DYNAMIC FRACTIONAL RESOURCE SCHEDULING

FOR CLUSTER PLATFORMS

A DISSERTATION SUBMITTED TO THE
GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

DECEMBER 2010

By

Mark Lee Stillwell

Dissertation Committee:
Henri Casanova, Chairperson
Edoardo Biagioni
Philip Johnson
Stephen Itoga
Galen Sasaki

This work is dedicated to my parents, my wife, and my son.

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Henri Casanova, for his supervision, his professionalism, and his patience. His comments and feedback throughout the process of developing my research project and writing this dissertation were both instrumental and essential.

I would like to acknowledge the contributions of my my research collaborators: Thanks go to Dr. Frédéric Vivien for his insights into the theoretical problem and his suggestions on how to properly develop and test the heuristic algorithms. Thanks go to David Schanzenbach for his programming work and his help with the Xen hypervisor and Usher virtual machine management system.

I also owe a debt of gratitude to the remaining members of my dissertation committee, Professors Philip Johnson, Edo Biagioni, Stephen Itoga, and Galen Sasaki for their time, encouragement, and efforts, and to the Department of Information & Computer Sciences at the University of Hawai'i at Mānoa for its academic and financial support.

I would like to separately thank my wife, without whom none of this would be possible.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

The experiments using our prototype system were carried out on machines owned by the Deparment of Computer Science and Engineering at the University of California, San Diego with the permission of Geoffrey Voelker. System administration and much assistance were provided by Gjergji Zyba.

ABSTRACT

This research focuses on the problem of job scheduling on homogeneous computational clusters. Clusters are widely used today for a variety of purposes, including high-performance scientific computing and Internet service hosting. While clusters may have impressive aggregate performance metrics, they are really only collections of fairly modest machines, which makes scheduling jobs for the best performance a non-trivial problem. Most clusters also need to be shared among users to amortize their start-up and maintenance costs, and ensuring that these users are treated fairly further adds to the difficulty. Existing approaches to scheduling attempt to address both of these issues, but have several limitations.

We propose a novel approach, called Dynamic Fractional Resource Scheduling (DFRS), to sharing homogeneous cluster computing platforms among competing jobs. The key features of DFRS are that it leverages existing virtual machine technology in order to share resources more efficiently and it defines and optimizes a user-centric metric that captures notions of both performance and fairness. In this dissertation we explain the principles behind DFRS and its advantages over the current state of the art, develop a theoretical model of resource sharing, design heuristics to optimize the proposed metric within the given framework, implement and run simulations comparing DFRS to traditional approaches using popular and accepted performance metrics, and finally develop and test a prototype implementation based on existing technologies. Our results show that it is possible to develop heuristic algorithms that give results reasonably close to theoretical bounds for a variety of cases, that resource requirements are well within the capabilities of modern systems, and that for some scenarios DFRS can provide orders-of-magnitude levels of improvement in performance over current approaches.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

A *cluster* is a near-homogeneous collection of computers, or *nodes*, connected to each other by a high-speed interconnect. Many clusters are *commodity clusters* built from off-the-shelf components intended for the personal computer market. Commodity cluster nodes are generally re-purposed desktop computers. Other clusters use specialty rack-mounted hardware, custom compute nodes [1], or cluster-specific interconnects such as InfiniBand or Myrinet [2]. Throughout this work we use the terms cluster and commodity cluster interchangeably.

Clusters are widely used today. Since their advent in the mid 1990s they have largely replaced traditional proprietary supercomputers for scientific and high-performance computing applications, and now represent over 80% of the machines listed on the top 500 index [3]. There is a great deal of demand for cluster computing in the sciences, with medical imaging [4], climate simulations [5] and protein folding simulation [6] being only a few of the better-known potential applications. Clusters also have uses beyond those traditionally reserved for supercomputers, including graphics rendering [7] and data processing [8–10]. Their distributed nature makes them ideal for providing load-balancing and redundancy in the event of failure, which is why they are also used for service hosting [11, 12].

Several factors have driven increased cluster adoption. The first is that they offer a major improvement in price performance ratio over traditional proprietary solutions [2]. Additionally, the more recent development of cluster-as-a-service and Grid technologies has given companies like Amazon the means and motivation to make cluster computing available even to casual users [12]. Compared to single-system approaches they are very configurable at time of purchase and they are easy to upgrade by purchasing additional nodes. Repairs and maintenance are relatively inexpensive. The use of commodity hardware means that clusters take advantage of the research and development cycle of desktop computing systems, and so node and network performance both improve rapidly over time [2].

While clusters generally provide an improvement in cost over traditional supercomputers [2], they still represent significant investments in infrastructure and maintenance. Though it is possible to put together a usable system from standard PCs and networking equipment (the so called "LOBOS" or Lots-Of-Boxes-On-Shelves approach), higher-end systems will use specialty rack-mounted hardware and high-speed interconnects [2]. Sizeable systems can have enormous power requirements and often require dedicated cooling [13, 14]. Properly maintaining the system hardware and software may require dedicated staff, as with any large system installation. For these reasons, sharing clusters among as many users as possible is necessary for both spreading costs and justifying said costs by maintaining high levels of utilization. Thus, clusters are typically expected to run multiple instances of user applications, or *jobs*, in sequence and, when possible, in parallel. Each of these jobs has its own individual resource requirements and the assignment of jobs to nodes in space and/or time in such a way that these various requirements can be met is referred to as *job scheduling*.

Several factors make scheduling on clusters difficult. While generally compared to traditional supercomputing systems using aggregate measures of computational power such as Giga- or Tera-Flops (floating-point operations per second), clusters are really only collections of fairly modest machines. Large computations and high-throughput operations must be broken down into smaller pieces, called *tasks*, which are then scheduled to be run on individual nodes. High inter-node communication latencies and the lack of a globally accessible memory address space mean that approaches targeting shared memory supercomputers are usually inappropriate for use on clusters [2].

Current solutions are unsatisfying for a number of reasons. The most-commonly used schedulers at production high-performance and scientific computing installations do not seek to maximize an objective metric based on performance or fairness, and it is well known that a there is a disconnect between the desires of users and schedules achieved in practice [15, 16]. Furthermore, the most sophisticated algorithms rely on unreliable end-user generated estimates of program runtime [17, 18]. In the service hosting arena, much of the existing work shows an engineering-heavy bias, with little theoretical justification [12, 19–22].

1.1 Problem Statement

We assume the existence of a cluster with a fixed number of homogeneous nodes and a set of jobs that need to be run on the cluster. Each node supplies a number of resources (CPU, memory, network bandwidth, etc.), and each task of a job, if allowed to run unimpeded on a node, will tend to saturate a fraction (possibly 0%, 100%, or somewhere in between) of each resource. Depending on the scenario, jobs may be submitted all together at once or individually over time. We study scheduling algorithms that take as input a number of nodes and a set of jobs, and output a schedule that allocates tasks to nodes, possibly over time. The problem is to develop scheduling algorithms that perform well by aggregate metrics of user and/or administrator concerns, like average job performance or schedule fairness, in both the average and worst case.

1.2 Approach

We address the cluster scheduling problem and extend the theoretical literature by designing a new model of resource sharing and developing algorithms based on it. Our goal is to create a well-defined optimization problem and then find heuristics to solve it. This is consistent with a number of lines of existing research [23–27], but our work possesses several key advantages: For one thing, we seek an approach that can account for an arbitrary number of resource dimensions instead of only one or two. For another, we define and optimize a universally applicable metric that captures notions of performance and fairness across different application domains.

We consider several different scenarios, including an off-line resource allocation problem for stable workloads of long-running jobs and an on-line resource allocation problem where temporary jobs are submitted by users at arbitrary times. We also examine some of the resource costs associated with our approach, particularly the bandwidth consumed by migrating tasks between nodes. Finally, we develop and evaluate a prototype system to show that our model of resource sharing is feasible in practice.

1.3 Organization

This dissertation is organized as follows: In Chapter 2 we discuss existing approaches to the job scheduling problem and their shortcomings. In Chapter 3 we develop a new model of resource sharing and formulate several versions of the Dynamic Fractional Resource Scheduling problem. In Chapter 4 we study the off-line problem in detail, establish a theoretical model and complexity bounds, propose a number of algorithms and perform simulation experiments to determine their relative performance under a number of different scenarios. In Chapter 5 we study the on-line problem, establish complexity results and a method for computing the maximum stretch lower bound for an instance using a clairvoyant algorithm in an idealized setting, establish a proof of the competitive ratio for the non-clairvoyant case, propose a number of algorithms and evaluate them in simulation using a combination of real workloads and synthetic workloads based on a well established model. In Chapter 6 we study the trade-offs between performance and network bandwidth consumption. In Chapter 7 we describe a prototype implementation of our system and give some results for simple validation experiments of a technique for the on-line estimation of dynamic resource needs. Finally, in Chapter 8 we summarize our results and the research contribution of this dissertation.

CHAPTER 2
CURRENT APPROACHES AND THEIR LIMITATIONS

For the reasons outlined in Chapter 1, cluster scheduling has been an active area of research for as long as there have been clusters. Since most clusters have traditionally been used for scientific and high performance computing, the existing literature tends to look at the problem from this perspective. In the past few years there has been more interest in using clusters for service hosting and data processing applications, however much of the initial infrastructure was designed by industry with little concern for theory. More recent papers have begun to address this lack of academic rigor, and service hosting has become something of a hot topic for research.

In this chapter we discuss the cluster scheduling literature, beginning with a review of approaches to scheduling for high performance computing (Section 2.1) and continuing with a look at proposed methods of dynamically allocating resources in service hosting environments (Section 2.2).

2.1 Scientific and HPC Workloads

Scientific and HPC workloads are composed of jobs created independently by end users in response to their own individual needs. These jobs usually involve large computations and may require access to a significant fraction of the available system resources. They are generally deterministic in nature, requiring minimal interaction and communication with outside systems, and complete after a finite period of time.

2.1.1 Batch Scheduling

The most commonly used cluster sharing technique for scientific or high-performance computing workloads is *batch scheduling*. In this paradigm users create requests to run programs using a number of nodes, and these requests are then put into queues. At some future time when appropriate resources are available the request can be removed from the queue and used to create a job which is then given exclusive access to a set of nodes until it either completes or is canceled by the user or the system [28]. In the discussion that follows we equate requests to run jobs with the jobs themselves and refer only to jobs.

The most obvious way to select jobs from the queue is on a first-come-first-served basis; that is, in the order that they are submitted. However, this can lead to unnecessarily long wait times: If a job at the head of the queue is blocked because there are insufficient nodes available to run it, then other jobs may also have to wait even when there are sufficient nodes available to run them. For this reason most batch schedulers use a first-come-first-served strategy augmented with some sort of *backfilling* algorithm that selects runnable tasks out of turn from the waiting queue [29].

Unrestricted backfilling can result in jobs that require a small number of nodes constantly jumping to the head of the queue, which can cause unbounded wait times for jobs requiring a larger number of nodes, a phenomenon referred to as *starvation*. Conservative backfilling techniques alleviate this problem by requiring the user to specify an estimated job run time and then scheduling jobs so that no job should be delayed from when it would run if the scheduler became a pure first-come-first-served scheduler at the time the job is submitted. Smaller jobs with relatively short run times may thus be allowed to start early without delaying larger jobs. Aggressive, or *EASY*, backfilling is more widely used than conservative backfilling and represents a compromise of sorts, as a reservation is only made for the first job in the queue [17, 29], which both simplifies the scheduler and allows for more backfilling.

Unfortunately, it is well known that the user runtime estimates required for backfilling are highly unreliable [18, 30]. Studies have also shown that this inaccuracy can have a significant impact on system performance [31]. Additionally, many systems that require runtime estimates will kill jobs that exceed these estimates by a certain threshold. This can compound the problem by encouraging users to make very conservative runtime estimates in order to ensure that their jobs are not killed after they start. Thus, it is generally believed that a system that did not require these types of runtime estimates would be desirable.

The integral (i.e., whole-node) allocations used by batch scheduling virtually guarantee poor utilization at both the node and individual resource level. At the node level, since job sizes in terms of nodes and run times are essentially arbitrary, the job scheduling problem is equivalent to bin-packing. This suggests that at any given moment some fraction of the usable nodes will be sitting idle, even when jobs are waiting to run in the queue. In fact, studies have shown that FCFS only manages to achieve around 40-60% node utilization, while EASY does somewhat better at around 70% node utilization [32]. The resource level problem comes from the fact that jobs receive exclusive access to nodes for their duration, and so any unused or underutilized resources will remain at least partially idle until the current job completes. Since nodes need to be provided with sufficient resources of each type to run the most demanding job, even on active nodes it is likely that some of the available resources will not be used. For example, in a 2006 log of a large Linux cluster [33], more than 95% of the jobs used under 40% of the available node memory, and more than 27% of the jobs effectively used less than 50% of the available node CPU resources (due to time spent performing I/O or network communication and synchronization). Similar observations have been made repeatedly in the literature [34–39].

2.1.2 Gang Scheduling

In light of the observations from Section 2.1.1, it seems likely that some form of time-sharing is necessary in order to achieve higher utilization levels and responsiveness in parallel computing [40]. Time-sharing is particularly useful for interactive applications as it allows the system to run more jobs simultaneously, and thus reduces the wait time between when a job is submitted and when it starts running (also called the *response time*). Intelligent scheduling policies can also make use of time-sharing to improve overall resource utilization, for example by scheduling a CPU intensive job and an I/O intensive job to the same node [41].

There is, however, a potential problem with using time-sharing on the nodes of parallel clusters: When a large job is allocated to nodes that use locally scheduled time-sharing the individual tasks may all start to run during different time-slices, or they may start out together and become out of sync due to clock-drift or the interference of other jobs on the system. If this occurs and the job requires fine-grained or interactive communication between tasks on different nodes then the progress of the job may grind nearly to a halt as tasks spend most of their allocated time-slices waiting for messages from other tasks that are sleeping. This is called *process thrashing* and the most straightforward way to ensure that it is avoided is to *co-schedule* the individual tasks of a parallel job in a coordinated manner so that they run at the same time [42].

Explicit co-scheduling, or *Gang scheduling*, is a paradigm in which the entire parallel machine is multiplexed via distributed context switching so that processes within the same job are run within the same time slices. Gang scheduling has inspired a considerable amount of academic research for a number of years [43–47]. While there have been several promising implementations of the concept [48–50], gang scheduling remains very much a curiosity in HPC circles and is not generally found in production environments (a notable exception being the CM5 [35], no longer in use). This is largely because the promised benefits have been difficult to realize [46].

One problem with gang scheduling is that it requires *coordinated* distributed context switching. This is high-overhead in terms of complexity and delay, and thus imposes the additional consideration that context switching should be performed as infrequently as possible. This limitation on the granularity of the time-slicing combined with the nature of the switch, wherein the entire parallel system is effectively replicated several times over, means that gang scheduling cannot solve the fundamental problems associated with batch scheduling. That is, while gang scheduling is likely to improve overall system utilization, it is still possible to have “holes” in the final schedule, so that some machines may sit idle for entire time slots even when there are jobs that need to be run [51]. Additionally, gang-scheduling can provide poor levels of resource sharing even given a perfect schedule that contains no holes: consider that if there are N time slots then an I/O bound job will receive $1/N$th of the CPU, even if its pattern of activity leaves the CPU idle for much of that time [52]. Thus, while gang scheduling may represent an improvement over batch scheduling, it still suffers from the problems discussed in the previous section.

Another problem with gang scheduling is simply that the original models did not take into account the memory needs of running jobs when packing them into time slots [36]. Because of this some jobs may be forced to swap to disk, and such delays are compounded in parallel applications by process swapping as previously described. Research indicates that most jobs leave enough system memory free to allow some degree of node sharing between jobs [35, 36, 38, 39, 53], and when memory needs are taken into account the real performance improvements of gang scheduling over pure space-sharing approaches can be significant [36, 37, 54].

2.1.3 Co-Scheduling

The problems of overhead and fragmentation have led to interest in more relaxed time-sharing schemes that still manage to avoid process thrashing. These more flexible techniques are generally referred to as co-scheduling, though it should be noted that true co-scheduling requires a globally synchronized clock [42], and so most of these techniques are better thought of as relaxed or approximate co-scheduling. Flexible co-scheduling monitors network activity in order to classify processes and then only uses gang-scheduling for those tasks that seem to require it [46, 55]. Implicit co-scheduling is based on the idea of two-phase waiting: processes which are blocking for communication events spin in place for a period of time, usually a little over the time required for two context-switches, before finally blocking and yielding the processor [56, 57]. Dynamic co-scheduling boosts the priority level of processes receiving communications, on the theory that this will keep parallel applications loaded across multiple nodes during periods of intense interactive communication [58]. Buffered co-scheduling buffers non-blocking communication events and waits to send messages until a globally coordinated synchronization event [59]. Coordinated co-scheduling is a more advanced technique that takes into account both sender and receiver side events when making local scheduling decisions [60]. Some authors even use the term co-scheduling to denote completely uncoordinated time-sharing on nodes [61].

What these different co-scheduling techniques have in common is that they attempt to minimize the amount of globally synchronized state that must be maintained and make decisions locally in order to make better use of local node resources. Experiments using both simulations and real systems seem to show that while some programs may suffer slightly from imperfect task synchronization [62], implicit co-scheduling frequently provides better over-all performance than explicit co-scheduling, possibly because of lower overheads and better system utilization [46, 63, 64].

2.1.4 Discussion

We contend that some form of approximate co-scheduling is probably the most efficient mechanism for sharing node resources on scientific workloads. However, the research on this topic has focused largely on mechanisms and low-level implementation details. Decisions about how to place jobs, and particularly which tasks should share nodes, have been based on fairly simple criteria, such as memory capacity [63] or arbitrary restrictions on the multiprogramming level [46, 65]. Many papers even make the expedient assumption that all parallel jobs run on all the available nodes [56, 57, 65–67]. At least one study has shown that proper task placement is important in a cluster that uses time sharing nodes under a number of different co-scheduling schemes [68], but this work focuses on the performance of running jobs and does not suggest algorithms or address overall schedule quality in an on-line setting. The focus of our research is this neglected, but important, problem of task placement and sub-node-level resource allocation.

The use of time-sharing and/or virtual machine technology to create "virtual clusters" has been explored previously in the literature [69–74]. Again, the focus here has been on lower level details, such as selecting appropriate host nodes in distributed and/or federated systems [75], properly distributing and instantiating virtual machine images [76], and allowing for check-pointing parallel jobs [69, 77].

The scheduling algorithms considered have been for the most part refinements or extensions to existing schemes, such as combining best-effort and reservation based jobs [78]. Fallenbeck et. al. developed a novel solution that allows two virtual clusters to coexist on the same physical cluster, with each physical node mapped to two virtual nodes, but their implementation is meant to solve a problem specific to a particular site and does not add to the literature in terms of general scheduling algorithms [79]. The system proposed by Ruth et. al. attempts to dynamically migrate tasks from "overloaded" hosts, but their definition of overloaded is vague and they do not propose a well-defined objective function [80].

One interesting use of time-sharing on cluster nodes suggested by Kalé would be to allow for limited malleability by consolidating multiple tasks from the same job, and only tasks from the same job, onto a single physical node when necessary to achieve global performance objectives [81]. A similar idea was espoused by Feldmann [82]. The model that we propose is much more ambitious and gains the purported advantages as a subset.

The VSched component of the Virtuoso system [83] can allow for distributed co-scheduling of serial and parallel jobs as we propose to do. However, VSched uses a fairly rough-grained time-sharing scheme and requires users or administrators to explicitly specify the amount of CPU time a given application should be assigned per period. It then uses a simple admission control mechanism to determine which applications can run. Our work focuses on automating such decisions by assigning applications a fair share of the resources they are capable of using through the optimization of a well defined objective function.

An additional problem with the scheduling paradigms proposed so far is that they do not optimize a user-centric objective function; it is known that there is a sharp disconnect between user concerns (low job turn-around time, fairness) and the schedules implemented in practice [15, 16]. Batch schedulers provide myriad configuration parameters by which cluster administrators can influence scheduling behaviors, but these parameters are not directly related to user-centric measures of performance.

2.2 Service Hosting Workloads

In contrast to the HPC scheduling paradigm, in which jobs exist for a finite period of time and may require significant inter-node communication over their lifespan, jobs in the service hosting environment tend to be long-lived and require little inter-node communication. The primary goals in this environment are to ensure performance isolation and keep the system in compliance with service level agreements (SLAs) [19, 84]. This latter concern can be construed to encompass both maintaining a particular minimum performance profile on active nodes and providing redundancy in the event of node-level hardware failure. Some works attempt to address issues of task-interdependencies and organize services into different levels, or *tiers*, with the tasks on one level being dependent on the output of those below [85]. We opt for a more abstract model in order to focus purely on the resource-allocation problem and so do not consider these issues.

Commercial products such as Amazon's Elastic Compute Cloud (EC2) [12] or Linode [86] give users access to statically allocated resources on shared machines (EC2 allows users to dynamically adjust their resource shares through an API, but provides no global mechanism for finding optimal allocations). Early deployments of these types of system were largely done "in the field" and the existing academic literature often reflects this, with many papers having a strong focus on experiments using benchmarks and real machines, but providing few theoretical models [19–22]. Some recent work, such as [87], has done much to correct this oversight.

The Océano project allows administrators to specify both statically and dynamically allocated nodes in order to allow for overbooking of resources while maintaining "flexible" SLAs [88]. Dynamically allocated nodes are maintained as blanks or templates and then instantiated as needed; when demand drops again they can be flushed and put back into the pool of unallocated nodes. This policy is reactive and works on the node-level, in contrast to more recent approaches which frequently make use of predictive models and usually allocate sub-node-level resources, such as CPU or memory.

The authors of [89] propose to place applications onto a shared cluster using intelligent "overbooking," i.e., sharing servers among application instances. An important contribution of this work is the validation of a number of application profiling techniques for obtaining statistical bounds on resource usage and minimum resource needs. Both this work and ours account for application QoS requirements, but the approach in [89] attempts to maximize resource provider revenue while we choose to focus on application performance and fairness in terms of resource shares. We also seek to develop intelligent algorithms inspired by the theoretical literature that go beyond the common-sense but simple heuristics described in [89].

In [23] the resource allocation problem is formulated as a constrained optimization problem, where each service is characterized by its desired resource share. One difference with our work is that we explicitly consider multiple resource dimensions while the authors of [23] consider a server as a monolithic resource. Also, the approach described in [23] first optimizes a linear objective function, namely, the average deviation between a service's resource share and its desired resource share. This can lead to unfair schedules and so the authors of [23] propose a second optimization step, this time with a quadratic objective function, that includes a bias term to improve fairness. By contrast, we propose to use a linear objective function that naturally captures fairness, as inspired by the theoretical job scheduling literature [90, 91].

The approach in [84] allocates applications to nodes based on historic data from a distributed set of "sensors" and then dynamically migrates applications based on feedback from those same sensors. A primary goal is to generate "robust" allocations in order to minimize the number of migrations. Toward this goal a number of different heuristics were proposed and evaluated, including both simple greedy approaches as well as genetic algorithms and simulated annealing. A key difference between this work and our own is that we attempt to find algorithms that maximize a linear objective function of performance, while the approach described in [84] is only concerned with finding valid resource allocations that satisfy minimum QoS requirements.

Several works propose resource allocation schemes that attempt to maximize user-specified or empirically derived utility functions [92–96]. In this work we do not consider utilities, but, instead, consider solely the resource share allocated to a service as the sole performance metric. This metric is generic, correlated to popular metrics of performance such as service response time, directly allows the specification of minimum resource shares, and allows us to formalize and provide algorithmic solutions to the resource allocation problem.

2.3 Conclusion

In this chapter we reviewed the cluster scheduling literature. We first explored the literature on scheduling for HPC workloads, including well established techniques such as Batch and Gang scheduling. We also discussed the literature on co-scheduling, and concluded that efficient algorithms for resource allocation in a co-scheduling environment are likely to lead to the best performance. We next explored the literature on scheduling for service hosting environments and discussed the need to formulate the resource allocation problem as an optimization problem with a well-defined objective function. We contend that such a formulation could be used to develop resource allocation algorithms that would lead to good performance for both HPC and service hosting applications.

CHAPTER 3
DYNAMIC FRACTIONAL RESOURCE SCHEDULING

In this dissertation we propose to leverage recent technological advances to reformulate and re-conceive the problem of scheduling user jobs on clusters. We call our approach *Dynamic Fractional Resource Scheduling* (DFRS) and contend that, within this new framework, it is possible to define a performance metric that is universally applicable to different types of workloads. That is, algorithms that do well at optimizing for this metric will also do well when when evaluated using metrics used traditionally in diverse contexts, namely, high-performance computing and service hosting. We explicitly seek to find polynomial-time algorithms that achieve good average case performance, which separates our work from approaches that require the solution of NP-complete constraint satisfaction problems [93, 94, 97–99].

This chapter provides an introduction to the DFRS approach. In Section 3.1 we give explanations of our basic assumptions and our chosen optimization objective function and also provide descriptions of three different versions of the problem that we will investigate further in future chapters. In Section 3.2 we justify our assumptions by explaining how enabling technologies can be applied in real world applications.

3.1 The DFRS Approach

3.1.1 System Overview

We consider a cluster of homogeneous machines, or *nodes*, to which users or the administrator submit requests to run *jobs*. These jobs consist of one or more *tasks*, where a task is a more or less independent instance of a program. From an administrator standpoint, the only distinction between a set of tasks from the same job and a set of independent tasks from different jobs is that the first set is generated by a single user request. We propose to run these tasks within independent virtual machines to allow greater flexibility and control by the administrator without requiring modifications to the software developed and run by users.

Resources, such as memory or disk space, that cannot be effectively multiplexed by some sort of time-sharing scheme are referred to as *fixed*, while those, such as CPU, network, or disk I/O bandwidth, that can be are referred to as *fluid* (the corresponding terminology in [25] is “load-independent” and “load-dependent”, while [89] uses the terms “spacial” and “temporal”). For each fixed resource a task has a *requirement* in that resource equal to a certain fraction of what is available on a node. A task cannot run when allocated less than this amount of the given resource. For each fluid resource a task has a *need* in that resource equal to the fraction of the time the task would keep that resource busy if it were running alone on the node. That is, a task’s need in a fluid resource is equal to its maximum rate of consumption of that resource and is thus the amount of the resource the task would need to be allocated in order to avoid experiencing degraded performance. For example, a task might have a fixed resource requirement of 50% of the memory available on the system and a fluid resource need of 40% of the CPU. As tasks within the same job are all instances of the same program and operate on similar amounts of data, they should have similar fixed resource requirements and fluid resource needs. Since the tasks of a job are generally instances of the same program working toward a common goal on similar amounts of data, we assume that they all have the same resource requirements.

When multiple tasks are running on the same node their fixed resource requirements and fluid resource needs will stack by simple addition. That is, if two tasks are running on one node and the first task uses 25% of the CPU and 37% of the memory, while the second uses 50% of the CPU and 40% of the memory, then the two tasks together use 75% of the CPU and 77% of the memory, leaving 25% of the CPU and 23% of the memory free to run additional tasks. While the total fixed resource requirements of the tasks on a given node can never exceed 100%, the fluid resource needs are not under this constraint. Instead, when tasks are placed together on a node so that the total of one or more of their fluid resource needs exceeds 100% the performance of some of those tasks must be degraded so that the actual utilization levels of the given resources are within the node's capacity.

To address this issue, for each job we define a value between 0 and 1 called *yield* of the job. Note that different jobs may have different yield values. The fluid resource needs of the tasks of a job are all scaled by the job yield, and for an allocation of tasks to a node to be valid, in addition to satisfying the constraint on the fixed resources, the total of the yield-scaled fluid resource needs in every dimension must also be less than or equal to 100%. Returning to the example from the previous paragraph, if a third task is added to the node with needs and requirements of 75% of the CPU and 20% of the memory, then the three tasks together use 97% of the system memory, which is allowed, but would require 150% of the CPU to run without degraded performance, which is impossible. The corresponding jobs of the tasks can be assigned yield values of 0.6, resulting in an actual CPU utilization of $150\% \times 0.6 = 90\%$. Restricting access to resources uniformly in this manner is equivalent to slowing the internal clock of the job by the same factor [100], and so provides a simple model of job performance degradation. This model is reasonable for self-contained jobs and single-tier services that do not make explicit use of the system clock. For example, consider a simple web server application. Some percentage of the CPU is used to process requests at a given rate. If CPU consumption is throttled down to, say, half of this percentage, then requests are processed at half the original rate. As a result, consumption rates for other resources, such as network bandwidth, are also halved. We leave a discussion of multi-tier services for Chapter 4 and deeper exploration of this topic outside the scope of this dissertation. While it is likely that some tasks could benefit from nonuniform resource throttling levels (e.g., an application that alternates between computation and network I/O, but sends the same 10MB file every 5 minutes, regardless of the rate of progress through its computations), such flexibility would add a great deal of complexity to the model for uncertain benefit.

Note that for the sake of simplicity we have assumed that fixed resource requirements and fluid resource needs are independent. This is not always the case for certain applications. For example, in Doyle et. al. [19] the authors consider the case of a web server that uses a RAM cache. With a larger fraction of the RAM space of a server, the RAM cache has a larger hit rate, leading to reduced CPU and I/O activities. It is thus possible to trade off CPU and I/O bandwidth for RAM space. In this work we do not model such inter-dependencies between resource needs.

3.1.2 Optimization Objective Function

We have stated that we model job scheduling as an optimization problem with a well defined objective function. More specifically, for any particular scheduling event, assuming that all of the jobs under consideration can be supported by the system, we choose to optimize a metric based on the yield as defined in the Section 3.1.1.

We note that resource management objectives are generally expressed based on metrics related to user concerns, such as service response time or throughput. This work relies on the fact that these higher level metrics are directly related to resource fractions allocated to jobs, and thus to the yield. This observation has been made repeatedly in the literature and several models that link resource shares to response time and or throughput have recently been developed [101–108]. For instance, in [103], the authors model response time as a function of CPU and memory resource fractions allocated to services. These models are validated for real-world services (Tomcat, MySQL). In [104] a model of response time as a function of allocated CPU fraction is developed for a Web application benchmark (RUBBoS), using linear interpolation. Similar models for response time and for throughput are proposed and validated in [105] for two multi-tier application benchmarks (RUBiS and TPC-W). A response time model for TPC-W is also proposed in [106], while [102] proposes a response time model for several synthetic applications. We conclude that metrics based directly on resource shares, in our case the yield, are reasonable stand-ins for metrics based directly on user performance concerns.

Since we wish to measure overall schedule quality, it is essential to aggregate the yield metrics of individual jobs into a well-defined objective function. It is well known that optimizing functions based on average performance (e.g., average stretch) is prone to unfairness and starvation [91]. Instead, we choose to focus on algorithms for maximizing the minimum yield. Maximizing the minimum helps to improve the overall average performance, while still respecting the importance of fairness by making sure that the "least happy" job is as happy as possible.

3.1.3 Off-line, On-line, and Adaptation Scenarios

In this dissertation we will explore three different versions of the Dynamic Fractional Resource Scheduling problem.

First, we consider an **off-line** resource allocation problem for serial jobs. In this version of the problem we assume that all of the jobs under consideration are static and eternal, that their requirements and needs are known, and that they are to be scheduled to run at the same time. Thus, this version of the problem is concerned only with the allocation of system resources to individual tasks, and does not consider the evolution of jobs over time. It is broadly applicable to any environment for which the workload is reasonably stable over a large portion of the scheduling epoch. For example, a service hosting environment with a mostly constant load. This version of the problem may not provide a realistic model for many real-world systems, but it provides a reasonable basis upon which to build more complicated, but relevant, scenarios.

Next, we look at an **on-line** scheduling problem that allows for temporary, parallel jobs. We primarily study situations where job resource requirements are fixed throughout a job's runtime. Note that since we allow for the preemption and migration of jobs this also covers situations in which job resource needs are allowed to change: whenever a resource need changes we simply say that the job has completed and a new job with different resource needs has been submitted. This version of the problem is appropriate to both HPC batch scheduling environments and service hosting environments where jobs may be parallel or have fluctuating resource needs.

Finally, in response to the problem of excessive migration/preemption costs we study an **adaptation** version of the problem. This version of the problem is similar to the off-line version, but we assume that a mapping already exists between some of the tasks and the cluster nodes, and the goal is to find a mapping that optimizes the objective function without exceeding some bound on the total cost of migration. The adaptation problem then can serve as the basis of a more refined version of the on-line problem with a bounded rate of resource consumption for migration and preemption.

3.2 Practical Considerations

In our overview of a system for Dynamic Fractional Resource Scheduling, we have made several assumptions about the capabilities of that system that are not found in the traditional job scheduling literature. In particular we have assumed that it is possible to time-share fluid resources, that it is possible to externally enforce bounds on rates of resource consumption, that it is possible to preempt jobs and to migrate tasks, and that job resource requirements and needs are known. In this section we attempt to justify these assumptions based on recent advances in virtual machine technology and a brief survey of the literature on the discovery of job resource needs.

3.2.1 Virtual Machine Technology

Though it has existed since the 1960s, virtual machine technology had long been thought of as inefficient and slow, and so was not traditionally considered as an option for performance sensitive production workloads [109]. Since the mid-1990s huge strides made in this area, and implementations such as Xen can allow a single server to run dozens of virtual machines with minimal overhead [110, 111]. In fact, studies even suggest that thousands of virtual machines can be simultaneously launched across a cluster with little difficulty [112]. Consequently, there has been a movement toward server consolidation in the data center to minimize idle resources and their associated costs [113]. This increased adoption has led to further advances, such as wide-spread hardware support for virtualization [114, 115].

Unsurprisingly, there has also been a movement in the HPC community, with similar goals of consolidation, increasing utilization, platform standardization, reductions in administrative overhead, and reducing energy consumption [53, 116, 117]. Another oft-cited use for virtual machines is increasing availability and reliability [77, 118]. Concerns about the potential performance impacts have slowed HPC adoption; However, recent studies indicate that the overheads and penalties associated with applying this technology to HPC workloads are manageable [116, 119–125]. Proper handling of the memory cache hierarchy and I/O and scaling across large numbers of cores are still potential issues [113, 126–129], but there are already groups working to resolve them [121, 130–135].

For our purposes, the most important capability provided by virtualization is the ability to pool and redistribute multiple discrete time-shared resources. For example, the Xen Credit CPU scheduler can allow 3 serial jobs to each receive 1/3 the total CPU capacity of a dual-core system [136]. Further, studies have shown that VM monitors can enforce constraints on resource utilization accurately and reliably in the presence of competing processes [137].

It is well established within the job scheduling literature that preemption can be used to improve both the average and worst-case performance of running jobs [40, 138, 139]. Virtual machine technology makes it fairly simple to save the state of a single VM to disk and stop execution, and then resume at a later date. Temporarily suspending a parallel job that consists of tasks executing on multiple hosts is considerably more difficult, but systems capable of doing so have already been developed [69, 77, 98].

Another capability offered by virtual machine technology that we can take advantage of is the migration of processes between nodes [140]. Migration has been shown to lead to significant performance improvements under gang scheduling [51, 141, 142], and it seems reasonable to assume that this would carry over to time-sharing schemes based on virtualization. Additionally, migration allows for better adaptation to unforeseen events, which can be important for on-line algorithms. Another potential use for migration is to free up space on an already loaded machine for high-priority tasks [143]. Modern virtual machine platforms even allow for “live” migration of processes, meaning that jobs do not need to be completely stopped, minimizing slowdowns and allowing for the migration of interactive jobs, though at an additional cost in computation and communication [144].

Several groups in academia and the private sector are currently working on platforms for centralized control of virtualized resources in a distributed environment [23, 71, 73, 97, 99, 114, 145–150]. These platforms generally allow a central controller to create VMs, specify resource consumption levels, migrate VMs between nodes, suspend running instances to disk, and, when necessary, delete unruly instances. We base our model on such a system and the capabilities it offers. The Entropy system recently developed by Hermenier et. al. [97–99] in fact implements all of the basic system capabilities that we propose to exploit as well as its own set of resource allocation algorithms, but their approach is based on searching for solutions to an NP-complete constraint satisfaction problem, while our approach is to develop polynomial time heuristic algorithms for a well-defined optimization problem.

3.2.2 Discovery of Resource Requirements and Needs

We have given an overview of our problem framework wherein task resource requirements and needs are given as fractions of the resources available on a node. An important question is how to determine the actual values of these resource fractions. While it is possible, even likely, that for certain resources such as RAM or disk space these values could be specified by the user, it is unlikely that many users would be able to provide precise resource requirements and needs in general, particularly for dynamic resources such as the CPU. While there are no studies of user accuracy in estimating resource utilization, we base this assumption on studies that have shown that users are generally inaccurate when asked to estimate program runtime [18, 30, 151].

Another approach, used in [89], is to rely on benchmarking of the services. This is most reasonable for a shared hosting platform that hosts a moderate number of well-known services, each for long periods of time. Going beyond simple static benchmarking, it is also possible to build analytical models of resource needs and of their temporal trends [19, 20, 92, 101, 146, 152, 153], and even to augment these models to account for virtualization overheads [154].

