Delta}{6}$. Using similar analysis as before, the approximation ratio of this solution is $\geq \frac{\Delta/6}{(\frac{1}{\beta} + 2)\Delta} \geq \frac{\beta}{6(\beta + 2)}$. The other case where $\Delta_1 \leq \Delta_2$ is symmetric. We set all the values in S_1 and the whole coordinates to 0, and round up the $\frac{n}{3}$ values in S_2 with the highest gain, to get a solution with value $\geq \frac{\Delta}{6}$ and approximation ratio $\geq \frac{\beta}{6(\beta + 2)}$.

To finish the proof, we have that in the case where $\Delta \leq \beta F^*$, we can find an integer solution with approximation ratio $\geq \frac{1}{\beta + 1}$, and in all the other cases the approximation ratio is $\geq \frac{\beta}{6(\beta + 2)}$. Setting these two expressions to be equal and solving for β , we get $\beta = \frac{5 + \sqrt{73}}{2}$, and so the approximation ratio is > 0.128 . $\square$

4.6 Experimental Evaluation

We performed experiments to evaluate the speed of our optimal allocation algorithm ALLOC and the quality of its solutions. We compared the running times of ALLOC and the optimal dynamic programming algorithm from [43]. We also compared the maximum throughput found by ALLOC with the throughput of the greedy algorithm [63, 33], which is frequently used in practice. As stated earlier, this algorithm iteratively assigns each unit of resource to the process whose throughput improves most with that additional unit. All our algorithms are implemented in C on a regular desktop PC with 4 cores and 4GB memory. As we currently do not implement our approximation algorithm APPROXALLOC, we do not report the results for the time being. We will continue to implement APPROXALLOC and compare the speed of APPROXALLOC with the existing approximation algorithm in [46].

4.6.1 Performance comparison with dynamic programming

To compare the speed of ALLOC and dynamic programming, we varied the amount of resource m and the number of segments s in the throughput functions. We first varied m from 1K to 20K, while fixing the number of processes at $n = 4$ and the number of function segments at $s = 2$. We used the average timing results from 100 randomly generated instances. Each instance consists of n throughput functions with s random concave or convex segments. Figure 4.2 illustrates the trend in the running times. For $m \leq 3K$, ALLOC is slower than DP due to the overhead from the upper envelope calculations. But for $m > 3K$, ALLOC becomes progressively faster; it is 4.6 times faster for $m = 20K$. We also conducted some experiments for larger values of m ; these were constrained by the amount of time DP took to run. For $m = 100K, 200K, 500K$ and $1M$, ALLOC ran 22, 32, 105 and 215 times faster than DP, resp.

Next, we varied s from 2 to 12 while fixing $n = 4$ and $m = 5K$. For each s , we again took the average time from 100 random runs. Figure 4.3 shows that as s increases, our running time initially increases nearly linearly, but then plateaus after $s = 6$. This is because for large s , the domains of each throughput function's segments get smaller, and

Figure 4.2: Running time of ALLOC versus DP for varying m .

Figure 4.3: Running time ALLOC for varying s .

the segments become less likely to cross each other. As the running time of ALLOC depends on the number of crossings, it stops increasing for large s . The unevenness observed for large s is due to the relatively small number of random trials we ran.

4.6.2 Quality comparison with greedy

We compared the optimal throughput output by ALLOC with the throughput from the greedy algorithm. The results are relatively insensitive to variations in m and s , as long as these values are not too small, so we fixed $m = 5K$ and $s = 3$ in our experiments. We created random throughput functions as before, by creating concave or convex segments of random widths. Using $n = 4$, we found that our algorithm produces 33% better results than greedy. We also experimented with different ways of generating random throughput functions, and found that they have a significant effect on the quality comparison, especially as n changes. Some types of functions result in very large differences between ALLOC and greedy, while for others the two algorithms perform almost the same. As we currently do not have a uniform way of generating these functions, we do not report our results for the time being. We will continue to study the relationship between properties of the throughput

functions and the quality comparison as n varies.

4.7 Summary

In this chapter, we studied how to allocate a shared resource among a set of processes with nonconcave utility functions to maximize their total utility. A key observation is that though many realistic utility functions are not concave, they can be divided into of a small number of concave or convex segments. We present both optimal and FPTAS algorithm leveraging the observation. Our optimal algorithm runs in $O(s\alpha(m)m(\log m)^2)$ time, and our FPTAS runs in $O(\frac{s}{\epsilon}\alpha(\frac{n}{\epsilon})n^2 \log \frac{n}{\epsilon} \log m)$ time for any $\epsilon > 0$. Compared to existing algorithms, both of our algorithms are almost $O(m)$ times faster. Experiments show that our exact algorithm runs up to 215 times faster than the exact dynamic programming based algorithm, and gives solutions that are 33% better than the greedy heuristic.

Chapter 5

Utility Maximization Distributed Resource Allocation

5.1 Introduction

In this chapter, we study efficient ways to run a set of threads on multiple servers. Our problem consists of two steps. First, each thread is assigned to a server. Subsequently, the resources at each server are allocated among the threads assigned to it. Each thread obtains a certain utility based on the resources it obtains, and the goal is to maximize the total utility of all the threads. The problem is called distributed resource allocation problem in our thesis.

The two steps in distributed resource allocation problem correspond to the *thread assignment* and *resource allocation* problems, both of which have been studied extensively in the literature. However, to our knowledge, these problems have not been studied together in the unified context considered in this paper. Existing works on resource allocation [63, 83, 55, 54, 62] largely deal with dividing the resources on a single server among a given set of threads. It is not clear how to apply these algorithms when there are multiple servers, since there are many possible ways to initially assign the threads to servers, and certain assignments result in low overall performance regardless of how resources are subsequently allocated. Likewise, existing works on thread assignment [59, 60, 57, 58] often overlook the resource allocation aspect. Threads typically request a certain amount of resources¹, and once assigned they are allocated precisely the resources they request, without

¹The request may be a scalar when there is a single type of resource, or a vector for multiple resource types.

any adjustments based on the requests of other threads assigned to the same server. This can also lead to suboptimal performance. For example, consider a thread that obtains x^β utility when allocated x amount of resource, for some $\beta \in (0, 1)^2$, and suppose the thread requests z resources, for an arbitrary z . Then when there are n threads and one server with C resources, $\frac{C}{z}$ of the threads would be allocated z resources while the rest receive 0, leading to a total utility of $Cz^{\beta-1}$, which is constant in n . However, the optimal allocation gives $\frac{C}{n}$ resources to each thread and has total utility $C^\beta n^{1-\beta}$, which is arbitrarily better for large n .

The distributed resource allocation problem combines thread assignment and resource allocation. We model each thread using a utility function. We assume the utility functions are nondecreasing and concave. The goal is to simultaneously find an assignment and allocations for each server that maximize the total utility of all the threads. We first show that the problem is NP-hard, even when there are only two servers. In contrast, the problem is efficiently solvable when there is a single server. Next, we present an approximation algorithm that finds a solution with utility $\alpha = 2(\sqrt{2} - 1) > 0.828$ times the optimal. The algorithm relates the optimal solution to a single server problem to an approximately optimal solution to the multiple server problem. The algorithm runs in time $O(mn^2 + n(\log mC)^2)$, where n and m are the number of threads and servers, resp., and C is the amount of resource on each server. We then present a faster algorithm with $O(n(\log mC)^2)$ running time and the same approximation ratio. Finally, we conduct experiments to test the performance of our algorithm. We used several types of synthetic threads, and show that our algorithm obtains at least 99% of the maximum utility on average for all the thread types. We also compared our algorithm with several simple but practical assignment and allocation heuristics, and show that our algorithm performs on average 3.9 to 5.7 times better.

The rest of the chapter is organized as follows. In Section 5.2, we describe related works on thread assignment and resource allocation. Section 5.3 formally defines our model and the distributed resource allocation problem, followed by Section 5.4 which proves the

²For $\beta = \frac{1}{2}$, this is known as the “square root rule” [5, 53] and is used to model a thread’s throughput as a function of its cache usage.

problem is NP-hard. Section 5.5 presents an approximation algorithm and its analysis, and Section 5.6 proposes a faster algorithm. In Section 5.7, we describe our experimental results. Finally, Section 5.8 concludes the chapter.

5.2 Related Works

There is a large body of work on resource allocation for a single server. Fox *et al.* [63] considered concave utility functions and proposed a greedy algorithm finding an optimal allocation in $O(nC)$ time, where n is the number of threads and C is the amount of resource on the server. Galil [83] proposed an improved algorithm with $O(n(\log C)^2)$ running time. Resource allocation for nonconcave utility functions is weakly NP-complete. However, Lai and Fan [62] identified a structural property of real-world utility functions that leads to fast parameterized algorithms. In a multiprocessor setting, utility typically correspond to cache miss rates. Miss rate curves can be determined by running threads multiple times using different cache allocations. Qureshi *et al.* [55] proposed efficient hardware based methods to minimize the overhead of this process, and also designed algorithms to partition a shared cache between multiple threads using methods based on [63]. Zhao *et al.* [40] proposed software based methods to measure page misses and allocate memory to virtual machines in a multicore system. Chase *et al.* [61] used utility functions to dynamically determine resource allocations among users in hosting centers and maximize total profit.

