MANAGING TAIL LATENCY IN INTERACTIVE SERVICES FOR MULTICORE SERVERS

BY MD EHTESAMUL HAQUE

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Thu D. Nguyen and Ricardo Bianchini

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2016

ABSTRACT OF THE DISSERTATION

Managing Tail Latency in Interactive Services for Multicore Servers

by MD EHTESAMUL HAQUE

Dissertation Directors: Thu D. Nguyen and Ricardo Bianchini

Interactive services such as Web search, recommendations, games, and finance must respond quickly to satisfy customers. Achieving this goal requires optimizing *tail* (e.g., 99th+ percentile) latency. Unfortunately, interactive services often show variability in service demand, with the tail latency being defined by a small number of long requests. Providers can keep the tail latency under a target by giving extra resources to long requests. This is challenging because (1) service demand is unknown when requests arrive; (2) blindly giving extra resources to all requests quickly oversubscribes resources; and (3) giving extra resources to the numerous short requests does not improve tail latency.

In this dissertation, we propose judicious allocation of processor cores in multicore servers to interactive service requests for optimizing tail latency. In the first part of the dissertation, we introduce Few-to-Many (FM) *incremental parallelization* for current multicore servers. FM dynamically increases parallelism to utilize idle cores and reduce tail latency. FM uses request service demand profiles and total number of cores in an offline phase to compute a policy, represented as an interval table, which specifies when and how much software parallelism to add. At runtime, FM adds parallelism to use extra cores as specified by the interval table indexed by dynamic system load and

request execution time progress. The longer a request executes, the more parallelism (and higher number of cores) FM allocates to it. We evaluate FM in Lucene, an open-source enterprise search engine, and in Bing, a commercial Web search engine. FM improves the 99th percentile response time up to 32% in Lucene and up to 26% in Bing, compared to prior state-of-the-art parallelization. Compared to running requests sequentially in Bing, FM improves tail latency by a factor of two.

In the second part, we demonstrate how to tradeoff tail latency and energy consumption on emerging heterogeneous processors with fast cores and energy-efficient slow cores on the same chip. We introduce *Adaptive Slow-to-Fast (AS2F)* scheduling algorithm, which migrates requests from slow to fast cores to give powerful cores to long requests. We use control theory to design threshold-based migration policies that manage latency and energy efficiency trade-offs for heterogeneous processors with any number of core types. We demonstrate several configurations that (1) improve energy-efficiency while meeting a target tail latency, (2) optimize average and tail latency, or (3) optimize energy only. AS2F can judiciously exploit slower cores to reduce energy by up to a *factor of 2.1* while meeting the tail latency target, compared to only optimizing latency. This configuration consumes only 20% more energy than an oracular policy that has perfect knowledge of request lengths and arrival rate.

Overall, our experience and results show that FM and AS2F can be powerful techniques for managing tail latency and energy, smartly utilizing hardware resources, and exploiting emerging heterogeneous processors.

Acknowledgements

First and foremost, I would like to thank my advisers Professor Thu D. Nguyen and Dr. Ricardo Bianchini. Their enormous support and guidance was a constant source of motivation for me. Without their vision and knowledge, this dissertation would not have been a reality. I am grateful to them for believing in me, constantly challenging me to bring out the best and encouraging me through difficult steps. I would also like to thank my committee members Professor Abhishek Bhattacharjee and Dr. David Meisner for their feedback on the work and comments on this dissertation.

I was fortunate to get a chance to work with an excellent set of collaborators from Microsoft Research. I would like to thank Dr. Yuxiong He, Dr. Sameh Elnikety and Professor Kathryn S. McKinley for their participation, support and guidance on the work. I would also like to thank Professor Santosh Nagarakatte for many useful discussion on processor architecture and Professor Xiaorui Wang for help on control theory.

I had the privilege of working with an excellent set of colleagues and mentors from Panic, Dark and RUArch lab. I would like to thank Kien Le, Íñigo Goiri, Luiz Ramos, Tuan Phan, Qingyuan Deng, Cheng Li, William Katsak, Daniela Vianna, Guilherme Cox, Ioannis Manousakis, Binh Pham, Zi Yan and Jan Vesely.

I would like to thank Toufiq Parag, Nishat Islam, Pavel Mahmud, Rajat Roy, Rezwana Karim and Nak Islam for their help to settle in New Jersey. Finally, I thank my loving wife Umme Atia Nazia for being patient while I was working for deadlines.