A more direct approach is to monitor services as they run to determine their needs [22, 25, 95, 96, 102, 155]. Such VM instance resource usage monitoring can be based on facilities like XenMon [156]. Of course, this approach can also be used as an enhancement to approaches that use predictive models [22, 146, 153], and such models can be built on-line from runtime observations [146].

An even more advanced technique would be to perform discovery via a combination of introspection and configuration variation. With introspection, one can, for instance, deduce CPU needs by inferring process activity inside of a VM [157], and memory pressure by inferring memory page eviction activity [158]. With configuration variation one can vary the amount of resources given to VM instances, track how they respond to the addition or removal of resources, and infer resource needs [157, 158].

We conclude that the on-line discovery of resource requirements and needs is a challenging, but tractable, problem. When studying our algorithms in simulation we therefore assume that these values are known.

3.3 Conclusion

In this chapter we have given our model and objective function, explained the underlying assumptions and how they are reasonable given current technology, and discussed three different versions of the DFRS problem:

1. an off-line problem targeting service hosting environments with static workloads
2. an on-line problem targeting dynamic workloads as found in high-performance computing environments
3. an adaptation problem that bounds migration costs for workloads with evolving requirements

The following three chapters explore each of these problems in greater depth. In particular, we give a formalization of the off-line problem along with its proof of NP-completeness and show that the other two problems can be seen as straightforward extensions. For each version of the problem we develop and evaluate heuristic algorithms based on detailed simulation experiments. Finally, where appropriate, we compare our approach to the current state of the art.

CHAPTER 4
THE OFF-LINE PROBLEM

In this chapter we frame the off-line resource allocation problem for continually executing jobs in an idealized service hosting environment as a constrained optimization problem, for which efficient algorithms can be developed. We make the following contributions: (i) We define the resource allocation problem for a static workload of services that are each fully contained in a single VM instance; this definition accounts for multiple resource dimensions, supports a mix of best-effort and QoS scenarios, and promotes performance, fairness, and high resource utilization (Section 4.1); (ii) We establish the complexity of the problem and give a mixed integer linear program formulation that leads to a bound on the optimum (Section 4.1); (iii) We propose several algorithms to solve the problem (Section 4.2); (iv) We evaluate these algorithms in simulation (Sections 4.3 and 4.4); and (v) we discuss how our approach can be extended to services that comprise multiple VM instances (Section 4.5).

4.1 Problem Definition

We consider a set of nodes N and a set of serial jobs J. Since the jobs are serial, we equate them with their tasks. There are n_s fixed resources and n_d fluid resources and each job $j \in J$ can be denoted by a pair of vectors $(\vec{s_j}, \vec{d_j})$ such that $\vec{s_j} \in [0,1]^{n_s}$ and $\vec{d_j} \in [0,1]^{n_d}$. $\vec{s_j}$, the fixed resource requirement vector, represents the fractional amounts of each of the fixed resources that are required by the job to run, while $\vec{d_j}$, the fluid resource need vector, represents the peak rates at which the job would use the each of the fluid resources if run alone on the system as a fraction of the maximum rate of consumption for each resource. A solution is a number $\alpha \in [0,1]$ and a mapping

$f : J \rightarrow N$, and a valid solution is a pair that fulfills the following conditions:

$$\forall n \in N \quad \| \sum_{f(j)=n}^{J} \vec{s_j} \|_\infty \leq 1 \quad \text{and} \quad \forall n \in N \quad \alpha \times \| \sum_{f(j)=n}^{N} \vec{d_j} \|_\infty \leq 1 \, .$$

These equations simply state that the total amount of any resource used on any node cannot exceed 100% of what is available.

In general, the goal of the optimization problem is to find a valid solution that maximizes α, which corresponds to the minimum yield. The definition of the yield, its correlation to other metrics of performance, and the reasons for choosing the minimum as the objective target are given in Section 3.1.2. An additional concern is that some jobs may have Quality-of-Service requirements that constrain them to function with a minimum allowed yield value or suffer a failure condition. In this case, for each fluid resource a job will have a *constrained fluid need* representing the minimum amount of the fluid resource that the job could be allocated without suffering a failure condition; this value is equal to the product of the fluid need and the QoS requirement. For example, consider a job with a CPU need of 0.6 and a QoS constraint of 0.4: This job has a constrained fluid need (i.e., minimum allowed allocation) for the CPU resource equal to $0.6 \times 0.4 = 0.24$.

To compare yields across jobs with various minimum yield requirements, we define the *scaled yield* of a job as follows:

$$\text{scaled yield} = \frac{\text{yield} - \text{minimum allowed yield}}{1 - \text{minimum allowed yield}} \, .$$

Note that in the case that a job has no specified QoS constraint (i.e., the minimum allowed yield is 0) the scaled yield of a job or service reduces to the yield as defined normally.

Consider two jobs with minimum yield requirements of 0.2 and 0.4, respectively. Then a yield of 0.8 for the first job and of 0.85 for the second achieve an identical scaled yield of 0.75 for both jobs. That is, each job experiences a yield that is 75% of the way between its minimum and maximum yield values. For a best-effort job the minimum yield is 0, and the scaled yield is equal to the yield. The above equation is undefined for a job whose minimum yield is equal to 1; Such a job is always "happy" in a valid resource allocation, and we defined its scaled yield as 1 in a valid allocation and 0 otherwise. For the remainder of this chapter, we use the term "yield" to mean "scaled yield," while we sometimes explicitly refer to the "unscaled yield."

4.1.1 Theoretical Background, Complexity, and (In-)Approximability

The connection between bin packing and multiprocessor scheduling was made more than 30 years ago [159]. While in the past it was assumed that jobs would run in sequence on the machine to which they were assigned, and thus that one bin dimension was purely temporal, more recently authors have considered scenarios similar to the one that we propose where gang scheduling and/or virtual machine technology are used to time-share the CPU between competing jobs [51, 160]. These more recent works provide reasonable models of CPU scheduling, but fail to take into account situations where multiple resources are under consideration, in which case vector bin packing is a more appropriate model [161–169]. There are also many situations in which the number of bins is fixed and it is acceptable to slightly overload the bins, so long as the degree of overload is minimized. This altered version of the problem is called extensible bin-packing or bin-packing with extendible bins [170–172]. Our work incorporates ideas from both the vector and extensible variants of bin packing, and is thus similar to the problem explored by Epstein [173], but whereas Epstein allows for a finite number of arbitrarily configured types of vector bins, we instead focus on maximizing a linear scaling factor that is applied to a subset of the dimensions under consideration, making our work more directly applicable to the cluster resource allocation problem. The use of independent vectors to denote time- and space-shared resources makes our work similar to that of Garofalakis et. al. in [174], but we ensure greater fairness by optimizing minimum instead of average performance.

Given the general connection between scheduling and bin packing it should come as no surprise that our problem is NP-hard; nonetheless we provide a brief proof for the sake of rigor and completeness: The goal of the *decision* version of our problem is to determine whether or not a valid allocation exists for particular instance of the problem and a given α. Since checking a given solution requires a number of vector multiplications and additions equal to the number of jobs, followed by a number of comparison operations equal to the number of nodes times the number of dimensions under consideration, the problem is clearly in NP. Any instance of the one-dimensional bin packing problem can be trivially reduced to an instance of the off-line problem, and so the off-line problem is NP-complete in the strong sense [175].

As the problem is NP-complete, no polynomial time algorithm can provide optimal solutions for every instance of our problem unless $P = NP$, which is considered to be unlikely. Instead, we explore the multi-dimensional vector packing literature in search of practical algorithms that have good average and worst case behavior. Versions of standard greedy algorithms (First Fit, Best Fit, Worst Fit, Next Fit, etc.) have been proposed [164, 169], their worst-case behaviors have been studied [162, 164], as well as their average behavior over practical instances [169, 176]. Results show that these algorithms can have worst case performance guarantees no better than $D + \delta$, where D is the number of resource dimensions and δ is some constant ≤ 1. A polynomial-time algorithm with a guarantee of $D + \varepsilon$, for any $\varepsilon > 0$, can be obtained by reusing the $(1 + \varepsilon)$-guaranteed bin packing algorithm in [177]. This guarantee was improved in [178], which proposes an algorithm with a $O(\ln D)$ performance guarantee for fixed D (the guarantee approaches $2 + \ln D$ for large D). More recently, [161] gave a complex algorithm with a performance guarantee that can be brought arbitrarily close to $1 + \ln D$. Several authors have also studied vector packing specifically for the case where $D = 2$ [163, 179, 179–182].

4.1.2 Mixed-Integer Linear Program Formulation

We formulate the problem as a Mixed Integer Linear Program (MILP), i.e., a linear program with rational and integer variables. We consider $J > 0$ jobs, indexed by $j = 1, \ldots, J$. The cluster comprises $N > 0$ identical physical nodes, indexed by $n = 1, \ldots, N$. Each node provides D types of resources, indexed by $d = 1, \ldots, D$. Fractions of these resources can be allocated to jobs. For each job j, r_{jd} denotes its resource need for resource type d, as a resource fraction between 0 and 1. δ_{jd} is a binary value that is 1 if r_{jd} is a fixed requirement, and 0 if r_{jd} is a fluid need; Note that this allows the MILP to solve a slightly more general version of the problem, wherein the same resource can be fluid for one job and fixed for another. While this poses some interesting possibilities, most of the current literature maintains a fundamental distinction between these two types of resources [22, 25, 102] and so we leave a study of the more general problem for future work. We use $\hat{\alpha}_j$ to denote the minimum unscaled yield requirement of job j, a value between 0 and 1.

We can now define the variables of our linear program. We define a binary variable e_{jn} that is 1 if job j runs on node n and 0 otherwise. We denote by α_{jn} the unscaled yield of job j on node n, which must be equal to 0 if the job does not run on the node. With these definitions the constraints of our linear program are as follows, with Y denoting the minimum yield:

$$Y \in \mathbb{Q}^+ \tag{4.1}$$

$$\forall j, n \quad e_{jn} \in \{0, 1\} \quad \alpha_{jn} \in \mathbb{Q} \tag{4.2}$$

$$\forall j \quad \sum_n e_{jn} = 1 \tag{4.3}$$

$$\forall j, n \quad 0 \leq \alpha_{jn} \leq e_{jn} \tag{4.4}$$

$$\forall n, d \quad \sum_j r_{jd}(\alpha_{jn}(1 - \delta_{jd}) + e_{jn}\delta_{jd}) \leq 1 \tag{4.5}$$

$$\forall j \quad \sum_n \alpha_{jn} \geq \hat{\alpha}_j + Y(1 - \hat{\alpha}_j) \tag{4.6}$$

Constraints (4.1) and (4.2) define the domains of our variables. Constraint (4.3) states that a job runs on exactly one node. Note that only one of the terms in the summation is non-zero. Constraint (4.4) states that a job can achieve an unscaled yield greater than 0 only on the node on which it runs. Constraint (4.5) states that the fraction of resource d on node n that is allocated to jobs is at most 1. The expression in the summation is explained as follows: If job j has a fluid need in resource d, then $\delta_{jd} = 0$ and the fraction of resource d used by job j on node n is $r_{jd} \times \alpha_{jn}$ (the maximum usable fraction multiplied by the yield). If instead job j has a fixed requirement in resource d, then $\delta_{jd} = 1$ and the fraction of resource d used by job j on node n is simply r_{jd}. Finally, Constraint (4.6) states that the minimum yield, Y, is no greater than the yield of any job. This constraint is written so that it also subsumes the requirement that the unscaled yield assigned to a job be larger than the job's specified minimum. The objective is to *maximize* Y.

4.2 Algorithms

4.2.1 Exact Solution

An exact solution to the problem can be computed by solving the MILP provided in Section 4.1.2. However, the time required to compute such a solution cannot be bounded by any polynomial in the size of the input (i.e., solving an MILP is not in P-time) and in fact for moderately sized problem instances real solvers may require an impractical amount of time (hours or days) to compute a solution. We use a publicly available MILP solver, the Gnu Linear Programming Kit (GLPK), to compute exact solutions for small problem instances (few nodes, few jobs) in under an hour. When available, we denote this exact solution by OPT.

4.2.2 Greedy Algorithms

In this section we propose greedy algorithms. These algorithms are not identical to greedy algorithms for vector packing [164, 169], due to the presence of fluid resource needs and differing objective function, but are inspired by similar ideas. The standard approach is to sort the jobs in some order, and then to pick a node for each job in order. We consider the following seven options to sort jobs: (S1) randomly; (S2) by decreasing maximum fluid need; (S3) by decreasing sum of fluid needs; (S4) by decreasing maximum fixed requirement or constrained fluid need; (S5) by decreasing sum of fixed requirements and constrained fluid needs; (S6) by decreasing maximum fixed requirement or fluid need; (S7) by decreasing sum of fixed requirements and fluid needs. We do not consider increasing orders since they are known to be inferior to decreasing orders for the vast majority of bin packing problem instances.

We also consider seven options to pick a node for a given job, j, provided that the node can accommodate the job's resource requirements. Let d_f be the index of the resource corresponding to the maximum fluid resource need of job j, i.e., r_{jd_f}. Let d_r be the index of the resource corresponding to the maximum fixed requirement or constrained fluid need of job j, i.e., r_{jd_r}. Let J_n be the set of the jobs already placed on node n. Two options are: (P1) pick node n with the smallest $\max_{j' \in J_n} r_{j'd_f}$; and (P2) pick node n with the smallest $\sum_{j' \in J_n} r_{j'd_f}$. In other words, P1 (resp. P2) places job j on the node that has the smallest maximum (resp. sum) of the fluid needs of jobs already placed on that node for resource d_f. These two options are oblivious to fixed requirements and constrained fluid needs. The other approach is to be oblivious to fluid resource needs. This can be done using standard best fit or worst fit placement, evaluating the load of each node n based on $\max_{j' \in J_n} r_{j'd_r}$ or $\sum_{j' \in J_n} r_{j'd_r}$. We term P3 and P4 the two corresponding best fit approaches, and P5 and P6 the two corresponding worst fit approaches. Finally, one can simply use first fit placement, placing a job on the first node that can accommodate its fixed requirements and constrained fluid needs, which we call P7.

Combining all job sorting and node picking options above, we obtain $7 \times 7 = 49$ different greedy algorithms, which we name GREEDY_SX_PY, where X $\in$ {1, 2, 3, 4, 5, 6, 7} and Y $\in$ {1, 2, 3, 4, 5, 6, 7}. All these algorithms are straightforward to implement with complexity at most $O(J \log J + JN)$ for a fixed number of resource dimensions. These algorithms subsume the greedy algorithms proposed in the applied literature [25, 89, 183].

All these algorithms could be augmented with a backtracking feature (with some bound to avoid exponential complexity) to enhance the search for a feasible solution. This technique was evaluated in previous work and shown to be unsuccessful for moderately large program instances, even with only two resource dimensions [184, 185]. Thus, we do not attempt backtracking for any of the above algorithms.

4.2.3 Relaxed LP Solution and its Uses

As we have stated, for large problem instances the MILP formulation cannot be solved in reasonable time. However, for these instances we can efficiently solve a *relaxed* version of the problem in which all variables are assumed to be rational. The obtained solution may be infeasible as jobs could be split across nodes, but it does provide an upper bound on the maximum minimum yield obtained when solving the MILP. It also forms a good basis for comparing various heuristics to the optimal: if a heuristic achieves a minimum yield close to the upper bound on the optimal, then it is even closer to the optimal. We call this upper bound LPBOUND. If the rational LP can be solved (i.e., the aggregate resource capacities can meet all fixed requirements and constrained fluid needs), then it has an immediate solution:

$$Y = \min\left(1, \min_{d \in NZ} \frac{N - \sum_i r_{jd}(\hat{\alpha}_j(1-\delta_{jd}) + \delta_{jd})}{\sum_j (1-\hat{\alpha}_j) r_{jd}(1-\delta_{jd})}\right) ,$$

where NZ is the set of indices $d \in \{1, \ldots, D\}$ such that $\sum_j (1-\hat{\alpha}_j) r_{jd}(1-\delta_{jd})$ is non-zero (i.e., it is the set of resource dimensions for which at least one job with a minimum allowed yield less than one has a fluid need greater than zero). This maximum minimum yield is achieved by the trivial allocation $e_{jn} = 1/N$ and $\alpha_{jn} = \frac{1}{N}\left(\hat{\alpha}_j + Y(1-\hat{\alpha}_j)\right)$, for all j and n.

Another use for the solution to the rational LP is that it may point the way toward good solutions of the MILP. A well-known idea is to round off to integer values the rational values assigned to integer variables, in our case the e_{jn} variables. For instance, if the relaxed solution produces some e_{jn} equal to 0.98, it seems likely that this value is equal to 1 in the solution of the MILP. Given a solution to the rational LP, we use the rounding approach in [186]: for each job j, taken in an arbitrary order, allocate job j to node n with probability e_{jn}. For each node n that cannot accommodate job j due to resource constraints, then e_{jn} is set to 0 and other probabilities are scaled accordingly. We call this algorithm RRND (Randomized Rounding). Similar techniques have been applied to other job scheduling problems in the past [187].

The trivial solution given above for the rational LP is a very poor starting point for rounding off e_{jn} values. Since all these values are identical, a job j is equally likely to be allocated to any node, which amounts to a random greedy allocation. A good starting point would be a solution of the rational LP in which e_{jn} values are diverse and distributed over the interval $[0, 1]$. We use GLPK, which uses the simplex algorithm to compute a solution in polynomial time. It turns out that, in practice, this solution leads to e_{jn} values that are well distributed in the interval $[0, 1]$.

One problem with RRND is that job j may not fit on any node n for which $e_{jn} > 0$ due to resource constraints. In this case the algorithm fails. To address this problem we can first set each zero e_{jn} to a small value ε (we use $\varepsilon = 0.01$). We call this algorithm RRNDNZ (Randomized Rounding Non-Zero). For those problem instances for which RRND provides solutions RRNDNZ should provide nearly identical solutions. But RRNDNZ should also provide solutions for some instances for which RRND fails.

Another approach, termed "diving" in [179], consists in solving the rational LP iteratively, each time fixing the e_{jn} variable closest to 0 or 1 to that value. This requires at most $J \times N$ rational LP resolutions, each time solving a LP with one fewer variable. Another approach, requiring at most J rational LP resolutions, consists in fixing at each iteration all N e_{jn} variables for a given j, picking the j with the largest $\max_h e_{jn}$ at each iteration. We term the first approach SLOWDIVING and the second approach FASTDIVING.

4.2.4 Genetic Algorithm

We implement a genetic algorithm similar to that used in [24] based on the GAlib library [188]. Each chromosome is a 1-D integer array of length J, in which the j-th value is equal to n if job j is allocated to node n. An initial chromosome is obtained by assigning each job to a random node. The mutation operator randomly swaps two jobs between two different nodes. Simply moving a job to a random node, instead of swapping, proved less effective in practice. We use a one-point crossover operator, by which two parent chromosomes are each cut into two segments and two new chromosomes are obtained by concatenating these segments. A new generated chromosome (initial, after mutation, or after crossover) may not correspond to a feasible resource allocation. We allow for infeasible chromosomes in our population. However, after an infeasible chromosome is generated, we use a greedy algorithm that attempts to make the chromosome feasible. This algorithm goes through the nodes in an arbitrary order, and for each overloaded node attempts to move jobs to other less loaded nodes. This approach, which reduces the diversity of the chromosome population and biases it toward feasible allocations, proved dramatically beneficial in practice. The fitness of an infeasible genome is defined as the number of nodes that are not overloaded in the mapping corresponding to the genome, and is thus between 0 and N. The fitness of a feasible genome is defined as $N(1 + Y)$, where Y is the achieved minimum yield. This fitness is thus between N and $2N$. We use a population size of 100, running for 2,000 generations, with a mutation probability of 0.1 and a crossover probability of 0.25. These parameters were estimated empirically based on calibration experiments with 792 instances (one instance for each experimental scenario, as described in Section 4.3).

4.2.5 Vector Packing Algorithms

As stated at the beginning of this section, our scheduling problem is closely related to *vector packing*. However, there are important differences. The most significant of these is that in the scheduling problem jobs may have fluid resource needs. This difference can be addressed as follows: Consider an instance of the problem and a fixed value of the yield, Y, that must be achieved for each job. Fixing Y amounts to making all fluid needs behave like fixed requirements. The problem then becomes exactly vector packing with the goal being to fit all of the items into at most a number of bins equal to the number of available nodes. A binary search on Y can be used to find the highest value for which the problem can be solved. Given a vector packing algorithm ALG, we term this general approach VP_ALG. We consider the following vector packing algorithms:

Best Fit and First Fit Algorithms [162, 164] – Standard Best Fit (BF) and First Fit (FF) algorithms are among the first algorithms used for solving vector packing problems and both rely on pre-sorting of the input vectors. We use three approaches to sort the vectors, as outlined in [164]: by decreasing sum of the coordinates (SUM), by decreasing maximum of the coordinates (MAX), and by decreasing lexicographical order (LEX). For fixed number of dimensions D, these algorithms can be implemented straightforwardly with complexity $O(J \log J + JN)$. We obtain 6 new algorithms: VP_BFSUM, VP_BFMAX, VP_BFLEX, VP_FFSUM, VP_FFMAX, and VP_FFLEX. An intriguing heuristic is presented in [169] as an add-on to any algorithm A that first sorts all vectors according to some criteria. Given the assignment of vectors to bins produced by A, one computes a metric called the "degree of dominance," which quantifies, for each dimension, the probability that this dimension causes bin capacities to be exceeded. One then re-sorts all vectors based on a sum of their coordinates weighted by their degrees of dominance, and apply algorithm A with this order. We have implemented this heuristic for all 6 algorithms but did not observe a single case in which it led to an improvement in our experiments over 72,900 problem instances.

Permutation Pack (PP) and Choose Pack (CP) Algorithms [166] – These algorithms attempt to balance the load of the dimensions of each bin. The PP algorithm places each of the J vectors in one of $D!/(D-w)!$ lists, where w is an integer between 1 and D. Each list contains the vectors with a common permutation of their largest w dimensions. For instance, for $w = 2$ and $D = 3$, there would be 6 lists, for all combinations (i,j), with $i, j \in \{1, \ldots, D\}$. List ($i$,$j$) contains the vectors whose i-th coordinate is larger that their j-th coordinate, which is larger than all their other coordinates. Vectors in each list are then sorted according to some criterion. We use four standard options: by decreasing sum of the coordinates (SUM), by decreasing maximum of the coordinates (MAX), by decreasing difference of the largest and smallest coordinate (DIFF), and by decreasing ratio between the largest and smallest coordinate (RATIO). In [184, 185], the four corresponding increasing orders were evaluated and, unsurprisingly, found to be consistently outperformed by the decreasing orders. The algorithm then starts filling bins with vectors, each time attempting to reduce the resource load imbalance in a bin. This is done by considering the current bin, and determining which w resource dimensions are least loaded for that bin, say, in the case $w = 2$, dimension i and then dimension j. The algorithm then first looks in list (i,j) for the first vector that can fit in the bin, hoping to reduce the resource imbalance. If no such vector can be found, then the algorithm relaxes the ordering of the components and searches in other lists (i.e., trying list (i,k), where k is the third least loaded resource of the bin, etc.). If no vector can fit in the current bin, then a new bin is added and the process is repeated until all vectors are placed in bins. The CP algorithm is a relaxation of the PP algorithm in that it does not enforce any ordering between the w coordinates of vectors, and thus needs "only" $D!/w!(D-w)!$

lists. The empirical results in [166] show that $w = 2$ leads to good results and we use this value in this chapter. For fixed D and w, both algorithms have complexity $O(J \log J)$. With the CP and PP algorithms, and the four options to sort vector lists, we obtain 8 new algorithms: VP_CPSUM, VP_CPMAX, VP_CPDIFF, VP_CPRATIO, VP_PPSUM, VP_PPMAX, VP_PPDIFF, and VP_PPRATIO.

The $O(\ln D)$ Guaranteed Algorithm [178] – This polynomial-time algorithm solves a rational linear program formulation of the vector packing problem, which leads to a bounded number of non-integral assignments of vectors to bins. Additional bins are then created in a greedy fashion to accommodate all vectors with non-integral assignments. We do not implement the algorithm in [161], in spite of its impressive $(1 + \ln D)$ guarantee. This algorithm formulates the vector packing problem as a set cover problem. Unfortunately, the instance of the set cover problem can be (and in our case, is) exponential in the size of the instance of the vector packing problem. Although the authors state that an approximation of the set cover problem instance could be formulated, no guidance is provided in [161]. Furthermore, the algorithm has high complexity regardless. We opt for the simpler guaranteed algorithm in [178] instead, which we call VP_CHEKURI.

4.3 Experimental Methodology

We evaluate our algorithms using a collection of randomly generated synthetic problem instances for J jobs and N nodes. We generate instances with D resource dimensions, where D is even. For each job, the first $D/2$ resource dimensions correspond to fixed requirements and the last $D/2$ resource dimensions to fluid needs.

All resource needs are sampled from a normal probability distribution with mean μ and standard deviation σ. Each job has a probability ρ to have a QoS requirement. We arbitrarily assume all QoS requirements to be 0.5 (i.e., half the job's fluid needs must be met). Experiments with other values, or with random values for all jobs, have led to similar conclusions regarding the relative performance of our algorithms. Depending on N and J, the aggregate resource needs of the jobs may overcome aggregate node capacities in one or more resource dimensions. We parameterize the overall resource load as follows. For each of the resource dimensions with fixed requirements, we scale the requirements in that dimension by a single factor, so that the aggregate node capacities can accommodate the aggregate job requirements in that dimension while a given fraction of the aggregate node capacities remains free. We call this free fraction $slack$. A lower value denotes an instance that is more difficult to solve. Some of our generated instances may not have solutions, especially for low slack values and/or large numbers of jobs with QoS requirements.

Unless specified otherwise, we use $N = 64$, $J = 100, 200, 500$, $D = 2, 4, 6$, $\mu = 0.5$, $\sigma = 0.25, 0.5, 1.0$, $\rho = 0.0, 0.25, 0.5$, and $slack = 0.1, 0.2, \ldots, 0.9$. This corresponds to $1 \times 3 \times 3 \times 1 \times 3 \times 3 \times 9 = 729$ scenarios. For each scenario we generate 100 random samples, for a total of 72,900 individual instances. Algorithm execution times are measured on a dedicated 3.2GHz Intel Xeon processor and averaged over 100 sample problem instances with N and J values as indicated in the text, and with $\mu = 0.5$, $\sigma = 0.5$, $\rho = 0.25$, and $slack = 0.5$. Over our entire set of 72,900 instances, there are 5,320 instances for which no algorithm was able to compute a valid allocation (7.45$\%$ of the instances).

Algorithm evaluation must account for two criteria: (i) how often an algorithm successfully computes a solution; and (ii) how good that solution is compared to those from other algorithms. For each algorithm we compute two metrics. The first is the *failure rate (fr)*, i.e., the percentage of instances for which it fails to find a solution. The second metric is the *distance from bound (dfb)*, i.e., the difference between the achieved minimum yield and LPBOUND. The *dfb* is computed for all instances for which the algorithm successfully produces a solution. We report average values over these instances as well as 90th percentile values (i.e., the value below which 90% of the *dfb* values fall). *dfb* values are absolute, and we also present results for relative percent *dfb* values (relative to LPBOUND). Low values for both metrics are desirable.

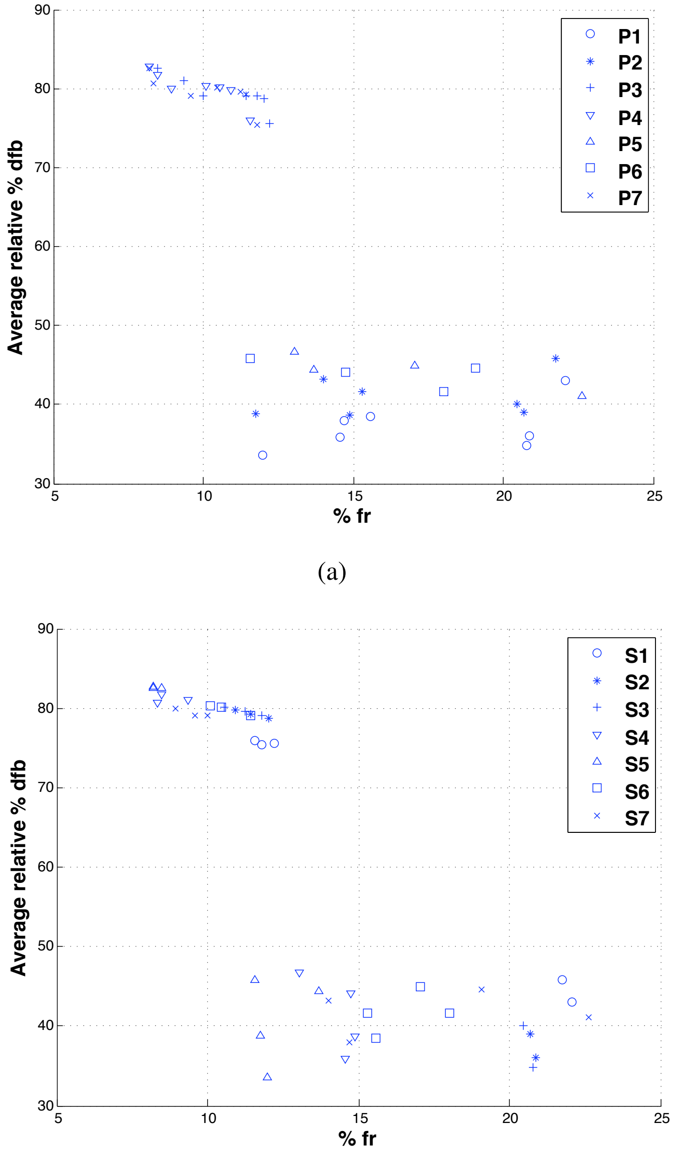
4.4 Experimental Results

4.4.1 Greedy Algorithms

In this section we evaluate the 49 greedy algorithms described in Section 4.2.2.

Figure 4.1 shows one data point for each algorithm, with the x-coordinate being the algorithm's *fr* and the y-coordinate being the algorithm's average relative *dfb*. Algorithms located toward the left and the bottom of the figure are preferable. For better readability, Figure 4.1(a) differentiates algorithms by their node selection strategy (P1 to P7), while Figure 4.1(b) differentiates algorithms by their job sorting strategy (S1 to S7).

Algorithms fall in two clusters depending on whether they use the P1, P2, P5, or P6 (Cluster #1), or the P3, P4, or P7 (Cluster #2) node selection strategy. Algorithms in Cluster #1 lead to lower *dfb* (by about 30 points or more), but to higher *fr* (by as much as about 10 points). However, some of the algorithms in Cluster #1 lead to *fr* as low as 12%. The algorithms in Cluster #1 sacrifice *fr* in two ways. Those that use P1 or P2 ignore non-fluid resource needs and thus solely attempt to optimize the yield without also considering the fixed requirements or constrained fluid needs. Those that use P5 or P6 pay more attention to fixed requirements and constrained fluid needs but use a Worst Fit



(a)

(b)

Figure 4.1. Bi-criteria graphical comparison of all GREEDY_Sx_Py algorithms, averaged over all 72,900 instances.

strategy by which they leave resources as free as possible while mapping jobs to nodes. This leads to better opportunities for optimizing the yield, but also to more failures.
In terms of job sorting strategies, S5 and, to a lesser extent, S4 are best. These are the two sorting strategies that consider both fixed requirements and constrained fluid needs. Overall, using the maximum of resource needs both for job sorting and for node selection is marginally more effective than using their sum, but no strong empirical claim can be made.

Expectedly, we find that each algorithm, even GREEDY_S1_P7 (random job sorting, random node selection), is best for at least some fraction of our instances. Furthermore, it is difficult to identify clear trends with respect to our instance parameters. We note that the time to execute one of the GREEDY_Sx_Py algorithm is relatively low even for moderately large instances, e.g., below 0.1 seconds for instances with $N = 64$ nodes and $J = 4,096$ jobs. A brute-force approach is then to combine all algorithms: run all 49 GREEDY_Sx_Py algorithms and pick the best successfully produced resource allocation, if any. We call this approach GREEDY.

The GREEDY approach may prove too expensive for some problem instances. For example, for $N = 1,024$ nodes and $J = 16,384$ jobs, some of the greedy algorithms require up to 3 seconds. Based on our experimental results, we can identify and rank 9 GREEDY_Sx_Py algorithms that beat or equal their competitors for more than 15% of the instances (x∈{2, 3, 5, 6, 7} and y=1, and x∈{2, 3, 6} and y=2). We also find that algorithm GREEDY_S5_P4 leads to the highest success rate (it fails for only 0.04% of the instances for which some other GREEDY_Sx_Py is successful), even though it almost never outperforms the previous 9 algorithms in terms of minimum yield when these algorithms succeed. Therefore, a reasonable approach, which we call GREEDYLIGHT, consists in using only these 10 algorithms. Furthermore, GREEDYLIGHT tries the 10 algorithms in sequence, stopping as soon as an algorithm produces a resource

Table 4.1. Average *dfb*, 90th percentile *dfb*, and *fr*, for the LP-based algorithms and Greedy, over 48,600 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
RRnd	0.58 (78.33%)	0.86 (98.52%)	66.56
RRndNZ	0.58 (77.95%)	0.89 (98.28%)	22.02
FastDiving	0.60 (75.03%)	0.84 (94.39%)	78.02
SlowDiving	0.57 (72.75%)	0.81 (93.60%)	77.92
Greedy	0.21 (29.17%)	0.38 (51.49%)	7.50

allocation. The sequence order is that of increasing empirical average *dfb* as observed in our experiments. Out of out the 72,900-5,320 = 67,580 instances for which Greedy succeeds, GreedyLight fails for only 203 instances (or 0.44%). When both algorithms succeed, Greedy outperforms GreedyLight in 20.78% of the cases, in which case if leads to a minimum yield that is relatively better by 20.26% on average. The Greedy and GreedyLight algorithms provide good baselines against which to evaluate more sophisticated algorithms.

4.4.2 LP-based Algorithms

In this section we evaluate the algorithms described in Section 4.2.3: RRnd, RRndNZ, FastDiving, and SlowDiving. Due to the large execution times of these algorithms we do not present results for $N = 500$, reducing the number of tested instances from 72,900 to 48,600. We compare these algorithms with the Greedy algorithm.

Table 4.2. Average execution times for LP-based algorithms with 64 nodes and and 64, 256 and 512 jobs.

	Average Execution Time (sec)		
Algorithm	$J = 128$	$J = 256$	$J = 512$
RRND	16.30	61.70	255.44
RRNDNZ	16.74	61.15	250.83
FASTDIVING	32.42	113.89	416.32
SLOWDIVING	382.58	1771.35	6704.79
GREEDY	0.05	0.10	0.24

Table 4.1 shows aggregate results for all five algorithms over all our problem instances. The striking observation is that GREEDY largely outperforms all algorithms that rely on solving a rational relaxation of the MILP problem formulation. GREEDY fails to compute a solution in only 3,646 of the 48,600 instances, or 7.50%. There is no instance for which GREEDY fails and one of its competitors succeeds. By contrast, RRND, FASTDIVING, and SLOWDIVING exhibit high failure rates above 65%. RRNDNZ has a much lower failure rate at 22.02%. This is expected since RRNDNZ was designed as an improvement to RRND precisely to reduce the likelihood of failure (in fact, RRNDNZ always succeeds when RRND succeeds). SLOWDIVING exhibits a slightly lower failure rate than FASTDIVING, which again is expected as SLOWDIVING is “more careful” when rounding off rationals to integers.

In terms of *dfb*, we see that GREEDY also leads to a drastic improvement relative to the other algorithms, both for the average and the 90th percentile. In spite of their use of more sophisticated methods for rounding rationals to integers, SLOWDIVING and FASTDIVING are not significantly closer to LPBOUND than RRND and RRNDNZ. RRND and RRNDNZ lead to similar *dfb*. SLOWDIVING provides a marginal improvement over FASTDIVING.

GREEDY runs orders of magnitude faster than the other four algorithms. Table 4.2 shows average algorithm execution times. Faster execution times could be achieved for

Table 4.3. Average *dfb*, 90th percentile *dfb*, and *fr*, for the Greedy and GA algorithms, over 72,900 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
Greedy	0.16 (30.06%)	0.34 (56.25%)	7.82
GA	0.24 (42.46%)	0.41 (68.18%)	9.57

all the LP-based algorithm by using a faster (commercial) linear program solver such as CPLEX [189]. Regardless, the execution times would likely still prohibit the use of the algorithms in practice.

The conclusion is that although algorithms that solve a rational relaxation of similar scheduling problems have been used successfully in the literature, in our context they perform poorly. One intuitive reason for this result is that binary e_{jn} variables are difficult to compute by rounding off rational values. There may simply not be a good way to round off a value of, say, 0.51 or 0.49 to either 0 or 1 without leading to a schedule that is far from the optimal schedule. Greedy successfully solves more instances, leads to better minimum yields, and runs orders of magnitude faster. In the rest of this paper we exclude results for RRnd, RRndNZ, FastDiving, and SlowDiving.