There has also been much work on assigning threads to cores in multicore architectures. Becchi *et al.* [59] proposed a scheme to determine an optimized assignment of threads to heterogeneous cores. They characterized the behavior of each thread on a core by a single value, its IPC (instructions per clock). In contrast, we model threads using utility functions giving the performance for different resource allocations. Radojković *et al.* [60] proposed a statistical approach to assign threads on massively multithreaded processors, choosing the best assignment out of a large random sample. The results in [59] and [60] are based on simulations and provide no theoretical guarantees. A problem related to thread assignment is *application placement*, in which applications with different resource requirements need to be mapped to servers while satisfying certain quality of service guarantees. Urgaonkar

et al. [57] proposed offline and online approximation algorithms for application placement. The offline algorithm achieves a $\frac{1}{2}$ approximation ratio. However, these algorithms are not directly comparable to ours, as they consider multiple types of resources while we only consider one type.

Co-scheduling is a technique which divides a set of threads into subsets and executes each subset together on one chip of a multicore processor. It is frequently used to minimize cache interference between threads. Jiang et al. [64] proposed algorithms to find optimal co-schedules for pairs of threads, and also approximation algorithms for co-scheduling larger groups of threads. Tian *et al.* [65] also proposed exact and approximation algorithm for co-scheduling on chip multiprocessors with a shared last level cache. Zhuralev *et al.* [66] survey scheduling techniques for sharing resources in multicores. The works on co-scheduling require measuring the performance from running different groups of threads together. When co-scheduling large groups, the number of measurements required becomes prohibitive. In contrast, we model threads by utility functions, which can be determined by measuring the performance of individual threads instead of groups.

The distributed resource allocation problem can be seen as a multiple-choice multiple knapsack problem. Here, there are multiple knapsacks with different capacities, and also multiple classes of items, where each item has a weight and value. The goal is to select one item from each class to fit the knapsacks and maximize the total value. The multiple choice knapsack problem involves a single knapsack, and corresponds to resource allocation with a single server. Lawler [69] proposed a $1 - \epsilon$ approximate algorithm that runs in $O(\frac{nc}{\epsilon})$ time, while Gens and Levner [68] proposed a $\frac{1}{5}$ approximate algorithm running in $O(nc \log n)$ time. Here, n is the number of processes (threads) and c is the total resource size (which equals mC in the context of distributed resource allocation problem). The multiple knapsack problem involves only one class of items. Neebe *et al.* [70] proposed a branch-and-bound algorithm, and Chekuri *et al.* [71] proposed a PTAS. However, we are not aware of any works combining the multiple choice and multiple knapsack problems. Our work is a first step in this direction, for the case when the ratios of values to weights in each item class is concave.

5.3 Model

In this section we formally define our model and problem. We consider a set of m homogeneous servers $s_1, \dots, s_m$, where each server has $C' > 0$ amount of resources. The homogeneity assumption is reasonable in a number of settings. For example, multicore processors typically contain shared caches of the same size, and datacenters often have many identically configured servers for ease of management. Homogeneous servers have also been widely studied in the literature [57, 96]. We also have n threads $t_1, \dots, t_n$. We imagine the threads are performing long-running tasks, and so the set of threads is static and all threads are active. Note a thread can be active even when it gets zero amount of resource, since for example a thread can run slowly when it gets no cache resource. Let S and T denote the set of servers and threads, respectively. Each thread t_i is characterized by a *utility function* $f_i : [0, C] \rightarrow \mathbb{R}^{\geq 0}$, giving its performance as a function of the resources it receives. We assume that f_i is nonnegative, nondecreasing and concave. The concavity assumption models a diminishing returns property frequently observed in practice [55]. While it does not apply in all settings, concavity is a common assumption used in many utility models, especially on cache and memory performance [5, 53].

Our goal is to assign the threads to the servers in a way that respects the resource bounds and maximizes the total utility. While a solution to this problem involves both an assignment of threads and allocations of resources, for simplicity we use the term assignment to refer to both. Thus, an *assignment* is given by a vector $[(r_1, c_1), \dots, (r_n, c_n)]$, indicating that each thread t_i is allocated c_i amount of resource on server s_{r_i} . Let S_j be the set of threads assigned to server s_j . That is, $S_j = \{i \mid r_i = j\}$. Then for all $1 \leq j \leq m$, we require $\sum_{i \in S_j} c_i \leq C$, so that the threads assigned to s_j use at most C resources. We assume that every thread is assigned to some server, even it receives 0 resources on the server. The *total utility* from an assignment is $\sum_{j=1}^m \sum_{i \in S_j} f_i(c_i) = \sum_{i \in T} f_i(c_i)$. The AA (*assign and allocate*) problem is to find an assignment that maximizes the total utility.

5.4 Hardness of the Problem

In this section, we show that it is NP-hard to find an assignment maximizing the total utility, even when there are only two servers. Thus, it is unlikely there exists an efficient optimal algorithm for the distributed resource allocation problem. This motivates the approximation algorithms we present in Sections 5.5 and 5.6.

Theorem 5.1. *Finding an optimal distributed resource allocation is NP-hard, even when there are only two servers.*

Proof. We give a reduction from the NP-hard *partition problem* [72] to the distributed resource allocation problem with two servers. In the partition problem, we are given a set of numbers $S = \{c_1, \dots, c_n\}$ and need to determine if there exist a partition of S into sets S_1 and S_2 such that $\sum_{i \in S_1} c_i = \sum_{i \in S_2} c_i$. Given an instance of partition, we create a distributed resource allocation instance A with two servers each with $C = \frac{1}{2} \sum_{i=1}^n c_i$ amount of resources. There are n threads $t_1, \dots, t_n$, where the i 'th thread has utility function f_i defined by

$$f_i(x) = \begin{cases} x & \text{if } x \leq c_i \\ c_i & \text{otherwise} \end{cases}$$

The f_i functions are nondecreasing and concave. We claim the partition instance has a solution if and only if A 's maximum utility is $\sum_{i=1}^n c_i$. For the if direction, let $A^* = [(r_1^*, c_1^*), \dots, (r_n^*, c_n^*)]$ denote an optimal solution for A , and let S_1 and S_2 be the set of threads assigned to the servers 1 and 2, resp. We show that S_1, S_2 solve the partition problem. We first show that $c_i^* = c_i$ for all i . Indeed, if $c_i^* < c_i$ for some i , then $f_i(c_i^*) < c_i$, while $f_j(c_j^*) \leq c_j$, for all $j \neq i$. Thus, $\sum_{j=1}^n f_j(c_j^*) < \sum_{j=1}^n c_j$, which contradicts the assumption that A^* 's utility is $\sum_{j=1}^n c_j$. Next, suppose $c_i^* > c_i$ for some i . Then since A^* is a valid assignment, we have $\sum_{i \in S_1} c_i^* - \sum_{i \in S_2} c_i^* \leq C + C = \sum_{i=1}^n c_i$, and so there exists $j \neq i$ such that $c_j^* < c_j$. But then $f_j(c_j^*) < c_j$ and $f_k(c_k^*) \leq c_k$ for all $k \neq j$, so A^* 's total utility is $\sum_{k=1}^n f_k(c_k^*) < \sum_{i=1}^n c_i$, again a contradiction. Thus, we have $c_i^* = c_i$ for all i , and so $\sum_{i \in S_1} c_i^* + \sum_{i \in S_2} c_i^* = \sum_{i=1}^n c_i = 2C$. So, since $\sum_{i \in S_1} c_i^* \leq C$ and $\sum_{i \in S_2} c_i^* \leq C$, then $\sum_{i \in S_1} c_i^* = \sum_{i \in S_2} c_i^* = C$. Hence, $\sum_{i \in S_1} c_i = \sum_{i \in S_2} c_i = C$, and S_1, S_2 solve the partition problem.

For the only if direction, suppose S_1, S_2 are a solution to the partition instance. Then since $\sum_{i \in S_1} c_i = \sum_{i \in S_2} c_i = C$, we can assign the threads with indices in S_1 and S_2 to servers 1 and 2, resp., and get a valid assignment with utility $\sum_{i=1}^n f_i(c_i) = \sum_{i=1}^n c_i$. This is a maximum utility assignment for A , since $f_i(x) \leq c_i$ for all i . Thus, the partition problem reduces to the distributed resource allocation problem, and so the latter is NP-hard. $\square$

5.5 Approximation Algorithm

In this section, we present an algorithm to find an assignment with total utility at least $\alpha = 2(\sqrt{2} - 1) > 0.828$ times the optimal in $O(mn^2 + n(\log mC)^2)$ time. The algorithm consists of two main steps. The first transforms the utility functions, which are arbitrary nondecreasing concave functions and hence difficult to work with algorithmically, to piecewise linear ones. We show that threads with the original utility functions and the linearized ones have the same maximum total utility. Next, we find an α -approximate optimal thread assignment for the linearized functions. This leads to an α approximate solution for the original concave problem.

5.5.1 Linearization

To describe the linearization procedure, we start with the following definition.

Definition 5.1. *Given an distributed resource allocation problem A with m servers each with C amount of resources, and n threads with utility functions $f_1, \dots, f_n$, a super-optimal allocation for A is a set of values $\hat{c}_1, \dots, \hat{c}_n$ that maximizes the quantity $\sum_{i=1}^n f_i(\hat{c}_i)$, subject to $\sum_{i=1}^n \hat{c}_i \leq mC$. Call $\sum_{i=1}^n f_i(\hat{c}_i)$ the super-optimal utility for A .*

To motivate the above definition, note that for any valid assignment $[(r_1, c_1), \dots, (r_n, c_n)]$ for A , we have $\sum_{i=1}^n c_i \leq mC$, and so the utility of the assignment $\sum_{i=1}^n f_i(c_i)$ is at most the super-optimal utility of A . Let F^* and $\hat{F}$ denote A 's maximum utility and super-optimal utility, resp. Then we have the following.

Lemma 5.1. $F^* \leq \hat{F}$.