Dedication

To my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this dissertation, we address the problem of managing tail latency in interactive services. Interactive services often consists of many short and a few long requests. So, these services can have high tail latencies even though average response time is low. We focus on intelligent allocation of resources (e.g., processor cores) to requests so that long requests are allocated extra (or more powerful) resources to finish faster. In this chapter, we first motivate the problem, then give an overview of our solutions and explain our contributions.

1.1 Motivation

Interactive online services are ones where users are actively waiting for service responses. Notable examples include Web search, financial trading, games, and online social networks. These services require consistently low response times to attract and retain users [68, 133]. Interactive service providers therefore define strict targets for *tail latencies* — 99th percentile or higher response times [33, 34, 72, 153] to deliver consistently fast responses to user requests. The lower the tail latency, the more competitive the service. For example, half a second delay in responses can cause around $315M revenue loss for Microsoft Bing [49].

Reducing tail latency is challenging, in part because requests exhibit highly variable demand. For example in finance servers and Web search, most user search requests are short, but a significant percentage are long [33, 72, 129]. Figure 1.1 shows the histogram of measured response time for 30000 web search request from Microsoft Bing. Although the median is only 15 ms, the 99th percentile latency is 185 ms. Other work on search

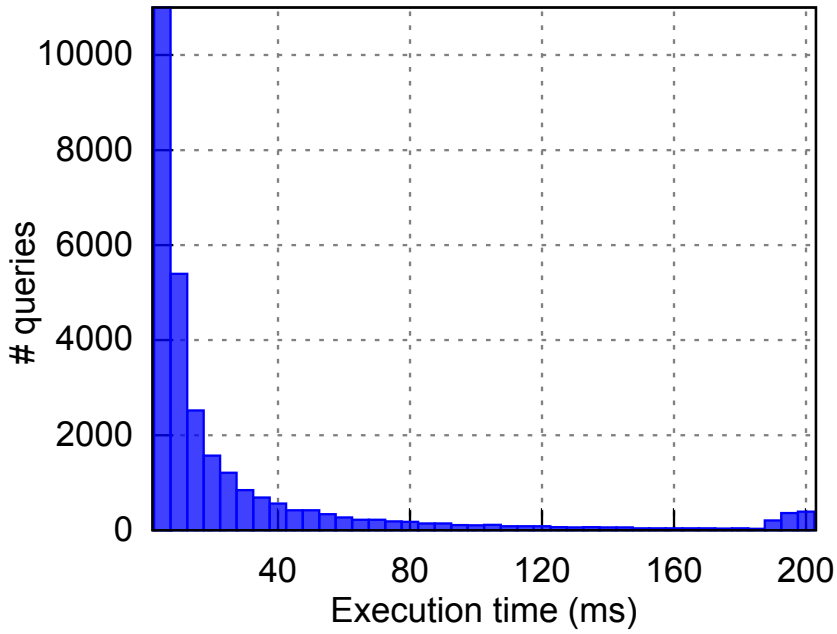


Figure 1.1: Bing demand distribution histogram for 30K requests.

engines also shows that in a distributed system, the longest requests (99th-percentile execution times) are up to a factor of 10× larger than the average execution times, and even 100× larger than the median [84]. While other sources of variability, such as interference from other workloads, hardware variability, and network congestion, contribute to tail latency, prior works establish that long requests in computationally intensive workloads are a primary factor in tail latency. Thus, a critical component of reducing tail latency is reducing the execution time of long requests.

Processing user queries in interactive services often require the processing of very large data sets. For example, Web search queries require accessing indexes of billions of Web pages. To keep the response time fast, the processing is distributed among multiple servers. Thus, the processing of a single request can span 100–1000s of servers. Reducing each server's tail latency is critical when a request spans many servers and responses are aggregated from these servers. In this case, the slower servers typically dominate the response time [33, 89]. Figure 1.2 shows the probability of the service getting delayed for various probability of a single server getting delayed. It says, for example, if a service request requires responses from 100 worker servers and each server has a 1% probability of getting delayed then the service will get delayed with 63%