4.4.3 Genetic Algorithm

In this section we evaluate the genetic algorithm, GA, described in Section 4.2.4. Table 4.3 shows results for the Greedy and GA algorithms, computed over all problem instances. We see that GA is outperformed by Greedy in terms of average *dfb*, 90th percentile *dfb*, and *fr*. Over all feasible instances, GA outperforms Greedy in 9.96% of the cases, and in these cases it leads to a minimum yield on average 26.64% higher than Greedy. By contrast, Greedy outperforms GA in 79.08% of the feasible instances, in which case it leads to a minimum yield that is on average 32.65% higher than GA.

Table 4.4. Average *dfb*, 90th percentile *dfb*, and *fr*, for the VP-based, Greedy, and GreedyLight algorithms, for 72,900 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
GreedyLight	0.16 (31.49%)	0.35 (56.25%)	8.16
Greedy	0.16 (30.07%)	0.34 (61.01%)	7.73
VP_PPRatio	0.08 (14.54%)	0.17 (28.32%)	15.81
VP_PPDiff	0.08 (13.67%)	0.16 (21.10%)	15.35
VP_FFLex	0.07 (12.85%)	0.15 (27.86%)	15.45
VP_PPMax	0.07 (13.08%)	0.15 (26.67%)	14.99
VP_PPSum	0.07 (12.84%)	0.15 (26.39%)	14.93
VP_CPRatio	0.07 (11.09%)	0.14 (21.21%)	11.45
VP_BFLex	0.06 (12.15%)	0.14 (27.10%)	13.75
VP_CPDiff	0.06 (10.19%)	0.12 (21.10%)	8.70
VP_CPMax	0.05 (10.10%)	0.11 (20.60%)	8.43
VP_CPSum	0.05 (9.92%)	0.11 (20.40%)	8.20
VP_BFMax	0.04 (11.39%)	0.11 (29.40%)	8.48
VP_FFMax	0.04 (11.26%)	0.11 (29.33%)	8.33
VP_BFSum	0.04 (10.95%)	0.10 (28.72%)	7.91
VP_FFSum	0.04 (10.95%)	0.10 (28.40%)	7.91

We conclude that the genetic algorithm approach is less effective than the greedy approach for our problem. Although we use a population of 100 genomes in GA, we have experimented with population sizes up to 2,000 and did not observe significant improvements. We have seen marginal improvements when increasing the number of generations from 100 to 2,000, suggesting that further increasing the number of generations could be beneficial. However, the execution time of GA is at least one order of magnitude larger than that of Greedy (for example, GA requires over thirteen seconds to compute a solution when $N = 512$ while Greedy requires less than a quarter of a second), and increasing the number of generations further is not practical.

4.4.4 Vector Packing Algorithms

In this section we evaluate the algorithms in Section 4.2.5, which use a vector packing (VP) approach. We also include Greedy and GreedyLight in this comparison.

We do not present results for VP_CHEKURI. Due to its computational cost, we ran this algorithm only on instances with $N = 100$. VP_CHEKURI leads to much higher failure rates than all its competitors and lower yields in all instances. It is also orders of magnitude more computationally expensive in practice, due to solving a large LP at each iteration of the binary search.

Table 4.4 summarizes the results averaged over all 72,900 instances. Rows of the table are sorted by decreasing average *dfb* and, for equal *dfb*, by decreasing *fr*. In terms of *dfb*, GREEDY and GREEDYLIGHT are outperformed by all VP-based algorithms. Furthermore, both have failure rates that are not significantly better than those of the least failure-prone VP-based algorithms. All VP-based algorithms exhibit comparable behavior, with no clear clustering of the algorithms when looking at averages.

We seek more insight into our results using one-to-one algorithm comparisons via a *domination* relationship. For two algorithms A and B, we define the following two measures: (i) $\mathcal{S}_{A,B}$: the percentage of instances for which A succeeds and B fails; and (ii) $\mathcal{Y}_{A,B}$: the average percent minimum yield difference between A and B, relative to the minimum yield achieved by B, computed on instances for which both algorithms succeed. For both measures, a positive value means an advantage of A over B. We say that "algorithm A dominates algorithm B" if $\mathcal{S}_{A,B} \geq 0.5\%$ and $\mathcal{Y}_{B,A} \leq 3\%$, or, $\mathcal{S}_{B,A} \leq 0.5\%$ and $\mathcal{Y}_{A,B} \geq 3\%$. That is, algorithm A dominates algorithm B if it is significantly more successful and not significantly less effective at maximizing minimum yield, or if it is significantly more effective at maximizing minimum yield and not significantly less successful. We say that two algorithms are equivalent if neither algorithm dominates the other. We picked 0.5% to mean "not significant" and 3% to mean "significant." We experimented with values higher that 3% for significance and found that for these values very few dominance relationships could be established, as many algorithms are close to each other in terms of *dfb* and *fr*.

We established domination relationships based on our experimental results considering all our experiments ($D \in \{2, 4, 6\}$), the two subsets for $D \in \{2, 4\}$ and $D \in \{4, 6\}$, and the three subsets for $D = 2$, $D = 4$, and $D = 6$, for a total of $1 + 2 + 3 = 6$ result subsets. The goal of considering these subsets is to determine whether the number of resource dimensions has an impact on the relative performance and the algorithms.

We found that each Permutation Pack (PP) algorithm is dominated by its Choose Pack (CP) counterpart, for all result subsets. We found that VP_CPRATIO is dominated by VP_CPDIFF across all result subsets. Among the remaining three CP algorithms, VP_CPSUM is the only one that is not dominated for any result subset. VP_CPDIFF is not dominated for subsets $D = 2, 4, 6$, $D = 2, 4$, and $D = 2$, while VP_CPMAX is not dominated for subsets $D = 4, 6$, $D = 4$, and $D = 6$. This indicates that VP_CPMAX is preferable to VP_CPDIFF for problems with more resource dimensions, while the situation is reversed for problems with lower resource dimensions. Regardless, we conclude that VP_CPSUM is the algorithm of choice among all PP and CP algorithms.

Among the algorithms that use a Best Fit or First Fit approach, we found that algorithms using lexicographical ordering (VP_FFLEX and VP_BFLEX) are each dominated by both of their counterparts on all result subsets. VP_FFSUM and VP_FFMAX are equivalent for high resource dimensions (result subsets $D = 4, 6$ and $D = 4$), but VP_FFSUM dominates VP_FFMAX for all other result subsets. Conclusions are identical for VP_BFSUM and VP_BFMAX. The VP_FFSUM and VP_BFSUM algorithms are equivalent on all result subsets.

We are left with the VP_CPSUM, VP_BFSUM, and VP_FFSUM algorithms. These three algorithms are equivalent on all our result datasets, by our definition of equivalence. The left-hand side of Table 4.5 shows the $\mathcal{S}$ and $\mathcal{Y}$ measures for VP_CPSUM vs. VP_FFSUM and VP_BFSUM. We see that the $\mathcal{S}$ values 0.56% and 0.67% are only slightly above our 0.5% insignificance threshold. When considering all

Table 4.5. Some of the $\mathcal{S}$ and $\mathcal{Y}$ values pertaining to the VP_CPSum algorithm, computed over all our problem instances.

Measure	Value (%)
$\mathcal{S}_{\text{VP\_CPSum,VP\_FFSum}}$	0.02
$\mathcal{Y}_{\text{VP\_CPSum,VP\_FFSum}}$	6.93
$\mathcal{S}_{\text{VP\_FFSum,VP\_CPSum}}$	0.56
$\mathcal{Y}_{\text{VP\_FFSum,VP\_CPSum}}$	-1.11
$\mathcal{S}_{\text{VP\_CPSum,VP\_BFSum}}$	0.04
$\mathcal{Y}_{\text{VP\_CPSum,VP\_BFSum}}$	7.28
$\mathcal{S}_{\text{VP\_BFSum,VP\_CPSum}}$	0.67
$\mathcal{Y}_{\text{VP\_BFSum,VP\_CPSum}}$	-1.29

Measure	Value (%)
$\mathcal{S}_{\text{VP\_CPSum,Greedy}}$	0.01
$\mathcal{Y}_{\text{VP\_CPSum,Greedy}}$	42.23
$\mathcal{S}_{\text{Greedy,VP\_CPSum}}$	0.99
$\mathcal{Y}_{\text{Greedy,VP\_CPSum}}$	-22.34
$\mathcal{S}_{\text{VP\_CPSum,GreedyLight}}$	0.00
$\mathcal{Y}_{\text{VP\_CPSum,GreedyLight}}$	68.61
$\mathcal{S}_{\text{GreedyLight,VP\_CPSum}}$	0.66
$\mathcal{Y}_{\text{GreedyLight,VP\_CPSum}}$	-24.05

our problem instances, it turns out that VP_CPSum is outperformed by VP_FFSum (resp. VP_BFSum) for 68.97% (resp. 76.26%) of instances for which both algorithms succeed. However, in these cases, its minimum yield is only outperformed by 4.01% (resp. 3.93%) on average. However, when VP_CPSum outperforms VP_FFSum (resp. VP_BFSum) it does so by 27.00% (resp. 18.70%) on average. We conclude that the best algorithm among the VP-based ones is VP_CPSum. In terms of computational demands, the execution times of the three algorithms are comparable. For instances with $J \times N = 8,388,608$ (e.g., $J = 8,192$ jobs running on a cluster with $N = 1024$ nodes), the average execution time of VP_CPSum, VP_FFSum, and VP_BFSum are 1.38, 1.50, and 1.62 seconds, respectively.

VP_CPSum is also preferable to the Greedy and GreedyLight approaches, as can be seen in the right-hand side of Table 4.5. While $\mathcal{S}_{\text{Greedy,VP\_CPSum}}$ and $\mathcal{S}_{\text{GreedyLight,VP\_CPSum}}$ are low (above the 0.5% threshold but below 1%), $\mathcal{Y}$ values show that VP_CPSum largely outperforms Greedy and GreedyLight in terms of minimum yield. Furthermore, VP_CPSum is also less computationally demanding than Greedy and GreedyLight. For the aforementioned instances with $J \times N = 8,388,608$, the execution time of VP_CPSum is 1.38 seconds while that of Greedy is 54.56 seconds and that of GreedyLight is 1.75 seconds.

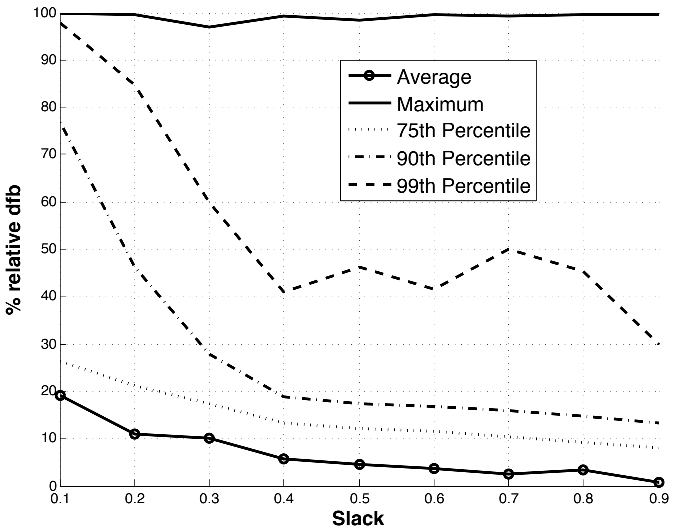


Figure 4.2. Percent relative *dfb* values for VP_CPSum vs. the $slack$.

We conclude that among all the algorithms considered in this chapter, the VP_CPSum algorithm is the algorithm that should be used in practice for computing resource allocations. This conclusion holds when taking further subsets of our results with respect to the σ, ρ, and $slack$ parameters.

4.4.5 Impact of Instance Parameters

In this section we study the effects of our instance parameters on the behavior of VP_CPSum. Figure 4.2 plots average, maximum, 75th percentile, 90th percentile, and 99th percentile of percent relative *dfb* values for VP_CPSum versus the $slack$. We see that *dfb* average values roughly decrease as the $slack$ increases. This is expected since higher $slack$ means an easier resource allocation problem. In our most difficult instances, $slack = 0.1$, VP_CPSum's relative *dfb* is on average 19.23%, which is reasonably close to LPBound. The 75th percentile curve is close to the average curve, denoting that for the bulk of our experiments VP_CPSum is still close to LPBound. The 90th and 99th percentile curves are expectedly higher, and the maximum curve remains close to 1.0.

Table 4.6. VP_CPSum's *dfb* values and relative *dfb* values (in parentheses), when fixing the memory slack and one of the parameters defining the instances.

Fixed param.	Param. Value	$slack = 0.1$		$slack = 0.4$	
		average	90th perc.	average	90th perc.
D	2	0.05 (16.51%)	0.13 (74.82%)	0.00 (0.19%)	0.24 (10.7%)
	4	0.09 (16.20%)	0.22 (41.61%)	0.03 (4.00%)	0.08 (15.61%)
	6	0.18 (31.83%)	0.57 (94.93%)	0.09 (13.15%)	0.17 (22.19%)
ρ	0.00	0.10 (16.90%)	0.49 (79.83%)	0.03 (5.14%)	0.10 (13.6%)
	0.25	0.08 (25.45%)	0.17 (79.13%)	0.04 (7.00%)	0.11 (18.92%)
	0.50	0.09 (14.91%)	0.22 (54.87%)	0.06 (1.13%)	0.16 (24.25%)
J	100	0.10 (10.29%)	0.21 (21.29%)	0.06 (6.00%)	0.17 (17.2%)
	200	0.15 (26.60%)	0.51 (84.90%)	0.05 (9.78%)	0.10 (20.12%)
	500	0.02 (15.02%)	0.11 (77.77%)	0.00 (0.01%)	0.02 (20.54%)
σ	0.25	0.10 (20.83%)	0.33 (80.62%)	0.03 (4.38%)	0.11 (17.30%)
	0.50	0.09 (20.09%)	0.20 (76.19%)	0.04 (6.36%)	0.13 (18.79%)
	1.00	0.08 (16.51%)	0.21 (70.71%)	0.04 (6.03%)	0.14 (19.69%)

This means that regardless of the $slack$ value, i.e., of the difficulty of the problem, there are still some instances that are hard to solve for our algorithm.

Table 4.6 shows VP_CPSum's absolute and relative *dfb* values for subsets of the instances when fixing one of the parameters that define our instances. The table shows average and 90th percentile values, for $slack = 0.1$ and $slack = 0.4$. We see that higher D increases the *dfb*. In the difficult case $slack = 0.1$, the relative *dfb* is at most 31.83% when $D = 6$. Higher values of ρ do not have much of an effect on the already difficult $slack = 0.1$ case, but do make the easier $slack = 0.4$ case more difficult. This is because with more jobs having QoS requirements, the resource allocation problem becomes more difficult. The σ parameter does not have a significant impact on the results. High values of the J parameter lead to low *dfb* values. This is because with more jobs, and keeping the $slack$ constant, the resource allocation problem becomes easier (many smaller jobs are easier to pack into nodes than fewer bigger jobs). Overall, this table demonstrates that, even when considering several subsets of our results, VP_CPSum is not far from LPBound on average (at most 31.83%).

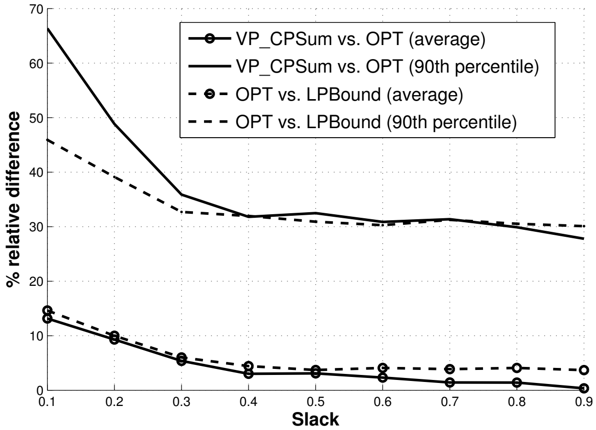


Figure 4.3. Percent relative *dfb* values for VP_CPSUM and OPT, vs. $slack$.

An important observation here is that VP_CPSUM is no further from OPT than from LPBOUND since LPBOUND is an upper bound on OPT. We conclude that VP_CPSUM produce resource allocations that are, on average, within roughly 30% of the optimal.

4.4.6 OPT vs. LPBOUND

So far we have used *dfb* as our measure of goodness for resource allocations, that is the distance to the LPBOUND upper bound on the optimal minimum yield. While a low *dfb* is certainly desirable, there remains the question of how tight the upper bound is. In this section we compare VP_CPSUM and LPBOUND to the optimal solution OPT computed by solving the MILP formulation of the resource allocation problem given in Section 4.1.2. Since solving a MILP takes exponential time, we use a set of "small" instances with $N = 4$, $J = 8, 10, 12$, and setting the other parameter values as previously. We found that out of these 72,900 instances, the MILP solver in GLPK failed to find a solution for 4,325 instances, or 5.93%. We assume that these instances have no feasible solution; more than 80% of them have $slack = 0.1$.

Figure 4.3 plots relative percent differences between VP_CPSUM and OPT, and between OPT and LPBOUND, both for the average and the 90th percentile. We see that, expectedly, all values decrease as $slack$ increases. Let us first examine the difference between OPT and LPBOUND (dashed lines on the figure). For the most difficult scenarios, i.e., $slack = 0.1$, OPT is on average 14.63% away from LPBOUND. The 90th percentile for $slack = 0.1$ is reasonable at 45.91%, showing that for the bulk of the instances LPBOUND is a relatively tight upper bound. We conclude that, at least for small instances, LPBOUND is a tight upper bound on optimal. Turning our attention to the distance between VP_CPSUM and OPT (solid lines), we see that roughly the same observations can be made even though the 90th percentile values for low $slack$ are a bit higher (up to 66.35% for $slack = 0.1$). We conclude that VP_CPSUM is roughly as far from OPT as OPT is from LPBOUND. In other terms, the relative difference between VP_CPSUM and OPT is about half of that between VP_CPSUM and LPBOUND. While impossible to verify, if this observation also holds true for large problem instance, then halving the values in Figure 4.2 and Table 4.6 would provide reasonable estimates of how far VP_CPSUM is from optimal.

4.4.7 Optimizing Average Yield

Once an allocation with a given maximum minimum yield, say $\mathcal{Y}$, has been produced, there may be excess resources available. To further improve resource utilization one can then maximize the average yield while preserving $\mathcal{Y}$ as the maximum minimum yield. This optimization can be framed as a MILP, simply replacing the objective function by the average yield, and adding the constraint $Y \geq \mathcal{Y}$. Unfortunately, the MILP cannot be solved in polynomial time and solving it would require developing and evaluating a number of heuristics. Instead, we opt for a simple solution: we enforce that the e_{jn} values computed by the minimum yield maximization procedure be kept constant. In other words, we do not allow average yield maximization to change the mapping of jobs to nodes. We only allow it to change allocated resource fractions. In this case, the MILP becomes a rational LP since all integer variables have become constants. It turns out that a simple greedy algorithm solves this LP. First, set the yield of each job to the minimum yield and update their resource fractions accordingly. Then, for each node, evaluate which job could get the highest yield increase given the remaining available resource fractions. Increase the yield of that job as much as possible. If free resources remain, repeat that procedure until no job can see its yield further increased. The optimality of this process is easily proved via a typical exchange argument.

Note that if the system is truly under subscribed, then some nodes can be turned off. Our algorithm can be used to determine resource allocation with different possible numbers of nodes and thus provide guidance on whether and how many nodes could be turned off without impacting the minimum yield in an unacceptable manner.

For each resource allocation we compute an overall cluster *utilization* metric. This metric accounts for the aggregate percentage of resources used in the cluster, excluding resources that correspond exclusively to fixed requirements. Indeed, the utilization of fixed resources is dictated by the requirements of the jobs, since in a successful allocation all such requirements must be met. This utilization is then a fixed quantity for a given problem instance, regardless of the resource allocation algorithm used, and we simply do not account for it.

Table 4.7 shows average utilization values computed over several subsets of our 72,900 instances, before and after average yield maximization, when using the VP_CPSUM algorithm. We see that as the number of resource dimensions increases, cluster utilization decreases. This is expected as with more resource dimensions vector packing is more difficult. We also see that, with the exception of the $D = 2$ results for $J = 200$ and $J = 500$, a larger number of jobs increases utilization. Again, this

Table 4.7. Average cluster utilization.

		Util. (%)	Opt. Util. (%)
$D = 2$	$J = 100$	77.75	78.32
	$J = 200$	98.03	99.98
	$J = 500$	95.70	99.97
$D = 4$	$J = 100$	72.87	77.06
	$J = 200$	88.60	94.51
	$J = 500$	94.76	96.74
$D = 6$	$J = 100$	70.07	75.77
	$J = 200$	83.15	90.92
	$J = 500$	93.36	95.82

is expected given that it is easier to pack many small vectors than fewer larger vectors. Results regarding how utilization varies with slack (not shown in the table) show that utilization improves marginally as the slack increases. We found that instance parameters ρ and σ had negligible impact on cluster utilization. Finally, we see that our average yield optimization step does improve cluster utilization noticeably. Larger improvements could be achieved by removing the constraint that the mapping of jobs to nodes be unchanged, but this would require the development of efficient average yield optimization heuristics.

4.5 Multi-VM jobs

So far we have assumed that each job consists of a single VM instance. There are, however, at least two compelling reasons why multi-VM jobs may be useful. First, a job could be implemented as a data-parallel application that consists of identical communicating tasks, with each task running inside a VM instance. This may be necessary in case the job requires an aggregate amount of memory beyond what a single node can provide, which is often the case for data-parallel applications in scientific domains. Second, a job may naturally consist of two or more components that must live within different VM instances and that have related resource needs (e.g., in steady-state a component uses half as much resources as another component). We discuss both these cases hereafter, explaining how our approach can be applied to each.

4.5.1 Data-parallel jobs

Our algorithms, and thus the VP_CPSUM algorithm, can be used almost directly to handle the data-parallel job scenario. In such a scenario, the job computes as fast as its slowest task. As a result, all tasks, which have identical resource needs, should be given identical resource fractions to avoid wasting resources. All tasks within a job should then experience the same yield. All tasks in the job can then be considered as individual jobs. This approach amounts to equating the yield of a data-parallel job with that of each of its individual tasks. Therefore, there is no incentive to implement a job in a data-parallel fashion simply for the purpose of achieving higher yield. Furthermore, our algorithms for single-task jobs can be employed *directly* (see for instance the experimental results in [185]).

There is a single difference between our approach for single-task jobs and that for data-parallel jobs. Recall that after computing a resource allocation that maximizes the minimum yield, our approach proceeds with an average yield maximization step. In Section 4.4.7, we have seen that, in the case of single-task jobs, average yield optimization can be formulated as a MILP. When not allowing average yield maximization to change the mapping of jobs to nodes (but only the resource fractions), this MILP becomes a rational LP. In the case of single-VM jobs, it turns out that the rational LP can be solved directly via a simple greedy algorithm. Unfortunately, the same algorithm cannot be applied to data-parallel jobs. One must then solve the rational LP, which can be done in polynomial time. For the sake of completeness we now give the formulation of this LP.

Let T_j be the number of tasks of job j. Let us consider a given mapping of tasks to nodes with an achieved minimum yield Y. The mapping of a task to a node is fully defined via a binary value e_{jtn} that is 1 if task t, $0 \leq t < T_j$, of job j is allocated to node n, and 0 otherwise. Defining α_{jtn} as the unscaled yield of task t of job j on node n, we can now write the following constraints, which are similar to the ones in Section 4.1.2. The main difference is that the e_{jtn} values and the Y value are constants rather than variables:

$$\begin{aligned}
& X \in \mathbb{Q}^+ && (4.7)\\
\forall j,t,n \quad & e_{jtn} \in \{0,1\} \quad \alpha_{jtn} \in \mathbb{Q} && (4.8)\\
\forall j,t,n \quad & 0 \leq \alpha_{jtn} \leq e_{jtn} && (4.9)\\
\forall j,t \quad & \sum_n \alpha_{jtn} \geq \hat{\alpha}_j && (4.10)\\
\forall j,t>0 \quad & \sum_n \alpha_{jtn} = \sum_n \alpha_{j0n} && (4.11)\\
\forall n,d \quad & \sum_{j,t} r_{jd}(\alpha_{jtn}(1-\delta_{jd}) + e_{jtn}\delta_{jd}) \leq 1 && (4.12)\\
\forall j,t \quad & \sum_n \alpha_{jtn} \geq \hat{\alpha}_j + Y(1-\hat{\alpha}_j) && (4.13)\\
& X = \sum_{j \ s.t.\ \hat{\alpha}_j<1,t} \frac{(\sum_n \alpha_{jtn}) - \hat{\alpha}_j}{T_j(1-\hat{\alpha}_j)} && (4.14)
\end{aligned}$$

Constraint (4.11) ensures that all tasks within a job have the same unscaled yield, which implies identical resource allocations. Constraint (4.13) ensures that the minimum yield is not reduced by average yield optimization. Constraint (4.14) defines X as the sum of the yields of all jobs with a minimum yield strictly lower than 1 (all other jobs have by definition a scaled yield equal to 1). Therefore X is linearly and positively correlated with the average yield computed over all jobs. The optimization objective is to maximize X. All variables, the y_{jtn} values and X, are rational making this program solvable in polynomial time.

4.5.2 Multi-Instance jobs

In the case of a job that is implemented with multiple components each running inside its own VM instance, our approach can be easily used if a linear relationship exists between the fluid resource needs of job components. For instance, consider a simple scenario in which nodes provide two resources, $r1$ and $r2$. Consider two components of the same job, A and B, each of them in its own VM instance. Say that component A's fluid need in resource $r1$ is 40$\%$ and component A's fluid need in resource $r2$ is 20$\%$. If, for each resource, component B requires half the amount required by component A then one can just set its fluid needs to 20$\%$ and 10$\%$ for resources $r1$ and $r2$, respectively, and treat both components as independent jobs. Before the average yield optimization step, all jobs have the same yield. Similarly to the data-parallel job case, the only modification to our approach is the resolution of a rational LP for average yield optimization. The fact that one component requires half as many resources as that required by another component could be specified by an operator. Alternately, the operator could simply specify that the two components are related, and the relationship could be discovered using the techniques described in Section 3.2.2.

As recognized in [153], multi-tier jobs raise a number of challenges. First, the relationship between the resource consumption of tiers are not necessarily linear or even uniform across resource dimensions (e.g., component A requires half the CPU of component B but twice the bandwidth), in which case our approach would need to be modified. Second, in a multi-tier scenario the number of instances for each tier may not be specified a-priori. Automatically deciding the number of instances is outside the scope of this paper, and raises the difficult “shifting bottleneck” issue identified in [153].

4.6 Conclusion

In this chapter we studied the resource allocation problem in shared hosting platforms for static workloads with nodes that provide multiple types of resources. We gave a formulation of the problem that supports a mix of QoS and best-effort scenarios, and that attempts to maximize a generic objective function, the minimum yield. We explained how an (in practice reasonably tight) upper bound on the optimal yield can be computed, and how the average yield can be maximized as a way to increase cluster utilization. We proposed and evaluated several classes of algorithms over a wide range of simulation scenarios. From our experimental results we conclude that performing a binary search over the yield and solving the resource allocation problem for a fixed yield using a vector packing algorithm is the best approach, and that vector packing algorithms that reason on the sum of the resource needs of the jobs are the most effective. Among those algorithms under consideration, the one that makes use of the Choose Pack vector packing algorithm from [166] runs in only a few seconds and was the most effective. Most notably, for our experimental scenarios it outperformed a greedy approach that combines many greedy algorithms, as well as linear program relaxations and a genetic algorithm approach. We conclude that we have found an algorithm that runs quickly and computes resource allocations that are close to the optimum.

Preliminary results upon which the work in this chapter is based were previously published in the *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid* [185] and the contents of this chapter have been published in the *Journal of Parallel and Distributed Computing* [190].

CHAPTER 5
THE ON-LINE PROBLEM

In this chapter we consider the on-line scheduling of temporary tasks. An example of an environment this problem would apply to would be a high-performance computing cluster intended to run scientific workloads. We make the following contributions: (i) We define the clairvoyant and non-clairvoyant versions of the on-line DFRS scheduling problem (Section 5.1); (ii) We establish the complexity of these problems and obtain absolute bounds on the performance of any DFRS scheduling algorithm for either scenario (Section 5.1); (iii) We propose several heuristic algorithms for solving the DFRS scheduling problem (Section 5.2) and (iv) We evaluate our proposed algorithms in simulation using a combination of synthetic and real workloads (Sections 5.3 and 5.4).

5.1 Problem Definition

Consider a set of nodes, N, that must service a stream of user jobs J. Each job j has a release time, r_j, before which the scheduler cannot start the job and has no information about the it, a number of tasks, T_j, fixed resource requirement and fluid resource need vectors $\vec{s_j}$ and $\vec{d_j}$ as described in the formulation of the off-line problem in Section 4.1, and a processing time, p_j, that is not known until the task completes. The value of p_j is the number of seconds in which the job would complete if running alone in the system (i.e., the run time). The scheduler can assign job tasks to nodes, migrate tasks between nodes, or set job yield values subject to constraints described in Section 4.1. Preempting jobs and migrating some (or all) of the tasks of a job to a different set of nodes both have costs, both in terms of network and I/O bandwidth consumed and in terms of penalizing the performance of the selected job.

As the problem is now on-line, the yield of a job can vary over the time it spends in the system, and the yield values it is assigned throughout its execution will affect its completion time. For each job the value $\alpha_j(t)$ represents the yield of the job at time t. When a job is not running $\alpha_j(t)$ is taken to be zero. The *virtual time* of a job at time t is the total subjective time experienced by the job between its submission and time t, and is equal to $\int_{r_j}^{t} \alpha_j(\tau) d\tau$. For example, a job that starts and runs for 10 seconds with a yield of 1.0, that is then paused for 2 minutes, and then restarts and runs for 30 seconds with a yield 0.5 has experienced 25 total seconds of virtual time ($10 \times 1.0 + 120 \times 0.0 + 30 \times 0.5$). A job completes when its virtual time equals or (in the case of discrete time units) exceeds p_j.

An important consideration for any formalized study of a scheduling problem is the metric by which the schedules generated by competing algorithms should be evaluated. In Chapter 4, which examined on the off-line problem, we chose to focus on the maximizing the minimum yield. However, this metric is instantaneous and is thus not be appropriate for an on-line setting with a significant temporal component. One reasonable and well established metric of individual job performance in HPC environments with on-line job submissions is the *stretch* (also referred to as the slowdown), which is defined as the ratio of the time that a given job spends in the system divided by the time the job would spend if alone in the system. Comparing the stretches of different jobs in a workload can give information about the quality of a schedule, and comparing metrics based on aggregates of stretch values over a range of different workloads can give information about the relative performance and fairness of different scheduling algorithms.

Approaches that seek to improve average stretch (such as found in [191] and [192]) are likely to be more fair than those that seek to minimize makespan (as in [193]). However, some jobs may still have to wait an inordinate, or even unbounded, amount of time [91]. Instead, we choose to minimize the maximum stretch as this is known to provide a reasonable compromise between performance and fairness while still avoiding starvation [90].

5.1.1 Related Work

Our research is related to several previous works that have explored algorithmic issues pertaining to bin packing and/or multiprocessor scheduling. There are obvious connections to *fully dynamic* bin packing, a formulation where items may arrive or depart at discrete time intervals and the goal is to maintain the maximum number of bins required while limiting re-packing, as studied by Ivkovic and Lloyd [194]. Coffman studies bin stretching, a version of bin packing in which a bin may be stretched beyond its normal capacity [170]. Epstein studies the on-line bin stretching problem as a scheduling problem with the goal of minimizing makespan [172].

Yossi Azar has studied on-line load balancing of temporary tasks on identical machines with assignment restrictions [195, 196]. The problem therein is that of assigning incoming tasks to nodes permanently. Each task has a weight and a duration. The weight is known when the task arrives, but the duration is not known until the job completes. The goal is to minimize the maximum load on any machine over time.

Other works have explored the problem of scheduling jobs without knowledge of their processing time. The famous “scheduling in the dark” approach [197] shows that in the absence of knowledge giving equal resource shares to job is theoretically sound. Our problem is also related to thread scheduling done in operating system kernels, given that thread processing times are unknown.

5.1.2 Theoretical Difficulty of Maximum Stretch Minimization

In this section we assess the theoretical complexity of minimizing the maximum stretch. Our purpose of this section is two-fold. First, we study the clairvoyant scenario (i.e., we assume that the scheduler has full knowledge of all jobs at the outset, making the problem effectively an off-line search for a solution) so that we can derive a lower-bound on the optimal maximum stretch. Second, we quantify the difficulty of the non-clairvoyant case, as this better corresponds to the situation facing real-world algorithms. We do not consider static resource requirements or even multiple resource dimensions. Our results from Chapter 4 show that the inclusion of even a single static resource requirement is enough to make the problem NP-hard, while, as we will show in the following section, the problem poses significant theoretical challenges even with only a single fluid resource dimension. The results in Section 5.4 show that the algorithms we propose in Section 5.2 achieve performance reasonably close to the aforementioned bound.

Computation of the Theoretical Bound for the Clairvoyant Case

We refer the reader to the terminology defined in Sections 4.1 and 5.1. In this section we assume n_s (the number of fixed resource dimensions) is equal to 0 while n_d (the number of fluid resource dimensions) is equal to 1. By an abuse of notation, in the following proofs we ignore $\vec{s_j}$ and refer to $\vec{d_j}$ as d_j, a scalar quantity.

The tasks of a job are, like any program, sequences of discrete events. This means that each task will saturate the fluid resource for small amounts of time, and the fluid resource need thus represents the maximum average utilization over some interval of bounded size. Furthermore, while the tasks of a parallel job must progress together, inter-task communication events are limited in their frequency. Based on these ideas, we posit the existence a time quantum $\mathcal{Q}$. If the execution of a job j can be broken down into a sequence of non-overlapping intervals no larger than $\mathcal{Q}$ where the average fluid resource utilization does exceed the need and all tasks proceed equally on each interval, then on any particular sub-interval the fluid resource utilization of j is not constrained and the tasks of j can progress independently. We make the reasonable assumption that the time-scale for resource scheduling is much smaller than that of job scheduling, and so for any two distinct job scheduling events (particularly, job submission and completion) the distance in time between these events, if nonzero, will be larger than $\mathcal{Q}$.

Theorem 1. *A target value $\mathcal{S}$ for the maximum stretch defines a deadline $D_j = r_j + \mathcal{S} \times p_j$ for the execution of each job j. The set of job release dates and deadlines, $\{r_j\}_{j \in J} \cup \{D_j\}_{j \in J}$, defines a set of n_t non-overlapping time intervals, $I_1, ..., I_{n_t}$. We restrict our selection of $\mathcal{S}$ to values for which none of these intervals is smaller than $\mathcal{Q}$. Then there exists a valid schedule whose maximum stretch is no greater than $\mathcal{S}$ if and only if the following linear system has a solution, where β_j^t is the fraction of job j executed during the time interval I_t:*

$$\forall j \quad \sum_t \beta_j^t = 1 \tag{5.1}$$

$$\forall j, t \quad r_j \geq \sup I_t \Rightarrow \beta_j^t = 0 \tag{5.2}$$

$$\forall j, t \quad D_j \leq \inf I_t \Rightarrow \beta_j^t = 0 \tag{5.3}$$

$$\forall j, t \quad \beta_j^t p_j \leq \sup I_t - \inf I_t \tag{5.4}$$

$$\forall t \quad \sum_j \beta_j^t d_j T_j p_j \leq N\left(\sup I_t - \inf I_t\right) \tag{5.5}$$

Proof. The constraints in the given linear system state that:

- Each job must be fully processed (Constraint (5.1));
- No work can be done on a job before its release date (Constraint (5.2));

- No work can be done on a job after its deadline (Constraint (5.3));
- A task cannot run longer, during a time interval, than the length of the time interval (Constraint (5.4));
- The cumulative resources used by the different tasks during a time interval cannot exceed what is available during that interval (Constraint (5.5)).

These conditions are necessary. We now show that they suffice to insure the existence of a schedule that achieves the desired maximum stretch (i.e., there exists a schedule in which each job completes before its deadline).