Also, since the f_i functions are nondecreasing, we can assume WLOG the following.

Lemma 5.2. $\sum_{i=1}^n \hat{c}_i = mC$.

Since the utility functions are concave, the super-optimal utility of A can be computed efficiently in $O(n(\log mC)^2)$ time using the algorithm in [83]. In the remainder of this section, fix A to be an distributed resource allocation problem consisting of m threads with C resource each and n threads with utility functions $f_1, \dots, f_n$. Let $[\hat{c}_1, \dots, \hat{c}_n]$ be a super-optimal allocation for A . We define the *linearized* version of A to be another distributed resource allocation problem B with the same set of servers and threads, but where the threads have utility functions $g_1, \dots, g_n$, where each g_i is defined by

$$g_i(x) = \begin{cases} f_i(\hat{c}_i) \frac{x}{\hat{c}_i} & \text{if } x < \hat{c}_i \\ f_i(\hat{c}_i) & \text{otherwise} \end{cases} \quad (5.1)$$

We state a lemma we will use later.

Lemma 5.3. For any $i \in T$ and $x \in [0, C]$, $f_i(x) \geq g_i(x)$.

Proof. Since f_i is concave, then for $x \in [0, \hat{c}_i]$, we have $f_i(x) \geq \frac{\hat{c}_i - x}{\hat{c}_i} f_i(0) + \frac{x}{\hat{c}_i} f_i(\hat{c}_i) \geq g_i(x)$, where the equality follows because $f_i(0) \geq 0$. Also, for $x > \hat{c}_i$, $f_i(x) \geq f_i(\hat{c}_i) = g_i(x)$. $\square$

5.5.2 Solving the linearized problem

We now describe an α approximation algorithm for the linearized problem. The pseudocode is given in Algorithm 5.1. The algorithm takes as input a super-optimal allocation $[\hat{c}_1, \dots, \hat{c}_n]$ for A and the resulting linearized utility functions $g_1, \dots, g_n$, as described in Section 5.5.1. Variable C_j represents the amount of resource left on server j , and R is the set of unassigned threads. The outer loop of the algorithm runs until all threads in R have been assigned. During each loop, U is the set of (thread, server) pairs, where the server has at least as much remaining resources as the thread's super-optimal allocation. If any such pairs exist, then in line 6 we find the thread in U with the greatest utility. Otherwise, in line 9 we find the thread that can obtain the greatest utility using the remaining resources of any server. In both cases we assign the thread in line 13 to the server giving it the greatest utility. Any ties are broken arbitrarily. Lastly, we update the server's remaining resources accordingly.

Algorithm 5.1

Input: Super-optimal allocation $[\hat{c}_1, \dots, \hat{c}_n]$, and $g_1, \dots, g_n$ as defined in Equation 5.1

```

1:  $C_j \leftarrow C$  for  $j = 1, \dots, m$ 
2:  $R \leftarrow \{1, \dots, n\}$ 
3: while  $R \neq \emptyset$  do
4:    $U \leftarrow \{(i, j) \mid (i \in R) \wedge (1 \leq j \leq m) \wedge (\hat{c}_i \leq C_j)\}$ 
5:   if  $U \neq \emptyset$  then
6:      $(i, j) \leftarrow \operatorname{argmax}_U g_i(\hat{c}_i)$ 
7:      $c_i \leftarrow \hat{c}_i$ 
8:   else
9:      $(i, j) \leftarrow \operatorname{argmax}_{i \in R, 1 \leq j \leq m} g_i(C_j)$ 
10:     $c_i \leftarrow C_j$ 
11:   end if
12:    $C_j \leftarrow C_j - c_i$ 
13:    $r_i \leftarrow j$ 
14:    $R \leftarrow R - \{i\}$ 
15: end while
16: return  $(r_1, c_1), \dots, (r_n, c_n)$ 

```

5.5.2.1 Analysis

We now analyze the quality of the allocation produced by Algorithm 5.1. We first define some notation. Let $D = \{i \in T \mid c_i = \hat{c}_i\}$ be the set of threads whose allocation in Algorithm 5.1 equals their super-optimal allocation, and let $E = T - D$ be the remaining threads. We say the threads in D are *full*, and the threads in E are *unfull*. The full threads have the same utility in the super-optimal allocation and the allocation produced by Algorithm 5.1. Note that full threads are the ones computed in line 6, and unfull threads are computed in line 9. Thus, to show Algorithm 5.1 achieves a good approximation ratio it suffices to show the utility of the unfull threads in Algorithm 5.1 is sufficiently large compared to their utility in the super-optimal allocation. We first show some basic properties about the unfull threads.

Lemma 5.4. *At most one thread from E is allocated to any server.*

Proof. Suppose for contradiction there are two threads $t_a, t_b \in E$ allocated to a server s_k , and assume that t_a was allocated before t_b . Consider the time of t_b 's allocation, and let S_j denote the set of threads allocated to a server s_j . We have $\sum_{i \in S_k} c_i = C$, because $t_a \in E$, and so it was allocated all of s_k 's remaining resources in lines 10 and 12 of Algorithm

5.1. Also, $\sum_{i \in S_j} c_i = C$ for any $j \neq k$. Indeed, if $\sum_{i \in S_j} c_i < C$ for any $j \neq k$, then s_j has more remaining resources than s_k , and so t_b would be allocated to s_j instead of s_k because it can obtain more utility. Thus, together we have that when t_b is allocated, $\sum_{i \in T} c_i \geq \sum_{j=1}^m \sum_{i \in S_j} c_i = mC$. Now, $c_i \leq \hat{c}_i$ for all $t_i \in T$. Also, since $t_a, t_b \in E$, then $c_a < \hat{c}_a$ and $c_b < \hat{c}_b$. Thus, we have $\sum_{i \in T} \hat{c}_i > \sum_{i \in T} c_i \geq mC$, which is a contradiction because $\sum_{i \in T} \hat{c}_i = mC$ by Lemma 5.2. $\square$

Lemma 5.5. $|E| \leq m - 1$.

Proof. Lemma 5.4 implies that $|E| \leq m$, so it suffices to show $|E| \neq m$. Assume for contradiction $|E| = m$. Then by Lemma 5.4, for each server s_k there exists a thread $t_a \in E$ allocated to s_k . t_a receives all of s_k 's remaining resources, and so $\sum_{i \in S_k} c_i = C$ after its allocation. Then after all m threads in E have been allocated, we have $\sum_{i \in T} c_i = mC$. But since $c_a < \hat{c}_a$ for all $t_a \in E$, and $c_i \leq \hat{c}_i$ for all $t_i \in T$, we have $\sum_{i \in T} \hat{c}_i > \sum_{i \in T} c_i = mC$, which is a contradiction. Thus, $|E| \neq m$ and the lemma follows. $\square$

The next lemma shows that the total resources allocated to the unfull threads in Algorithm 5.1 is not too small compared to their super-optimal allocation.

Lemma 5.6. $\sum_{i \in E} c_i \geq \frac{|E|}{m} \sum_{i \in E} \hat{c}_i$.

Proof. We first partition the servers into sets U and V , where $U = \{j \in S \mid S_j \subseteq D\}$ is the set of servers containing only full threads, and $V = S - U$ are the servers containing some unfull threads. Let $C_j = C - \sum_{i \in S_j} c_i$ be the amount of unused resources on a server s_j at the end of Algorithm 5.1. Then $C_j = 0$ for all $j \in V$, since the unfull thread in S_j was allocated all the remaining resources on s_j . So, we have $\sum_{j \in U} C_j = \sum_{j \in S} C_j = \sum_{j \in S} (C - \sum_{i \in S_j} c_i) = mC - \sum_{i \in T} c_i$, and so $\sum_{i \in T} c_i = mC - \sum_{j \in U} C_j$.

Next, we have $\sum_{i \in T} c_i = \sum_{i \in D} \hat{c}_i + \sum_{i \in E} c_i = mC - \sum_{i \in E} \hat{c}_i + \sum_{i \in E} c_i$. The first equality follows because $c_i = \hat{c}_i$ for $i \in D$, and the second equality follows because $D \cup E = T$ and $\sum_{i \in T} \hat{c}_i = mC$. Combining this with the earlier expression for $\sum_{i \in T} c_i$, we have $mC - \sum_{j \in U} C_j = mC - \sum_{i \in E} \hat{c}_i + \sum_{i \in E} c_i$, and so

$$\sum_{i \in E} c_i + \sum_{j \in U} C_j = \sum_{i \in E} \hat{c}_i. \quad (5.2)$$

Now, assume for contradiction that $\sum_{i \in E} c_i < \frac{|E|}{m} \sum_{i \in E} \hat{c}_i$. Then by Equation 5.2 we have

$$\sum_{i \in U} C_i > \frac{m - |E|}{m} \sum_{i \in E} \hat{c}_i. \quad (5.3)$$

We have $|V| = |E|$, since by Lemma 5.4 each server in V contains only one unfull thread. Thus $|U| = m - |V| = m - |E|$. Using this in Equation 5.3, we have that there exists an $j \in U$ with

$$C_j \geq \frac{1}{|U|} \sum_{i \in U} C_i > \frac{1}{m} \sum_{i \in E} \hat{c}_i. \quad (5.4)$$

We claim that for all $i \in E$, $c_i \geq C_j$. Indeed, suppose $c_i < C_j$ for some i , so that $C_j > 0$. Thread t_i was not allocated to s_j , since then t_i would use all the remaining resources on s_j and $C_j = 0$, a contradiction. But since $C_j > c_i$, t_i should be allocated to s_j because it can obtain greater utility on s_i than its current server, which is also a contradiction. Thus, $c_i \geq C_j$ for all $i \in E$. Using this and Equation 5.4, we have

$$\sum_{i \in E} c_i \geq \sum_{i \in E} C_j = |E| C_j > \frac{|E|}{m} \sum_{i \in E} \hat{c}_i$$

However, this contradicts the assumption that $\sum_{i \in E} c_i < \frac{|E|}{m} \sum_{i \in E} \hat{c}_i$. Thus, the lemma follows. $\square$

Let $\gamma = \max_{i \in E} g_i(\hat{c}_i)$ be the maximum super-optimal utility of any thread in E . The following lemma says that any of first m threads allocated by Algorithm 5.1 is given its super-optimal allocation and has utility at least γ .