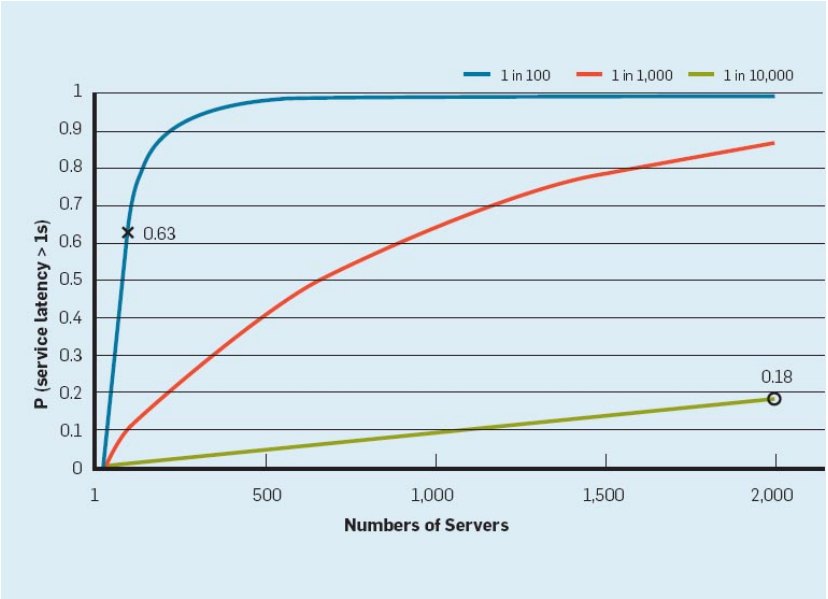


Figure 1.2: Probability of one second service latency vs. individual server latency as number of servers increases. If 1 out of 100 servers shows one second latency, the service will have 63% probability of showing 1 second or more while aggregating response from 100 servers (marked with x). Even if 1 out of 10000 servers takes more than 1 second, roughly one-fifth users will experience 1 second service latency when the service aggregates response from 2000 servers (marked with o). [33]

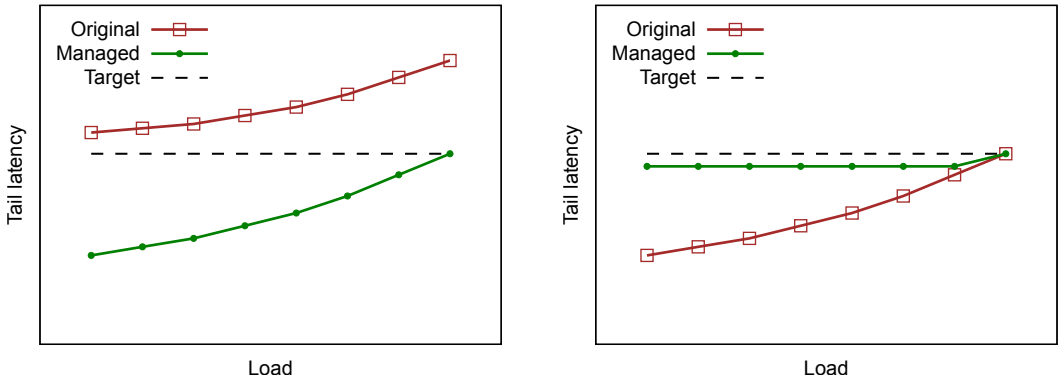


Figure 1.3: Possible objectives for tail latency management.

probability. Thus, it is very important to improve the tail latency performance of individual servers. In this dissertation, we focus on managing tail latency at the server level.

While low latency is important, it may not be profitable to achieve the minimum tail latency all the time. Lowering the latency beyond a certain value (which varies depending on service) often has minimal impact on user satisfaction while incurring a disproportionately large cost. So, providers often define a target tail latency for interactive services. This presents interesting trade-offs for service providers in that some providers want minimum tail latency with low risk of missing the target while others want minimum cost by achieving just the target. Figure 1.3 shows some example objectives for different providers. Researchers have looked at scenarios like Figure 1.3(a) where the service exhibits high tail latency and providers wants to minimize tail latency [69]. Researchers have also looked at scenarios like Figure 1.3(b) where the providers want to achieve a specific tail latency, but would prefer to save resources/cost (e.g., energy consumption) rather than reducing the tail latency further below the target. For example, authors in [109] exploited the slack in tail latencies and reduced energy consumption. Since electricity represents a large operating expense for datacenter operators [16, 93], reducing energy within a target tail latency can increase profitability.

1.2 Dissertation Goal and Structure

Managing tail latency is an interesting and challenging problem. There have been numerous works at different levels of the software stack (see Chapter 3). However, the majority of the work considered interference and transient factors as the sources of high tail latency. This dissertation shows how to reduce tail latency when request length differ and some requests are long because they require more processing. We show how to manage tail latency in the two scenarios discussed above. In the first scenario, we reduce tail latency by exploiting dynamic parallelism on current (homogeneous) multicore servers (Chapter 4). In the second scenario, we exploit the slack in tail latency to reduce energy consumption on emerging heterogeneous multicore servers (Chapter 5).