From any solution of the linear system we can build a valid schedule. We show how to build the schedule for a single interval I_t, the whole schedule being obtained by concatenating all interval schedules. For each job j, any of its T_j tasks receives a cumulative resource allocation equal to $\beta_j^t p_j d_j$ during interval I_t. Let a_j^t and b_j^t be two integers such that $\frac{a_j^t}{b_j^t} = \beta_j^t p_j d_j$. Without loss of generality, we can assume that all integers b_j^t are equal to a constant b, that is $\forall j \in J, \forall t \in [1, n_t], b_j^t = b$. Let $\mathcal{R}$ be any value smaller than $\mathcal{Q}$ such that there exists an integer λ such that $(\sup I_t - \inf I_t) = \lambda \times \mathcal{R}$. Then, during each of the λ sub-intervals of I_t of size $\mathcal{R} < \mathcal{Q}$, we greedily schedule the tasks on the node in any order: starting at time 0, we first run the first task on the first node at 100% resource utilization for a time $\frac{a_{j_1}^t}{b\mathcal{R}}$, where j_1 is the job this task belongs to. Then we run the second task on the first node at 100% resource utilization for a time $\frac{a_{j_2}^t}{b\mathcal{R}}$, where j_2 is the job this task belongs to. If there is not enough remaining time on the first node to accommodate the second task, we schedule the second task on the first node for all the remaining time, and we schedule the remaining of the task on the second node starting at the beginning of the sub-interval (thanks to our assumption on task migration). We proceed in this manner until every task have been scheduled.

We now show that this schedule is valid. Note that our construction ensures that no task is run simultaneously on two different nodes. Indeed, this could only happen if, for a task of some job j, we had:

$$\frac{a_j^t}{\lambda b} > \mathcal{R} \quad \Leftrightarrow \quad \beta_j^t p_j d_j = \frac{a_j^t}{b} > \sup I_t - \inf I_t$$

which is forbidden by Constraint (5.4). Then, by construction, the resource utilization of the platform does not exceed what is available. Also, for any interval I_t there is a set of sub-intervals of size no larger than $\mathcal{Q}$, such that on on every sub-interval no task uses more than its resource need on the average and all the tasks of each job are processed equally. □

Since all variables of the linear system are rational, one can check in polynomial time whether it has a solution. Using a binary search, one can use the above theorem to find an approximation of the optimal maximum stretch. (In practice, the time quantum $\mathcal{Q}$ is small enough that the restriction on values for $\mathcal{S}$ can be safely ignored.) In fact, one can find the optimal value in polynomial time using a binary search and a version of the linear system tailored to check the existence of a solution for a range of stretch values (see [91, Section 6] for details). While the underlying assumptions in Theorem 1 are not met in practice (e.g., there are static resource constraints, migration causes overhead, job submission and run-times are not known in advance), the optimal maximum stretch computed via this theorem provides a lower bound on the optimal maximum stretch.

Competitive Ratio of The Non-Clairvoyant Case

On-line maximum stretch minimization is known to be theoretically difficult. Even in a clairvoyant scenario there does not exist any constant-ratio competitive algorithm [91]. Recall that an algorithm for an on-line problem has a competitive ratio of γ if it leads to results worse than an optimal algorithm for the off-line problem by at most a factor γ. In this work we study an on-line, non-clairvoyant scenario. However, unlike the work in [91], we consider that time-sharing of compute nodes is allowed. The question then is whether this added time-sharing capability can counter-balance the fact that the scheduler does not know the processing time of jobs when they arrive in the system, thus changing the results in [91]. In general, bounds on the competitive ratios of on-line algorithms can be expressed as a function of the number of jobs submitted to the system, or as a function of Δ, the ratio between the processing time of the largest and shortest jobs.

In this section we assume that we have one single-core node at our disposal or, equivalently, that all jobs are perfectly parallel. We show that, in spite of this simplification, the problem is very difficult (i.e., competitive ratios are large). As a result, the addition of time-sharing does not change the overall message of the work in [91].

Our first result is that the bound derived for on-line algorithms in a clairvoyant settings without time-sharing holds in our non-clairvoyant, time-sharing context.

Theorem 2. *There is no $\frac{1}{2}\Delta^{\sqrt{2}-1}$-competitive preemptive time-sharing on-line algorithm for minimizing the maximum stretch if there are at least three jobs in the instance that have distinct processing times.*

This result is valid for both clairvoyant and non-clairvoyant scenarios and is established by the proof of Theorem 14 in [91], which holds when time-sharing is allowed. Surprisingly, we were not able to increase this bound by taking advantage of non-clairvoyance. However, as seen in the next theorem, non-clairvoyance makes it possible to establish a very large bound with respect to the number of jobs.

Theorem 3. *There is no (preemptive) on-line algorithm for the non-clairvoyant minimization of max-stretch whose competitive ratio is strictly smaller than n, where n is the number of jobs submitted to the system.*

Proof. By contradiction, let us hypothesize that there exists an algorithm with a competitive ratio strictly smaller than $n - \varepsilon$ for some $\varepsilon > 0$.

We consider an instance with n jobs that are all released at time 0, with processing times large enough such that all jobs are kept running until time n regardless of what the algorithm does. The job that has received the smallest cumulative virtual time up to time n has received at most 1 unit of virtual time (one nth of the n time units). We sort the jobs in increasing order of the cumulative virtual time each has received up to time n. We construct our instance so that the i-th job in this order has processing time λ^{i-1}. Note that our assumption of what happened prior to time n is valid for any value of λ no smaller than n. The completion time of job 0, i.e., the job that has received the smallest cumulative virtual time up to time n, is at least n. Consequently, its stretch is no smaller than n because its processing time is $\lambda^0 = 1$.

A possible schedule would have been to execute jobs in order of increasing processing time. The stretch of the job of processing time λ^{i-1} would then be:

$$\frac{\sum_{j=1}^{i} \lambda^{j-1}}{\lambda^{i-1}} = \frac{\lambda^i - 1}{\lambda^{i-1}(\lambda - 1)} \xrightarrow[\lambda \to +\infty]{} 1.$$

Therefore, if λ is large enough (and greater than n) no job has a stretch greater than $1 + \frac{\varepsilon}{n}$ in this schedule. Consequently, the competitive ratio of our hypothetical algorithm is no smaller than:

$$\frac{n}{1 + \frac{\varepsilon}{n}} \geq n(1 - \frac{\varepsilon}{n}) = n - \varepsilon \ ,$$

which is a contradiction. □

The EQUIPARTITION algorithm, which gives each job an equal share of the platform, is known to deliver good performance in some non-clairvoyant settings [197]. We therefore assess its performance for maximum stretch minimization.

Theorem 4. *Let n denote the number of jobs submitted to the system and Δ the ratio between the processing times of the largest and the smallest jobs. Then, in a non-clairvoyant scenario,*

1. EQUIPARTITION *is exactly an n-competitive on-line algorithm for maximum stretch minimization;*

2. *There exists an instance for which the maximum stretch realized by* EQUIPARTITION *is at least $\frac{\Delta+1}{2+\ln(\Delta)}$ times the optimal.*

To put the performance of EQUIPARTITION into perspective, FCFS is exactly Δ-competitive [91].

Proof.

Competitive ratio as a function of n – At time t, EQUIPARTITION gives each of the $m(t)$ not-yet-completed jobs a share of the node resource equal to $\frac{1}{m(t)} \geq \frac{1}{n}$. Hence, no job has a stretch greater than n and the competitive ratio of EQUIPARTITION is no greater than n. We conclude using Theorem 3.

Competitive ratio as a function of Δ – Let us consider n jobs as follows. Jobs j_1 and j_2 are released at time 0 and have the same processing time. For the remaining jobs $j_3, \ldots, j_n$, each job j_i is released at time $r_{j_i} = r_{j_{i-1}} + p_{j_{i-1}}$. Job processing times are defined so that, using EQUIPARTITION, all jobs complete at time $r_{j_n} + n$. Therefore, job j_i is executed during the time interval $[r_{j_i}, r_{j_n} + n]$. There are two active jobs during the time interval $[r_{j_1} = r_{j_2} = 0, r_{j_3}]$, each receiving one half of the node's processing time. For any $i \in [3, n]$, there are i active jobs in the time interval $[r_{j_i}, r_{j_{i+1}}]$, each receiving a fraction $1/i$ of the processing time. Finally, there are n jobs active in the time interval $[r_{j_n}, r_{j_n} + n]$, each receiving a fraction $1/n$ of the resource available over the interval.

The goal of this construction is to have the nth job experience a stretch of n. However, by contrast with the previous theorem, the value of Δ is "small," leading to a large competitive ratio as a function of Δ, but smaller than n. Formally, to define the job processing times, we write that the processing time of a job is equal to the cumulative virtual time it is given between its release date and its deadline using EQUIPARTITION:

$$\begin{aligned}
\forall i \in [1,2] \quad p_{j_i} &= \frac{1}{2}(r_{j_3} - r_{j_1}) + \sum_{k=3}^{n-1} \frac{1}{k}(r_{j_{k+1}} - r_{j_k}) + \frac{1}{n}((r_{j_n} + n) - r_{j_n}) \\
&= \frac{1}{2}p_{j_1} + \sum_{k=3}^{n-1} \frac{1}{k} p_{j_k} + 1 \\
\forall i \in [3,n] \quad p_{j_i} &= \sum_{k=i}^{n-1} \frac{1}{k}(r_{j_{k+1}} - r_{j_i}) + \frac{1}{n}((r_{j_n} + n) - r_{j_n}) = \sum_{k=j}^{n-1} \frac{1}{k} p_{j_k} + 1
\end{aligned}$$

We first infer from the above system of equations that $p_{j_n} = 1$ (and the nth job has a stretch of n). Then, considering the equation for p_{j_i} for $i \in [3, n-1]$, we note that $p_{j_i} - p_{j_{i+1}} = \frac{1}{i} p_{j_i}$. Therefore, $p_{j_i} = \frac{i}{i-1} p_{j_{i+1}}$ and, by induction, $p_{j_i} = \frac{n-1}{i-1}$. We also have $p_{j_2} - p_{j_3} = \frac{1}{2} p_{j_1} = \frac{1}{2} p_{j_2}$. Therefore, $p_{j_2} = 2p_{j_3} = n - 1$.

Now let us consider the schedule that, for $i \in [2, n]$, executes job j_i in the time interval $[r_{j_i}, r_{j_{i+1}} = r_{j_i} + p_{j_i}]$, and that executes the first job during the time interval $[r_{j_n} + p_{j_n} = r_{j_n} + 1, r_{j_n} + n]$. With this schedule all jobs have a stretch of 1 except for the first job. The maximum stretch for this schedule is thus the stretch of the first job. The makespan of this job, i.e., the time between its release data and its completion, is:

$$\begin{aligned}
\sum_{i=1}^{n} p_{j_i} = 2p_{j_1} + \sum_{i=3}^{n} p_{j_i} = 2(n-1) + \sum_{i=3}^{n} \frac{n-1}{i-1} &= (n-1)\left(1 + \sum_{i=2}^{n} \frac{1}{i-1}\right) \\
&= (n-1)\left(1 + \sum_{i=1}^{n-1} \frac{1}{i}\right).
\end{aligned}$$

The first job being of size $n - 1$, its stretch is thus: $1 + \sum_{i=1}^{n-1} \frac{1}{i} = 2 + \sum_{i=2}^{n-1} \frac{1}{i}$. Using a classical bounding technique:

$$\sum_{i=2}^{n-1} \frac{1}{i} \leq \sum_{i=2}^{n-1} \int_{i-1}^{i} \frac{1}{x} dx = \int_{1}^{n-1} \frac{1}{x} dx = \ln(n-1).$$

The competitive ratio of EQUIPARTITION on that instance is no smaller than the ratio of the maximum stretch it achieves (n) and that of the maximum stretch of any other schedule on that instance. Therefore, the competitive ratio of EQUIPARTITION is no smaller than:

$$\frac{n}{2 + \sum_{i=2}^{n-1} \frac{1}{i}} \geq \frac{n}{2 + \ln(n-1)} = \frac{\Delta + 1}{2 + \ln(\Delta)}$$

as the smallest job —the nth one— is of size 1, and the largest ones —the first two jobs— are of size $n - 1$.

□

5.2 Algorithms

We seek to develop algorithms that perform well in terms of maximum stretch, but without any knowledge of job processing times. The theoretical results from the previous section indicate that the problem is “hopeless” in the sense that no algorithm can be designed to have a low worst-case competitive ratio because of the large number of jobs and huge differences in job run-times found in HPC workloads. Instead, we focus on developing non-guaranteed algorithms (i.e., heuristics) that perform well in practice (i.e., usually close to the off-line bound developed in Section 5.1.2). Additionally, because of the on-line nature of the problem these algorithms should compute schedules quickly. In the design and evaluation of our algorithms we restrict ourselves to a single fluid and a single fixed resource dimension that we assume, without loss of generality, to represent the CPU and memory requirements of running jobs.

Since we assume no knowledge of job processing times, it is difficult to predict how resource allocation decisions made at one moment in time will affect the final stretch values for running jobs. Existing approaches often use user-supplied run-time estimates in making resource allocation decisions, but as discussed in Section 2.1.1 these estimates are generally unreliable and their use can severely impact performance. Instead of relying on estimates or trying to build complicated predictive statistical models, we opt to focus on maximizing the minimum yield at a number of points in time. We contend that this strategy will result in low maximum stretch values, as both the yield and the stretch capture notions of job "happiness", and are thus related. In fact, the yield can be seen as the inverse of an instantaneous stretch.

Our basic approach is to consider a number of "scheduling events" over the course of time and to try to maximize the minimum yield of the running jobs at each scheduling event using a subset of the algorithms discussed in Chapter 4. Since it is possible that at times the total resource demand of waiting jobs may overwhelm what is available, we also define a notion of job priority in order to allow some jobs to be temporarily preempted and removed from consideration.

We present first two approaches for mapping tasks to compute nodes in a way that optimizes the minimum yield: a local optimization heuristic (Section 5.2.1) and a global optimization heuristic (Section 5.2.2). The efficient use of both these approaches mandates the introduction of a priority function (Section 5.2.3). We then specify when task mapping algorithms should be invoked (Section 5.2.4) and how to derive resource allocations from task mappings (Section 5.2.5). All the above algorithms optimize minimum yield in the hope of achieving good job stretches. For comparison purposes, in Section 5.2.6 we propose an algorithm that attempts to optimize the maximum stretch directly.

5.2.1 Greedy Task Mapping

The basic Greedy algorithm allocates nodes to an incoming job j without interfering with tasks of other jobs that may currently be running. It first identifies the nodes that have sufficient available memory to run at least one task of job j. For each of these nodes it computes its *CPU load* as the sum of the CPU needs of all the tasks currently allocated to it. It then assigns one task of job j to the node with the lowest CPU load, thereby picking the node that can lead to the optimal yield for the task. If after this allocation that node no longer has sufficient remaining memory to accommodate another task of job j, it is removed from consideration. If necessary memory resources are available, all tasks of the job j are allocated to nodes in this manner. This is roughly analogous to the process used by the GREEDY_S1_P1 and GREEDY_S1_P2 algorithms in the previous chapter. That is, tasks are assigned to nodes in essentially random order (since the algorithm has no control over the order in which jobs are submitted) and it always places tasks on nodes that have the lowest load in the single fluid resource dimension under consideration.

A clear weakness of the basic Greedy algorithm is its admission policy. If a short-running job is submitted to the cluster but cannot be executed immediately due to memory constraints, then it is postponed. However, since we assume no knowledge of job processing time, there is no way to correlate how long a job is postponed with its processing time. In fact, a job could be postponed for an arbitrarily long period of time, leading to unbounded maximum stretch. The only way to circumvent this problem is to force the admission of all newly submitted jobs. This can be accomplished by pausing (via preemption) and/or moving (via migration) tasks of one or more jobs that are currently running. A method for selecting which jobs should be paused or moved, which relies on a notion of job priority, is described in Section 5.2.3.

5.2.2 Task Mapping as Vector Packing

The Greedy approach described in the previous section is incremental. It builds a solution through a succession of locally optimal decisions for each task, but the final solution may be far from being globally optimal. Another approach is to compute a global solution from scratch, and then preempt and/or migrate tasks in order to implement the resulting task mapping. We apply one of the vector-packing based algorithms described in Chapter 4 for this purpose. The VP_CPSUM algorithm was previously identified as the best of these algorithms in general. However, the difference in performance between VP_CPSUM and VP_CPMAX is small, and our previous research [184, 185] had led us to believe that VP_CPMAX had a slight edge in performance for a single fixed and single fluid resource dimension. Thus, the vector packing task placement algorithm chosen for this research study was VP_CPMAX, but in principle any vector packing algorithm could be used. We refer to the vector packing task mapping approach simply as MCB (short for "Multi-Capacity Bin-Packing", another name for vector packing).

In the event that the MCB algorithm cannot find a valid allocation for all of the jobs currently in the system at any yield value, it should remove some jobs from consideration and try again. The selection of which jobs to remove is based on the same job priority notion as that used by the greedy approach to select jobs to pause or move, and is described in the next section.

5.2.3 Prioritizing Jobs

When the system is oversubscribed, we use a priority function to decide which jobs to pause/move, when using Greedy, and which jobs to remove from consideration, when using MCB. This priority function is also used to decide which of the jobs that are currently paused should be restarted when more resources become subsequently available.

An intuitive choice for the priority function would be to simply take the inverse of the virtual time (as described in Section 5.1): then the shorter the virtual time, the higher the priority. A job that has not yet been allocated any CPU time has a zero virtual time, i.e., an infinite priority. This ensures that no job is left waiting at its release date, especially short jobs whose stretch would otherwise degrade the overall performance. This rationale for using the inverse of the subjective time experienced by the job as a priority function is similar to that found in [198].

Experimental results, not presented here, show that using the inverse of the virtual time as a priority function leads to good performance. Unfortunately, there is a problem with using a priority based solely on the virtual time: The virtual time of a paused job remains constant, which can lead to starvation. Thus, the priority function should also consider the current *flow time* of a job, i.e., the time elapsed since its submission. The goal is to prevent starvation by ensuring that the priority of any paused job increases with time and tends to infinity.

Based on these ideas, we considered using the ratio of flow time to virtual time as a priority function, but preliminary experiments showed that this leads to significantly poorer performance in practice than not using the flow time at all. As a consequence, we define the priority function as: $\text{priority} = \frac{\text{flow time}}{(\text{virtual time})^2}$. The power of two in the denominator increases the importance of the virtual time with respect to the flow time, thereby giving an advantage to short-running jobs, while still escalating the priority of jobs that are paused for long periods of time. When necessary, we break ties between equal-priority jobs by considering their order of submission.

We always consider jobs for pausing or moving, and we always remove jobs from consideration for scheduling, by increasing order of priority. Conversely, we always consider jobs for resuming by decreasing order of priority. The use of the priority function is mandatory to ensure that MCB always finds a valid allocation. In the event that the MCB algorithm cannot find a valid allocation for all of the considered jobs at any yield value, the lowest priority job (among those considered) is removed from consideration and MCB is called on the new set of jobs. The use of the priority function is not mandatory for Greedy but enables us to define two enhanced variants. The GreedyP algorithm is like Greedy except that if an incoming job cannot be started then some of the running jobs are paused. To do so, this algorithm goes through the list of currently running jobs in order of increasing priority and marks them as candidates for pausing until the incoming job could be started if all these candidates were indeed paused. It then goes through the list of these marked jobs in decreasing order of priority and determines for each whether it could instead be left running due to sufficient available memory. After this step, running jobs that are still marked as candidates for pausing are paused, and the new job is started. The GreedyPM algorithm further extends the GreedyP algorithm with the capability of moving rather than pausing running jobs. This is done by going through the list of jobs to be paused in decreasing order of priority and trying to reschedule them using the Greedy algorithm.

5.2.4 When to Compute New Task Mappings

So far, we have not stated *when* our task mapping algorithms should be invoked. The most obvious choice is to apply them each time a new job is submitted to the system and each time some resources are freed due to a job completion. The MCB algorithm attempts a global optimization and, thus, can (theoretically) "reshuffle" the whole mapping each time it is invoked[1]. One may thus fear that applying MCB on each submission could lead to a prohibitive number of preemptions and migrations. By contrast, Greedy has low overhead and the addition of a new job should not be overly disruptive to currently

[1] In practice this does not happen because the algorithm is deterministic and always considers the tasks and the nodes in the same order.

running jobs, particularly if they do not share any nodes, but exclusive use of Greedy can generate allocations that use cluster resources inefficiently. For both these reasons we experiment with a periodic use of MCB. More specifically, we consider algorithms that:

- upon job submission, either do nothing or apply Greedy, GreedyP, GreedyPM or MCB;
- upon job completion, either do nothing or apply Greedy or MCB;
- apply or do not apply MCB periodically.

We use a multi-part scheme for naming our algorithms, using '/' to separate the parts. The first part corresponds to the policy used for scheduling jobs upon job submission, followed by a "*" if the jobs are also scheduled opportunistically upon job completion (using MCB if MCB was used on job submission, and using Greedy if Greedy, GreedyP, or GreedyPM was used upon job submission). If the algorithm applies MCB periodically, the keyword "per" is added as a second part to the name of the algorithms. For example, the GreedyP*/per algorithm performs a Greedy allocation with preemption upon job submission, opportunistically tries to start currently paused jobs using the Greedy algorithm whenever a job completes, and periodically applies the MCB algorithm to balance the overall workload. We consider all combinations listed in Table 5.1 (the last algorithm in the table is described in Section 5.2.6).

The MCB algorithm has the opportunity to re-map any running job. This is a strength, as it enables to attempt a global optimization. This is also a weakness as it can pause and/or move a job that has just started, which can induce a large stretch on a short job. To mitigate this behavior, we introduce two parameters that attempt to minimize unfavorable migrations initiated by MCB. If set, the MFT parameter (respectively, the MVT parameter), stipulates that jobs whose flow-times (resp., virtual times) are smaller than a given bound may be paused in order to run higher priority jobs, but, if they continue running, their current node mapping must be maintained. Jobs whose flow-times (resp., virtual times)

Table 5.1. On-line scheduling algorithms

Name	Action on submission	Action on completion	Periodic action
Greedy*	Greedy	Greedy	none
GreedyP*	GreedyP	Greedy	none
GreedyPM*	GreedyPM	Greedy	none
Greedy/per	Greedy	none	MCB
GreedyP/per	GreedyP	none	MCB
GreedyPM/per	GreedyPM	none	MCB
Greedy*/per	Greedy	Greedy	MCB
GreedyP*/per	GreedyP	Greedy	MCB
GreedyPM*/per	GreedyPM	Greedy	MCB
MCB*	MCB	MCB	none
MCB/per	MCB	none	MCB
MCB*/per	MCB	MCB	MCB
/per	none	none	MCB
/stretch-per	none	none	MCB-stretch

are greater than the specified bound may be moved as previously. Migrations initiated by the GreedyPM algorithm are not affected by these parameters. This algorithm uses migrations to allow the execution of higher priority jobs. The result of applying these bounds would be that those jobs which would be migrated would instead be paused in favor of running lower priority jobs that had previously been paused. The use of the MFT or MVT parameter is indicated by an additional part in the algorithm name (e.g., MCB*/per/MFT=300).

5.2.5 Resource Allocation

Once tasks have been mapped to nodes it is necessary to decide on appropriate CPU allocations for each job (recall that all tasks in a job are given identical CPU allocations). All previously described algorithms use the following procedure: First all jobs are assigned yield values of $1/\max(1,\Lambda)$, where Λ is the maximum CPU load over all nodes. This maximizes the minimum yield given the current mapping of tasks to nodes. After this step there may be remaining CPU resources on at least some of the nodes that can be used for further improvement without changing the mapping of tasks to nodes. We use two different approaches to exploit remaining resource fractions.

Average Yield Optimization

Once the task mapping is fixed and the maximized minimum yield computed, we can write a rational linear program to find the resource allocation that maximizes the average yield under the constraint that no job is given a yield lower than the maximized minimum. This approach was previously discussed in Section 4.4.7. Algorithms that use this second-phase optimization procedure have "OPT=AVG" as an additional part of their names.

Max-min Yield Optimization

As an alternative to maximizing the average yield under the given constraints, we also consider an approach that iteratively maximizes the minimum yield. At each step the minimum yield is maximized using the procedure described at the beginning of this section. Those jobs whose yield cannot be further improved because of memory resource constraints are removed from consideration, and the minimum is further improved for the remaining jobs. This process continues until there are no more jobs that can be improved. While this algorithm may not do as good a job of maximizing resource utilization as average yield optimization, it can be argued that it is more fair. This type of max-min optimization is commonly used to allocate bandwidth to competing network flows [199, Chapter 6]. Algorithms that use this second-phase optimization procedure have "OPT=MIN" as an additional part of their names.

5.2.6 Optimizing the Stretch Directly

All algorithms described thus far optimize the minimum yield as a way to optimize the maximum stretch. We validate this approach by developing and comparing with an algorithm that attempts to minimize the maximum stretch directly. In lieu of providing the algorithm with knowledge of job processing times we instead restrict it to the periodic case so that it can have knowledge of the time between scheduling events. Thus, this algorithm, which we call /stretch-per, should be directly comparable with /per. /stretch-per uses a multi-capacity or vector bin packing approach called MCB-stretch that is similar to MCB, but with the following differences: At scheduling event i, since we assume no knowledge of job processing times, the best estimate of the stretch of job j is the ratio of its flow time (time since submission) to its virtual time: $\hat{S}_j(i) = flowtime_j(i)/vt_j(i)$. Assuming that the job continues running until scheduling event $i + 1$, then $\hat{S}_j(i + 1) = flowtime_j(i + 1)/vt_j(i + 1) = (flowtime_j(i) + T)/(vt_j(i) + \alpha_j(i) \times T)$, where T is the scheduling period and $\alpha_j(i)$ is the yield that MCB-stretch assigns to job j between scheduling events i and $i + 1$. Similar to the binary search on the yield, here we do a binary search to minimize $\hat{S}(i + 1) = \max_j \hat{S}_j(i + 1)$. At each iteration of the binary search, a target value $\hat{S}(i + 1)$ is tested. From this value the algorithm computes the yield for each job j by solving the above equation for $\alpha_j(i)$ (if, for any job, $\alpha_j(i) > 1$, then the target value $\hat{S}(i + 1)$ is infeasible and the iteration fails). At that point, CPU requirements are defined and the vector-packing algorithm can be applied to try to produce a task-mapping for the attempted $\hat{S}(i + 1)$ value. This is repeated until the lowest feasible such value is found. Note that since the stretch is an unbounded positive value, the algorithm actually performs a binary search over the inverse of the stretch, which is between 0 and 1. If the MCB-stretch algorithm cannot find a valid allocation for any value of estimated stretch, then the job of lowest priority is removed from consideration and the search is re-initiated for the new set of jobs.

Once a mapping of jobs to nodes has been computed each task is initially assigned a CPU allocation exactly equal to the amount of resources it needs to reach the desired stretch. For the resource allocation improvement phase we use algorithms similar to those in Section 5.2.5, except that the first (OPT=AVG) seeks to minimize the average stretch and the second (OPT=MAX) iteratively minimizes the maximum stretch. They are analogous to their yield-based counterparts.

5.3 Experimental Methodology

5.3.1 Discrete Event Simulator

We have developed a discrete event simulator that implements our scheduling algorithms and takes as input a number of nodes and a list of jobs. Each job is described by a submit time, a required number of tasks, one CPU need and one memory requirement specification (since all tasks within a job have the same needs and requirements), and a processing time. Jobs are allocated shares of memory and CPU resources on simulated compute nodes. As stated previously in Section 3.2.1, the use of VM technology allows the CPU resources of a (likely multi-core) node to be shared precisely and fluidly as a single resource [136]. Thus, for each simulated node, the total amount of allocated CPU resource is simply constrained to be less than or equal to 100%. However, when simulating a multi-core node, then 100% CPU resource utilization can be reached by a single task only if that task is CPU-bound and is implemented using multiple threads (or processes). A CPU-bound sequential task can use at most $100/n\%$ of the node's CPU resource, where n is the number of processor cores.

The question of properly accounting for preemption and migration overheads is a complicated one. For this reason we provide two versions of each simulation experiment:

one where the overhead is zero and one where this overhead is 5 minutes of wall clock time, whatever the job's characteristics and the number of its tasks being migrated, which is justifiably high [2]. We call this overhead the *rescheduling penalty*. In the real world there are facilities available that can allow for the live migration of a running task between nodes [144], but in order to avoid introducing additional complexity we make the pessimistic assumption that all migrations are carried out through a pause/resume mechanism. Experiments on real systems have shown that the use of live migration can cut the rescheduling penalty to approximately 20 seconds [80].

Note that none of the scheduling algorithms are aware of the rescheduling penalty or try to schedule around it. Based on preliminary results, we opt for a default period equal to twice the rescheduling penalty for all periodic algorithm, i.e., 10 minutes. The MFT and MVT parameters for MCB and MCB-stretch are evaluated using time bounds equal to one-half penalty and one penalty (i.e., 300s and 600s).

We consider two batch scheduling algorithms: FCFS and EASY. FCFS (First Come First Serve), often used as a baseline comparator in the literature, holds incoming jobs in a queue and assigns them to nodes in order as nodes become available. EASY [17], which is representative of production batch schedulers, is similar to FCFS but enables backfilling to reduce resource fragmentation. EASY gives the first job in the queue a reservation for the earliest possible time it would be able to run with FCFS, but other jobs in the queue are scheduled opportunistically as nodes become available, as long as they do not interfere with the reservation for the first job. EASY thus improves on FCFS by allowing small jobs to run while large jobs are waiting for a sufficiently large number of nodes. A drawback of EASY is that it requires estimations of job processing

[2] Consider a 128-task job with 1 TB total memory, or 8 GB per task (our simulations are for a 128-node cluster). Current technology affords aggregate bandwidth to storage area networks up to tens of GB/sec for reading and writing [200]. Conservatively assuming 10 GB/sec, moving this job between node memory and secondary storage can be done in under two minutes.

times. In all simulations we conservatively assume that EASY has perfect knowledge of job processing times. While this seems a best-case scenario for EASY, studies have shown that for some workloads some batch scheduling algorithms can, surprisingly, produce better schedules when using non-perfectly accurate processing times (e.g., using inaccurate user-provided estimates, multiplying the perfectly accurate estimate by a certain factor). We refer the reader to the discussion in [151] for more details. At any rate, in those studies the potential advantage of using inaccurate estimates is shown to be relatively small, while our results show that our approach outperforms EASY by orders of magnitude. Our conclusions thus still hold when EASY uses non-accurate processing time estimates.

5.3.2 Workloads

Real-World Workload

We perform experiments using a real-world workload from a well-established on-line repository [33]. Most publicly available logs provide standard information such as job arrival times, start time, completion time, requested duration, size in number of nodes, etc. For our purpose, we need to quantify the fraction of the resource allocated to jobs that are effectively used. We selected the “cleaned” version of the HPC2N workload [201] from [33], which is a 182-week trace from a 120-node dual-core cluster running Linux that has been scrubbed to remove workload flurries and other anomalous data that could skew the performance comparisons on different scheduling algorithms [202]. A primary reason for choosing this workload was that it contains almost complete information regarding memory requirements, while other publicly available workloads contain no or little such information.

The HPC2N workload required some amount of processing for use in our simulation experiments. First, job per-processor memory requirements were set as the maximum of either requested or used memory as a fraction of the system memory of 2GB, to a minimum of 10%. Of the 202,876 jobs in the trace, only 2,142 ($\sim$ 1%) did not give values for either used or requested memory and so were assigned the lower bound. Second, the swf file format [33] contains information about the required number of "processors," but not the number of tasks, and so this value had to be inferred. For jobs that required an even number of processors and had a per-processor memory requirement less than 50% of the available node memory, we assumed that the job used a number of multi-threaded tasks equal to half the number of processors. In this case, we assume that each task has a CPU need of 100% (saturating the two cores of a dual-core node) and the memory requirement was doubled from its initial per-processor value. For jobs requiring an odd number of processors or more than 50% of the available node memory per processor, we assumed that the number of tasks was equal to the number of processors and that each of these tasks had a CPU need of 50% (saturating one core of a dual-core mode). Since we assume CPU-bound tasks, performance degradation due to CPU resource sharing will be maximal. Consequently, our assumptions are detrimental to our approach and should benefit batch scheduling algorithms. We split the HPC2N workload into week-long segments, resulting in 182 different experimental scenarios

Synthetic Workloads

We also use synthetic workloads based on the model by Lublin et. al. [203], augmented with additional information as described hereafter. There are a number of reasons for preferring synthetic workloads to real workloads for this type of relative performance evaluation: Real workloads are often of poor quality, and often do not contain all of the information that we require. Further, real traces may be misleading, or lacking in critical information, such as system down times that might affect jobs running on the system [203]. Also, a real workload trace only provides a single data point, and may not be generally representative [204]. That is, the idiosyncrasies of a trace from one site may make it inappropriate for evaluating or predicting performance at another site [205]. Further, a real workload trace is the product of an existing system that uses a particular scheduling policy, and so may be biased or affected by that policy, while synthetic traces can provide a more neutral environment [205]. Finally, real workloads may contain spurious or anomalous events like user flurries that can confound evaluations of the performance of scheduling algorithms [202, 206]. In fact, long workload traces often contain such events, and including them can seriously impact relative performance evaluation [207].

For the synthetic workloads we arbitrarily assume quad-core nodes, meaning that a sequential task would use at most 25% of a node's CPU resource. Due to the lack of real-world data and models regarding the CPU needs of HPC jobs, we make pessimistic assumptions similar to those described in the previous section. We assume that the task in a one-task job is sequential, but that all other tasks are multi-threaded. We assume that all tasks are CPU-bound: CPU needs of sequential tasks are 25% and those of other tasks are 100%.

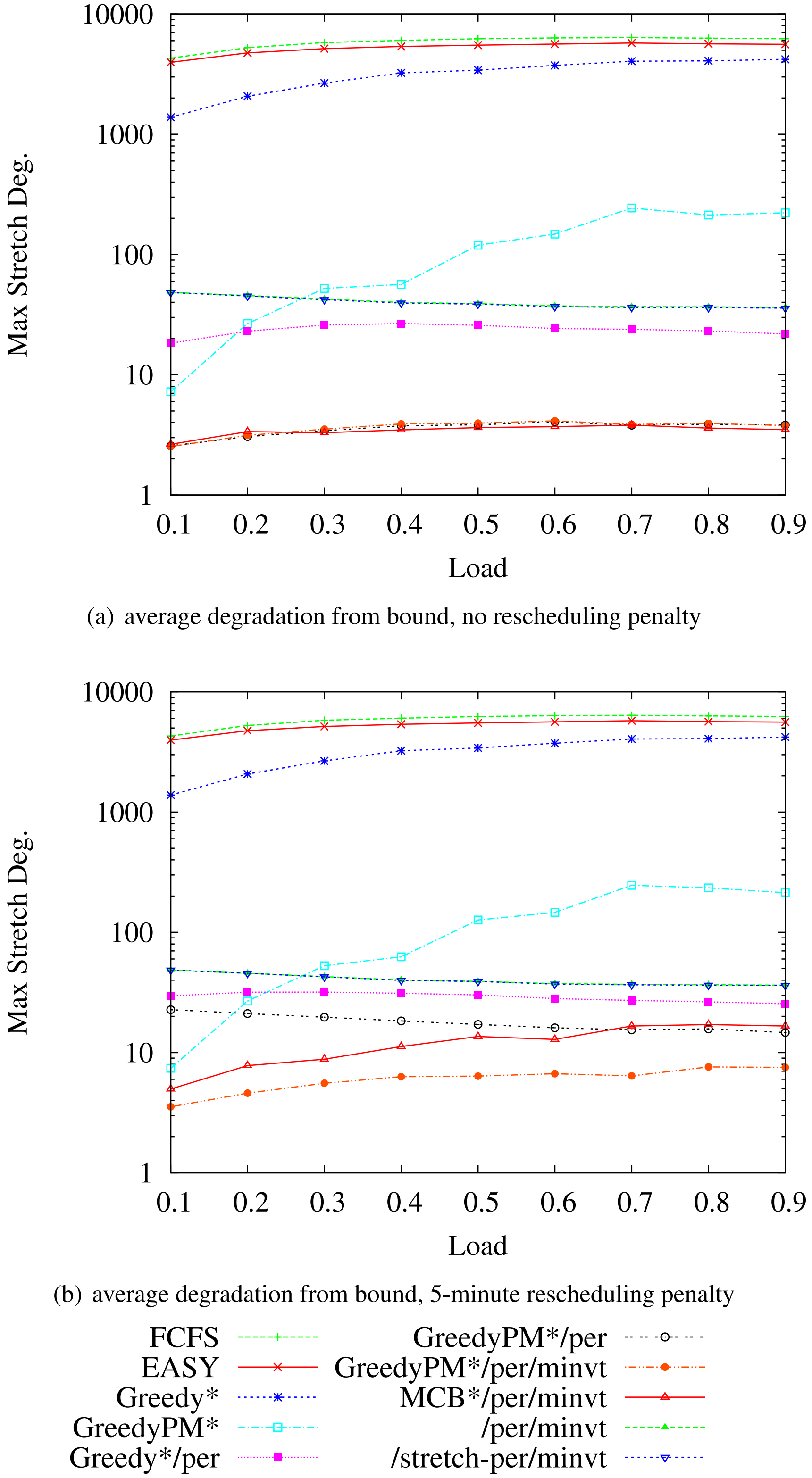
The general consensus is that there is ample memory available for allocating multiple HPC job tasks on the same node [35, 37–39], but no explicit model is available in the literature. We opt for a simple model suggested by data in Setia et. al. [36]: 55% of the jobs have tasks with a memory requirement of 10%. The remaining 45% of the jobs have tasks with memory requirements $10 \times x\%$, where x is an integer value uniformly distributed over {2,. . . ,10}.