Lemma 5.7. *Let i be one of the first m threads allocated by Algorithm 5.1. Then $i \in D$ and $g_i(c_i) \geq \gamma$.*

Proof. To show $i \in D$, note that the m servers all had C resource at the start of Algorithm 5.1, and fewer than m threads were allocated before t_i . So in t_i 's iteration there was at least one server with C resource, and so t_i can obtain $\hat{c}_i$ resource on one of those servers, and $i \in D$.

To show $g_i(c_i) \geq \gamma$, suppose the opposite, and let $j \in E$ be such that $g_j(\hat{c}_j) = \gamma$. Since $\hat{c}_j \leq C$, and since in t_i 's iteration there was some server with C resource, then in that iteration Algorithm 5.1 would have obtained greater utility by allocating t_j instead of t_i , which is a contradiction. Thus, $g_i(c_i) \geq \gamma$. $\square$

Lemma 5.7 implies there are at least m threads in D , and so we have the following.

Corollary 5.1. $\sum_{i \in D} g_i(c_i) \geq m\gamma$.

The next lemma shows that for the threads in E , threads with higher slopes in the nonconstant portion of their utility functions are allocated more resources.

Lemma 5.8. *For any two threads $i, j \in E$, if $\frac{g_i(\hat{c}_i)}{\hat{c}_i} > \frac{g_j(\hat{c}_j)}{\hat{c}_j}$, then $c_i \geq c_j$.*

Proof. Suppose for contradiction $c_i < c_j$, and suppose first that t_i was allocated before t_j . Then when t_i was allocated, there was at least one server with c_j or more remaining resources. We have $\hat{c}_i > c_j$, since otherwise t_i can be allocated $\hat{c}_i$ resources, so that $i \notin E$. Now, since $\hat{c}_i > c_j > c_i$, then t_i could obtain greater throughput by being allocated c_j instead of c_i amount of resources. This is a contradiction.

Next, suppose t_j was allocated before t_i . Then when t_j was allocated, there was a server with at least c_j amount of resources. Again, we have $\hat{c}_i > c_j$. Indeed, otherwise we have $\hat{c}_i \leq c_j$, and $\hat{c}_j > c_j$ since $j \in E$, and so t_i can be allocated its super-optimal allocation while t_j cannot. But Algorithm 5.1 prefers in line 4 to allocate threads that can receive their super-optimal allocations, and so it would allocate t_i before t_j , a contradiction. Thus, $\hat{c}_i > c_j$. However, this means that in the iteration in which t_j was allocated, t_i can obtain greater utility than t_j , since $g_i(c_j) = c_j \frac{g_i(\hat{c}_i)}{\hat{c}_i} > c_j \frac{g_j(\hat{c}_j)}{\hat{c}_j} = g_j(c_j)$, where the first equality follows because $\hat{c}_i > c_j$, the inequality follows because $\frac{g_i(\hat{c}_i)}{\hat{c}_i} > \frac{g_j(\hat{c}_j)}{\hat{c}_j}$, and the second equality follows because $\hat{c}_j > c_j$. Thus, t_i would be allocated before t_j , a contradiction. The lemma thus follows. $\square$

We now list several mathematical facts used in the rest of the proof. The first fact follows from the Cauchy-Schwarz inequality, the second from direct manipulation, and the third is Chebyshev's sum inequality.

Fact 5.1. *Given $a_1, \dots, a_n > 0$, we have $(\sum_{i=1}^n a_i)(\sum_{i=1}^n \frac{1}{a_i}) \geq n^2$.*

Fact 5.2. *Given $a, a', b, b' > 0$, if $\frac{a}{a'} < \frac{b}{b'}$, then $\frac{a}{a'} < \frac{a+b}{a'+b'} < \frac{b}{b'}$.*

Fact 5.3. *Given $a_1 \geq a_2 \geq \dots \geq a_n$ and $b_1 \geq b_2 \geq \dots \geq b_n$, we have $\sum_{i=1}^n a_i b_i \geq (\frac{1}{n} \sum_{i=1}^n a_i)(\sum_{i=1}^n b_i)$.*

We now prove a lower bound on a certain function.

Lemma 5.9. *Let $A, d > 0$, and $0 < a_1 \leq a_2 \leq \dots \leq a_n$. Also, let $\beta = (A + \sum_{i=1}^n a_i z_i) / (A + \sum_{i=1}^n z_i)$, where each $z_i \in [0, d]$. Then*

$$\beta \geq \min_{j=1, \dots, n} \left(\frac{A + \sum_{i=1}^j a_i d}{A + jd}, 1 \right)$$

Proof. If $a_1 \geq 1$, then Fact 5.2 implies that $\beta \geq 1$, and the lemma holds. Otherwise, suppose $a_1 < 1$. Then differentiating β with respect to z_1 , we get

$$\beta'(z_1) = \frac{(a_1 - 1)A + \sum_{i=2}^n (a_1 - a_i)z_i}{(A + \sum_{i=1}^n z_i)^2}$$

Since $a_1 \leq a_2 \leq \dots \leq a_n$ and $a_1 < 1$, we have $\beta'(z_1) < 0$. Thus, $\beta(z_1)$ is minimized for $z_1 = d$, and we have

$$\beta \geq \frac{A + a_1 d + \sum_{i=2}^n a_i z_i}{A + d + \sum_{i=2}^n z_i}$$

To simplify this expression, suppose first that $(A + a_1 d) / (A + d) \leq a_2$. Then we have

$$\beta \geq \frac{A + a_1 d + \sum_{i=2}^n a_i z_i}{A + d + \sum_{i=2}^n z_i} \geq \frac{A + a_1 d}{A + d}.$$

The second inequality follows because $a_2 \leq \dots \leq a_n$ and by Fact 5.2. Thus, the lemma is proved. Otherwise, $(A + a_1 d) / (A + d) > a_2$, and so

$$\frac{A + a_1 d + \sum_{i=2}^n a_i z_i}{A + d + \sum_{i=2}^n z_i} \geq \frac{A + a_1 d + a_2 d + \sum_{i=3}^n a_i z_i}{A + 2d + \sum_{i=3}^n z_i}$$

We can simplify the latter expression in a way similar to above, based on whether $(A + a_1 d + a_2 d) / (A + 2d) \leq a_3$. Continuing this way, if we stop at the j 'th step, then $\beta \geq (A + \sum_{i=1}^j a_i d) / (A + jd)$. Otherwise, after the n 'th step, we have $\beta \geq (A + \sum_{i=1}^n a_i d) / (A + nd)$. In either case, the lemma holds. $\square$

Since for $i \in E$, $\hat{c}_i > c_i$ and by definition of g , we have $g(c_i) = \frac{g_i(\hat{c}_i)}{\hat{c}_i} c_i$ for $i \in E$. Algorithm 5.1 produces an allocation $c_1, \dots, c_n$ with total utility $G = \sum_{i \in D} g_i(\hat{c}_i) + \sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i} c_i$. We now prove that this allocation is an α approximation to the super-optimal utility $\hat{F} = \sum_{i \in T} f_i(\hat{c}_i)$.

Lemma 5.10. $G \geq \alpha \hat{F}$, where $\alpha = 2(\sqrt{2} - 1) > 0.828$.

Proof. We have $\hat{F} = \sum_{i \in T} f_i(\hat{c}_i) = \sum_{i \in T} g_i(\hat{c}_i)$ by the definition of the g_i 's. Thus,

$$\begin{aligned} \frac{G}{\hat{F}} &= \frac{\sum_{i \in D} g_i(\hat{c}_i) + \sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i} c_i}{\sum_{i \in D} g_i(\hat{c}_i) + \sum_{i \in E} g_i(\hat{c}_i)} \\ &\geq \frac{m\gamma + \sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i} c_i}{m\gamma + \sum_{i \in E} g_i(\hat{c}_i)} \\ &\geq \frac{m\gamma + (\sum_{i \in E} c_i / |E|) \sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i}}{m\gamma + \sum_{i \in E} g_i(\hat{c}_i)} \\ &\geq \frac{m\gamma + (\sum_{j \in E} \hat{c}_j / m) \sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i}}{m\gamma + \sum_{i \in E} g_i(\hat{c}_i)} \end{aligned}$$