In chapter 4, we show how to reduce tail latency with the judicious use of parallelism at the level of an individual request. The objective of this work was similar to the one shown in Figure 1.3(a). Internet services already employ datacenter-scale parallelism where many servers perform the computationally intensive work [82, 9, 115, 129, 65, 152, 140] and then return their results to a few servers that aggregate responses [14, 24, 86]. This technique can be extended for requests running on each individual server, where request processing times can be reduced by dividing the work among multiple threads working simultaneously. However, simply parallelizing all requests is not desirable. This decision would waste resources on short-running requests, which have a smaller impact on the tail latency and often do not scale as well as long-running requests [82]. Resources would also be wasted by using more parallelism than the requests (even long-running ones) can effectively exploit. As parallelism introduces overhead and speedup is usually sub-linear, it is less desirable under moderate or high system load, when contention for resources is higher. Otherwise, it may introduce too many threads in the system, degrading average and tail latency. We therefore seek to parallelize only the long requests, which contribute the most to tail latency. However, when a request arrives, we do not know its service demand and it is hard to predict the execution time of each starting request online [111]. Thus, we need a solution that uses the information as they are revealed and decide the parallelism degree for the requests. We introduce *incremental* parallelism to reduce tail latency, which dynamically adds parallelism to an individual request based on the requests' progress and the system load.

Few-to-Many (FM) increases parallelism by increasing the number of worker threads for a request, from 1 to a given maximum, over the duration of its execution. Short requests execute sequentially, saving resources, and long requests execute in parallel, reducing tail latency. This approach naturally limits the resources assigned to short-running and long-running requests. The key challenge for dynamic parallelization is determining *when* to increase the parallelism and by *how much*, as a function of hardware resources, parallelism efficiency, dynamic load, and individual request progress.

FM has an offline and an online phase. The offline phase takes as input a demand profile of requests with (1) their individual sequential and parallel execution times, and

(2) hardware parallelism (core resources). We perform a scalability analysis to determine a maximum degree of software parallelism to introduce. Although the individual requests submitted to a service change frequently, the demand profile of these requests changes slowly [111, 129], making periodic offline or online processing practical. The offline phase computes a set of schedules that specify a time and degree of parallelism to introduce based on dynamic load and request progress. The algorithm that produces these schedules seeks to fully utilize available hardware parallelism at all loads. At low load, FM aggressively parallelizes requests. At moderate to high load, FM runs short requests sequentially and incrementally adds parallelism to long requests to reduce their tail latency. Online, each request self-schedules, adding parallelism to itself based on its current progress and the instantaneous system load. Decentralized self-scheduling limits synchronization costs and therefore improves scalability. We evaluate FM in Lucene, an open-source enterprise search engine and Bing, a commercial Web search engine. Our results show that FM can significantly reduce tail latency.

Chapter 5 focuses on scheduling requests on a heterogeneous processor to meet a target tail latency while reducing energy consumption. Processor designers are trying to add more processing capacity without significantly increasing energy consumption. This trend has fueled the design of heterogeneous processors. These processor chips include multiple fast and slow cores with the same Instruction Set Architecture (ISA) [97, 98, 107, 61, 127]. Asymmetric multicore processors (AMPs) have the potential to improve both performance and energy when schedulers can match workloads to core capabilities [25, 98, 107, 132, 130].

The objective of this work is similar to Figure 1.3(b) using heterogeneous processors. This is challenging because requests compete for cores, migration introduces overhead, and determining which request should use which core when must be computed quickly. Although the distribution over all requests is known, the service demand of an individual request is unknown. Executing a request that turns out to have small computational demand on a fast core wastes processor energy, but executing a request with large computational demand on a slow core results high tail latency. Even if we know the service demand of a request, the core with the appropriate speed may not be available.

We present *Adaptive Slow-to-Fast (AS2F)*, a new general and configurable algorithm that can effectively reduce tail and average latency, and can reduce energy while meeting a target tail latency constraint on AMPs with N different core types. Adaptive S2F schedules independent, concurrent interactive service requests. It exploits in new ways two observations from prior work: (i) long requests reveal themselves online as they stay in the system longer [33, 69, 77, 129, 130] and (ii) we only consider slow to fast migration policies, since for any optimal schedule, there is an equivalent schedule that migrates from slow to fast cores [129, 130].