We generated 100 distinct traces of 1,000 jobs using the Lublin model [203] and annotated them with CPU needs and memory requirements as described. The generated traces assume a 128-node cluster and thus contain jobs with between 1 and 128 tasks. Generally the time between the submission of the first job and the submission of the last job is on the order of 4-6 days. Next, in order to provide a way to systematically study how different algorithms perform on problems with different levels of difficulty, we multiplied the inter-arrival times of jobs in each generated trace by 9 computed constants in order to create 9 new traces with identical job mixes but offered load [37], or *load*, levels of 0.1 to 0.9 in increments of $.1$. Thus, from the 100 initial traces we created 900 *scaled* traces.

5.4 Experimental Results

For a given problem instance, and for each algorithm, we define the *degradation from bound* as the ratio between the maximum stretch achieved by the algorithm on the instance and the theoretical lower bound on the optimal maximum stretch obtained in Section 5.1.2. A value of x means that the algorithm achieves a maximum stretch equal to x times the theoretical lower bound, so lower values denote better performance.

Recall that Table 5.1 lists 14 general combinations of mechanisms for mapping tasks to processors. All these combinations can use either OPT=AVG or OPT=MIN to compute resource allocations once a mapping has been determined. Furthermore, the last 11 combinations in Table 5.1 use the MCB algorithm and thus can use MFT=300, MFT=600, MVT=300, MVT=600, or no mechanism to limit task remapping. Therefore, the total number of potential algorithms is $3 \times 2 + 11 \times 2 \times 5 = 116$. However, the full results (available in the Appendix B in Tables B.1, B.2, and B.3) show that on average OPT=MIN is never worse and often slightly better than OPT=AVG. Consequently, we present here results only for algorithms that use OPT=MIN. Furthermore, we found that among the mechanisms for limiting task remapping, MVT is always better than MFT, and slightly



(a) average degradation from bound, no rescheduling penalty

(b) average degradation from bound, 5-minute rescheduling penalty

Figure 5.1. Average degradation from bound vs. load for selected algorithms on the scaled synthetic dataset. Each data point shows average values over 100 instances.

better with the larger 600s bound. Accordingly we also exclude algorithms that use MFT, and algorithms that use MVT with a 300s bound. These exclusions reduce the number of algorithms under consideration to $3 + 11 \times 2 = 25$.

Figure 5.1 shows plots of average degradation factors vs. load for selected algorithms when applied to scaled synthetic workloads, using a logarithmic scale on the y-axis, for both the no migration penalty and 5-minute migration penalty cases. As plotting the data for all of the 25 remaining algorithms would be impractical, we instead choose to plot results for a subset of representative algorithms to show how the addition of each of our policies and mechanisms affects performance in both the no-migration-penalty and 5-minute migration penalty case. Figure 5.1(a) shows the results for the ideal case in which there is no rescheduling penalty, and Figure 5.1(b) shows results for the case where there is a pessimistic 5-minute rescheduling penalty. Algorithm names have been shortened because of space constraints and the plotted algorithms all use the OPT=MIN second phase optimization strategy and assume MVT=600 if MVT is specified.

The first observation from these figures is that one can design algorithms that achieve maximum stretch values several orders of magnitude lower than EASY, even under pessimistically high rescheduling penalties. Furthermore, the achieved values are, on average, surprisingly low. Indeed, whatever the system load, our best algorithms are always within a factor of seven on the average from a theoretical lower bound that ignores both memory constraints and rescheduling penalties. Furthermore, while the algorithms are executed in an on-line, non-clairvoyant context, the computation of the bounds relies on knowledge of both the release dates and processing times of all jobs. Expectedly, the addition of the 5-minute rescheduling penalty has a large negative impact on the performance of the algorithm that uses MCB upon job submission and completion submission (MCB*/per), though it is still about half an order of magnitude better than the batch scheduling algorithms. This is due to job thrashing induced by the constant re-

Table 5.2. Average degradation from bound results for the real-world trace, unscaled synthetic traces, and scaled synthetic traces for selected algorithms and assuming a 5-minute rescheduling penalty.

Algorithms	Real-world	Unscaled syn.	Scaled syn.
FCFS	3,578.5	5,457.2	5,869.3
EASY	3,041.9	4,955.4	5,262.0
Greedy*/OPT=MIN	949.8	2,435.0	3,204.3
GreedyP*/OPT=MIN	13.5	37.5	115.7
GreedyPM*/OPT=MIN	13.8	33.8	124.0
Greedy/per/OPT=MIN	28.3	30.1	29.3
GreedyP/per/OPT=MIN	18.5	20.1	17.8
GreedyPM/per/OPT=MIN	18.4	20.2	17.9
Greedy*/per/OPT=MIN	24.3	30.4	29.1
GreedyP*/per/OPT=MIN	17.9	20.3	17.9
GreedyPM*/per/OPT=MIN	17.9	20.3	17.9
GreedyP/per/OPT=MIN/MVT=600	8.9	5.9	7.3
GreedyPM/per/OPT=MIN/MVT=600	8.8	5.9	7.3
GreedyP*/per/OPT=MIN/MVT=600	6.9	4.9	6.1
GreedyPM*/per/OPT=MIN/MVT=600	6.9	4.8	6.1
MCB*/OPT=MIN/MVT=600	12.0	6.9	13.2
MCB/per/OPT=MIN/MVT=600	10.8	8.1	11.0
MCB*/per/OPT=MIN/MVT=600	13.6	7.8	12.2
/per/OPT=MIN/MVT=600	105.0	43.0	40.4
/stretch-per/OPT=MAX/MVT=600	105.0	43.0	40.2

mapping of tasks for running jobs. The overall conclusion from these results is that, to achieve good performance, all our techniques should be combined: an aggressive greedy job admission policy, a periodic use of the MCB vector-packing task mapping algorithm, an opportunistic use of resources freed upon job completion, and a grace period that prevents migration of jobs that have just begun executing. Hereafter only results including the 5-minute rescheduling penalty are presented.

To cross-validate this conclusion we now study the performance results for all experiments using the real-world HPC2N trace, the original unscaled synthetic traces directly from the Lublin model, and the scaled synthetic traces (i.e., the results presented in Figure 5.1(b)). Table 5.2 presents average degradation from bound values for all 9 greedy combinations with no mechanism for limiting remapping, and results for 4 selected greedy combinations with MVT=600, as explained hereafter. It also presents results for the 3 MCB combinations, for the /per algorithm, and for the /stretch-per algorithm. All these are with MVT=600. We leave results without MVT=600 for the 3 MCB combinations out of the table because these algorithms perform very poorly. Because they apply MCB upon each job arrival, they lead to inordinate amounts of remapping and thus are more than one order of magnitude further away from the bound than when MVT=600 is used. For /per and /stretch-per, the addition of MVT=600 has little effect since the scheduling period is no shorter than 600s. We are left with the 18 algorithms in the table, which we discuss hereafter. Data for all algorithms, including additional statistical information, such as standard deviations and maximum values are available in Appendix B in Tables B.1, B.2, and B.3.

The results in Table 5.2 are mostly consistent across our three sets of traces, with a few exceptions. As in Figure 5.1(b), we see that EASY outperforms FCFS, while our proposed algorithms outperform EASY by several orders of magnitude, thereby showing that DFRS is an attractive alternative to batch scheduling.

The table shows results for 4 groups of greedy algorithms. In the first group of 3 are algorithms that do not apply MCB periodically (i.e., those without “per " in their names). On average, these algorithms lead to results poorer than the 6 algorithms in the next 2 groups, which do apply MCB periodically, for our synthetic traces. For real-world traces, we see that they instead lead to results better than that of the other 6 algorithms. However, examining the comprehensive tables from the appendix shows that they lead to standard deviation and maximum values that are orders of magnitude larger. Finally, we see that these algorithms are outperformed by the algorithms in the last set of greedy algorithms, even for the real-world traces. We conclude that applying MCB periodically is beneficial for greedy algorithms. The results also show that GreedyP is better than Greedy, demonstrating that the use of preemption is indeed beneficial. However, the use of GreedyPM does not lead to significant further improvement and can lead to a slight decrease in performance. It turns out that the jobs migrated in the GreedyPM approach are often low priority and thus have a high probability of being preempted at an upcoming scheduling event. Finally, for the first 3 groups of greedy algorithms, we see that scheduling jobs opportunistically (i.e., as done by algorithms with * in their names) also seems to have limited impact on the performance.

The last group of 4 greedy algorithms shows results for the GreedyP and GreedyPM approaches, with or without opportunistic scheduling, but with the MVT=600 feature to limit task remapping. We see that these algorithms outperform all previously discussed algorithms. For these algorithms the use of opportunistic scheduling leads to improvements for all 3 sets of traces. In this case, GreedyP and GreedyPM leads to similar results and are the best greedy algorithms overall, including in terms of standard deviation.

The 3 algorithms in the next group all use MCB to assign tasks to nodes rather than a greedy approach. They all use MVT=600 because without limiting task remapping they all lead to performance poorer by orders of magnitude due to job thrashing. Overall, while these algorithms perform very well, they are outperformed by the best greedy algorithms.

The next algorithm, /per/OPT=MIN/MVT=600, simply applies MCB periodically without taking action upon job arrival or job completion, and is outperformed by the best greedy algorithm more than 5-fold. This result confirms the notion that a scheduling algorithm must react to job submissions. The algorithm in the last row of the table optimizes the stretch directly. It performs more than one order of magnitude worse than our best yield-based algorithm. This demonstrates that yield optimization is a good approach and that a better algorithm for minimizing the stretch directly may not be achievable.

Our overall conclusion from the results in Table 5.2 is again that, to achieve good performance, all our techniques should be combined: an aggressive greedy job admission policy, a periodic use of the MCB vector-packing algorithm, an opportunistic use of resources freed upon job completion, and a grace period that prevents remapping of tasks that have just begun executing. While our DFRS algorithms are executed in an on-line, non-clairvoyant context, the computation of the bound on the optimal performance relies on knowledge of both the release dates and processing times of all jobs. Furthermore, the bound ignores memory constraints. Nevertheless, in our experiments, our best algorithms are on average at most a factor of 7 away from this loose bound. We conclude that our algorithms lead to good performance in an absolute sense.

Our simulations show that the use of preemption and/or migration, at least when used in conjunction with adequate algorithms, provides benefit in terms of stretch even with a high rescheduling penalty. One may wonder, however, whether the running time of these algorithms would not be prohibitive in practice. To investigate this question we instrumented MCB to record the time required for each scheduling event. We ran the simulator using the MCB* algorithm on a system with a 3.2GHz Intel Xeon CPU and 4GB RAM for the 100 unscaled traces generated by the Lublin model, which resulted in a total of 197,808 observations. For 67.25% of these observations MCB computed allocations for 10 or fewer jobs in less than 0.001 seconds. The remaining observations were for 11 to 102 jobs. The average compute time was about 0.25 seconds with the maximum under 4.5 seconds. Since typical job inter-arrival times are orders of magnitude larger [203], we conclude that our best algorithms can be used in practice.

5.5 Conclusion

In this chapter we studied the on-line scenario for Dynamic Fractional Resource Scheduling. We examined its theoretical difficulty in both the clairvoyant and non-clairvoyant scenarios, devised an algorithm for computing a lower bound on the maximum stretch in an idealized scenario, proposed several DFRS algorithms and compared them to standard batch scheduling approaches through simulations using both real-world and synthetic workloads. Our algorithms were given no knowledge of job run-times, while batch scheduling algorithms were provided with perfect estimates. We found that several DFRS algorithms lead to dramatic improvement over batch scheduling in terms of an established metric of schedule quality, the maximum (bounded) stretch. We found that to achieve the best performance it is necessary to combine a number of different approaches: periodic re-allocation of resources by an intelligent vector-packing algorithm, greedy scheduling with preemption for newly submitted jobs, opportunistic greedy scheduling of paused jobs when resources become available, and a restriction on initiating migrations for jobs that have not yet progressed beyond a threshold amount of virtual time. While our algorithms were executed in an on-line, non-clairvoyant context, we found that, even when a heavy performance penalty was assumed for initiating job preemption and migration, the best of them were less than a factor of seven away on the average from a theoretical bound that assumed knowledge of job run-times times, penalty-free migration, and infinite system memory.

One issue is that the consumption of network bandwidth (while it should be within the capacity of modern systems as explained in Section 5.3.1) could potentially be of concern to some administrators. In the Chapter 6 we explore some of the trade-offs between bandwidth consumption and performance and evaluate techniques to help reduce that consumption when necessary.

Preliminary results upon which the work in this chapter was based were previously published in the *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium* [208]. A new paper has been written based on the results of the experiments presented in this chapter and will be submitted for publication in the *IEEE Transactions on Parallel and Distributed Systems* [209].

CHAPTER 6
THE ADAPTATION PROBLEM

In Chapter 5 we approached the on-line resource allocation problem as a sequence of instances of the off-line problem, which allowed us to apply some of the off-line algorithms we developed in Chapter 4. Our results indicated that this approach generally leads to good maximum stretch values–within a factor of about 7 from a loose theoretical bound on the average, and several orders of magnitude better than FCFS and EASY batch scheduling. While we showed in Section 5.3.1 that the bandwidth requirements of our approach should be within the capacity of a modern cluster system, network usage may still be a concern for some administrators. In this chapter we attempt to address the problem of bandwidth utilization more directly. More specifically, we make the following contributions: (i) We evaluate the average bandwidth utilized by our approach in Chapter 5 when applied to heavy loads (Section 6.1); (ii) We define the problem for the general case of explicitly bounding the bandwidth used for migration at each resource allocation event (Section 6.2); (iii) We establish the complexity of the problem (Section 6.2.1); (iv) We provide a mixed-integer linear program formulation that can be used to find an optimal solution for any instance of the general problem (Section 6.2.2); and (v) We study the effect of varying the rescheduling period on both the performance and bandwidth consumption for our best-performing algorithm from Chapter 5 on real and synthetic workloads (Section 6.3).

6.1 Bandwidth Consumption for Previous Results

Table 6.1 shows the bandwidth consumption for the algorithms discussed in Table 5.2 in Section 5.4 (results for all algorithms can be found in Appendix B in Table B.4) using our synthetic scaled trace data, and only for traces with load levels 7 or higher (i.e., high-

Table 6.1. Preemption and migration bandwidth costs for selected algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Bandwidth consumption (GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
Greedy*/OPT=MIN	0.00	0.00	0.00	0.00
GreedyP*/OPT=MIN	0.06	0.17	0.00	0.00
GreedyPM*/OPT=MIN	0.03	0.07	0.02	0.05
Greedy/per/OPT=MIN	0.48	1.08	0.21	0.60
GreedyP/per/OPT=MIN	0.50	1.11	0.20	0.60
GreedyPM/per/OPT=MIN	0.49	1.10	0.21	0.60
Greedy*/per/OPT=MIN	0.50	1.29	0.27	0.66
GreedyP*/per/OPT=MIN	0.58	1.37	0.28	0.65
GreedyPM*/per/OPT=MIN	0.56	1.37	0.29	0.66
GreedyP/per/OPT=MIN/MVT=600	0.49	1.11	0.18	0.57
GreedyPM/per/OPT=MIN/MVT=600	0.49	1.10	0.18	0.57
GreedyP*/per/OPT=MIN/MVT=600	0.56	1.36	0.24	0.63
GreedyPM*/per/OPT=MIN/MVT=600	0.54	1.34	0.26	0.62
MCB*/OPT=MIN/MVT=600	0.13	0.37	0.53	1.51
MCB/per/OPT=MIN/MVT=600	0.53	1.12	0.43	1.12
MCB*/per/OPT=MIN/MVT=600	0.54	1.11	0.56	1.53
/per/OPT=MIN/MVT=600	0.49	1.08	0.19	0.58
/stretch-per/OPT=MAX/MVT=600	0.28	0.64	0.37	0.78

load traces). This data was obtained from the same set of simulation experiments as discussed in Sections 5.3 and 5.4. The table shows average and maximum values due to preemptions and migrations computed over all traces in GB/sec, assuming that nodes have 8GB of RAM. Information about the number of preemptions and migrations per hour and per job can be found in Appendix B in Tables B.5 and B.6 respectively.

The main observation from the results in Table 6.1 is that the total bandwidth consumption is, as expected, within the capabilities of modern high-end cluster systems for all algorithms, even under high load conditions. For instance, the GreedyPM*/per/OPT=MIN/MVT=600 algorithm, identified as best in Section 5.4, has a total average bandwidth requirement of 0.80 GB/sec. Even accounting for maximum bandwidth requirements, i.e., for the trace that causes the most traffic due to preemptions and migrations, this algorithm still uses under 2 GB/sec. Such numbers represent only a fraction of the bandwidth capacity of current interconnect technology for cluster platforms [200], though the value is still high enough to be a concern for some administrators.

6.2 Problem Definition

For the sake of completeness, we provide a definition of the adaptation version of the DFRS scheduling problem. This version extends the off-line version described in Section 4.1, and models the need to move from an existing resource allocation to a new allocation that will provide a better value for the minimum yield. Since we assume that job resource needs have changed, the existing allocation may be poor, or even invalid. We again assume serial jobs and thus equate jobs with their tasks. An instance of this problem thus consists of a set of nodes, N, a set of jobs J and a mapping f from J to N representing the current job placements. Each job j also has an associated value c_j that represents the cost of migrating the job to a new node (new jobs can be represented with an arbitrary mapping and zero migration cost). These values might represent bytes transferred, elapsed milliseconds, or any other cost associated with migration. The goal of the adaptation problem is to find a value $\alpha \in [0, 1]$ and a mapping f' from J to H that is valid as defined previously, and with the additional constraint that the total cost to migrate jobs does not exceed a static bound C. That is:

$$\sum_{j \in J}^{f'(j) \neq f(j)} c_j \leq C$$

6.2.1 Computational Complexity

As before, the decision problem is to determine whether or not a solution exists for a particular α. Given a solution it can be checked in polynomial time, and so the problem is in NP. The off-line problem trivially reduces to the adaptation problem, and so the adaptation problem is also NP-complete in the strong sense.

6.2.2 MILP Formulation

We provide a MILP formulation of the adaptation problem. It is essentially the same as that of the off-line problem given in Section 4.1.2, with a few additions. The values $\{\bar{e}_{jn}\}_{j\in J,n\in N}$ are constants representing the current allocation of jobs to nodes. A given $\bar{e}_{jn}$ is 1 if and only of job j is located on node n, and is 0 otherwise. For any job j there is at most one value n for which $\bar{e}_{jn}$ is allowed to be nonzero. The other constants and variables are the same as given in the formulation of the off-line problem:

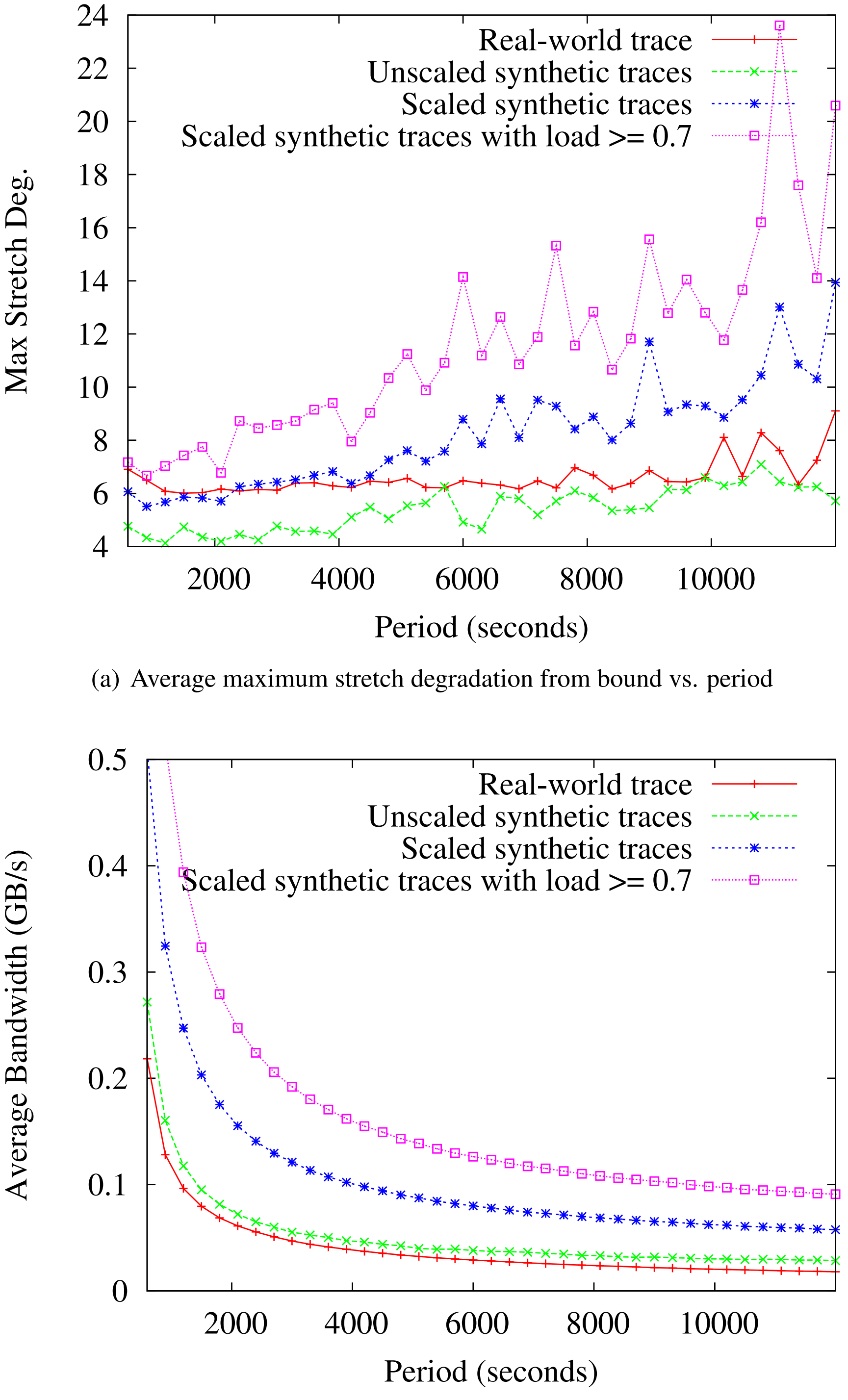
$$\begin{align}
& Y \in \mathbb{Q}^{+} \tag{6.1}\\
\forall j,n \quad & e_{jn} \in \{0,1\} \quad \alpha_{jn} \in \mathbb{Q} \tag{6.2}\\
\forall j \quad & \sum_{n} e_{jn} = 1 \tag{6.3}\\
\forall j,n \quad & 0 \leq \alpha_{jn} \leq e_{jn} \tag{6.4}\\
\forall n,d \quad & \sum_{j} r_{jd}(\alpha_{jn}(1-\delta_{jd}) + e_{jn}\delta_{jd}) \leq 1 \tag{6.5}\\
\forall j \quad & \sum_{n} \alpha_{jn} \geq \hat{\alpha}_j + Y(1-\hat{\alpha}_j) \tag{6.6}\\
& \sum_{j=1}^{J}\sum_{n=1}^{N}(1-\bar{e}_{jn})e_{jn}c_j \leq C \tag{6.7}
\end{align}$$

The only change from the linear program given in Section 4.1.2 is the addition of Constraint 6.7 to bound migration. The objective is, again, to maximize Y, which represents the minimum (scaled) yield.

6.3 Reducing Bandwidth Consumption

As the formal version of the adaptation problem is NP-complete, we focus on pragmatic solutions to the problem. In particular, we hope to show that we can reduce the average bandwidth consumption of the on-line algorithms developed in Chapter 5 without making huge sacrifices in performance. Returning to the results summarized in Table 6.1, we can see that the more aggressive algorithms that use MCB to remap tasks upon job submission and completion (when a * is present) have bandwidth requirements that are higher than their counterparts that use Greedy. The purely greedy algorithms that do not use MCB at all have very low bandwidth requirements, while the purely periodic use of MCB has bandwidth requirements of around 0.68 GB/sec on average. This suggests that most of the bandwidth consumption for our algorithms is caused by the use of MCB. Thus, an obvious way to try to reduce bandwidth consumption is to reduce the use of MCB. However, as we discovered in Chapter 5, some use of MCB is necessary in order to achieve good performance. We propose to vary the rescheduling period of our best algorithm from Chapter 5 (which uses GreedyPM upon job submission and completion) and study how this affects both the maximum stretch achieved and the bandwidth consumed.

Figure 6.1 shows the results of varying the rescheduling period from 600 to 12,000 seconds, in increments of 300 seconds, for the GreedyPM*/per/OPT=MIN/MVT=600 algorithm. This means that the period varies from 2x to 20x the rescheduling penalty. We do not use a period equal to the penalty cost as for some traces it can result in a



(a) Average maximum stretch degradation from bound vs. period

(b) Average bandwidth consumption vs. period

Figure 6.1. Effects of varying the rescheduling period on maximum stretch degradation from bound and bandwidth consumption, for our three sets of traces and the set of scaled synthetic traces with loads ≥ 7.

situation where no job ever makes any progress because of thrashing. The graphs contain lines representing average values over all three sets of traces, as well as the subset of the scaled synthetic traces with "heavy" load values greater-than-or-equal-to 7. Figure 6.1(a) shows the effect of varying the period on the maximum stretch degradation from bound, while Figure 6.1(b) shows the effect of varying the period on the amount of bandwidth consumed by job preemptions and migrations initiated by the algorithm.

The main observation from Figure 6.1(a) is that for all of the sets of traces under consideration the average maximum stretch degradation increases slowly as the period increases. The largest increase in degradation from bound is seen for the scaled synthetic traces, for which an increase of the period by a factor 20 leads to an increase of the maximum stretch degradation by less than a factor 3. Results for the unscaled synthetic traces and the real-world trace show much smaller increases (by less than a factor 1.5 and a factor 2, respectively). Recall from Section 5.4 that EASY leads to maximum stretch degradation orders of magnitude larger than our algorithms. We conclude that increasing the period significantly, i.e., up to 20x the rescheduling penalty, still allows our algorithms to widely outperform EASY.

By contrast, we see by looking at Figure 6.1(b) that as the period increases the bandwidth consumed by preemptions and migrations rapidly declines. For all of the sets of traces under consideration the largest improvements can be seen by increasing the period from 600 to 2,000 seconds, with additional significant improvement gained by moving from 2,000 to 4,000 seconds. After this the rate of improvement starts to level off.

This is good news for our approach as it suggests that by increasing the rescheduling period from 600 seconds to something slightly over an hour, we can keep our average maximum degradation from best values close to the theoretical bound, while reducing bandwidth costs significantly.

6.4 Conclusion

In this chapter we studied the problem of adapting an existing allocation of jobs to nodes to a new one with greater performance, while limiting the costs associated with job migration. We gave a theoretical formulation of the problem assuming a constant bound on migration, proved its theoretical complexity, and provided a mixed-integer linear program that can be used to find an optimal solution of any instance of the problem. We also studied the costs of migration associated with our previously established results from Chapter 5, showed that they are within the capabilities of modern systems, and demonstrated that they can be reduced further without significantly impacting the performance of our algorithms by the simple expedient of increasing the size of the rescheduling period.

Preliminary results upon which the work in this chapter was based were previously published in the *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium* [208]. A new paper has been written based on the results of the experiments presented in this chapter and will be submitted for publication in the *IEEE Transactions on Parallel and Distributed Systems* [209].

CHAPTER 7
FEASIBILITY STUDY OF DFRS IN PRACTICE

In this chapter we detail the first steps we have taken in applying our knowledge of the abstract DFRS scheduling problem toward creating a workable and useful real-world system. There are two critical points that we propose to address: The first is to identify a suitable platform for development. The second is to determine how to quickly and accurately assess task resource requirements and needs. Our specific contributions in this chapter are: (i) We identify our chosen development platform and detail how we implement our DFRS based scheduling algorithms (Section 7.1); (ii) We propose a basic approach for CPU resource needs discovery and allocation (Section 7.2); and (iii) We evaluate our proposed approach through experiments conducted on a real testbed system using synthetic traces and simulated applications (Sections 7.3 and 7.4).

7.1 Platform Description

We have chosen to base our implementation on the Usher system [147] from the Computer Science and Engineering department at the University of California, San Diego (UCSD). Usher can interface with individual instances of the Xen virtual machine hypervisor to configure and deploy virtual machines or collections of virtual machines across a cluster with low overhead, and has been successfully used for this purpose at UCSD for some time. Additionally, it is highly scriptable, giving plugin modules written in the Python programming language the ability to place, schedule, and migrate virtual machines between hosts at will.

Usher consists of two parts: the *local node manager* (LNM), that runs on each compute node, and a *central controller* that sends commands to the LNMs. The LNM software runs inside a special administrative domain, effectively a separate virtual machine with its own allocated resources, called dom0 in the Xen documentation, and communicates directly with Xen using the provided API. The LNM software is designed to perform local management tasks such as monitoring VM resource usage (e.g., CPU, network usage) and executing the commands sent by central controller (e.g., initializing, migrating, and powering off VM instances). The central controller can be accessed directly by client programs such as Ush or Plusher [210], and also provides an interface for plugins written in Python [211].

Plugins are exposed to a set of events, the most important of which for our purposes are:

***start* –** A VM instance has been started.

***power off* –** A VM instance has been turned off.

***periodic* –** A timer that fires at a user defined interval.

***migrate* –** A VM instance has been migrated.

These events and many more may be used to trigger the execution of specialized code [212]. Plugins have access to information about the VM instances and LNMs via the central controller. Plugins also have the ability to issue commands through the central controller to affect individual VM instances. The Usher plugin interface thus provides an ideal development platform for implementing our DFRS algorithms.

7.2 Resource Allocation and Discovery

For this portion of our research we choose to focus on discovering only the CPU needs of running VM instances. While the moment to moment CPU usage of a VM instance can be quite erratic, it is nonetheless true that some jobs can require more use of the CPU than others over a given period of time. The actual allocation of the CPU to running VM instances is carried out by Xen.

7.2.1 VM Caps and Weights in Xen

The Xen hypervisor provides two different methods for managing the proportion of available CPU cycles allocated to competing VM instances on the same node: *caps* and *weights*. A cap represents an absolute bound on the percentage of available CPU cycles used by a VM instance: the total of all the caps of the VM instances running on a node can add up to at most 100%, and if a VM instance uses less than its cap the excess will not be available to its competitors. For this reason, when caps are used Xen is said to be in non-work-conserving mode. As an alternative to caps, Xen also provides a way to specify weights for VM instances. In the event that all of the VM instances on a node are CPU-bound, each VM instance will receive a proportion of the available CPU cycles equal to its weight divided by the total of the weights of all the VM instances on the node. If a VM instance requires less than its proportional share, then the remaining CPU cycles will be apportioned to the other VM instances in proportion to their relative weights, and so on until there are no extra cycles remaining. When Xen is using weights and not caps it is said to be in work-conserving mode [136].

We use weights rather than caps in order to keep Xen in work-conserving mode, and thus avoid forcing the CPU to sit idle while there is work to be done. Though there is no explicit requirement to do so, we keep the impact of different weights easy to understand by requiring that the weights of running VM instances on a node sum up to 100. As an example, if one VM instance is given a weight of 70 while another is given a weight of 30, then the first should be assigned 70% of the available CPU cycles while the second gets 30% of the available cycles. If the first VM instance requires less than 70% of the available cycles then the remainder could potentially be used by the second VM instance.

Using weights instead of absolute caps on CPU utilization thus helps the system to be more responsive to actual needs when they are out of sync with estimates. A version of the system that used caps instead of weights was also implemented, but was found to perform poorly because of these types of inefficiencies.

7.2.2 Algorithms

Our basic approach is to periodically assign a majority of the available CPU cycles to one of the VM instances running on a node and then use the average utilization over this time as an estimate of the CPU need for that VM instance. These estimates can then be used to make resource allocation decisions. We study two different versions of the same basic algorithm: DYNAMIC, short for "dynamic CPU need discovery", and DYNAMIC+MIGRATION. These algorithms are identical except that DYNAMIC+MIGRATION can initiate live migrations [144] to improve the load balance.

We have implemented our algorithms in Python as a plugin to Usher. The algorithms proceed periodically in three phases: a discovery phase, an allocation phase, and an execution phase. The purpose of the discovery phase is to accurately discover the needs of one of the running VM instances on each node. The purpose of the allocation phase is to allocate resources and (in the case of DYNAMIC+MIGRATION) allow time for migration. The purpose of the execution phase is to allow the VM instances to run as efficiently as possible without being disturbed by discovery or migration (though it should be noted that all of the VM instances will be executed for some portion of the time during each of the previous two phases). In our experiments we arbitrarily set all three phases to last 60 seconds, for an overall period of 3 minutes.

At the beginning of the discovery phase the system first identifies the least-recently-discovered VM instance on each node by comparing the recorded times of their most-recent discovery events. In the unlikely event of a tie (which is possible if migration is enabled) then the VM instance with the lower current need estimate is selected. Further ties are broken arbitrarily. The selected VM instance is assigned a weight of 100 minus the number of other VM instances present on the node, while the remaining VM instances are assigned weights of 1. At the end of this phase the CPU need estimate for the selected VM instance is set to the average CPU utilization over the phase and the time of the discovery event is recorded.

At the beginning of the allocation phase new resource allocations are computed by first maximizing the minimum yield for VM instances at their current locations. Any remaining weight (out of the 100 points under consideration) is allocated in such a way as to maximize the average yield (given the lower bound on the minimum) as discussed in Section 4.4.7. For DYNAMIC+MIGRATION only, all of the current need estimates are also given to the MCB algorithm (discussed in Chapter 5), which then generates a new task mapping, and then resource allocations are also computed for this new mapping. Migrations are initiated only if they are expected to result in an improvement of the minimum yield greater than a percentage threshold (10% in our experiments) over not migrating. All migrations should be completed by the end of this phase. If no migrations are initiated, either because they are disabled in the case of DYNAMIC or because improvement from migration does not meet the threshold requirement in the case of DYNAMIC+MIGRATION, the remainder of this phase operates as an additional execution phase, as detailed below.

During the execution phase all VM instances are run with their assigned weights, but these weights may be adjusted due to activity. If a task uses less CPU than the current estimate then the estimate is adjusted downward and the "slack", or leftover CPU, is distributed to the remaining VM instances equally, and this slack, if used, can result in raising the need estimates of the VM instances to which it is assigned. If a VM instance's need estimate is adjusted downward, then a discovery event is recorded for it, meaning that it is not likely to be selected during the next discovery phase. A discovery event is not recorded for a VM instance if its estimate is adjusted upward, as the actual need might still be even higher.

7.3 Experimental Methodology

Our testbed cluster consists of 4 compute nodes and 1 controller node all with 2.8GHz Intel Xeon CPUs and 1GB of ram located at UCSD, but dedicated to our exclusive use. The compute nodes run instances of Xen and the LNM software, and are used to execute the VM instances. The controller node runs an instance of the Usher central controller, as well as other required network services (most notably, NFS and Bind).

We generate a synthetic trace for each running VM instance. Each trace consists of a sequence of time and CPU need values. To simulate a Poisson process the time value is initialized to 0 and incremented at each step by a number of seconds chosen from an exponential random distribution, with λ, the mean value, equal to 20 minutes in our experiments. CPU needs are selected from a uniform random distribution scaled to return values between 0 and 1. The total amount of time covered by each trace is 12 hours. These traces are run through a simulated application that performs a tight computational loop, with CPU utilization set to the specified values through the use of cpulimit [213]. While these traces are not necessarily representative of production workloads, they allow us to examine the system under a range of conditions and thus ensure that resource need discovery and resource allocation are functioning properly.

It should be noted that while our algorithms only attempt to discover CPU need values, active VM instances still have memory requirements. Each VM instance is allocated 96MB of memory, while 192MB of memory is allocated to dom0 on each node. Also, during live migration a VM instance will need to be allocated memory on both the source and destination system, and so we conservatively wish to ensure that half of the system memory remains free at all times. For this reason we assign all running VM instances a memory requirement of 0.25, meaning that at most 4 can be assigned to any node.

We consider two different scenarios. In the first, “balanced”, scenario three normal VM instances with randomly generated workloads are assigned to each node. In the second, “unbalanced”, scenario we add two “hogs”, VM instances configured to run at 100% CPU utilization, each to a different initial node.

7.4 Experimental Results

We evaluate our algorithms against standard Xen with each VM instance assigned the same weight. It should be noted that since the standard Xen algorithm will never discover that the total needs of VM instances running on a node exceed 100%, it will also never attempt to migrate VM instances between nodes.