Recall that $\gamma = \max_{i \in E} g_i(\hat{c}_i)$. The first inequality follows because of Corollary 5.1 and because $\hat{c}_i \geq c_i$ for all i . The second inequality follows because by Lemma 5.8, threads $i \in E$ with larger values of $\frac{g_i(\hat{c}_i)}{\hat{c}_i}$ also have larger values of c_i . Thus, we can apply Fact 5.3 to bring the term $\sum_{i \in E} c_i / |E|$ outside the sum $\sum_{i \in E} \frac{g_i(\hat{c}_i)}{\hat{c}_i} c_i$. The last inequality follows because of Lemma 5.6. Now, assume WLOG that the elements in E are ordered by non-increasing value of $\hat{c}_i$, so that $\frac{1}{\hat{c}_1} \leq \frac{1}{\hat{c}_2} \leq \dots \leq \frac{1}{\hat{c}_{|E|}}$. Let E_i denote the first i elements of E in this order. For any $i \in E$, we have $g_i(\hat{c}_i) \in [0, \gamma]$. Thus, applying Lemma 5.9 to the last expression above, letting $g_i(\hat{c}_i)$ play the role of z_i and $\frac{\sum_{j \in E} \hat{c}_j}{m\hat{c}_i}$ play the role of a_i , and noting $\frac{G}{\hat{F}} \leq 1$, we have

$$\begin{aligned} \frac{G}{\hat{F}} &\geq \min_{i=1, \dots, |E|} \left(\frac{m\gamma + (\sum_{j \in E} \hat{c}_j / m) \sum_{j \in E_i} \frac{\gamma}{\hat{c}_j}}{m\gamma + i\gamma} \right) \\ &\geq \min_{i=1, \dots, |E|} \left(\frac{m - \frac{1}{m} (\sum_{j \in E_i} \hat{c}_j) (\sum_{j \in E_i} \frac{1}{\hat{c}_j})}{m - i} \right) \\ &\geq \min_{i=1, \dots, |E|} \left(\frac{m - \frac{i^2}{m}}{m + i} \right) \end{aligned}$$

The second inequality follows by simplification and because $\sum_{j \in E} \hat{c}_j \geq \sum_{j \in E_i} \hat{c}_j$ for any i . The last inequality follows by Fact 5.1. It remains to lower bound the final expression. Treating i as a real value and taking the derivative with respect to i , we find the minimum value is obtained at $i = (\sqrt{2} - 1)m$, for which $\frac{G}{\hat{F}} \geq 2(\sqrt{2} - 1) = \alpha$. Thus, the lemma is proved. $\square$

5.5.3 Solving the concave problem

To solve the original distributed resource allocation problem with concave utility functions $f_1, \dots, f_n$, we run Algorithm 5.1 on the linearized problem to obtain an allocation $c_1, \dots, c_n$, then simply output this as the solution to the concave problem. The total utility of this solution is $F = \sum_{i \in T} f_i(c_i)$. We now show this is an α approximation to the optimal utility F^* .

Theorem 5.2. $F \geq \alpha F^*$, and Algorithm 5.1 achieves an α approximation ratio.

Proof. We have $F = \sum_{i \in T} f_i(c_i) \geq \sum_{i \in T} g_i(c_i) \geq \alpha \hat{F} \geq \alpha F^*$, where the first inequality follows because $f_i(c_i) \geq g_i(c_i)$ by Lemma 5.3, the second inequality follows by Lemma 5.10, and the last inequality follows by Lemma 5.1. $\square$

Next, we give a simple example that shows our analysis of Algorithm 5.1 is nearly tight.

Theorem 5.3. *There exists an instance of distributed resource allocation where Algorithm 5.1 achieves $\frac{5}{6} > 0.833$ times the optimal total utility.*

Proof. Consider 3 threads, and 2 servers each with one (divisible) unit of resource. Let

$$f_1(x) = \begin{cases} 2x & \text{if } x \in [0, \frac{1}{2}] \\ 1 & \text{if } x > \frac{1}{2} \end{cases}$$

Also, let $f_2(x) = x$. Suppose the first two threads both have utility functions f_1 , and third thread has utility function f_2 . The super-optimal allocation is $[\hat{c}_1, \hat{c}_2, \hat{c}_3] = [\frac{1}{2}, \frac{1}{2}, 1]$. Algorithm 5.1 may assign threads 1 and 2 to different servers, with $\frac{1}{2}$ resource each, then assign thread 3 to server 1 with $\frac{1}{2}$ resource. This has a total utility of $2\frac{1}{2}$. On the other hand, the optimal assignment is to put threads 1 and 2 on server 1 and thread 3 on server 2. This has a utility of 3. $\square$

Lastly, we analyze Algorithm 5.1's time complexity.

Theorem 5.4. *Algorithm 5.1 runs in $O(mn^2 + n(\log mC)^2)$ time.*

Proof. Computing the super-optimal allocation takes $O(n(\log mC)^2)$ time using the algorithm in [83]. Then the algorithm runs n loops, where in each loop it computes the set U with $O(mn)$ elements. Thus, the theorem follows. $\square$

5.6 A Faster Algorithm

In this section, we present a faster approximation algorithm that achieves the same approximation ratio as Algorithm 1 in $O(n(\log mC)^2)$ time. The pseudocode is shown in Algorithm 5.2.

The algorithm also takes as input a super-optimal allocation $\hat{c}_1, \dots, \hat{c}_n$, which we compute as in Section 5.5.1. It sorts the threads in nondecreasing order of $g_i(\hat{c}_i)$. It then takes threads $m+1$ to n in this ordering, and resorts them in nondecreasing order of $g_i(\hat{c}_i)/\hat{c}_i$. Next, it initializes $C_1, \dots, C_m$ to C , and stores them in a max heap H . C_j represents the amount of remaining resources on server j . The main loop of the algorithm iterates through the threads in order. Each time it chooses the server with the most remaining resources, allocates the minimum of the thread's super-optimal allocation and the server's remaining resources to it, and assigns the thread to the server. Then H is updated accordingly.

Algorithm 5.2

Input: Super-optimal allocation $[\hat{c}_1, \dots, \hat{c}_n]$, and $g_1, \dots, g_n$ as defined in Equation 5.1

```

1: sort threads in nonincreasing order of  $g_i(\hat{c}_i)$  as  $t_1, \dots, t_n$ 
2: sort  $t_{m+1}, \dots, t_n$  in nonincreasing order of  $g_i(\hat{c}_i)/\hat{c}_i$ 
3:  $C_j \leftarrow C$  for  $j = 1, \dots, m$ 
4: store  $C_1, \dots, C_m$  in a max-heap  $H$ 
5: for  $i = 1, \dots, n$  do
6:    $j \leftarrow \operatorname{argmax}_{1 \leq j \leq m} C_j$ 
7:    $c_i \leftarrow \min(\hat{c}_i, C_j)$ 
8:    $C_j \leftarrow C_j - c_i$ , and update  $H$ 
9:    $r_i \leftarrow j$ 
10: end for
11: return  $(r_1, c_1), \dots, (r_n, c_n)$ 
```

5.6.1 Algorithm Analysis

We now show Algorithm 5.2 achieves a $2(\sqrt{2}-1)$ approximation ratio, and runs in $O(n(\log mC)^2)$ time. The proof of the approximation ratio uses exactly the same set of lemmas as in Section 5.5.2.1. The proofs of most of the lemmas are also similar. Rather than replicating all the proofs from Section 5.5.2.1, we will go through each lemma and point out any differences in the proof. Please refer to Section 5.5.2.1 for the definitions, lemma statements and original proofs.

- *Lemmas 5.1, 5.2, 5.3* These lemmas deal with the super-optimal allocation, which is the same in Algorithms 5.1 and 5.2.
- *Lemma 5.4* The proof of this lemma depended on the fact in Algorithm 5.1 that if we assign a second E thread to a server, then all the other servers have no remaining resources. This is also true in Algorithm 5.2, since in line 6 we assign a thread to a server with the most remaining resources, so that if when we assign a second E thread t to a server s , there was another server s' with positive remaining resources, we would assign t to s' instead. a contradiction.
- *Lemma 5.5* This follows by exactly the same arguments as the original proof.
- *Lemma 5.6* The only statement we need to check from the original proof is that for all $i \in E$, $c_i \geq C_j$, where C_j is defined as in Equation 5.4. But this is true in Algorithm 5.2 because if there were any $c_i < C_j$, line 6 of Algorithm 5.2 would assign thread i to server j instead of i 's current server, a contradiction. All the other statements in the original proof then follow.
- *Lemma 5.7* This follows because lines 1 and 2 of Algorithm 5.2 show that the first m threads allocated have at least as much super-optimal utility as the remaining $n - m$ threads. Also, the first m threads must be in D , since there is always a server with C resources during the first m iterations of Algorithm 5.2. Thus, all threads in E are among the last $n - m$ allocated, and their maximum super-optimal utility is no more than the minimum utility of any D thread.
- *Corollary 5.1* This follows immediately from Lemma 5.7.
- *Lemma 5.8* As we stated above, all threads in E must be among the last $n - m$ allocated by Algorithm 5.2. That is, they are among threads $t_{m+1}, \dots, t_n$. But in line 2 these threads are sorted in nondecreasing order of $g_i(\hat{c}_i)/\hat{c}_i$. Thus, the lemma follows.
- *Facts 5.1 to 5.3, Lemma 5.9* These follow independently of Algorithm 5.2.

- *Lemma 5.10* The proof of this lemma used only the preceding lemmas, not any properties of Algorithm 5.1. Thus, it also holds for Algorithm 5.2.

Given the preceding lemmas, we can state the approximation ratio of Algorithm 5.2. The proof of the theorem is the same as the proof of Theorem 5.2, and is omitted.

Theorem 5.5. *Let F be the total utility from the allocation produced by Algorithm 5.2, and let F^* be the optimal total utility. Then $F \geq \alpha F^*$.*

Lastly, we analyze Algorithm 5.2's time complexity.

Theorem 5.6. *Algorithm 5.2 runs in $O(n(\log mC)^2)$ time.*

Proof. Finding the super-optimal allocation takes $O(n(\log mC)^2)$ using the algorithm in [83]. Steps 1 and 2 take $O(n \log n)$ time. Each iteration of the main for loop takes $O(\log m)$ time to extract the maximum element from H and update H . Thus, the entire for loop takes $O(n \log m)$ time. In practice, $C \gg n$, so the overall running time is dominated by the time to find the super-optimal allocation. $\square$

5.7 Experimental Evaluation

To evaluate the quality of the solutions produced by Algorithm 5.2, we compare the total utility it achieves with the super-optimal (SO) utility, which is at least as large as the optimal utility. We also compare Algorithm 5.2 with several other simple but practical heuristics, UU, UR, RU and RR. The UU (uniform-uniform) heuristic assigns threads in a round robin manner to the servers, and allocates the threads assigned to a server the same amount of resources. UR (uniform-random) assigns threads in a round robin manner, and allocates threads a random amount of resources on each server. RU (random-uniform) assigns threads to random servers, and equally allocates resources on each server. Finally, RR (random-random) randomly assigns threads and allocates them random amounts of resource. While simplistic, these heuristics may be similar to techniques used in practice, and hence give a reasonable comparison.

Our experiments use threads with random utility functions generated according to various probability distributions, as follows. We fix an amount of resources C on each server,

Figure 5.1: Performance of Algorithm 1 versus SO, UU, RU, UR, and RR as a function of β under the uniform and normal distributions.

and set the value of the utility function at 0 to be 0. Then we generate two values v and w according to the distribution H , conditioned on $w \leq v$, and set the value of the utility function at $\frac{C}{2}$ to v , and the value at C to $v + w$. Lastly, we apply cubic interpolation using Matlab's `pchip` function to the three generated points to produce a smooth concave utility function. In all the experiments, we set the number of servers to be $m = 8$, and test the effects of varying different parameters. One parameter is $\beta = \frac{n}{m}$, the average number of threads per server. The results below show the average performance from 1000 random trials. Performance represents the ratio of Algorithm 5.2's utility versus the utility achieved by other algorithms.

5.7.1 Uniform and normal distributions

We first consider Algorithm 5.2's performance compared to SO, UU, UR, RU and RR on threads with utility functions generated according to the uniform and normal distributions. Figures 5.1(a) and 5.1(b) show the ratio of Algorithm 5.2's total utility versus the utility of the other algorithms, for β varying between 1 to 15. The behaviors for both distributions are similar. Compared to SO, our performance never drops below 0.99, meaning that Algorithm 5.2 always achieves at least 99% of the optimal utility. The ratios compared to UU, UR, RU and RR are always above 1, so that we always perform better than the simple heuristics. For small values of β , UU performs well. Indeed, for $\beta = 1$, UU achieves the optimal

Figure 5.2: Performance of Algorithm 1 versus SO,UU,RU,UR, and RR as a function of β and α under the power law distribution.

utility because it places one thread on each server and allocates it all the resources. UR does not achieve optimal utility even for $\beta = 1$, since it allocates threads a random amount of resources. RU and RR may allocate multiple threads per server, and also do not achieve the optimal utility. As β grows, the performance of the heuristics get worse relative to Algorithm 5.2. This is because as the number of threads grows, it becomes more likely that some threads have very high maximum utility. These threads need to be assigned and allocated carefully. For example, they should be assigned to different servers and allocated as much resources as possible. The heuristics likely fail to do this, and hence obtain low performance. The performance of UR and RR, and UU and RU converge as β grows. This is because both random and uniform assignment assigns the threads roughly evenly between the servers for large β . Also, the performances of UU and RU are substantially better than UR and RR, which indicates that the way in which resources are allocated has a bigger effect on performance than how threads are assigned, and that uniform allocation is generally better than random allocation.

5.7.2 Power law distribution

We now look at the performance of Algorithm 5.2 using threads generated according to the power law distribution. Here, each value x has a probability $\lambda x^{-\alpha}$ of occurring, for a normalization factor λ and $\alpha > 1$. Figure 5.2(a) shows the effect of varying β while fixing $\alpha = 2$. Here we see the same trends as those under the uniform and normal distributions,

Figure 5.3: Performance of Algorithm 1 versus SO, UU, RU, UR, and RR as a function of β , γ and θ under the discrete distribution.

namely that Algorithm 5.2 always performs very close to optimal, while the performance of the heuristics gets worse with increasing β . However, the rate of performance degradation is faster than with the uniform and normal distributions. This is because the power law distribution with $\alpha = 2$ is more likely to generate threads with very different maximum utilities. These threads must be carefully assigned and allocated, which the heuristics fail to do. For $\beta = 15$, Algorithm 5.2 is 3.9 times better than UU and RU, and 5.7 times better than UR and RR.

Figure 5.2(b) shows the effect of varying α , using a fixed $\beta = 5$. Algorithm 5.2's performance is nearly optimal. In addition, the performance of the heuristics improves as α increases. This is because for higher values of α , it is unlikely that there are threads with very high maximum utilities. So, since the maximum utilities of the threads are roughly the same, almost any even assignment of the threads works well. Despite this, we still observe that UU and RU perform better than UR and RR. This is because when the threads are roughly the same, then due to the concavity of the utility functions, the optimal allocation is to give each thread the same amount of resources. This is done by UU and RU, but not by UR and RR, which allocate resources randomly.

5.7.3 Discrete distribution

Lastly, we look at the performance using utility functions generated by a discrete distribution. This distribution takes on only two values ℓ, h , with $\ell < h$. γ is a parameter that controls the probability that ℓ occurs, and $\theta = \frac{h}{\ell}$ is a parameter that controls the relative size of the values. Figure 5.3(a) shows Algorithm 1's performance as we vary β , fixing

$\gamma = 0.85$ and $\theta = 5$. The same trends as with the other distributions is observed. Figure 5.3(b) shows the effect of varying γ , when $\beta = 5$ and $\theta = 5$. Our algorithm achieves the lowest performance for $\gamma = 0.75$, when we achieve 97.5% of the super-optimal utility. The four heuristics also perform worst for this value. For γ close to 0 or 1, all the heuristics perform well, since these correspond to instances where either h or l is very likely to occur, so that almost all the threads have the same maximum utility. Lastly, we consider the effect of varying θ . Here, as θ increases, the difference between the high and low utilities becomes more evident, and the effect of poorly assigning threads or misallocating resources becomes more serious. Hence, the performance of the heuristics decrease with θ . Meanwhile, Algorithm 1 always achieves over 99% of the optimal utility.

5.8 Summary

In this chapter, we studied the problem of simultaneously assigning threads to servers and allocating server resources to maximize total utility. Thread assignment and resource allocation are investigated separately extensively in the existing works. However, to our knowledge there has been work to combine two aspects together. In this chapter, we studied the two aspects together in the unified context. In the problem, each thread was modeled by a concave utility function. We showed that the problem is NP-hard, even when there are only two servers. We also presented two algorithms with approximation ratio $2(\sqrt{2} - 1) > 0.828$, running in times $O(mn^2 - n(\log mC)^2)$ and $O(n(\log mC)^2)$, resp. We tested our algorithms on multiple types of threads, and found that we achieve over 99% of the optimal utility on average. We also perform up to 5.7 times better than several heuristic methods.

Chapter 6

Conclusions and Future Work

In this chapter, we conclude the thesis and propose several directions to continue our research.

6.1 Conclusions

Two opposing trends distribution and consolidation are occurring in today's computer systems, ranging from multicore to clouds. The systems where threads are consolidated contain numerous limited resources (e. g. cache, memory) to run their hosted threads, leading to the fact that their resource are shared among threads. In resource sharing, allocating shared resource effectively requires computer system to be aware of each thread's resource demand, and a flexible method to capture the resource demand of a thread is by utility function. In resource sharing, properly allocating shared resource is crucial to achieve high overall performance. In resource allocation, makespan and total utility are two important metrics to evaluate the quality of allocation.

In the thesis, we investigated how to allocate a shared resource among threads with utility functions to achieve high overall performance in terms of makespan and total utility. Many threads have concave utility functions from observations and this property has certain theoretical justifications. We first studied how to allocate a shared resource to multiple threads with concave utility functions to minimize the makespan in Chapter 3. We show that the characteristics of optimal schedule and present an efficient approximation algorithm to find an approximately optimal schedule based on the characterization. Simulation results show that our algorithm can produce schedules between 1.22 to 1.59 better than those from existing allocation algorithms.

Also, we observe many realistic utility functions are not concave. We present efficient algorithms to allocate a shared resource among multiple threads with nonconcave utility functions to maximize the total utility in Chapter 4. A key observation is that though the utility functions are nonconcave, they can be divided into a small number of concave or convex segments. We present both efficient optimal algorithm and approximate algorithm for the problem leveraging the observation. Our optimal algorithm runs in $O(s n \alpha(m) m (\log m)^2)$ time, and our approximate algorithm finds an $1 - \epsilon$ optimal allocation in $O(\frac{s}{\epsilon} \alpha(\frac{n}{\epsilon}) n^2 \log \frac{n}{\epsilon} \log m)$ time for any $\epsilon > 0$; here, s is the number of segments, n is the number of processes, m is the amount of resource, α is the inverse Ackermann function. In most settings, s and α are very small. Both of the two algorithms are almost order m times faster than existing ones. Experiments show that our exact algorithm runs up to 215 times faster than the exact dynamic programming based algorithm, and gives solutions that are 33% better than the greedy heuristic.

In addition to centralized resource, there are lots of distributed resource in distributed systems. In Chapter 5, we introduced the distributed resource allocation problem. We want to simultaneously find an assignment and allocations for each thread that maximize the total utility of all the threads. We presented two algorithms with approximation ratio $2(\sqrt{2} - 1) > 0.828$, running in times $O(mn^2 + n(\log mC)^2)$ and $O(n(\log mC)^2)$, resp; here, m is the number of servers, and n is the number of threads, and C is the amount of resource on each server. We tested our algorithms on multiple types of threads, and found that we achieve over 99% of the optimal utility on average. We also perform up to 5.7 times better than several heuristic methods.