We compare two different metrics of performance: *average minimum yield* and the *minimum average yield*. The average minimum yield is the average second-by-second minimum yield value achieved over all VM instances. That is, we take the minimum yield over all VM instances at each second, and then average the values over seconds. It represents the average worst-case performance and is most relevant in situations such as service hosting where all of the VM instances need to maintain a certain minimum level of responsiveness. The minimum average yield is the minimum of the average second-by-second yields of all the VM instances. That is, for each VM instance we compute the average yield achieved over the seconds of the experiment, and then select the minimum of these values. It represents the worst average-case performance, is analogous to the maximum stretch, and is most relevant in situations such as high-performance computing

Table 7.1. Aggregate performance metrics for balanced workloads

Approach	Average Minimum Yield	Minimum Average Yield
Xen	0.346 / 0.352	0.536 / 0.601
DYNAMIC	0.270 / 0.246	0.619 / 0.609
DYNAMIC+MIGRATION	0.216 / 0.209	0.609 / 0.567

where all of the VM instances need to perform a certain amount of work over the trace runtime.

Table 7.1 shows our two aggregate metrics for the native Xen CPU scheduler and each of our algorithms when applied to the balanced workload over two separate runs of the experiment, with the input trace values for each run generated by the same methodology but using a different initial random seed value. The values before the slash are for the first experimental run, while those after the slash are for the second experimental run.

Expectedly, the method of CPU need discovery employed by the DYNAMIC and DYNAMIC+MIGRATION, which sets the weights of at least some of the VM instances to very low values 1/3 of the time, has a significant negative impact on the average minimum yield metric. We see that adding migration, which can increase the load on resources as VM instances may temporarily exist on multiple nodes, has additional, but lesser, negative impact on this metric.

When we instead consider the minimum average yield, we see that our algorithms seem to have some positive impact by this metric. In particular, the DYNAMIC algorithm performs the best over both runs of the experiment. For the balanced case, migration seems to have a negative impact by this metric as well, as DYNAMIC+MIGRATION does worse than DYNAMIC (though still better than Xen) for the first experiment, and performs worst out of the three algorithms for the second.

Table 7.2 shows the same two aggregate metrics for Xen, DYNAMIC, and DYNAMIC+MIGRATION, this time for unbalanced workloads that include two hogs, each

Table 7.2. Aggregate performance metrics for unbalanced workloads

Approach	Average Minimum Yield	Minimum Average Yield
Xen	0.238 / 0.237	0.257 / 0.259
DYNAMIC	0.124 / 0.123	0.362 / 0.348
DYNAMIC+MIGRATION	0.133 / 0.113	0.425 / 0.401

on a different node. We again consider two separate runs of the experiment with traces generated by distinct random seed values.

The first observation is that the addition of the two hogs has an adverse impact on the performance all algorithms by both metrics. We can see that Xen again performs best by the average minimum yield metric, and that DYNAMIC+MIGRATION is worse than DYNAMIC. The minimum average yield, however, tells a different story. For both runs of the experiment, by this metric DYNAMIC+MIGRATION performs significantly better than DYNAMIC, which in turn outperforms Xen by a large margin. This suggests that in some situations, particularly those with severe and/or unbalanced resource overloads, it may be worthwhile to make temporary sacrifices in performance to more accurately estimate needs and make better resource allocation decisions. It also suggests that the costs of migration can be overcome by the benefit from intelligent load balancing.

7.5 Conclusion

In this chapter we described the steps taken so far toward implementing a system that can make use of our DFRS algorithms. We identified a platform for development, proposed algorithms for allocating resources based on the on-line discovery of fluid resource needs, and evaluated them through experiments conducted using real machines and simulated applications. We considered two different algorithms that make use of the proposed technique for resource need estimation: the first simply uses estimates to allocate resources in a way that should maximize the minimum yield, while the second can also migrate tasks to achieve better load balancing. For the experiments, we generated synthetic traces based on a simple model and used these traces to control the CPU utilization levels of a tight computational loop that ran in each VM instance. We found the proposed approaches have a negative impact on the average moment-by-moment worst-case performance, but that they can also result in better overall performance for the worst performing job, particularly when workloads are unbalanced. The algorithm that makes use of migration has a greater cost in terms of moment-by-moment worst-case performance than the one that does not, but it can also provide a greater benefit to the overall performance of the worst performing job.

CHAPTER 8
CONCLUSION

In this dissertation we proposed a novel approach, called Dynamic Fractional Resource Scheduling (DFRS), to sharing homogeneous cluster computing platforms among competing jobs. The key features of DFRS are that it leverages existing virtual machine technology in order to share resources more efficiently and it defines and optimizes a user-centric metric that captures notions of both performance and fairness.

In Chapter 2 we described existing strategies for allocating cluster resources to competing jobs. We first explored the literature on scheduling for HPC workloads, including well established techniques such as Batch and Gang scheduling. We also discussed the literature on co-scheduling, and concluded that efficient algorithms for resource allocation in a co-scheduling environment are likely to lead to the best performance. Finally, we explored the literature on scheduling for service hosting environments and discussed the need to formulate the resource allocation problem as an optimization problem with a well-defined objective function.

In Chapter 3 we explained the basic concepts essential to understanding DFRS. First, we gave a new model of resource sharing based on modern virtual machine technology. Next, we developed an objective measure of instantaneous schedule quality, the minimum yield, that accounts for both performance and fairness. Finally, we developed three different versions of the DFRS scheduling problem: an off-line problem targeting service hosting environments with static workloads, an on-line problem targeting dynamic workloads as found in high-performance computing environments, and an adaptation problem that bounds migration costs for workloads with evolving requirements.

In Chapter 4 we studied the off-line problem, which focuses on resource allocation in shared hosting platforms for static workloads with nodes that provide multiple types of resources. We gave a formulation of the problem that supports a mix of QoS and best-effort scenarios, and that attempts to maximize a generic objective function, the minimum yield. We explained how an (in practice reasonably tight) upper bound on the optimal minimum yield can be computed, and how the average yield can be maximized as a way to increase cluster utilization. Finally, we proposed and evaluated several classes of algorithms over a wide range of simulation scenarios.

In Chapter 5 we studied the on-line problem, which focuses on job-scheduling strategies for high-performance and scientific computing environments. We examined its theoretical difficulty in both the clairvoyant and non-clairvoyant scenarios, devised an algorithm for computing a theoretical maximum stretch lower bound assuming an idealized scenario, proposed several DFRS algorithms and compared them to standard batch scheduling approaches using both real-world and synthetic workloads. In our simulations, our algorithms were given no knowledge of job processing times, while batch scheduling algorithms were provided with perfect estimates.

In Chapter 6 we studied the problem of adapting an existing allocation of jobs to nodes to a new one with greater performance, while limiting the costs associated with job migration. We gave a theoretical formulation of the problem assuming a constant bound on migration, proved its theoretical complexity, and provided a mixed-integer linear program that can be used to find an optimal solution of any instance of the problem. We also studied the costs of migration associated with our previously established results from Chapter 5 and explored one simple way to further reduce those costs.

In Chapter 7 we looked at a prototype implementation of a system based on our ideas. We described the development platform and how the system was implemented, proposed algorithms for resource allocation based on the on-line estimation of CPU needs, and performed a simple validation study to see how well these algorithms are likely to work in practice.

8.1 Contribution of this Dissertation

Cluster scheduling is an important and active area of research for a number of reasons. Today, clusters represent a majority of the machines on the Top500 list of supercomputers [3], and they are a ubiquitous presence in the research departments of industry and academia. They are also widely deployed in service hosting environments. However, since clusters are in fact only collections of fairly modest machines, scheduling jobs properly to meet the needs of large numbers of competing users is a non-trivial issue, and current approaches and techniques are not satisfactory for a variety of reasons (see Chapter 2).

In this work we make two major theoretical and algorithmic contributions to the cluster scheduling literature:

1. We propose a new metric, the minimum yield, that is correlated to established metrics of fairness and performance and formulate the scheduling problem as the constrained optimization of this metric.

2. We give algorithms that achieve results reasonably close to theoretical bounds for both the off-line and on-line cases, even under conservative assumptions. For the latter case our algorithms widely outperform state-of-the-art approaches while incurring reasonable extra network bandwidth consumption due to task migrations.

Though the majority of our results are obtained through the use of simulation experiments, we have ample reason to believe that our conclusions will carry through to real world systems. For the on-line case, we use a combination of representative synthetic traces based on an accepted model and real-world traces harvested from a production system. This is consistent with standard procedure in HPC scheduling research. We also implement a proof-of-concept prototype system that demonstrates some of our ideas and perform a simple evaluation study to show that it works reasonably well in practice for small problem instances.

8.2 Highlights of Scientific Findings

In this section we provide greater detail on some of our more significant scientific findings:

- Performing a binary search over the yield and solving the resource allocation problem for a fixed yield using a vector packing algorithm is the best approach for solving the off-line version of our problem, and vector packing algorithms that reason on the sum of the resource needs of the jobs are the most effective;
- Among those vector packing algorithms we considered the one that makes use of the Choose Pack vector packing algorithm from [166] runs in only a few seconds and is the most effective;
- Algorithms for the off-line case that achieve high minimum yield values can be adapted for use in the on-line case, where they achieve low maximum stretch values;
- To achieve the best performance in the on-line case it is necessary to combine a number of different approaches, more specifically: periodic re-allocation of resources by an intelligent vector-packing algorithm, greedy scheduling with preemption for newly submitted jobs, opportunistic greedy scheduling of paused jobs when resources become available, and a restriction on initiating migrations for jobs that have not yet progressed beyond a threshold amount of virtual time;
- For the on-line problem, the already low bandwidth consumption due to task migrations can be effectively reduced without significantly impacting performance

by selecting an appropriate rescheduling period from a wide range of periods that work well in practice.

8.3 Future Work

There are a number of different ways in which this research can be extended. Some of these important research directions are detailed here.

8.3.1 Algorithmic and Theoretical Extensions

Our current models and algorithms only consider homogeneous clusters, but in the real world it may be necessary to work with heterogeneous platforms for a variety of reasons. For example, when funding is secured to upgrade a cluster, advances in technology and market forces may have made higher-performance nodes with a different CPU architecture more cost-effective than nodes identical to those already present. A number of popular technological approaches, such as Grid or Cloud computing, allow for federating resources, from individual computers up to full clusters, that belong to different administrative domains. Thus, there is an obvious need for models and algorithms that consider scenarios with varying levels of heterogeneity.

Perhaps the most ambitious and widely applicable improvement would be to allow for arbitrary multi-objective optimization targets. While it is true that maximizing the minimum yield tends to improve both fairness and job performance, users and administrators may have other concerns that require consideration, or may want to balance performance against fairness in a way different from what can be achieved by focusing strictly on the yield. Allowing for arbitrary multi-objective functions would enable the system to take into account concerns such as user or job priority, energy consumption [214–216] (or other costs), or even a more formalized notion of job fairness as defined in [217]. These concerns could be balanced against each other non-linearly as well (e.g., the ratio of the cube of average response-time against the square of energy consumption). The system could even allow for arbitrary per-job user-specified measures of utility as described in [15].

Considering only the on-line scenario, there are several refinements we could make to our basic strategy of maximizing the objective function at different points in time. For example, our scheduling algorithms could be improved with a strategy for reducing the yield of long running jobs as a way to improve fairness and further decrease maximum stretch. This strategy, inspired by thread scheduling in operating systems kernels, would be particularly useful for mitigating the negative impact of long running jobs on shorter ones. Another possibility would be to compute deadlines for rescheduling based on when jobs achieve certain stretch values, rather than performing purely periodic rescheduling.

Another potential way to improve our approach to the on-line problem would be through the use of predictive models. While it is true that in the on-line scenario the system does not generally have any knowledge of future jobs–their eventual submission time, resource requirements, or runtime requirements–it is nonetheless also true that many, if not most, systems experience regular and predictable patterns of usage. If a statistical model can be made of these usage patterns then that model can be used to make more intelligent decisions about resource allocation strategies. For example, if a large job submitted during the daytime uses more than a threshold amount of run-time, then it may make sense to classify that job as “long-running” and then suspend it or run it with reduced resources until evening, when the overall machine usage is lower and users are less likely to submit short-running high-priority jobs. This is somewhat similar to the approach described by Barsanti and Sodan in [218].

8.3.2 Development and Evaluation of a Practical System

The system described in Chapter 7 was developed primarily to test intelligent strategies for research probing, and will require much additional work in order to make it fully practical for use by administrators and end-users. In particular, it requires a front-end interface for job submission and additional sensors allow for the extension of probing to additional resource dimensions.

We believe that the proposed approach for CPU needs discovery is a reasonable candidate for future, more complicated, studies targeting the use of DFRS on more realistic workloads. Examples of established synthetic service-hosting workloads that also provide simulated applications include RUBBoS [219], RUBiS [220], and TPC-W [221]. For high-performance computing applications the most well-established model for generating synthetic workloads is the one by Lublin et. al. [203] that we used in Chapter 5, however there is no well-established model for simulating resource usage by applications. One possible approach would be to run simulated applications based on portions of the NAS Parallel Benchmark suite [222].

We know that the proposed algorithm achieves better knowledge of resource needs than the naïve approach (which is, indeed, incapable of discovering true needs when a resource is overloaded), but has a high cost in terms of mis-allocated resources. There may be better general-case algorithms, and it is very likely that in the real-world some programs and/or resources may supply "hints" that can indicate likely future resource needs. By coupling such hints with predictive statistical models it may be possible, for some resources and some workloads, to achieve highly accurate predictions at little or no cost.

APPENDIX A
LIST OF PUBLICATIONS

– M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, “Resource allocation using virtual clusters,” in *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid’09)*, May 2009, pp. 260–267, 21% acceptance rate.

– M. Stillwell, “Dynamic fractional resource scheduling for cluster platforms,” in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium PhD Forum (IPDPS’10 Workshops)*, Apr. 2010, research poster presentation.

– M. Stillwell, F. Vivien, and H. Casanova, “Dynamic fractional resource scheduling for HPC workloads,” in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS’10)*, Apr. 2010, 24% acceptance rate.

– M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, “Resource allocation algorithms for virtualized service hosting platforms,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 9, pp. 962–974, 2010.

– M. Stillwell, F. Vivien, and H. Casanova, “Fine-grain dynamic resource allocation vs. batch scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, submitted for publication.

APPENDIX B
ADDITIONAL TABLES

Table B.1. Average degradation from bound results for the real-world HPC2N workload. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
FCFS	3,578.5	3,727.8	21,718.4
EASY	3,041.9	3,438.0	21,317.4
Greedy*/OPT=AVG	1,012.2	2,229.5	19,799.1
Greedy*/OPT=MIN	949.8	1,828.5	11,778.4
Greedy/per/OPT=AVG	28.9	27.0	212.9
Greedy/per/OPT=AVG/MFT=300	23.3	26.9	182.3
Greedy/per/OPT=AVG/MFT=600	23.5	27.6	212.2
Greedy/per/OPT=AVG/MVT=300	23.9	26.9	182.3
Greedy/per/OPT=AVG/MVT=600	23.8	27.8	182.3
Greedy/per/OPT=MIN	28.3	24.9	163.7
Greedy/per/OPT=MIN/MFT=300	23.4	26.0	152.0
Greedy/per/OPT=MIN/MFT=600	23.1	24.8	152.5
Greedy/per/OPT=MIN/MVT=300	23.5	26.9	182.8
Greedy/per/OPT=MIN/MVT=600	23.0	25.9	152.0
Greedy*/per/OPT=AVG	24.5	15.9	81.6
Greedy*/per/OPT=AVG/MFT=300	19.8	17.8	85.6
Greedy*/per/OPT=AVG/MFT=600	19.3	17.6	85.6
Greedy*/per/OPT=AVG/MVT=300	19.2	17.5	85.6
Greedy*/per/OPT=AVG/MVT=600	18.9	17.3	74.9
Greedy*/per/OPT=MIN	24.3	15.9	81.6
Greedy*/per/OPT=MIN/MFT=300	19.5	17.7	85.6
Greedy*/per/OPT=MIN/MFT=600	19.1	17.5	85.6
Greedy*/per/OPT=MIN/MVT=300	18.9	17.2	85.6
Greedy*/per/OPT=MIN/MVT=600	19.0	17.4	66.1
GreedyP*/OPT=AVG	20.4	116.7	1,254.2
GreedyP*/OPT=MIN	13.5	68.0	819.2
GreedyP/per/OPT=AVG	18.4	18.6	152.4
GreedyP/per/OPT=AVG/MFT=300	9.1	18.8	152.4
GreedyP/per/OPT=AVG/MFT=600	9.0	18.7	152.4
GreedyP/per/OPT=AVG/MVT=300	9.0	18.8	152.4
GreedyP/per/OPT=AVG/MVT=600	8.9	18.9	152.4
GreedyP/per/OPT=MIN	18.5	18.6	152.4

Table B.1. (Continued) Average degradation from bound results for the real-world HPC2N workload. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
GreedyP/per/OPT=MIN/MFT=300	9.1	18.8	152.4
GreedyP/per/OPT=MIN/MFT=600	9.0	18.9	152.4
GreedyP/per/OPT=MIN/MVT=300	9.0	18.9	152.4
GreedyP/per/OPT=MIN/MVT=600	8.9	18.9	152.4
GreedyP*/per/OPT=AVG	17.9	19.7	213.5
GreedyP*/per/OPT=AVG/MFT=300	8.2	19.4	213.5
GreedyP*/per/OPT=AVG/MFT=600	7.7	18.5	198.8
GreedyP*/per/OPT=AVG/MVT=300	7.0	14.0	149.3
GreedyP*/per/OPT=AVG/MVT=600	6.9	14.0	149.3
GreedyP*/per/OPT=MIN	17.9	19.6	213.5
GreedyP*/per/OPT=MIN/MFT=300	7.9	19.0	213.5
GreedyP*/per/OPT=MIN/MFT=600	7.5	18.0	198.8
GreedyP*/per/OPT=MIN/MVT=300	6.9	14.0	149.3
GreedyP*/per/OPT=MIN/MVT=600	6.9	14.2	149.3
GreedyPM*/OPT=AVG	14.1	72.7	880.1
GreedyPM*/OPT=MIN	13.8	68.2	819.2
GreedyPM/per/OPT=AVG	18.5	19.2	158.7
GreedyPM/per/OPT=AVG/MFT=300	9.1	19.4	158.7
GreedyPM/per/OPT=AVG/MFT=600	9.0	19.3	158.7
GreedyPM/per/OPT=AVG/MVT=300	9.1	19.4	158.7
GreedyPM/per/OPT=AVG/MVT=600	8.9	19.5	158.7
GreedyPM/per/OPT=MIN	18.4	18.8	158.7
GreedyPM/per/OPT=MIN/MFT=300	9.2	19.2	158.7
GreedyPM/per/OPT=MIN/MFT=600	9.1	19.3	158.7
GreedyPM/per/OPT=MIN/MVT=300	8.9	18.8	158.7
GreedyPM/per/OPT=MIN/MVT=600	8.8	18.9	158.7
GreedyPM*/per/OPT=AVG	17.8	19.3	198.6
GreedyPM*/per/OPT=AVG/MFT=300	8.2	19.1	198.6
GreedyPM*/per/OPT=AVG/MFT=600	7.8	19.0	198.8
GreedyPM*/per/OPT=AVG/MVT=300	7.0	14.1	149.6
GreedyPM*/per/OPT=AVG/MVT=600	6.9	14.2	149.6
GreedyPM*/per/OPT=MIN	17.9	19.3	198.6
GreedyPM*/per/OPT=MIN/MFT=300	8.1	19.2	198.6
GreedyPM*/per/OPT=MIN/MFT=600	7.9	19.1	198.8
GreedyPM*/per/OPT=MIN/MVT=300	6.9	14.3	149.6
GreedyPM*/per/OPT=MIN/MVT=600	6.9	14.4	149.6
MCB*/OPT=AVG	346.3	1,223.1	13,399.7
MCB*/OPT=AVG/MFT=300	44.7	144.2	1,082.8

Table B.1. (Continued) Average degradation from bound results for the real-world HPC2N workload. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
MCB*/OPT=AVG/MFT=600	20.6	63.9	672.0
MCB*/OPT=AVG/MVT=300	14.1	33.2	370.0
MCB*/OPT=AVG/MVT=600	12.1	32.8	370.0
MCB*/OPT=MIN	345.6	1,241.5	13,668.8
MCB*/OPT=MIN/MFT=300	44.7	138.4	1,126.3
MCB*/OPT=MIN/MFT=600	19.1	59.2	672.0
MCB*/OPT=MIN/MVT=300	14.0	33.2	370.0
MCB*/OPT=MIN/MVT=600	12.0	32.8	370.0
MCB/per/OPT=AVG	171.4	702.4	8,383.0
MCB/per/OPT=AVG/MFT=300	15.0	33.5	279.3
MCB/per/OPT=AVG/MFT=600	12.1	26.9	292.4
MCB/per/OPT=AVG/MVT=300	11.5	25.3	287.6
MCB/per/OPT=AVG/MVT=600	10.7	25.2	287.6
MCB/per/OPT=MIN	169.1	691.4	8,362.3
MCB/per/OPT=MIN/MFT=300	15.1	34.5	309.3
MCB/per/OPT=MIN/MFT=600	12.1	26.9	292.4
MCB/per/OPT=MIN/MVT=300	11.5	25.3	287.6
MCB/per/OPT=MIN/MVT=600	10.8	25.3	287.6
MCB*/per/OPT=AVG	394.5	1,562.9	17,862.0
MCB*/per/OPT=AVG/MFT=300	56.1	197.4	1,849.6
MCB*/per/OPT=AVG/MFT=600	23.6	71.3	799.3
MCB*/per/OPT=AVG/MVT=300	15.5	30.6	318.9
MCB*/per/OPT=AVG/MVT=600	13.7	30.3	318.9
MCB*/per/OPT=MIN	389.1	1,522.1	17,313.6
MCB*/per/OPT=MIN/MFT=300	56.8	198.7	1,738.4
MCB*/per/OPT=MIN/MFT=600	25.1	81.4	799.3
MCB*/per/OPT=MIN/MVT=300	15.4	30.6	318.9
MCB*/per/OPT=MIN/MVT=600	13.6	30.2	318.9
/per/OPT=AVG	105.0	445.6	5,011.9
/per/OPT=AVG/MFT=300	105.0	445.6	5,011.9
/per/OPT=AVG/MFT=600	105.0	445.6	5,011.9
/per/OPT=AVG/MVT=300	105.0	445.6	5,011.9
/per/OPT=AVG/MVT=600	105.0	445.6	5,011.9
/per/OPT=MIN	105.0	445.6	5,011.9
/per/OPT=MIN/MFT=300	105.0	445.6	5,011.9
/per/OPT=MIN/MFT=600	105.0	445.6	5,011.9
/per/OPT=MIN/MVT=300	105.0	445.6	5,011.9
/per/OPT=MIN/MVT=600	105.0	445.6	5,011.9

Table B.1. (Continued) Average degradation from bound results for the real-world HPC2N workload. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
/stretch-per/OPT=AVG	105.0	445.6	5,011.9
/stretch-per/OPT=AVG/MFT=300	105.0	445.6	5,011.9
/stretch-per/OPT=AVG/MFT=600	105.0	445.6	5,011.9
/stretch-per/OPT=AVG/MVT=300	105.0	445.6	5,011.9
/stretch-per/OPT=AVG/MVT=600	105.0	445.6	5,011.9
/stretch-per/OPT=MAX	105.0	445.6	5,011.9
/stretch-per/OPT=MAX/MFT=300	105.0	445.6	5,011.9
/stretch-per/OPT=MAX/MFT=600	105.0	445.6	5,011.9
/stretch-per/OPT=MAX/MVT=300	105.0	445.6	5,011.9
/stretch-per/OPT=MAX/MVT=600	105.0	445.6	5,011.9

Table B.2. Average degradation from bound results for the unscaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
FCFS	5,457.2	2,958.5	15,102.7
EASY	4,955.4	2,730.6	14,036.8
Greedy*/OPT=AVG	2,527.1	2,472.3	12,487.5
Greedy*/OPT=MIN	2,435.0	2,285.6	11,229.9
Greedy/per/OPT=AVG	30.0	10.2	58.1
Greedy/per/OPT=AVG/MFT=300	26.5	14.4	58.1
Greedy/per/OPT=AVG/MFT=600	25.6	14.2	57.8
Greedy/per/OPT=AVG/MVT=300	25.7	14.5	57.8
Greedy/per/OPT=AVG/MVT=600	25.5	14.2	57.8
Greedy/per/OPT=MIN	30.1	10.2	58.1
Greedy/per/OPT=MIN/MFT=300	26.0	14.3	58.1
Greedy/per/OPT=MIN/MFT=600	25.9	14.5	58.0
Greedy/per/OPT=MIN/MVT=300	25.9	14.5	57.9
Greedy/per/OPT=MIN/MVT=600	25.9	14.2	58.0
Greedy*/per/OPT=AVG	30.5	9.8	65.7
Greedy*/per/OPT=AVG/MFT=300	25.6	14.4	58.7
Greedy*/per/OPT=AVG/MFT=600	25.0	14.3	57.5
Greedy*/per/OPT=AVG/MVT=300	25.3	14.4	57.5
Greedy*/per/OPT=AVG/MVT=600	24.7	14.1	54.1
Greedy*/per/OPT=MIN	30.4	9.7	65.7
Greedy*/per/OPT=MIN/MFT=300	25.1	14.3	57.5
Greedy*/per/OPT=MIN/MFT=600	24.9	14.3	57.5
Greedy*/per/OPT=MIN/MVT=300	24.9	14.2	54.1
Greedy*/per/OPT=MIN/MVT=600	24.6	14.3	54.1
GreedyP*/OPT=AVG	32.7	146.9	1,230.9
GreedyP*/OPT=MIN	37.5	156.0	1,204.9
GreedyP/per/OPT=AVG	20.2	7.2	38.1
GreedyP/per/OPT=AVG/MFT=300	6.3	4.3	38.1
GreedyP/per/OPT=AVG/MFT=600	6.1	4.4	38.1
GreedyP/per/OPT=AVG/MVT=300	6.0	3.9	27.5
GreedyP/per/OPT=AVG/MVT=600	6.0	4.5	38.1
GreedyP/per/OPT=MIN	20.1	7.3	38.1
GreedyP/per/OPT=MIN/MFT=300	6.1	3.8	27.5
GreedyP/per/OPT=MIN/MFT=600	6.1	4.5	38.1
GreedyP/per/OPT=MIN/MVT=300	5.9	3.8	27.5
GreedyP/per/OPT=MIN/MVT=600	5.9	4.5	38.1
GreedyP*/per/OPT=AVG	20.4	6.8	32.0
GreedyP*/per/OPT=AVG/MFT=300	5.5	2.8	18.0

Table B.2. (Continued) Average degradation from bound results for the unscaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
GreedyP*/per/OPT=AVG/MFT=600	5.1	2.8	18.0
GreedyP*/per/OPT=AVG/MVT=300	4.9	2.4	13.6
GreedyP*/per/OPT=AVG/MVT=600	4.8	2.4	13.6
GreedyP*/per/OPT=MIN	20.3	6.8	32.0
GreedyP*/per/OPT=MIN/MFT=300	5.2	2.4	13.7
GreedyP*/per/OPT=MIN/MFT=600	5.0	2.7	18.0
GreedyP*/per/OPT=MIN/MVT=300	4.9	2.7	18.0
GreedyP*/per/OPT=MIN/MVT=600	4.9	2.9	19.2
GreedyPM*/OPT=AVG	28.2	104.4	676.2
GreedyPM*/OPT=MIN	33.8	154.0	1,321.7
GreedyPM/per/OPT=AVG	20.2	7.2	38.1
GreedyPM/per/OPT=AVG/MFT=300	6.3	3.7	27.5
GreedyPM/per/OPT=AVG/MFT=600	6.1	4.4	38.1
GreedyPM/per/OPT=AVG/MVT=300	6.2	4.5	38.1
GreedyPM/per/OPT=AVG/MVT=600	5.9	4.4	38.1
GreedyPM/per/OPT=MIN	20.2	7.3	38.1
GreedyPM/per/OPT=MIN/MFT=300	6.1	3.6	27.5
GreedyPM/per/OPT=MIN/MFT=600	6.0	4.4	38.1
GreedyPM/per/OPT=MIN/MVT=300	6.0	3.9	27.5
GreedyPM/per/OPT=MIN/MVT=600	5.9	4.5	38.1
GreedyPM*/per/OPT=AVG	20.4	6.8	32.0
GreedyPM*/per/OPT=AVG/MFT=300	5.5	2.6	13.7
GreedyPM*/per/OPT=AVG/MFT=600	5.0	2.5	13.7
GreedyPM*/per/OPT=AVG/MVT=300	4.9	2.5	13.8
GreedyPM*/per/OPT=AVG/MVT=600	4.8	2.4	13.6
GreedyPM*/per/OPT=MIN	20.3	6.9	32.0
GreedyPM*/per/OPT=MIN/MFT=300	5.3	2.7	18.0
GreedyPM*/per/OPT=MIN/MFT=600	4.9	2.5	13.7
GreedyPM*/per/OPT=MIN/MVT=300	4.9	2.7	18.0
GreedyPM*/per/OPT=MIN/MVT=600	4.8	2.4	13.6
MCB*/OPT=AVG	245.1	130.3	634.2
MCB*/OPT=AVG/MFT=300	18.0	23.2	206.3
MCB*/OPT=AVG/MFT=600	9.8	6.4	43.6
MCB*/OPT=AVG/MVT=300	8.6	5.6	43.9
MCB*/OPT=AVG/MVT=600	7.7	6.9	44.8
MCB*/OPT=MIN	233.2	117.1	634.2
MCB*/OPT=MIN/MFT=300	16.6	22.8	206.3
MCB*/OPT=MIN/MFT=600	9.9	8.1	51.3

Table B.2. (Continued) Average degradation from bound results for the unscaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
MCB*/OPT=MIN/MVT=300	9.2	8.0	65.3
MCB*/OPT=MIN/MVT=600	6.9	5.4	44.4
MCB/per/OPT=AVG	134.7	57.1	324.1
MCB/per/OPT=AVG/MFT=300	15.2	18.7	173.0
MCB/per/OPT=AVG/MFT=600	10.2	8.1	65.7
MCB/per/OPT=AVG/MVT=300	9.2	6.8	51.3
MCB/per/OPT=AVG/MVT=600	8.2	7.0	53.3
MCB/per/OPT=MIN	133.7	57.5	323.7
MCB/per/OPT=MIN/MFT=300	14.5	18.6	173.0
MCB/per/OPT=MIN/MFT=600	10.0	8.1	65.7
MCB/per/OPT=MIN/MVT=300	9.0	6.7	51.3
MCB/per/OPT=MIN/MVT=600	8.1	6.6	53.3
MCB*/per/OPT=AVG	252.1	126.3	634.2
MCB*/per/OPT=AVG/MFT=300	19.5	35.4	349.2
MCB*/per/OPT=AVG/MFT=600	10.7	5.6	37.1
MCB*/per/OPT=AVG/MVT=300	8.8	3.5	19.0
MCB*/per/OPT=AVG/MVT=600	7.8	3.8	21.4
MCB*/per/OPT=MIN	250.6	125.0	634.2
MCB*/per/OPT=MIN/MFT=300	19.0	35.3	349.2
MCB*/per/OPT=MIN/MFT=600	10.6	5.7	37.1
MCB*/per/OPT=MIN/MVT=300	8.9	3.5	19.0
MCB*/per/OPT=MIN/MVT=600	7.8	3.9	21.9
/per/OPT=AVG	43.1	19.7	134.7
/per/OPT=AVG/MFT=300	43.0	19.7	134.7
/per/OPT=AVG/MFT=600	43.0	19.7	134.7
/per/OPT=AVG/MVT=300	43.0	19.8	134.7
/per/OPT=AVG/MVT=600	43.1	19.7	134.7
/per/OPT=MIN	43.0	19.8	134.7
/per/OPT=MIN/MFT=300	43.0	19.8	134.7
/per/OPT=MIN/MFT=600	43.0	19.8	134.7
/per/OPT=MIN/MVT=300	43.0	19.8	134.7
/per/OPT=MIN/MVT=600	43.0	19.7	134.7
/stretch-per/OPT=AVG	43.1	19.5	134.7
/stretch-per/OPT=AVG/MFT=300	43.1	19.5	134.7
/stretch-per/OPT=AVG/MFT=600	43.1	19.5	134.7
/stretch-per/OPT=AVG/MVT=300	43.1	19.5	134.7
/stretch-per/OPT=AVG/MVT=600	43.0	19.6	134.7
/stretch-per/OPT=MAX	43.0	19.6	134.7

Table B.2. (Continued) Average degradation from bound results for the unscaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
/stretch-per/OPT=MAX/MFT=300	43.0	19.6	134.7
/stretch-per/OPT=MAX/MFT=600	43.0	19.6	134.7
/stretch-per/OPT=MAX/MVT=300	43.0	19.6	134.7
/stretch-per/OPT=MAX/MVT=600	43.0	19.6	134.7

Table B.3. Average degradation from bound results for the scaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
FCFS	5,869.3	2,789.1	17,403.3
EASY	5,262.0	2,588.9	14,534.1
Greedy*/OPT=AVG	3,326.7	2,561.2	18,310.2
Greedy*/OPT=MIN	3,204.3	2,517.5	19,129.2
Greedy/per/OPT=AVG	29.2	14.3	153.2
Greedy/per/OPT=AVG/MFT=300	27.4	15.5	153.2
Greedy/per/OPT=AVG/MFT=600	27.3	15.5	152.6
Greedy/per/OPT=AVG/MVT=300	27.5	15.8	153.2
Greedy/per/OPT=AVG/MVT=600	27.4	15.9	153.0
Greedy/per/OPT=MIN	29.3	14.3	153.2
Greedy/per/OPT=MIN/MFT=300	27.4	15.7	153.2
Greedy/per/OPT=MIN/MFT=600	27.4	15.9	152.6
Greedy/per/OPT=MIN/MVT=300	27.6	15.9	153.4
Greedy/per/OPT=MIN/MVT=600	27.0	15.5	152.8
Greedy*/per/OPT=AVG	29.2	11.9	101.4
Greedy*/per/OPT=AVG/MFT=300	26.7	13.6	87.1
Greedy*/per/OPT=AVG/MFT=600	25.5	13.2	95.2
Greedy*/per/OPT=AVG/MVT=300	25.6	13.0	81.0
Greedy*/per/OPT=AVG/MVT=600	25.4	13.1	95.2
Greedy*/per/OPT=MIN	29.1	12.3	101.4
Greedy*/per/OPT=MIN/MFT=300	26.4	13.2	87.1
Greedy*/per/OPT=MIN/MFT=600	25.5	13.3	103.9
Greedy*/per/OPT=MIN/MVT=300	25.4	13.0	81.0
Greedy*/per/OPT=MIN/MVT=600	25.1	13.0	95.2
GreedyP*/OPT=AVG	114.3	617.3	9,490.0
GreedyP*/OPT=MIN	115.7	644.0	10,354.2
GreedyP/per/OPT=AVG	18.0	9.7	84.6
GreedyP/per/OPT=AVG/MFT=300	7.7	7.9	84.6
GreedyP/per/OPT=AVG/MFT=600	7.5	7.8	84.6
GreedyP/per/OPT=AVG/MVT=300	7.4	7.8	84.6
GreedyP/per/OPT=AVG/MVT=600	7.3	8.4	96.8
GreedyP/per/OPT=MIN	17.8	9.6	84.6
GreedyP/per/OPT=MIN/MFT=300	7.6	7.9	84.6
GreedyP/per/OPT=MIN/MFT=600	7.3	7.6	84.6
GreedyP/per/OPT=MIN/MVT=300	7.3	7.7	84.6
GreedyP/per/OPT=MIN/MVT=600	7.3	8.5	96.8
GreedyP*/per/OPT=AVG	18.1	8.6	89.9
GreedyP*/per/OPT=AVG/MFT=300	7.1	5.6	90.2

Table B.3. (Continued) Average degradation from bound results for the scaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
GreedyP*/per/OPT=AVG/MFT=600	6.7	6.3	103.5
GreedyP*/per/OPT=AVG/MVT=300	6.5	6.7	103.5
GreedyP*/per/OPT=AVG/MVT=600	6.3	6.3	103.5
GreedyP*/per/OPT=MIN	17.9	8.6	89.9
GreedyP*/per/OPT=MIN/MFT=300	6.8	6.4	103.5
GreedyP*/per/OPT=MIN/MFT=600	6.3	5.4	90.2
GreedyP*/per/OPT=MIN/MVT=300	6.1	5.4	90.2
GreedyP*/per/OPT=MIN/MVT=600	6.1	6.3	103.5
GreedyPM*/OPT=AVG	124.9	658.4	9,404.5
GreedyPM*/OPT=MIN	124.0	673.5	9,598.8
GreedyPM/per/OPT=AVG	18.1	9.9	93.3
GreedyPM/per/OPT=AVG/MFT=300	7.8	7.5	84.6
GreedyPM/per/OPT=AVG/MFT=600	7.5	7.5	84.6
GreedyPM/per/OPT=AVG/MVT=300	7.4	7.5	84.6
GreedyPM/per/OPT=AVG/MVT=600	7.3	8.0	96.8
GreedyPM/per/OPT=MIN	17.9	9.8	93.0
GreedyPM/per/OPT=MIN/MFT=300	7.6	7.4	84.6
GreedyPM/per/OPT=MIN/MFT=600	7.4	7.5	84.6
GreedyPM/per/OPT=MIN/MVT=300	7.3	7.6	84.6
GreedyPM/per/OPT=MIN/MVT=600	7.3	8.1	96.8
GreedyPM*/per/OPT=AVG	18.2	8.7	89.9
GreedyPM*/per/OPT=AVG/MFT=300	7.1	5.2	80.1
GreedyPM*/per/OPT=AVG/MFT=600	6.8	6.4	103.5
GreedyPM*/per/OPT=AVG/MVT=300	6.4	5.6	90.2
GreedyPM*/per/OPT=AVG/MVT=600	6.5	6.6	103.5
GreedyPM*/per/OPT=MIN	17.9	8.6	89.9
GreedyPM*/per/OPT=MIN/MFT=300	6.9	6.5	103.5
GreedyPM*/per/OPT=MIN/MFT=600	6.4	6.3	103.5
GreedyPM*/per/OPT=MIN/MVT=300	6.3	5.6	90.2
GreedyPM*/per/OPT=MIN/MVT=600	6.1	5.4	90.2
MCB*/OPT=AVG	750.1	1,100.8	6,274.4
MCB*/OPT=AVG/MFT=300	121.9	351.2	3,609.6
MCB*/OPT=AVG/MFT=600	33.2	95.9	1,509.2
MCB*/OPT=AVG/MVT=300	15.3	20.9	270.9
MCB*/OPT=AVG/MVT=600	14.5	40.9	1,068.1
MCB*/OPT=MIN	742.4	1,103.0	6,130.4
MCB*/OPT=MIN/MFT=300	117.8	358.4	3,680.3
MCB*/OPT=MIN/MFT=600	31.7	78.0	1,216.7

Table B.3. (Continued) Average degradation from bound results for the scaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
MCB*/OPT=MIN/MVT=300	15.7	22.5	270.9
MCB*/OPT=MIN/MVT=600	13.2	21.6	270.9
MCB/per/OPT=AVG	155.8	122.8	913.3
MCB/per/OPT=AVG/MFT=300	23.5	26.0	231.5
MCB/per/OPT=AVG/MFT=600	15.8	19.2	231.4
MCB/per/OPT=AVG/MVT=300	12.1	12.3	127.5
MCB/per/OPT=AVG/MVT=600	11.1	12.6	127.5
MCB/per/OPT=MIN	153.0	118.1	909.5
MCB/per/OPT=MIN/MFT=300	22.1	24.0	231.5
MCB/per/OPT=MIN/MFT=600	15.2	18.9	231.4
MCB/per/OPT=MIN/MVT=300	12.3	14.2	223.0
MCB/per/OPT=MIN/MVT=600	11.0	12.6	127.5
MCB*/per/OPT=AVG	959.5	1,469.0	8,299.4
MCB*/per/OPT=AVG/MFT=300	168.2	516.8	5,469.8
MCB*/per/OPT=AVG/MFT=600	40.3	155.5	2,941.5
MCB*/per/OPT=AVG/MVT=300	14.2	15.7	195.7
MCB*/per/OPT=AVG/MVT=600	12.0	14.5	195.7
MCB*/per/OPT=MIN	956.8	1,486.7	8,398.3
MCB*/per/OPT=MIN/MFT=300	161.4	481.8	4,590.5
MCB*/per/OPT=MIN/MFT=600	37.9	126.9	2,400.6
MCB*/per/OPT=MIN/MVT=300	14.4	17.4	222.2
MCB*/per/OPT=MIN/MVT=600	12.2	15.3	195.7
/per/OPT=AVG	40.4	25.1	238.3
/per/OPT=AVG/MFT=300	40.4	25.0	238.3
/per/OPT=AVG/MFT=600	40.4	25.1	238.3
/per/OPT=AVG/MVT=300	40.4	25.0	238.3
/per/OPT=AVG/MVT=600	40.4	25.0	238.3
/per/OPT=MIN	40.4	25.1	238.3
/per/OPT=MIN/MFT=300	40.4	25.1	238.3
/per/OPT=MIN/MFT=600	40.4	25.1	238.3
/per/OPT=MIN/MVT=300	40.4	25.0	238.3
/per/OPT=MIN/MVT=600	40.4	25.0	238.3
/stretch-per/OPT=AVG	40.2	24.8	236.5
/stretch-per/OPT=AVG/MFT=300	40.2	24.8	236.5
/stretch-per/OPT=AVG/MFT=600	40.2	24.8	236.5
/stretch-per/OPT=AVG/MVT=300	40.2	24.8	236.5
/stretch-per/OPT=AVG/MVT=600	40.2	24.8	236.5
/stretch-per/OPT=MAX	40.2	24.8	236.9

Table B.3. (Continued) Average degradation from bound results for the scaled synthetic traces. All results are for a 5-minute rescheduling penalty.