6.2 Future works

In the future, there are several directions to continue our work. There are numerous applications and computer systems today, which have many different characteristics. More algorithms are a necessity to deal with different situations in the emerging computer applications and systems to strengthen the completeness of resource allocation. More algorithms are needed to deal with: (1) applications with dynamic resource behavior; (2) makespan

minimization in distributed resource allocation; (3) heterogeneous servers in distributed systems; (4) multiple type of resource; (5) online information of applications.

6.2.1 Dynamic Resource Allocation

Our algorithm assumes that the work speed function of all the threads is constant during the execution in Chapter 3. However, in certain applications, the work speed functions may vary over time. For example, [74] showed the miss rate of applications from CPU benchmarks as a function of the execution time moment. The miss rates of several applications (e. g. gcc, mesa) changes over time. This indicates that the work speed function of these applications also vary over time and our algorithm is not able to deal with these applications. Therefore, it would be useful to design an algorithm to deal with applications with dynamic behaviors.

We observe that though these applications have dynamic behaviors, they do not change arbitrarily. In particular, the behaviors of these applications can be divided into a small number of phases and in each phase, the behaviors of the applications keep nearly constant. Figure 6.1 demonstrates how L2 cache miss rate changes over time for the CPU benchmark *gcc*. There are three graphs. Each dot in these graphs represents the miss rate measured over a period of ten million instructions. Each graph shows the miss rates over a window of a half a billion instructions. The first graph shows the half-billion instruction window immediately after the program starts, the second shows the half-billion instruction window after one billion instructions have been executed, and the third after two billion instructions have been executed. From the second and third graph, we can see the phase behavior of *gcc*'s cache miss rate. We would like to leverage this observation to design efficient algorithms to deal with applications with dynamic behaviors. The problem seems to be solvable when a subproblem with fewer number of phases is solved. We may leverage the observation to design algorithms using dynamic programming method.

6.2.2 Makespan Minimization Distributed Resource Allocation

In the distributed resource allocation problem, we only consider to optimize the performance in terms of total utility. Makespan is also a very important performance goal in computer systems. Currently, parallel computing is widely used in many areas to achieve high

Figure 6.1: *Gcc*'s cache miss rate over time [95]

performance computing. One important type of parallel computing is task parallelism. In task parallelism, a task is divided into several independent subtasks, each is corresponding to different type of work, and the subtasks are executed by a set of concurrent threads. Each thread may have different work length, since each type of work is disparate. The completion time of the task is determined by the slowest thread. To save the runtime of the task, it is essential to minimize the finishing time of the slowest thread, which is makespan. However, makespan is a different performance metric compared to total utility, and the algorithms in chapter 5 does not work for the distributed resource allocation problem to minimize the makespan. Therefore, it is essential to propose algorithms to allocate distributed resource to minimize makespan.

The problem is similar with the problem in chapter 5 except the optimization goal. We wonder whether a general transformation of the algorithm in chapter 5 can work for makespan minimization. We are interested in investigating whether this is true.

6.2.3 Heterogeneous Distributed Resource Allocation

In the distributed resource allocation problem, we assume all servers are homogeneous. However, servers in practice may be heterogeneous. In a data center, heterogeneity occurs because servers are gradually provisioned and replaced over the last decades [75, 76]. At any time, a data center may host 3-5 server generations with a few hardware configurations per generation, in terms of the specific speeds and capacities of the processor, memory, storage and networking subsystem and it is common to have 10 to 40 configurations throughout the data center. For example, Table 5.1 illustrates the heterogeneity of servers in a Google's cluster. The heterogeneity of servers causes new distributed resource allocation problems

in two aspects. On one hand, the resource size of each server is different. On the other hand, the utility functions of a thread assigned on different servers are different. We would like to study the problem when there are different types of servers.

Number of Servers	CPUs	Memory
6732	0.50	0.50
3863	0.50	0.25
1001	0.50	0.75
795	1.00	1.00
126	0.25	0.25
52	0.50	0.12
5	0.50	0.03
5	0.50	0.97
3	1.00	0.50
1	0.50	0.06

Table 6.1: Server configurations in one of Google’s clusters [89, 90]. CPU and memory are normalized.

6.2.4 Multiple Type Resource Allocation

In the thesis, we assume the shared resource is single type. In other words, each thread has one-dimensional utility function. However, in many real applications, threads share multiple types of resources. For instance, a multiprocessor system runs multiple threads. The multiprocessor share its resource, such as cache, memory, disk among the threads. In cloud computing, one physical machines runs several virtual machines. The physical machine shares its resource, such as processing, memory, bandwidth to its virtual machines. In these applications, each thread has multi-dimensional utility functions. Therefore, the one-dimensional resource allocation algorithms proposed in the thesis cannot be directly applied. We would like to consider a more general system model incorporating multiple types of resources, and to study optimal allocation schemes in these settings.

The problem is essentially a multiple choice multiple dimensional knapsack problem. We may design algorithms for the problem based on existing algorithms for multiple choice multiple dimensional knapsack problem.

6.2.5 Online Resource Allocation

Our algorithms assume utility function of each thread is known. In many situations, these information cannot be known before running them. There has been much work on how to measure on the fly information of running threads. For example, [93, 77] proposed efficient software based techniques to measure online cache miss rate curves. We would like to integrate on the fly information to design efficient online algorithms for these situations.

Appendix

Author's Publication

- (i) Pan Lai, Rui Fan. Makespan Optimal Cache Partitioning. *Proceedings of the IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2013.
- (ii) Pan Lai, Rui Fan. Fast Optimal Nonconcave Resource Allocation. *34th IEEE Conference on Computer Communications (INFOCOM)*, 2015.
- (iii) Wei Zhang, Rui Fan, Fang Liu, Pan Lai. Energy-Aware Caching. *21st IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2015.
- (iv) Pan Lai, Rui Fan, Wei Zhang, Fang Liu. Utility Maximization Thread Assignment and Resource Allocation. *Accepted by 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

Full Terms of Abbreviation

Abbreviation	Full term
CMP	chip multiprocessor
IPC	instruction per cycle
MRC	miss rate curve
LRU	least recently used
VMM	virtual machine monitor
VM	virtual machine
MCKP	multiple choice knapsack problem
DP	dynamic programming
SO	super optimal
UU	uniform assignment uniform allocation
RU	random assignment uniform allocation
UR	uniform assignment random allocation
RR	random assignment random allocation

References

- [1] R. Mcmillan. Amazons secretive cloud carries 1 percent of the Internet. <http://www.wired.com/2012/04/amazon-cloud/>, 2012.
- [2] C.K. Chow. On optimization of storage hierarchies. *IBM Journal of R&D*, vol. 18, pp. 194-204, 1974.
- [3] C.K. Chow. Determination of caches capacity and its matching storage hierarchy. *IEEE Transaction on Computers*, vol. 25, pp. 157-164, 1976.
- [4] A. J. Smith. Cache memories. *Computing Surveys*, vol. 14, pp. 473 -528, 1982.
- [5] V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior, is it $\sqrt{2}$. *Proceedings of the 3rd Conference on Computing Frontiers*, 2006.
- [6] D. Thiebaut. *On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio*. *IEEE Transactions on Computers*, vol. 38, no. 7, 1989.
- [7] A. S. Cassidy. Beyond Amdahls Law: An objective function that links multiprocessor performance gains to delay and energy? In *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1110-1126, 2012.
- [8] R. Mattson, J. Geesei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 1970, 9(2):78-117.
- [9] T. King. Cache partitioning enhances multicore performance. *The Journal of Military Electronics & Computing*, 2013.
- [10] H. S. Stone, J. Turek and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, vol. 41, no. 9, 1992.

- [11] E. Suh, L. Rudolph, S. Devadas, Dynamic partitioning of shared cache memory. *Journal of Supercomputing Architecture*, 2002.
- [12] E. Suh, S. Devadas, and L. Rudolph, Analytical cache models with application to cache partitioning, *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, 2001.
- [13] E. Suh, L. Rudolph, S. Devadas, Dynamic cache partitioning for simultaneous multi-threading systems. *In the proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001
- [14] R. Iyer, CQoS: a framework for enabling QoS in shared caches of CMP platforms. *Proceedings of the 18th annual international conference on Supercomputing (ICS)*, 2004.
- [15] M. K. Qureshi, Y. N. Patt, Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. *International Symposium on Microarchitecture (MICRO)*, 2006.
- [16] J. Chang and G. S. Sohi, Cooperative cache partitioning for chip multiprocessors. *Proceedings of the 21st ACM International Conference on Supercomputing (ICS)*, June, 2007.
- [17] S. Kim, D. Chandra and Y. Solihin, Fair cache sharing and partitioning in a chip multiprocessor architecture. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.
- [18] L. Hsu, Steven K. Reinhardt, R. Iyer, S. Makineni, Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.
- [19] H. Dybdahl, P. Stenstrom, An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

- [20] M. Moreto , F. J. Cazorla, A. Ramirez , M. Valero. MLP-aware dynamic cache partitioning. *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*. 2008
- [21] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang and P. Sadayappan. Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems. *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.
- [22] S. Srikantaiah, R. Das, A. Mishra, C. Das, M. Kandemir. A case for integrated processor-cache Partitioning in chip multiprocessors. *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, 2009.
- [23] S. P. Muralidhara, M. Kandemir, P. Raghavan. Intra-application cache partitioning. *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010.
- [24] D. Sanchez, C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.
- [25] W. Wang, P. Mishra, S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multicore systems. *Design Automation Conference (DAC)*, 2011.
- [26] M. Kandemir, T. Yemliha, E. Kultursay. A helper thread based dynamic cache partitioning scheme for multithreaded applications. *Design Automation Conference (DAC)*, 2011.
- [27] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, B. Franke. Cooperative partitioning: energy-efficient cache partitioning for high-performance CMPs. *18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2012.
- [28] P. Zhou, V. Pandey, J. Sundarsn, A. Raghuraman, Y. Zhou and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. *Proceedings of the Eleventh*

- International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.
- [29] D. K. Tam, R. Azimi, L. B. Soares, M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.
- [30] L. He, Z. Yu, H. Jin. FractalMRC: online cache miss rate curve prediction on commodity systems. *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2012.
- [31] D. Minarolli, B. Freisleben. Utility-based resource allocation for virtual machines in cloud Computing. *IEEE Symposium on Computers and Communications*, 2011.
- [32] H. Yanagisawa, T. Osogami, R. Raymond. Dependable virtual machine allocation. *Proceedings of IEEE INFOCOM*, 2013.
- [33] H. S. Stone, J. Turek and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 1992, vol. 41, no. 9.
- [34] E. Suh, L. Rudolph, S. Devadas. Dynamic partitioning of shared resource memory. *Journal of Supercomputing Architecture*, 2002.
- [35] M. K. Qureshi, Y. N. Patt. Utility-based resource partitioning: a low-overhead, high-performance, runtime mechanism to partition shared resources. *IEEE/ACM International Symposium on Microarchitecture*, 2006.
- [36] S. Kim, D. Chandra and Y. Solihin. Fair resource sharing and partitioning in a chip multiprocessor architecture. *International Conference on Parallel Architectures and Compilation Techniques*, 2004.
- [37] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for Qos management. *Real-Time Systems Symposium*, 1997.
- [38] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for Qos-based resource allocation problems. *IEEE Real-Time Systems Symposium*, 1998.

- [39] P. Zhou, V. Pandey, J. Sundarsn, A. Raghuraman, Y. Zhou and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [40] W. Zhao, Z. Wang, Y. Luo. Dynamic memory balancing for virtual machines. *Conference on Virtual Execution Environments*, 2009.
- [41] Z. Galil, N. Megiddo. A fast selection algorithm and the problem of optimum distribution of effort. *Journal of ACM*, 1979, vol. 26, pp. 58-64.
- [42] N. Katoh, T. Ibaraki, H. Mine. A polynomial time algorithm for the resource allocation problem with a convex objective function. *The Journal of the Operational Research Society*, 1979, vol. 30, no. 5, pp. 449-455.
- [43] T. Ibaraki, N. Katoh. Resource allocation problems: algorithm approaches. *MIT Presss*, 1988, pp. 47-50.
- [44] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, 1989, vol. 33, pp. 169-174.
- [45] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 1986, 6:151-177.
- [46] H. Kellerer, U. Pferschy, D. Pisinger. Knapsack Problems. *Springer-Verlag Berlin Heidelberg*, 2004, pp. 329.
- [47] G.V. Gens and E.V. Levner. An approximate binary search algorithm for the multiple choice knapsack problem. *Information Processing Letters*, 1998, 67:261-265.
- [48] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 1979, vol. 4, No. 4, pp. 339-356.
- [49] B. Fox. Discrete optimization via marginal analysis. *Management Science*, 1966, vol. 13, pp. 909-918.
- [50] J. W. Lee, R. R. Mazumdar, N. B. Shroff. Downlink power allocation for multi-class CDMA wireless networks. *IEEE INFOCOM*, 2002.

- [51] A.K. Chandra, D.S. Hirschberg, C.K. Wong. Approximate algorithms for some generalized knapsack problems. *Theoretical Computer Science*, 1976. 3:293-304.
- [52] O. A. B. Yehuda, E. Posener, M. B. Yehuda, A. Schuster, A. Mualem. Ginseng: market-driven memory allocation. *International Conference on Virtual Execution Environments*, 2014.
- [53] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 1989, vol. 38, no. 7.
- [54] E. Suh, L. Rudolph, S. Devadas. Dynamic partitioning of shared resource memory. *Journal of Supercomputing Architecture*, 2002.
- [55] M. K. Qureshi, Y. N. Patt. Utility-based resource partitioning: a low-overhead, high-performance. runtime mechanism to partition shared resources. *IEEE/ACM International Symposium on Microarchitecture*, 2006.
- [56] B. Urgaonkar, P. Shenoy, T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.
- [57] B. Urgaonkar, A. Rosenberg, P. Shenoy. Application placement on a cluster of servers. *International Journal of Foundations of Computer Science*, 2007, vol. 18, no. 05, pp. 1023-1041.
- [58] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. *Proceedings of the 15th International Conference on World Wide Web*, 2006.
- [59] M. Becchi, P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *Proceedings of the 3rd Conference on Computing Frontiers*, 2006.
- [60] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, M. Valero. Optimal task assignment in multithreaded processors: a statistical approach. *Architectural Support for Programming Languages and Operating Systems*, 2012.

- [61] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, R. P. Doyle. Managing energy and server resources in hosting centers. *ACM Symposium on Operating Systems Principles*, 2001.
- [62] P. Lai, R. Fan. Fast optimal nonconcave resource allocation. *IEEE Conference on Computer Communications (INFOCOM)*, 2015.
- [63] B. Fox. Discrete optimization via marginal analysis. *Management Science*, 1966, vol. 13, pp. 909-918.
- [64] Y. L. Jiang, X. P. Shen, C. Jie, R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2008.
- [65] K. Tian, Y. L. Jiang, X. P. Shen, W. Z. Mao. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. *Proceedings of the 6th ACM conference on Computing Frontiers*, 2009.
- [66] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Survey*, 2012.
- [67] Z. Galil. A fast selection algorithm and the problem of optimum distribution of effort. *Journal of the Association for Computing Machinery*, 1979, vol. 26, no. 1, pp. 58-64.
- [68] G.V. Gens and E.V. Levner. An approximate binary search algorithm for the multiple choice knapsack problem. *Information Processing Letters*, 1998, vol. 67, 261-265.
- [69] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 1979, vol. 4, no. 4, pp. 339-356.
- [70] A. Neebe and D. Dannenbring. Algorithms for a specialized segregated storage problem. *Technical Report 77-5*, University of North Carolina, 1977.
- [71] C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. *Proceedings of the 11th annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.
- [72] M. Garey and D. Johnson. Computers and intractability: A guide to the theory of NP-Completeness. *W. H. Freeman & Co.*, 1979.

- [73] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
- [74] K. K. Agaram, S. W. Keckler, C. Lin, K. S. McKinley. Decomposing memory performance: data structures and phases. *Proceedings of the 5th International Symposium on Memory Management*, 2006.
- [75] C. Delimitrou, C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous data-centers. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages*, 2013.
- [76] R. Nathuji, C. Isci, E. Gorbato. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the Fourth International Conference on Autonomic Computing*, 2007.
- [77] L. He, Z. Yu, H. Jin. FractalMRC: online cache miss rate curve prediction on commodity systems. *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.
- [78] D. T. Chiou. Extending the reach of microprocessor: Column and Curious Caching. *PhD thesis*. Massachusetts Institute of Technology, 1999.
- [79] S. Cho, L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. *The 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.
- [80] D. Tam, R. Azimi, L. Soares, M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems & Computer Architecture (WIOSCA)*, 2007.
- [81] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating System Review*, 2002. 36:181-194.
- [82] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 1982, vol. 24, 197-208.

- [83] Z. Galil and N. Megiddo. A fast selection algorithm and the problem of optimum distribution of effort. *Journal of ACM*, 1979, vol. 26, 58-64.
- [84] O. Gross. A class of discrete type minimization problems. *Rand Corporation*, 1956.
- [85] N. Katoh, T. Ibaraki and H. Mine, A polynomial time algorithm for the resource allocation problem with a convex objective function. *Journal of Operational Research Society*, 1979, vol. 30, 449-455.
- [86] B. O. Koopman. The optimum distribution of effort. *Operations Research*, 1953, vol. 1, 52-63.
- [87] W. Shih. A new application of incremental analysis in resource allocations. *Operational Research Quarterly*, 1974, vol. 25, 587-597.
- [88] H. Ohtera and S. Yamada. Optimal allocation & control problems for software testing resources. *IEEE Transaction on Reliability*, 1990, 39(2), 171-176.
- [89] W. Wang, B. Li, B. Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. *Proceedings of IEEE INFOCOM*, 2014.
- [90] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch. Heterogeneity and dynamics of clouds at scale: Google trace analysis. *ACM Symposium on Cloud Computing*, 2012.
- [91] S. J. Tang, B. S. Lee, B. S. He. Dynamic job ordering and slot configurations for MapReduce workloads. *IEEE Transactions on Services Computing*, 2015.
- [92] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Operating System Review*, 2003, 37(5):164-177.
- [93] D. Tam. Operating system management of shared caches on multicore processors. *PhD Thesis*, University of Toronto, 2010.
- [94] H. Sasaki, T. Tanimoto, K. Inoue, H. Nakamura. Scalability-based manycore partitioning. *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

- [95] A. Fedorova. Operating system scheduling for chip multithreaded processors. *PhD Thesis*. Harvard University. 2006.
- [96] M. Lin, A. Wierman, L.L.H. Andrew, E. Thereska. Dynamic right-sizing for power-proportional data centers. *Proceedings of IEEE INFOCOM*, 2011.