Algorithm	Degradation from bound		
	avg.	std.	max
/stretch-per/OPT=MAX/MFT=300	40.2	24.8	236.9
/stretch-per/OPT=MAX/MFT=600	40.2	24.8	236.9
/stretch-per/OPT=MAX/MVT=300	40.2	24.8	236.9
/stretch-per/OPT=MAX/MVT=600	40.2	24.8	236.9

Table B.4. Preemption and migration bandwidth consumption for DFRS algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Bandwidth consumption (GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
Greedy*/OPT=AVG	0.00	0.00	0.00	0.00
Greedy*/OPT=MIN	0.00	0.00	0.00	0.00
Greedy/per/OPT=AVG	0.44	1.02	0.21	0.63
Greedy/per/OPT=AVG/MFT=300	0.44	1.04	0.20	0.62
Greedy/per/OPT=AVG/MFT=600	0.44	1.03	0.20	0.62
Greedy/per/OPT=AVG/MVT=300	0.44	1.03	0.20	0.63
Greedy/per/OPT=AVG/MVT=600	0.44	1.04	0.19	0.60
Greedy/per/OPT=MIN	0.48	1.08	0.21	0.60
Greedy/per/OPT=MIN/MFT=300	0.47	1.08	0.20	0.59
Greedy/per/OPT=MIN/MFT=600	0.47	1.08	0.19	0.58
Greedy/per/OPT=MIN/MVT=300	0.47	1.06	0.19	0.57
Greedy/per/OPT=MIN/MVT=600	0.47	1.07	0.18	0.58
Greedy*/per/OPT=AVG	0.45	1.28	0.28	0.69
Greedy*/per/OPT=AVG/MFT=300	0.45	1.27	0.27	0.66
Greedy*/per/OPT=AVG/MFT=600	0.44	1.26	0.26	0.67
Greedy*/per/OPT=AVG/MVT=300	0.44	1.26	0.26	0.65
Greedy*/per/OPT=AVG/MVT=600	0.44	1.26	0.25	0.65
Greedy*/per/OPT=MIN	0.50	1.29	0.27	0.66
Greedy*/per/OPT=MIN/MFT=300	0.50	1.29	0.26	0.65
Greedy*/per/OPT=MIN/MFT=600	0.50	1.29	0.26	0.63
Greedy*/per/OPT=MIN/MVT=300	0.50	1.27	0.26	0.63
Greedy*/per/OPT=MIN/MVT=600	0.49	1.27	0.24	0.62
GreedyP*/OPT=AVG	0.06	0.17	0.00	0.00
GreedyP*/OPT=MIN	0.06	0.17	0.00	0.00
GreedyP/per/OPT=AVG	0.46	1.05	0.20	0.64
GreedyP/per/OPT=AVG/MFT=300	0.46	1.07	0.19	0.61
GreedyP/per/OPT=AVG/MFT=600	0.46	1.06	0.19	0.60
GreedyP/per/OPT=AVG/MVT=300	0.46	1.07	0.19	0.62
GreedyP/per/OPT=AVG/MVT=600	0.46	1.05	0.18	0.60
GreedyP/per/OPT=MIN	0.50	1.11	0.20	0.60
GreedyP/per/OPT=MIN/MFT=300	0.49	1.11	0.19	0.58
GreedyP/per/OPT=MIN/MFT=600	0.49	1.10	0.18	0.58
GreedyP/per/OPT=MIN/MVT=300	0.49	1.10	0.18	0.57
GreedyP/per/OPT=MIN/MVT=600	0.49	1.11	0.18	0.57
GreedyP*/per/OPT=AVG	0.53	1.36	0.28	0.68
GreedyP*/per/OPT=AVG/MFT=300	0.52	1.35	0.27	0.69

Table B.4. (Continued) Preemption and migration bandwidth consumption for DFRS algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Bandwidth consumption (GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
GreedyP*/per/OPT=AVG/MFT=600	0.52	1.36	0.26	0.69
GreedyP*/per/OPT=AVG/MVT=300	0.52	1.35	0.26	0.66
GreedyP*/per/OPT=AVG/MVT=600	0.51	1.35	0.25	0.65
GreedyP*/per/OPT=MIN	0.58	1.37	0.28	0.65
GreedyP*/per/OPT=MIN/MFT=300	0.57	1.38	0.26	0.65
GreedyP*/per/OPT=MIN/MFT=600	0.57	1.38	0.26	0.65
GreedyP*/per/OPT=MIN/MVT=300	0.57	1.37	0.25	0.62
GreedyP*/per/OPT=MIN/MVT=600	0.56	1.36	0.24	0.63
GreedyPM*/OPT=AVG	0.03	0.08	0.02	0.05
GreedyPM*/OPT=MIN	0.03	0.07	0.02	0.05
GreedyPM/per/OPT=AVG	0.46	1.05	0.21	0.64
GreedyPM/per/OPT=AVG/MFT=300	0.46	1.05	0.20	0.64
GreedyPM/per/OPT=AVG/MFT=600	0.45	1.04	0.20	0.61
GreedyPM/per/OPT=AVG/MVT=300	0.45	1.06	0.20	0.62
GreedyPM/per/OPT=AVG/MVT=600	0.45	1.06	0.19	0.60
GreedyPM/per/OPT=MIN	0.49	1.10	0.21	0.60
GreedyPM/per/OPT=MIN/MFT=300	0.49	1.10	0.20	0.61
GreedyPM/per/OPT=MIN/MFT=600	0.49	1.10	0.19	0.58
GreedyPM/per/OPT=MIN/MVT=300	0.49	1.10	0.19	0.57
GreedyPM/per/OPT=MIN/MVT=600	0.49	1.10	0.18	0.57
GreedyPM*/per/OPT=AVG	0.51	1.33	0.29	0.68
GreedyPM*/per/OPT=AVG/MFT=300	0.50	1.33	0.28	0.69
GreedyPM*/per/OPT=AVG/MFT=600	0.50	1.35	0.27	0.68
GreedyPM*/per/OPT=AVG/MVT=300	0.49	1.33	0.27	0.67
GreedyPM*/per/OPT=AVG/MVT=600	0.49	1.34	0.26	0.65
GreedyPM*/per/OPT=MIN	0.56	1.37	0.29	0.66
GreedyPM*/per/OPT=MIN/MFT=300	0.55	1.36	0.27	0.66
GreedyPM*/per/OPT=MIN/MFT=600	0.55	1.36	0.27	0.65
GreedyPM*/per/OPT=MIN/MVT=300	0.55	1.36	0.27	0.64
GreedyPM*/per/OPT=MIN/MVT=600	0.54	1.34	0.26	0.62
MCB*/OPT=AVG	0.38	1.15	1.16	2.35
MCB*/OPT=AVG/MFT=300	0.18	0.99	0.86	2.98
MCB*/OPT=AVG/MFT=600	0.14	0.67	0.73	2.60
MCB*/OPT=AVG/MVT=300	0.13	0.48	0.61	2.09
MCB*/OPT=AVG/MVT=600	0.12	0.39	0.53	1.49
MCB*/OPT=MIN	0.42	1.26	1.17	2.36

Table B.4. (Continued) Preemption and migration bandwidth consumption for DFRS algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Bandwidth consumption (GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
MCB*/OPT=MIN/MFT=300	0.20	0.98	0.86	2.89
MCB*/OPT=MIN/MFT=600	0.15	0.78	0.75	2.72
MCB*/OPT=MIN/MVT=300	0.13	0.72	0.63	2.12
MCB*/OPT=MIN/MVT=600	0.13	0.37	0.53	1.51
MCB/per/OPT=AVG	0.52	1.07	0.69	2.92
MCB/per/OPT=AVG/MFT=300	0.50	1.05	0.55	2.14
MCB/per/OPT=AVG/MFT=600	0.49	1.04	0.51	1.66
MCB/per/OPT=AVG/MVT=300	0.49	1.07	0.47	1.32
MCB/per/OPT=AVG/MVT=600	0.49	1.06	0.43	1.18
MCB/per/OPT=MIN	0.56	1.10	0.69	3.03
MCB/per/OPT=MIN/MFT=300	0.54	1.10	0.55	2.04
MCB/per/OPT=MIN/MFT=600	0.53	1.09	0.51	1.68
MCB/per/OPT=MIN/MVT=300	0.53	1.11	0.47	1.40
MCB/per/OPT=MIN/MVT=600	0.53	1.12	0.43	1.12
MCB*/per/OPT=AVG	0.67	1.08	1.21	2.54
MCB*/per/OPT=AVG/MFT=300	0.54	1.05	0.89	3.12
MCB*/per/OPT=AVG/MFT=600	0.51	1.03	0.77	2.69
MCB*/per/OPT=AVG/MVT=300	0.50	1.07	0.65	2.08
MCB*/per/OPT=AVG/MVT=600	0.50	1.09	0.57	1.53
MCB*/per/OPT=MIN	0.72	1.15	1.21	2.68
MCB*/per/OPT=MIN/MFT=300	0.58	1.12	0.89	3.02
MCB*/per/OPT=MIN/MFT=600	0.55	1.10	0.77	2.76
MCB*/per/OPT=MIN/MVT=300	0.54	1.11	0.65	2.16
MCB*/per/OPT=MIN/MVT=600	0.54	1.11	0.56	1.53
/per/OPT=AVG	0.45	1.02	0.21	0.64
/per/OPT=AVG/MFT=300	0.45	1.03	0.21	0.64
/per/OPT=AVG/MFT=600	0.45	1.04	0.21	0.64
/per/OPT=AVG/MVT=300	0.45	1.02	0.21	0.63
/per/OPT=AVG/MVT=600	0.45	1.03	0.20	0.62
/per/OPT=MIN	0.49	1.07	0.21	0.62
/per/OPT=MIN/MFT=300	0.49	1.07	0.21	0.62
/per/OPT=MIN/MFT=600	0.49	1.07	0.21	0.62
/per/OPT=MIN/MVT=300	0.49	1.08	0.20	0.60
/per/OPT=MIN/MVT=600	0.49	1.08	0.19	0.58
/stretch-per/OPT=AVG	0.28	0.66	0.39	0.79
/stretch-per/OPT=AVG/MFT=300	0.28	0.66	0.39	0.78

Table B.4. (Continued) Preemption and migration bandwidth consumption for DFRS algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Bandwidth consumption (GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
/stretch-per/OPT=AVG/MFT=600	0.28	0.66	0.39	0.78
/stretch-per/OPT=AVG/MVT=300	0.28	0.68	0.38	0.79
/stretch-per/OPT=AVG/MVT=600	0.28	0.68	0.37	0.78
/stretch-per/OPT=MAX	0.28	0.65	0.39	0.81
/stretch-per/OPT=MAX/MFT=300	0.28	0.65	0.39	0.81
/stretch-per/OPT=MAX/MFT=600	0.28	0.65	0.39	0.81
/stretch-per/OPT=MAX/MVT=300	0.28	0.65	0.38	0.81
/stretch-per/OPT=MAX/MVT=600	0.28	0.64	0.37	0.78

Table B.5. Preemption and migration frequency in terms of number of preemption and migration occurrences per hour. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / hour			
	pmtn		mig	
	avg.	max	avg.	max
Greedy*/OPT=AVG	0.00	0.00	0.00	0.00
Greedy*/OPT=MIN	0.00	0.00	0.00	0.00
Greedy/per/OPT=AVG	30.12	75.96	38.81	124.56
Greedy/per/OPT=AVG/MFT=300	29.82	75.24	36.41	110.88
Greedy/per/OPT=AVG/MFT=600	29.63	75.96	35.46	112.32
Greedy/per/OPT=AVG/MVT=300	29.69	74.88	35.46	118.08
Greedy/per/OPT=AVG/MVT=600	29.55	74.52	33.91	107.64
Greedy/per/OPT=MIN	32.58	83.52	38.79	110.52
Greedy/per/OPT=MIN/MFT=300	32.28	82.80	36.23	107.28
Greedy/per/OPT=MIN/MFT=600	32.18	82.80	35.15	106.56
Greedy/per/OPT=MIN/MVT=300	32.25	82.80	35.11	103.68
Greedy/per/OPT=MIN/MVT=600	32.04	83.16	33.60	103.68
Greedy*/per/OPT=AVG	26.07	73.08	55.45	126.00
Greedy*/per/OPT=AVG/MFT=300	25.68	71.64	53.16	119.16
Greedy*/per/OPT=AVG/MFT=600	25.39	71.28	51.96	117.36
Greedy*/per/OPT=AVG/MVT=300	25.23	71.28	51.74	123.12
Greedy*/per/OPT=AVG/MVT=600	24.86	70.56	50.07	120.24
Greedy*/per/OPT=MIN	29.27	84.96	58.06	124.56
Greedy*/per/OPT=MIN/MFT=300	28.74	83.16	55.46	123.84
Greedy*/per/OPT=MIN/MFT=600	28.58	83.88	54.32	123.12
Greedy*/per/OPT=MIN/MVT=300	28.50	83.52	53.86	120.96
Greedy*/per/OPT=MIN/MVT=600	28.08	83.52	51.97	117.36
GreedyP*/OPT=AVG	5.78	20.52	0.00	0.00
GreedyP*/OPT=MIN	5.67	18.00	0.00	0.00
GreedyP/per/OPT=AVG	30.86	76.68	37.70	111.96
GreedyP/per/OPT=AVG/MFT=300	30.46	74.88	35.37	108.72
GreedyP/per/OPT=AVG/MFT=600	30.31	75.24	34.51	110.88
GreedyP/per/OPT=AVG/MVT=300	30.39	77.04	34.63	106.56
GreedyP/per/OPT=AVG/MVT=600	30.17	77.04	33.08	103.68
GreedyP/per/OPT=MIN	33.34	85.32	37.52	107.64
GreedyP/per/OPT=MIN/MFT=300	32.95	83.88	35.05	104.40
GreedyP/per/OPT=MIN/MFT=600	32.79	84.60	34.07	103.68
GreedyP/per/OPT=MIN/MVT=300	32.88	84.60	34.17	105.84
GreedyP/per/OPT=MIN/MVT=600	32.70	83.52	32.74	101.52
GreedyP*/per/OPT=AVG	37.08	83.52	56.42	123.12
GreedyP*/per/OPT=AVG/MFT=300	35.90	84.96	53.48	123.12

Table B.5. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per hour. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / hour			
	pmtn		mig	
	avg.	max	avg.	max
GreedyP*/per/OPT=AVG/MFT=600	35.58	84.60	52.38	120.60
GreedyP*/per/OPT=AVG/MVT=300	35.33	84.24	52.26	117.00
GreedyP*/per/OPT=AVG/MVT=600	34.70	82.80	50.41	114.84
GreedyP*/per/OPT=MIN	39.67	97.92	58.56	127.08
GreedyP*/per/OPT=MIN/MFT=300	38.67	98.28	55.64	124.20
GreedyP*/per/OPT=MIN/MFT=600	38.39	96.48	54.46	122.40
GreedyP*/per/OPT=MIN/MVT=300	38.27	97.56	53.82	119.52
GreedyP*/per/OPT=MIN/MVT=600	37.70	97.20	52.14	119.16
GreedyPM*/OPT=AVG	2.31	9.72	3.75	14.04
GreedyPM*/OPT=MIN	2.25	10.08	3.69	13.32
GreedyPM/per/OPT=AVG	30.36	77.04	39.63	114.84
GreedyPM/per/OPT=AVG/MFT=300	29.96	75.24	37.04	114.12
GreedyPM/per/OPT=AVG/MFT=600	29.79	77.40	36.13	109.80
GreedyPM/per/OPT=AVG/MVT=300	29.90	75.60	36.03	113.40
GreedyPM/per/OPT=AVG/MVT=600	29.70	77.04	34.83	116.28
GreedyPM/per/OPT=MIN	32.93	84.24	39.21	112.32
GreedyPM/per/OPT=MIN/MFT=300	32.46	84.60	36.78	108.36
GreedyPM/per/OPT=MIN/MFT=600	32.34	83.88	35.87	108.72
GreedyPM/per/OPT=MIN/MVT=300	32.44	84.24	35.78	108.72
GreedyPM/per/OPT=MIN/MVT=600	32.24	83.16	34.31	106.56
GreedyPM*/per/OPT=AVG	32.78	80.28	60.90	129.60
GreedyPM*/per/OPT=AVG/MFT=300	31.93	81.00	57.83	123.48
GreedyPM*/per/OPT=AVG/MFT=600	31.57	81.00	56.72	119.88
GreedyPM*/per/OPT=AVG/MVT=300	31.19	78.84	56.59	121.32
GreedyPM*/per/OPT=AVG/MVT=600	30.60	79.56	54.88	120.96
GreedyPM*/per/OPT=MIN	35.78	96.48	62.95	135.72
GreedyPM*/per/OPT=MIN/MFT=300	34.88	95.04	59.84	132.48
GreedyPM*/per/OPT=MIN/MFT=600	34.45	92.88	58.68	130.32
GreedyPM*/per/OPT=MIN/MVT=300	34.25	94.32	58.31	128.52
GreedyPM*/per/OPT=MIN/MVT=600	33.80	94.32	56.45	127.08
MCB*/OPT=AVG	56.54	214.56	470.14	998.28
MCB*/OPT=AVG/MFT=300	20.23	162.72	327.56	1,313.64
MCB*/OPT=AVG/MFT=600	12.86	90.36	279.56	1,201.68
MCB*/OPT=AVG/MVT=300	10.04	51.84	238.86	1,101.60
MCB*/OPT=AVG/MVT=600	9.27	37.80	206.62	822.60
MCB*/OPT=MIN	61.66	230.40	490.48	1,005.48

Table B.5. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per hour. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / hour			
	pmtn		mig	
	avg.	max	avg.	max
MCB*/OPT=MIN/MFT=300	21.60	174.96	337.86	1,343.16
MCB*/OPT=MIN/MFT=600	13.24	102.96	291.29	1,306.80
MCB*/OPT=MIN/MVT=300	10.25	78.48	247.80	1,083.24
MCB*/OPT=MIN/MVT=600	9.51	38.16	212.16	825.48
MCB/per/OPT=AVG	38.66	83.52	235.57	1,181.88
MCB/per/OPT=AVG/MFT=300	35.19	80.28	164.35	798.48
MCB/per/OPT=AVG/MFT=600	34.72	79.56	149.35	709.92
MCB/per/OPT=AVG/MVT=300	34.57	80.64	139.75	600.84
MCB/per/OPT=AVG/MVT=600	34.15	78.48	124.47	480.24
MCB/per/OPT=MIN	41.59	86.76	243.13	1,269.00
MCB/per/OPT=MIN/MFT=300	38.09	86.04	167.67	826.56
MCB/per/OPT=MIN/MFT=600	37.47	84.96	151.97	755.64
MCB/per/OPT=MIN/MVT=300	37.32	86.40	142.42	619.92
MCB/per/OPT=MIN/MVT=600	36.93	86.76	126.13	490.68
MCB*/per/OPT=AVG	70.98	163.44	461.18	1,045.44
MCB*/per/OPT=AVG/MFT=300	44.52	128.52	314.64	1,351.80
MCB*/per/OPT=AVG/MFT=600	38.07	83.52	263.46	1,245.60
MCB*/per/OPT=AVG/MVT=300	35.99	80.64	222.61	1,057.32
MCB*/per/OPT=AVG/MVT=600	35.09	77.04	191.91	795.24
MCB*/per/OPT=MIN	77.04	170.28	479.31	1,109.52
MCB*/per/OPT=MIN/MFT=300	47.63	135.00	325.09	1,370.16
MCB*/per/OPT=MIN/MFT=600	41.08	93.60	271.59	1,337.76
MCB*/per/OPT=MIN/MVT=300	38.88	86.40	227.80	1,129.32
MCB*/per/OPT=MIN/MVT=600	37.94	85.32	194.57	836.28
/per/OPT=AVG	31.29	76.32	38.76	113.76
/per/OPT=AVG/MFT=300	31.17	75.96	39.00	114.84
/per/OPT=AVG/MFT=600	31.23	75.96	38.96	113.40
/per/OPT=AVG/MVT=300	31.16	75.24	37.86	111.24
/per/OPT=AVG/MVT=600	31.04	75.24	36.62	111.60
/per/OPT=MIN	33.83	84.24	38.69	111.24
/per/OPT=MIN/MFT=300	33.83	84.24	38.69	111.24
/per/OPT=MIN/MFT=600	33.83	84.24	38.69	111.24
/per/OPT=MIN/MVT=300	33.88	83.52	37.62	107.64
/per/OPT=MIN/MVT=600	33.76	84.60	36.21	104.04
/stretch-per/OPT=AVG	20.64	43.20	62.89	140.04
/stretch-per/OPT=AVG/MFT=300	20.65	42.48	63.03	140.40

Table B.5. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per hour. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / hour			
	pmtn		mig	
	avg.	max	avg.	max
/stretch-per/OPT=AVG/MFT=600	20.62	43.56	62.88	138.96
/stretch-per/OPT=AVG/MVT=300	20.62	43.20	61.58	136.80
/stretch-per/OPT=AVG/MVT=600	20.60	43.56	59.78	132.48
/stretch-per/OPT=MAX	20.41	45.36	67.26	159.48
/stretch-per/OPT=MAX/MFT=300	20.41	45.36	67.26	159.48
/stretch-per/OPT=MAX/MFT=600	20.41	45.36	67.26	159.48
/stretch-per/OPT=MAX/MVT=300	20.35	46.44	65.60	158.40
/stretch-per/OPT=MAX/MVT=600	20.25	46.80	63.87	153.36

Table B.6. Preemption and migration frequency in terms of number of preemption and migration occurrences per job. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / job			
	pmtn		mig	
	avg.	max	avg.	max
Greedy*/OPT=AVG	0.00	0.00	0.00	0.00
Greedy*/OPT=MIN	0.00	0.00	0.00	0.00
Greedy/per/OPT=AVG	5.03	21.58	4.79	18.26
Greedy/per/OPT=AVG/MFT=300	4.98	21.52	4.52	15.80
Greedy/per/OPT=AVG/MFT=600	4.94	20.66	4.41	16.26
Greedy/per/OPT=AVG/MVT=300	4.95	21.03	4.40	17.08
Greedy/per/OPT=AVG/MVT=600	4.91	21.01	4.22	15.65
Greedy/per/OPT=MIN	5.41	21.76	4.81	16.17
Greedy/per/OPT=MIN/MFT=300	5.36	21.55	4.52	15.40
Greedy/per/OPT=MIN/MFT=600	5.33	21.31	4.39	15.27
Greedy/per/OPT=MIN/MVT=300	5.34	21.83	4.37	15.21
Greedy/per/OPT=MIN/MVT=600	5.29	21.94	4.20	14.83
Greedy*/per/OPT=AVG	3.95	20.26	6.59	17.29
Greedy*/per/OPT=AVG/MFT=300	3.89	19.76	6.36	16.62
Greedy*/per/OPT=AVG/MFT=600	3.85	19.57	6.23	16.52
Greedy*/per/OPT=AVG/MVT=300	3.81	19.60	6.18	16.42
Greedy*/per/OPT=AVG/MVT=600	3.75	19.24	5.98	16.01
Greedy*/per/OPT=MIN	4.49	22.55	6.94	17.65
Greedy*/per/OPT=MIN/MFT=300	4.41	22.07	6.66	17.31
Greedy*/per/OPT=MIN/MFT=600	4.38	21.89	6.55	17.38
Greedy*/per/OPT=MIN/MVT=300	4.37	21.71	6.48	17.26
Greedy*/per/OPT=MIN/MVT=600	4.29	21.80	6.25	16.53
GreedyP*/OPT=AVG	0.58	2.09	0.00	0.00
GreedyP*/OPT=MIN	0.57	2.04	0.00	0.00
GreedyP/per/OPT=AVG	5.16	21.11	4.67	16.20
GreedyP/per/OPT=AVG/MFT=300	5.09	21.05	4.41	15.45
GreedyP/per/OPT=AVG/MFT=600	5.05	20.97	4.32	15.12
GreedyP/per/OPT=AVG/MVT=300	5.06	20.52	4.33	15.29
GreedyP/per/OPT=AVG/MVT=600	5.02	20.36	4.14	14.68
GreedyP/per/OPT=MIN	5.54	22.20	4.67	15.76
GreedyP/per/OPT=MIN/MFT=300	5.47	21.45	4.39	15.09
GreedyP/per/OPT=MIN/MFT=600	5.45	21.41	4.28	15.30
GreedyP/per/OPT=MIN/MVT=300	5.46	21.94	4.29	15.04
GreedyP/per/OPT=MIN/MVT=600	5.41	21.37	4.11	14.29
GreedyP*/per/OPT=AVG	5.37	22.68	6.78	17.07
GreedyP*/per/OPT=AVG/MFT=300	5.21	22.27	6.47	17.30

Table B.6. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per job. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / job			
	pmtn		mig	
	avg.	max	avg.	max
GreedyP*/per/OPT=AVG/MFT=600	5.17	21.34	6.35	17.24
GreedyP*/per/OPT=AVG/MVT=300	5.12	21.58	6.32	16.78
GreedyP*/per/OPT=AVG/MVT=600	5.03	21.52	6.09	16.26
GreedyP*/per/OPT=MIN	5.87	24.87	7.09	17.91
GreedyP*/per/OPT=MIN/MFT=300	5.73	23.92	6.78	17.45
GreedyP*/per/OPT=MIN/MFT=600	5.69	24.22	6.65	17.17
GreedyP*/per/OPT=MIN/MVT=300	5.66	24.30	6.54	17.04
GreedyP*/per/OPT=MIN/MVT=600	5.56	23.88	6.34	16.48
GreedyPM*/OPT=AVG	0.24	1.11	0.37	1.18
GreedyPM*/OPT=MIN	0.23	1.19	0.36	1.22
GreedyPM/per/OPT=AVG	5.11	20.79	4.88	16.67
GreedyPM/per/OPT=AVG/MFT=300	5.03	20.61	4.60	15.78
GreedyPM/per/OPT=AVG/MFT=600	5.01	20.41	4.50	15.52
GreedyPM/per/OPT=AVG/MVT=300	5.02	20.99	4.48	15.41
GreedyPM/per/OPT=AVG/MVT=600	4.98	21.06	4.33	16.83
GreedyPM/per/OPT=MIN	5.52	21.65	4.85	16.03
GreedyPM/per/OPT=MIN/MFT=300	5.43	21.97	4.58	15.71
GreedyPM/per/OPT=MIN/MFT=600	5.41	21.39	4.49	15.59
GreedyPM/per/OPT=MIN/MVT=300	5.42	21.67	4.45	15.15
GreedyPM/per/OPT=MIN/MVT=600	5.37	22.19	4.28	15.07
GreedyPM*/per/OPT=AVG	4.88	22.02	7.30	18.27
GreedyPM*/per/OPT=AVG/MFT=300	4.76	21.49	6.98	17.71
GreedyPM*/per/OPT=AVG/MFT=600	4.70	21.30	6.85	17.02
GreedyPM*/per/OPT=AVG/MVT=300	4.65	21.03	6.81	17.11
GreedyPM*/per/OPT=AVG/MVT=600	4.55	20.70	6.61	17.24
GreedyPM*/per/OPT=MIN	5.42	24.00	7.59	18.32
GreedyPM*/per/OPT=MIN/MFT=300	5.29	23.42	7.27	18.23
GreedyPM*/per/OPT=MIN/MFT=600	5.23	23.76	7.14	17.80
GreedyPM*/per/OPT=MIN/MVT=300	5.19	23.12	7.07	17.32
GreedyPM*/per/OPT=MIN/MVT=600	5.11	23.23	6.84	16.94
MCB*/OPT=AVG	6.10	17.36	55.01	86.16
MCB*/OPT=AVG/MFT=300	2.02	14.26	31.45	80.96
MCB*/OPT=AVG/MFT=600	1.31	9.68	25.24	65.40
MCB*/OPT=AVG/MVT=300	1.03	2.88	20.89	42.69
MCB*/OPT=AVG/MVT=600	0.97	3.42	18.17	41.67
MCB*/OPT=MIN	6.67	18.83	57.46	90.29

Table B.6. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per job. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / job			
	pmtn		mig	
	avg.	max	avg.	max
MCB*/OPT=MIN/MFT=300	2.18	15.58	32.59	89.17
MCB*/OPT=MIN/MFT=600	1.37	8.67	26.28	68.39
MCB*/OPT=MIN/MVT=300	1.06	3.08	21.73	46.10
MCB*/OPT=MIN/MVT=600	1.00	2.89	18.68	41.15
MCB/per/OPT=AVG	6.34	25.03	25.02	41.57
MCB/per/OPT=AVG/MFT=300	6.04	24.52	18.31	36.68
MCB/per/OPT=AVG/MFT=600	5.97	24.57	16.87	35.67
MCB/per/OPT=AVG/MVT=300	5.94	25.07	15.76	33.88
MCB/per/OPT=AVG/MVT=600	5.85	24.90	14.18	33.19
MCB/per/OPT=MIN	6.83	26.38	25.74	43.24
MCB/per/OPT=MIN/MFT=300	6.51	26.03	18.75	38.89
MCB/per/OPT=MIN/MFT=600	6.42	26.10	17.23	36.68
MCB/per/OPT=MIN/MVT=300	6.39	26.37	16.13	36.79
MCB/per/OPT=MIN/MVT=600	6.30	26.13	14.45	34.48
MCB*/per/OPT=AVG	11.97	26.78	70.57	106.14
MCB*/per/OPT=AVG/MFT=300	7.54	26.28	39.94	105.55
MCB*/per/OPT=AVG/MFT=600	6.59	24.99	31.32	78.30
MCB*/per/OPT=AVG/MVT=300	6.24	25.69	25.53	53.96
MCB*/per/OPT=AVG/MVT=600	6.08	25.58	22.24	50.41
MCB*/per/OPT=MIN	13.01	28.93	73.61	117.15
MCB*/per/OPT=MIN/MFT=300	8.11	27.86	41.44	112.18
MCB*/per/OPT=MIN/MFT=600	7.13	27.14	32.52	81.60
MCB*/per/OPT=MIN/MVT=300	6.72	27.27	26.33	54.97
MCB*/per/OPT=MIN/MVT=600	6.57	26.52	22.55	50.29
/per/OPT=AVG	5.25	22.34	4.89	16.87
/per/OPT=AVG/MFT=300	5.24	22.85	4.92	17.02
/per/OPT=AVG/MFT=600	5.24	22.90	4.92	16.88
/per/OPT=AVG/MVT=300	5.23	22.16	4.78	16.52
/per/OPT=AVG/MVT=600	5.20	22.95	4.63	16.52
/per/OPT=MIN	5.65	23.23	4.90	16.58
/per/OPT=MIN/MFT=300	5.65	23.23	4.90	16.58
/per/OPT=MIN/MFT=600	5.65	23.23	4.90	16.58
/per/OPT=MIN/MVT=300	5.65	23.25	4.77	16.00
/per/OPT=MIN/MVT=600	5.63	22.78	4.59	15.51
/stretch-per/OPT=AVG	3.79	16.03	9.58	23.98
/stretch-per/OPT=AVG/MFT=300	3.79	15.86	9.60	24.02

Table B.6. (Continued) Preemption and migration frequency in terms of number of preemption and migration occurrences per job. Average and maximum values over scaled synthetic traces with load ≥ 0.7.

Algorithm	Occurrences / job			
	pmtn		mig	
	avg.	max	avg.	max
/stretch-per/OPT=AVG/MFT=600	3.78	16.13	9.58	23.79
/stretch-per/OPT=AVG/MVT=300	3.77	16.24	9.36	23.27
/stretch-per/OPT=AVG/MVT=600	3.77	16.00	9.11	22.63
/stretch-per/OPT=MAX	3.79	16.71	10.41	26.96
/stretch-per/OPT=MAX/MFT=300	3.79	16.71	10.41	26.96
/stretch-per/OPT=MAX/MFT=600	3.79	16.71	10.41	26.96
/stretch-per/OPT=MAX/MVT=300	3.78	17.13	10.14	26.75
/stretch-per/OPT=MAX/MVT=600	3.76	17.52	9.87	25.78

BIBLIOGRAPHY

[1] A. Buttari, J. Kurzak, and J. Dongarra, “Limitations of the PlayStation 3 for high performance cluster computing,” Innovative Computing Laboratory, University of Tennessee Knoxville, Tech. Rep. UT-CS-07-597, Apr. 2007.

[2] M. A. Baker, “Cluster computing whitepaper,” arXiv:cs/0004014v2, Dec. 2001.

[3] “Top 500 supercomputer sites.” [Online]. Available: http://www.top500.org/

[4] S. Warfield, F. Jolesz, and R. Kikinis, “A high performance computing approach to the registration of medical imaging data,” *Parallel Computing*, vol. 24, no. 9-10, pp. 1345–1368, 1998.

[5] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S.-i. Habata, M. Yokokawa, H. Itoh, and K. Otsuka, “A 26.58 Tflops global atmospheric simulation with the spectral transform method on the earth simulator,” in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov. 2002, pp. 1–19.

[6] Y. M. Rhee and V. S. Pande, “Multiplexed-replica exchange molecular dynamics method for protein folding simulation,” *Biophysical Journal*, vol. 84, no. 2, pp. 775–786, 2003.

[7] O. G. Staadt, J. Walker, C. Nuber, and B. Hamann, “A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering,” in *ACM SIGGRAPH Conference in Asia courses*, Dec. 2008.

[8] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec. 2002, pp. 137–150.

[9] “Apache hadoop project.” [Online]. Available: http://hadoop.apache.org/

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM Press, Mar. 2007, pp. 59–72.

[11] “IBM introduces ready-to-use cloud computing,” 2007. [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/22613.wss

[12] “Amazon elastic compute cloud.” [Online]. Available: http://aws.amazon.com/ec2

[13] J. G. Koomey, “Estimating total power consumption by servers in the U.S. and the world,” Feb. 2007. [Online]. Available: http://enterprise.amd.com/Downloads/svrpwrusecompletefinal.pdf

[14] “Report to congress on server and data center energy efficiency,” U.S. Environmental Protection Agency, Aug. 2007. [Online]. Available: http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf

[15] C. B. Lee and A. E. Snavely, “Precise and realistic utility functions for user-centric performance analysis of schedulers,” in *Proceedings of the 16th ACM International Symposium on High-Performance Distributed Computing*, Jun. 2007, pp. 107–116.

[16] U. Schwiegelshohn and R. Yahyapour, “Fairness in parallel job scheduling,” *Journal of Scheduling*, vol. 3, no. 5, pp. 297–320, 2000.

[17] D. Lifka, “The ANL/IBM SP scheduling system,” in *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1995, vol. 949, pp. 295–303.

[18] C. B. Lee and A. E. Snavely, “On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions,” *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 495–506, 2006.

[19] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, “Model-based resource provisioning in a web service utility,” in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*. USENIX, Mar. 2003, pp. 57–71.

[20] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, “Workload analysis and demand prediction of enterprise data center applications,” in *Proceedings of the 2007 IEEE International Symposium on Workload Characterization*, Sep. 2007, pp. 171–180.

[21] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper, “An integrated approach to resource pool management: Policies, efficiency and quality metrics,” in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society Press, Jun. 2008, pp. 326–335.

[22] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, “1000 islands: Integrated capacity and workload management for the next generation data center,” in

Proceedings of the 5th IEEE International Conference on Autonomic Computing, Jun. 2008, pp. 172–181.

[23] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster reserves: A mechanism for resource management in cluster-based network servers," in *Proceedings of the 2000 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Jun. 2000, pp. 90–101.

[24] J. Rolia, A. Andrzejak, and M. Arlitt, "Automating enterprise application placement in resource utilities," in *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, ser. Lecture Notes in Computer Science, M. Brunner and A. Keller, Eds. Springer, Oct. 2003, vol. 2867, pp. 118–129.

[25] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. N. Tantawi, "Dynamic placement for clustered web applications," in *Proceedings of the 15th International Conference on the World Wide Web*, 2006, pp. 595–604.

[26] T. Kimbrel, M. Steinder, M. Sviridenko, and A. N. Tantawi, "Dynamic application placement under service and memory constraints," in *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms*, ser. Lecture Notes in Computer Science, S. E. Nikoletseas, Ed. Springer, May 2005, vol. 3503, pp. 391–402.

[27] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proceedings of the 16th International Conference on the World Wide Web*, May 2007, pp. 331–340.

[28] B. N. Chun and D. E. Culler, "User-centric performance analysis of market-based cluster batch schedulers," in *Proceedings of the 2nd IEEE International Symposium*

on Cluster Computing and the Grid. IEEE Computer Society Press, May 2002, pp. 30–38.

[29] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, “Characterization of backfilling strategies for parallel job scheduling,” in *Proceedings of the 2002 International Conference on Parallel Processing Workshops*, Aug. 2002, pp. 514–522.

[30] A. W. Mu’alem and D. G. Feitelson, “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[31] S.-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, “The impact of more accurate requested runtimes on production job scheduling performance,” in *Proceedings of the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2002, vol. 2537, pp. 103–127.

[32] J. P. Jones and B. Nitzberg, “Scheduling for parallel supercomputing: A historical perspective on achievable utilization,” in *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1999, vol. 1659, pp. 1–16.

[33] D. G. Feitelson, “Parallel workloads archive.” [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[34] A. Acharya and S. K. Setia, “Availability and utility of idle memory in workstation clusters,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, pp. 35–46, 1999.

[35] D. G. Feitelson, “Memory usage in the LANL CM-5 workload,” in *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1997, vol. 1291, pp. 78–94.

[36] S. K. Setia, M. S. Squillante, and V. K. Naik, “The impact of job memory requirements on gang-scheduling performance,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 4, pp. 30–39, 1999.

[37] A. Batat and D. G. Feitelson, “Gang scheduling with memory considerations,” in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, May 2000, pp. 109–114.

[38] S.-H. Chiang and M. K. Vernon, “Characteristics of a large shared-memory production workload,” in *Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Jun. 2001, vol. 2221, pp. 159–187.

[39] H. Li, D. Groep, and L. Wolters, “Workload characteristics of a multi-cluster supercomputer,” in *Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2004, vol. 3277, pp. 176–193.

[40] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1997, vol. 1291, pp. 1–34.

[41] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 581–592, 2003.

[42] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd International Conferance on Distributed Computing Systems*, Oct. 1982, pp. 22–30.

[43] D. G. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *IEEE Computer*, vol. 23, no. 5, pp. 65–77, 1990.

[44] ——, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306–318, Dec. 1992.

[45] D. G. Feitelson and M. A. Jette, "Improved utilization and responsiveness with gang scheduling," in *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1997, vol. 1291, pp. 238–261.

[46] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling – a status report," in *Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2004, vol. 3277, pp. 1–16.

[47] F. Wang, M. Papaefthymiou, and M. S. Squillante, "Performance evaluation of gang scheduling for parallel and distributed multiprogramming," in *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1997, vol. 1291, pp. 277–298.

[48] A. Hori, H. Tezuka, and Y. Ishikawa, "Highly efficient gang scheduling implementation," in *Proceedings of the 1998 ACM/IEEE Conference on High Performance Networking and Computing*, Nov. 1998.

[49] M. A. Jette, "Performance characteristics of gang scheduling in multiprogrammed environments," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, Nov. 1997.

[50] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette, "An infrastructure for efficient parallel job execution in terascale computing environments," in *Proceedings of the 1998 ACM/IEEE Conference on High Performance Networking and Computing*, Nov. 1998.

[51] D. G. Feitelson, "Packing schemes for gang scheduling," in *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1996, vol. 1162, pp. 89–110.

[52] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "Implications of I/O for gang scheduled workloads," in *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Apr. 1997, vol. 1291, pp. 215–237.

[53] L. Cherkasova, D. Gupta, E. Ryabinkin, R. Kurakin, V. Dobretsov, and A. Vahdat, "Optimizing grid site manager performance with virtual machines," in *Proceedings of the 3rd USENIX Workshop on Real, Large Distributed Systems*, 2006. [Online]. Available: http://www.usenix.org/events/worlds06/tech/

[54] W. J. Leinberger, G. Karypis, and V. Kumar, "Gang scheduling for distributed memory systems," University of Minnesota Department of Computer Science and Engineering, Tech. Rep. 00-014, Feb. 2000.

[55] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, Apr. 2003.

[56] A. C. Arpaci-Dusseau, "Implicit coscheduling: coordinated scheduling with implicit information in distributed systems," Ph.D. Dissertation, University of California at Berkeley, Dec. 1998.

[57] ——, "Implicit coscheduling: coordinated scheduling with implicit information in distributed systems," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 283–331, 2001.

[58] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic coscheduling on workstation clusters," in *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Mar. 1998, vol. 1459, pp. 231–256.

[59] F. Petrini and W.-c. Feng, "Time-sharing parallel jobs in the presence of multiple resource requirements," in *Proceedings of the 6th Workshop on Job Scheduling*

Strategies for Parallel Processing, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, May 2000, vol. 1911, pp. 113–136.

[60] S. Agarwal, G. S. Choi, C. R. Das, A. B. Yoo, and S. Nagar, "Co-ordinated coscheduling in time-sharing clusters through a generic framework," in *Proceedings of the 5th IEEE International Conference on Cluster Computing*. IEEE Computer Society Press, Dec. 2003, pp. 84–91.

[61] A. Bouteiller, H.-L. Bouziane, T. Herault, P. Lemarinier, and F. Cappello, "Hybrid preemptive scheduling of MPI applications on grids," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Nov. 2004, pp. 130–137.

[62] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The interaction of parallel and sequential workloads on a network of workstations," *ACM SIGMETRICS Performance Evaluation Review*, vol. 23, no. 1, pp. 267–278, 1995.

[63] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das, "Coscheduling in clusters: Is it a viable alternative?" in *Proceedings of the 2004 ACM/IEEE Conference on High Performance Networking and Computing*, Nov. 2004.

[64] P. Strazdins and J. Uhlmann, "A comparison of local and gang scheduling on a Beowulf cluster," in *Proceedings of the 6th IEEE International Conference on Cluster Computing*. IEEE Computer Society Press, Sep. 2004, pp. 55–62.

[65] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Adaptive parallel job scheduling with flexible coscheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 11, pp. 1066–1077, 2005.

[66] C. Anglano, "A comparative evaluation of implicit coscheduling strategies fornetworks of workstations," in *Proceedings of the 9th IEEE International*

Symposium on High-Performance Distributed Computing, Aug. 2000, pp. 221–228.

[67] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das, "Alternatives to coscheduling a network of workstations," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 302–327, 1999.

[68] G. S. Choi, S. Agarwal, J.-H. Kim, A. B. Yoo, and C. R. Das, "Impact of job allocation strategies for communication driven coscheduling in clusters," in *Proceedings of 9th European International Conference on Parallel Processing*, ser. Lecture Notes in Computer Science, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds. Springer, Aug. 2003, vol. 2790, pp. 160–168.

[69] W. Emeneker, D. Jackson, J. Butikofer, and D. Stanzione, "Dynamic virtual clustering with Xen and Moab," in *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications Workshops*, ser. Lecture Notes in Computer Science, Dec. 2006, vol. 4331, pp. 440–451.

[70] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: resource management using virtual clusters on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 229–262, 2000.

[71] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum, "Sharing networked resources with brokered leases," in *Proceedings of the 2006 USENIX Annual Technical Conference*. USENIX, May/Jun. 2006.

[72] N. Kiyanclar, G. A. Koenig, and W. Yurcik, "Maestro-VC: A paravirtualized execution environment for secure on-demand cluster computing," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2006.

[73] L. Ramakrishnan, D. Irwin, L. Grit, A. Yumerefendi, A. Iamnitchi, and J. S. Chase, "Toward a doctrine of containment: Grid hosting and adaptive resource control," in *Proceedings of the 2006 ACM/IEEE Conference on High Performance Networking and Computing*, Nov. 2006.

[74] M. Rodriguez, D. Tapiador, J. Fontán, E. Huedo, R. S. Montero, and I. M. Llorente, "Dynamic provisioning of virtual clusters for grid computing," in *Proceedings of the 2008 Euro-Par Workshops*, ser. Lecture Notes in Computer Science, E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds. Springer, Aug. 2008, vol. 5415, pp. 23–32.

[75] S. Yamasaki, N. Maruyama, and S. Matsuoka, "Model-based resource selection for efficient virtual cluster deployment," in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, Nov. 2007.

[76] H. Nishimura, N. Maruyama, and S. Matsuoka, "Virtual clusters on the fly – fast, scalable, and flexible installation," in *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2007, pp. 549–556.

[77] W. Emeneker and D. Stanzione, "Increasing reliability through dynamic virtual clustering," in *Proceedings of the High Availability and Performance Computing Workshop*, Oct. 2006. [Online]. Available: http://xcr.cenit.latech.edu/hapcw2006/program

[78] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *Proceedings of the 17th ACM International Symposium on High-Performance Distributed Computing*, Jun. 2008, pp. 87–96.

[79] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, “Xen and the art of cluster scheduling,” in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, Nov. 2007.

[80] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen, “Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure,” in *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, Jun. 2006, pp. 5–14.

[81] L. V. Kalé, “Performance and productivity in parallel programming via processor virtualization,” in *Proceedings of the 1st Workshop on Productivity and Performance in High-End Computing*, Feb. 2004, pp. 40–49.

[82] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng, “Optimal online scheduling of parallel jobs with dependencies,” Carnegie Mellon University School of Computer Science, Tech. Rep. CMU-CS-92-189, Sep. 1992.

[83] B. Lin and P. A. Dinda, “Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling,” in *Proceedings of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*, Nov. 2005.

[84] S. Ali, J.-K. Kim, H. J. Siegel, and A. A. Maciejewski, “Static heuristics for robust resource allocation of continuously executing applications,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1070–1080, 2008.

[85] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, “Dynamic provisioning of multi-tier internet applications,” in *Proceedings of the 2nd IEEE International Conference on Autonomic Computing*, Jun. 2005, pp. 217–228.

[86] “Linode virtual hosting service.” [Online]. Available: http://linode.com/

[87] B. Urgaonkar, A. L. Rosenberg, and P. Shenoy, “Application placement on a cluster of servers,” *International Journal of Foundations of Computer Science*, vol. 18, no. 5, pp. 1023–1041, 2007.

[88] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishakumar, D. P. Pazel, J. Pershing, and B. Rochwerger, “Océano – SLA based management of a computing utility,” in *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, May 2001, pp. 855–868.

[89] B. Urgaonkar, P. Shenoy, and T. Roscoe, “Resource overbooking and application profiling in shared hosting platforms,” *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 239–254, 2002.

[90] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, “Flow and stretch metrics for scheduling continuous job streams,” in *Proceedings of the 9th Annual ACM-SIAM Symposium On Discrete Algorithms*, Jan. 1998, pp. 270–279.

[91] A. Legrand, A. Su, and F. Vivien, “Minimizing the stretch when scheduling flows of divisible requests,” *Journal of Scheduling*, vol. 11, no. 5, pp. 381–404, 2008.

[92] A. Chandra, W. Gong, and P. Shenoy, “Dynamic resource allocation for shared data centers using online measurements,” in *Proceedings of the 11th International Workshop on Quality of Service*, Jun. 2003, pp. 381–400.

[93] H. Nguyen Van, F. Dang Tran, and J.-M. Menaud, “Autonomic virtual resource management for service hosting platforms,” in *Proceedings of the ICSE 2009 Workshop on Software Engineering Challenges in Cloud Computing*, May 2009.

[94] ——, “SLA-aware virtual resource management for cloud infrastructures,” in *Proceedings of the 9th IEEE International Conference on Computer and Information Technology*, Oct. 2009, pp. 357–362.

[95] K. Shen, H. Tang, T. Yang, and L. Chu, "Integrated resource management for cluster-based internet services," in *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Dec. 2002.

[96] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef, "Performance management for cluster-based web services," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 12, pp. 2333–2343, Dec. 2005.

[97] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," INRIA, Tech. Rep. RR-6639, 2008.

[98] F. Hermenier, A. Lèbre, and J.-M. Menaud, "Cluster-wide context switch of virtualized jobs," INRIA, Tech. Rep. RR-6929, 2009. [Online]. Available: http://hal.inria.fr/inria-00383325/en/

[99] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Mar. 2009.

[100] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. M. Vahdat, and G. M. Voelker, "To infinity and beyond: Time warped network emulation," in *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, May 2006, pp. 87–100.

[101] M. N. Bennani and D. A. Menascé, "Resource allocation for autonomic data centers using analytic performance models," in *Proceedings of the 2nd IEEE International Conference on Autonomic Computing*, Jun. 2005, pp. 229–240.

[102] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé, "Utility-based placement of dynamic web applications with fairness goals," in *Proceedings of the*

11th Network Operations and Management Symposium. IEEE/IFIP, Apr. 2008, pp. 9–16.

[103] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai, “Translating service level objectives to lower level policies for multi-tier services,” *Cluster Computing*, vol. 11, no. 3, pp. 299–311, 2008.

[104] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, “Power and performance management of virtualized computing environments via lookahead control,” *Cluster Computing*, vol. 12, no. 1, pp. 1–15, 2009.

[105] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, “Adaptive control of virtualized resources in utility computing environments,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM Press, Mar. 2007, pp. 289–302.

[106] Y. Song, H. Wang, Y. Li, B. Feng, and Y. Sun, “Multi-tiered on-demand resource scheduling for VM-based data center,” in *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2009, pp. 148–155.

[107] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, “Utility functions in autonomic systems,” in *Proceedings of the 1st IEEE International Conference on Autonomic Computing*, May 2004, pp. 70–77.

[108] R. Wang and N. Kandasamy, “A distributed control framework for performance management of virtualized computing environments: some preliminary results,” in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, Jun. 2009, pp. 7–12. [Online]. Available: http://www.cs.duke.edu/nicl/acdc09/

[109] R. P. Goldberg, “Survey of virtual machine research,” *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[110] 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,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003, pp. 164–177.

[111] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews, “Xen and the art of repeated research,” in *Proceedings of the FREENIX Track of the 2004 USENIX Annual Technical Conference*, Jun./Jul. 2004, pp. 135–144.

[112] N. Bobroff, R. Coppinger, L. Fong, S. Seelam, and J. Xu, “Scalability analysis of job scheduling using virtual nodes,” in *Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Springer, May 2009, vol. 5798, pp. 190–206.

[113] L. Cherkasova, D. Gupta, and A. Vahdat, “When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments,” Hewlett-Packard Labs, Tech. Rep. HPL-2007-25, 2007.

[114] N. Bhatia and J. S. Vetter, “Virtual cluster management with Xen,” in *Proceedings of the 2007 Euro-Par Workshops*, ser. Lecture Notes in Computer Science, L. Bougé, M. Forsell, J. L. Träff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, Eds. Springer, Aug. 2007, vol. 4854, pp. 185–194.

[115] “Intel virtualization technology.” [Online]. Available: http://www.intel.com/technology/virtualization/index.htm

[116] C. Macdonell and P. Lu, “Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads,” in *Proceedings of the 2007 High Performance Computing & Simulation Conference*, Jun. 2007.

[117] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, “Virtualization for high-performance computing,” *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 8–11, 2006.

[118] F. Calzolari, “High availability using virtualization,” Ph.D. Dissertation, Universitá de Pisa, 2006.

[119] N. Barcelo, N. Legg, and T. Bressoud, “The performance cost of virtual machines on big data problems in compute clusters,” in *Proceedings of the Midstates Conference for Undergraduate Research in Computer Science and Mathematics*, Nov. 2008, pp. 22–29. [Online]. Available: http://www3.wooster.edu/cs/mcurcsm2008/papers/vmclusters.pdf

[120] W. Emeneker and D. Stanzione, “HPC cluster readiness of Xen and User Mode Linux,” in *Proceedings of the 8th IEEE International Conference on Cluster Computing*. IEEE Computer Society Press, Sep. 2006.

[121] W. Huang, J. Liu, B. Abali, and D. K. Panda, “A case for high performance computing with virtual machines,” in *Proceedings of the 20th Annual International Conference on Supercomputing*, Jun./Jul. 2006, pp. 125–134.

[122] A. Ranadive, M. Kesavan, A. Gavrilovska, and K. Schwan, “Performance implications of virtualizing multicore cluster machines,” in *Proceedings of the 2nd Workshop on System-level Virtualization for High Performance Computing*, Mar. 2008, pp. 1–8. [Online]. Available: http://www.csm.ornl.gov/srt/hpcvirt08/

[123] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, “Evaluating the performance impact of Xen on MPI and process execution for HPC systems,” in *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing*, Nov. 2006.

[124] ——, “Paravirtualization for HPC systems,” in *Proceedings of the 4th International Symposium on Parallel and Distributed Processing and Applications Workshops*, ser. Lecture Notes in Computer Science, Dec. 2006, vol. 4331, pp. 474–486.

[125] L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski, “The impact of paravirtualized memory heirarchy on linear algebra computational kernels and software,” in *Proceedings of the 17th ACM International Symposium on High-Performance Distributed Computing*, Jun. 2008, pp. 141–152.

[126] L. Cherkasova and R. Gardner, “Measuring CPU overhead for I/O processing in the Xen virtual machine monitor,” in *Proceedings of the 2005 USENIX Annual Technical Conference*, May/Jun. 2005.

[127] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya, “High-performance hypervisor architectures: Virtualization in HPC systems,” in *Proceedings of the 1st Workshop on System-level Virtualization for High Performance Computing*, Mar. 2007. [Online]. Available: http://www.csm.ornl.gov/srt/hpcvirt07/

[128] A. Warfield, S. Hand, T. Harris, and I. Pratt, “Isolation of shared network resources in XenoServers,” PlanetLab Project, Tech. Rep. PDN-02-2006, Nov. 2002. [Online]. Available: http://www.planet-lab.org/files/pdn/PDN-02-006/pdn-02-006.pdf

[129] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, “Concurrent direct network access for virtual machine monitors,” in *Proceedings*

of the 13th International Conference on High-Performance Computer Architecture. IEEE, Feb. 2007, pp. 306–317.

[130] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, “Xen and co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms,” in *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Jun. 2007.

[131] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, “Enforcing performance isolation across virtual machines in Xen,” in *Proceedings of the 7th ACM/IFIP/USENIX Middleware Conference*, Dec. 2006, pp. 342–362.

[132] J. Liu, W. Huang, B. Abali, and D. K. Panda, “High performance VMM-bypass I/O in virtual machines,” in *Proceedings of the 2006 USENIX Annual Technical Conference*. USENIX, May/Jun. 2006.

[133] K. Nesbit, J. Laudon, and J. E. Smith, “Virtual private caches,” in *Proceedings of the 34th International Symposium on Computer Architecture*, Jun. 2007, pp. 57–68.

[134] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, and M. Valero, “Multicore resource management,” *IEEE Micro*, May 2008.

[135] D. Ongaro, A. L. Cox, and S. Rixner, “Scheduling I/O in virtual machine monitors,” in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Mar. 2008.

[136] D. Gupta, L. Cherkasova, and A. M. Vahdat, “Comparison of the three CPU schedulers in Xen,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[137] D. Schanzenbach and H. Casanova, “Accuracy and responsiveness of CPU sharing using Xen’s cap values,” University of Hawai‘i at Mānoa Department of Information and Computer Sciences, Tech. Rep. ICS2008-05-01, May 2008. [Online]. Available: http://www.ics.hawaii.edu/research/tech-reports/ICS2008-05-01.pdf

[138] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon, “Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies,” in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 33–44.

[139] R. Kettimuthu, V. Subramani, S. Srinivasan, T. Gopalsamy, D. K. Panda, and P. Sadayappan, “Selective preemption strategies for parallel job scheduling,” *International Journal of High Performance Computing and Networking*, vol. 3, no. 2, pp. 122–152, 2005.

[140] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Black-box and gray-box strategies for virtual machine migration,” in *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, Apr. 2007, pp. 229–242.

[141] S. K. Setia, “Trace-driven analysis of migration-based gang scheduling policies for parallel computers,” in *Proceedings of the 1997 International Conference on Parallel Processing*, Aug. 1997, pp. 489–492.

[142] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, “An integrated approach to parallel scheduling using gang-scheduling, backfilling and migration,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 236–247, 2003.

[143] M. Zhao and R. J. Figueiredo, "Experimental study of virtual machine migration in support of reservation of cluster resources," in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, Nov. 2007.

[144] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, May 2005, pp. 273–286.

[145] L. Grit, D. Irwin, A. Yumerefendi, and J. S. Chase, "Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration," in *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing*, Nov. 2006.

[146] L. Grit, D. Irwin, V. Marupadi, P. Shivam, A. Yumerefendi, J. S. Chase, and J. Albrecht, "Harnessing virtual machine resource control for job management," in *Proceedings of the 1st Workshop on System-level Virtualization for High Performance Computing*, Mar. 2007. [Online]. Available: http://www.csm.ornl.gov/srt/hpcvirt07/

[147] M. McNett, D. Gupta, A. M. Vahdat, and G. M. Voelker, "Usher: An extensible framework for managing clusters of virtual machines," in *Proceedings of the 21st Large Installation System Administration Conference*, Nov. 2007, pp. 167–181. [Online]. Available: http://www.usenix.org/event/lisa07/tech/full_papers/

[148] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *Proceedings of the Conference on Cloud Computing and Its Applications*, Oct. 2008. [Online]. Available: http://www.cca08.org/papers.php

[149] "VirtualCenter." [Online]. Available: http://www.vmware.com/products/vi/vc

[150] "Citric XenServer enterprise edition." [Online]. Available: http://www.xensource.com/products/Pages/XenEnterprise.aspx

[151] C. B. Lee, Y. Schwartzman, J. Hardy, and A. E. Snavely, "Are user runtime estimates inherently inaccurate?" in *Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2004, vol. 3277, pp. 253–263.

[152] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak, "A capacity management service for resource pools," in *Proceedings of the 5th International Workshop on Software and Performance*, 2005, pp. 229–237.

[153] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 1, pp. 1–39, 2008.

[154] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and modeling resource usage of virtualized applications," in *Proceedings of the 9th ACM/IFIP/USENIX Middleware Conference*, Dec. 2008, pp. 366–387.

[155] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy, "An observation-based approach towards self-managing web servers," in *Proceedings of the 10th International Workshop on Quality of Service*, May 2002.

[156] D. Gupta, R. Gardner, and L. Cherkasova, "XenMon: QoS monitoring and performance profiling tool," Hewlett-Packard Labs, Tech. Rep. HPL-2005-187, 2005.

[157] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proceedings of the 2006 USENIX Annual Technical Conference*. USENIX, May/Jun. 2006, pp. 1–14.

[158] ——, "Geiger: Monitoring the buffer cache in a virtual machine environment," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, Oct. 2006, pp. 14–24.

[159] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, pp. 1–17, 1978.

[160] M. Bichler, T. Setzer, and B. Speitkamp, "Capacity planning for virtualized servers," in *Proceedings of the 16th Annual Workshop on Information Technologies & Systems*, 2006. [Online]. Available: http://papers.ssrn.com/sol3/JELJOUR_Results.cfm?form_name=journalbrowse&journal_id=1015620

[161] N. Bansal, A. Caprara, and M. Sviridenko, "Improved approximation algorithms for multidimensional bin packing problems," in *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Oct. 2006, pp. 697–708.

[162] J. Csirik, J. B. G. Frenk, M. Labbe, and S. Zhang, "On multidimensional vector bin packing," *Acta Cybernetica*, vol. 9, no. 4, pp. 361–369, 1990.

[163] B. T. Han, G. Diehr, and J. S. Cook, "Multiple-type, two-dimensional bin packing problems: Applications and algorithms," *Annals of Operations Research*, vol. 50, no. 1, pp. 239–261, 1994.

[164] L. T. Kou and G. Markowsky, "Multidimensional bin packing algorithms," *IBM Journal of Research and Development*, vol. 21, no. 5, pp. 443–448, 1977.

[165] W. J. Leinberger, V. Kumar, and G. Karypis, “Job scheduling in the presence of multiple resource requirements,” University of Minnesota Department of Computer Science and Engineering, Tech. Rep. 99-025, May 1999.

[166] W. J. Leinberger, G. Karypis, and V. Kumar, “Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints,” in *Proceedings of the 1999 International Conference on Parallel Processing*, Sep. 1999, pp. 404–412.

[167] W. J. Leinberger, G. Karypis, V. Kumar, and R. Biswas, “Load balancing across near-homogeneous multi-resource servers,” in *Proceedings of the 9th Heterogeneous Computing Workshop*, May 2000, pp. 60–71.

[168] W. J. Leinberger, “Scheduling heuristics for improved utilization in multi-resource parallel systems,” Ph.D. Dissertation, University of Minnesota, Oct. 2001.

[169] K. Maruyama, S. K. Chang, and D. T. Tang, “A general packing algorithm for multidimensional resource requirements,” *International Journal of Parallel Programming*, vol. 6, no. 2, pp. 131–149, 1977.

[170] E. G. Coffman, Jr. and G. S. Lueker, “Approximation algorithms for extensible bin packing,” *Journal of Scheduling*, vol. 9, no. 1, pp. 63–69, 2006.

[171] P. Dell’Olmo, H. Kellerer, M. G. Speranza, and Z. Tuza, “A 13/12 approximation algorithm for bin packing with extendable bins,” *Information Processing Letters*, vol. 65, no. 5, pp. 229–233, 1998.

[172] L. Epstein, “Bin stretching revisited,” *Acta Informatica*, vol. 39, no. 2, pp. 97–117, 2003.

[173] ——, “On variable-sized vector packing,” *Acta Cybernetica*, vol. 16, pp. 47–56, 2003.

[174] M. N. Garofalakis and Y. E. Ioannidis, “Parallel query scheduling and optimization with time- and space-shared resources,” in *Proceedings of the 23rd International Conference on Very Large Data Bases*, Aug. 1997, pp. 296–305.

[175] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. New York, USA: W.H. Freeman and Company, 1979.

[176] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt, “Toward effective multi-capacity resource allocation in distributed real-time and embedded systems,” in *Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2008, pp. 124–128.

[177] W. Fernandez de la Vega and G. S. Lueker, “Bin packing can be solved within $1+\epsilon$ in linear time,” *Combinatorica*, vol. 1, no. 4, pp. 349–355, 1981.

[178] C. Chekuri and S. Khanna, “On multi-dimensional packing problems,” *SIAM Journal on Computing*, vol. 33, no. 4, pp. 837–851, 2004.

[179] A. Caprara and P. Toth, “Lower bounds and algorithms for the 2-dimensional vector packing problem,” *Discrete Applied Mathematics*, vol. 111, no. 3, pp. 231–262, 2001.

[180] F. C. R. Spieksma, “A branch-and-bound algorithm for the two-dimensional vector packing problem,” *Computers and Operations Research*, vol. 21, no. 1, pp. 19–25, 1994.

[181] H. Kellerer and V. Kotov, “An approximation algorithm with absolute worst-case performance ratio 2 for two-dimensional vector packing,” *Operations Research Letters*, vol. 31, no. 1, pp. 35–41, 2003.

[182] H. Shachnai and T. Tamir, “Approximation schemes for generalized 2-dimensional vector packing with application to data placement,” in *Proceedings of the 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and the 7th International Workshop on Randomization and Approximation Techniques in Computer Science*, ser. Lecture Notes in Computer Science, S. Arora, K. Jansen, J. D. P. Rolim, and A. Sahai, Eds. Springer, Aug. 2003, vol. 2764, pp. 165–177.

[183] J. Gueyoung, K. Joshi, and M. Hiltunen, “Performance aware regeneration in virtualized multitier applications,” in *Proceedings of Proactive Failure Avoidance Recovery and Maintenance (PFARM)*, Jun. 2009.

[184] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, “Resource allocation using virtual clusters,” University of Hawai‘i at Mānoa Department of Information and Computer Sciences, Tech. Rep. ICS2008-09-01, Sep. 2008. [Online]. Available: http://www.ics.hawaii.edu/research/tech-reports/ics2008-09-15.pdf

[185] ——, “Resource allocation using virtual clusters,” in *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, May 2009, pp. 260–267, for a more detailed version, see [184].

[186] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, “Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms,” *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 365–381, 2006.

[187] A. S. Shulz and M. Skutella, “Scheduling unrelated machines by randomized rounding,” *SIAM Journal on Discrete Mathematics*, vol. 15, no. 4, pp. 450–469, 2002.

[188] "GAlib: A C++ library of genetic algorithm components." [Online]. Available: http://lancet.mit.edu/ga/

[189] "CPLEX." [Online]. Available: http://www.ilog.com/products/cplex/

[190] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 9, pp. 962–974, 2010.

[191] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, "Approximation algorithms for average stretch scheduling," *Journal of Scheduling*, vol. 7, no. 3, pp. 195–222, 2004.

[192] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke, "Online scheduling to minimize average stretch," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Oct. 1999, pp. 433–443.

[193] S. Albers, "Better bounds for online scheduling," *SIAM Journal on Computing*, vol. 29, no. 2, pp. 459–473, 1999.

[194] Z. Ivković and E. L. Lloyd, "Fully dynamic algorithms for bin packing: Being (mostly) myopic helps," *SIAM Journal on Computing*, vol. 28, no. 2, pp. 574–611, 1999.

[195] Y. Azar, A. Z. Broder, and A. R. Karlin, "On-line load balancing," in *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, Oct. 1992, pp. 218–225.

[196] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. R. Pruhs, and O. Waarts, "On-line load balancing of temporary tasks," *Journal of Algorithms*, vol. 22, no. 1, pp. 93–110, 1997.

[197] J. Edmonds, "Scheduling in the dark," in *Proceedings of the 31st ACM Symposium on Theory of Computing*, May 1999, pp. 179–188.

[198] N. Bansal, K. Dhamdhere, J. Könemann, and A. Sinha, "Non-clairvoyant scheduling for minimizing mean slowdown," *Algorithmica*, vol. 40, no. 4, pp. 305–318, 2004.

[199] D. P. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1992.

[200] S. Cochrane, K. Kutzer, and L. McIntosh, "Solving the HPC I/O bottleneck: Sun™ Lustre™ storage system," Sun BluePrints™ Online, Sun Microsystems, Apr. 2009.

[201] A. Sandgren, D. G. Feitelson, and M. Jack, "The HPC2N log," 2006. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html

[202] D. G. Feitelson and D. Tsafrir, "Workload sanitation for performance evaluation," in *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2006, pp. 221–230.

[203] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, 2003.

[204] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Mar. 1998, vol. 1459, pp. 1–24.

[205] V. Lo, J. Mache, and K. Windisch, “A comparative study of real workload traces and synthetic workload models for parallel job scheduling,” in *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer, Mar. 1998, vol. 1459, pp. 25–46.

[206] E. Frachtenberg and D. G. Feitelson, “Pitfalls in parallel job scheduling evaluation,” in *Proceedings of the 11th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2005, vol. 3834, pp. 257–282.

[207] D. Tsafrir and D. G. Feitelson, “Instability in parallel job scheduling simulation: The role of workload flurries,” in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, Apr. 2006.

[208] M. Stillwell, F. Vivien, and H. Casanova, “Dynamic fractional resource scheduling for HPC workloads,” in *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, Apr. 2010.

[209] ——, “Fine-grain dynamic resource allocation vs. batch scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, submitted for publication.

[210] “Usher clients.” [Online]. Available: http://usher.ucsd.edu/trac/wiki/UsherClients

[211] “Usher plugins.” [Online]. Available: http://usher.ucsd.edu/trac/wiki/UsherPlugins

[212] “Usher events.” [Online]. Available: http://usher.ucsd.edu/trac/wiki/UsherDevelopment#UsherEvents

[213] “cpulimit.” [Online]. Available: http://cpulimit.sourceforge.net/

[214] N. Bansal, T. Kimbrel, and K. R. Pruhs, "Dynamic speed scaling to manage energy and temperature," in *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Oct. 2004, pp. 520–529.

[215] D. J. Bradley, R. E. Harper, and S. W. Hunter, "Workload-based power management for parallel computer systems," *IBM Journal of Research and Development*, vol. 47, no. 5–6, pp. 703–718, 2003.

[216] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Oct. 2001, pp. 103–116.

[217] G. Sabin and P. Sadayappan, "Unfairness metrics for space-sharing parallel job schedulers," in *Proceedings of the 11th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson, E. Frachtenberg, L. Rudolph, and U. Schwiegelshohn, Eds. Springer, Jun. 2005, vol. 3834, pp. 238–256.

[218] L. Barsanti and A. C. Sodan, "Adaptive job scheduling strategies via predictive job resource allocation," in *Proceedings of the 12th International Workshop on Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Springer, Jun. 2006, vol. 4376, pp. 115–140.

[219] "RUBBoS: Bulletin board benchmark." [Online]. Available: http://jmob.ow2.org/rubbos.html

[220] "RUBiS: Rice University bidding system." [Online]. Available: http://rubis.ow2.org/

[221] “TPC-W: a transactional web e-Commerce benchmark.” [Online]. Available: http://www.tpc.org/tpcw/

[222] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, “The NAS parallel benchmarks 2.0,” NASA Advanced Supercomputing Division, Tech. Rep. NAS-95-020, 1995. [Online]. Available: http://www.nas.nasa.gov/News/Techreports/1995/PDF/nas-95-020.pdf