Adaptive S2F specifies thresholds at which requests should migrate to a faster core. Once a new request has run on a slow core for threshold amount of time, it is migrated to a free fast core. The approach ensures that slow requests will finish on slow cores and only long requests live long enough to migrate to fast cores. Intuitively, this threshold is higher for low loads and lower for high loads. Because, we need to migrate earlier (i.e. use fast cores more) to satisfy same latency target for higher loads. The key challenge for this method is to decide *how* to adapt this threshold based on system load. Offline, we use target tail latencies and workload distributions to design feedback-based controllers. We use control theory to determine migration thresholds and initial placement policies to balance the conflicting goals of reducing tail latency and reducing energy for an AMP with two or more core types. Online, each request periodically self schedules, migrating itself based on its execution time progress as it relates to the thresholds and system load. Our online scheduler is thus very efficient and highly scalable.

We show configurations of Adaptive S2F that mirror common service provider needs. (1) RTL reduces tail and average latency, and (2) EETL reduces energy for a given target tail latency. For our evaluation, we emulate various AMP configurations, since cycle-level multicore simulators are too slow or use sampling, which make them inappropriate for measuring tail latency for our workloads.

We evaluate AS2F in Lucene and a finance server that predicts prices of European options. Our results show that A2SF can efficiently exploit the trade-offs between energy consumption and tail latency targets.

Overall, FM and AS2F are powerful strategies for managing tail latency in interactive services. Although we evaluate them for few interactive service workloads, the results apply to other interactive services, such as online ads, financial recommendations, and games, that are computationally intensive, have stable workload distributions, and are easy to parallelize incrementally [65, 86].

Appendix A and Appendix B address efficient usage of aleternative energy sources (e.g, renewable energy sources) in large scale computer systems. In particular, we looked at "green" datacenters, i.e. datacenters partially powered by renewable energy such as solar or wind energy sources. In Appendix A, we first propose an approach for High Performance Computing cloud providers to offer such a Green SLA service. Specifically, each client job specifies a Green SLA, which is the minimum percentage of green energy that must be used to run the job. The provider earns a premium for meeting the Green SLA, but is penalized if it accepts the job but violates the Green SLA. We then propose (1) a power distribution and control infrastructure that uses a small amount of hardware to support Green SLAs, (2) an optimization-based framework for scheduling jobs and power sources that maximizes provider profits while respecting Green SLAs, and (3) two scheduling policies based on the framework. We evaluate our framework and policies extensively through simulations. Our main results show the tradeoffs between our policies, and their advantages over simpler greedy heuristics. We conclude that a Green SLA service that uses our policies would enable the provider to attract environmentally conscious clients, especially those who require strict guarantees on their use of green energy.

In appendix B, we propose GreenPar, a scheduler for parallel high-per-formance applications in green datacenters. GreenPar schedules the workload to maximize the green energy consumption and minimize the grid ("brown") energy consumption, while respecting a performance service-level agreement (SLA). When green energy is available, GreenPar increases the resource allocations of active jobs to reduce runtimes. When using brown energy, GreenPar reduces resource allocations within the constraints imposed by the performance SLA to conserve energy. GreenPar makes its decisions based on the speedup profile of each job. We have implemented GreenPar in a real solar-powered

datacenter. Our results show that GreenPar can increase the green energy consumption and reduce both the average job runtime and the brown energy consumption, compared to schedulers that are oblivious to on-site green energy.

1.3 Contributions

This dissertation makes the following contributions.

- We observe that emerging processors offer many opportunities to reduce tail latency and/or energy savings. Properly exploited, such opportunities can provide significant benefits to providers.
- We introduce Few-to-Many incremental parallelization for interactive services.
- We develop an FM scheduler that determines when and how much parallelism to introduce based on maximum software parallelism, hardware parallelism, dynamic load, and individual request progress to optimize tail latency.
- We evaluate FM in open-source and commercial search engines, using production workloads and service demand profiles.
- We introduce AS2F, an online algorithm to jointly optimize tail latency and energy consumption in interactive services by exploiting processor heterogeneity.
- We present methods to configure AS2F to support a wide range of objectives such as satisfying tail latency with minimum energy, optimize tail latency or optimize energy consumption to server requests. We also show how to use control theory to determine thresholds for these objectives.
- We study a wide set of heterogeneous processor configurations for different workloads and present interesting processor design trade-offs.
- We evaluate AS2F in open-source search engine and finance server on our in-house emulator.
- We show substantial improvements in tail latency and energy consumption over prior approaches that improve users' experiences and reduce service providers' infrastructure cost.

- We introduce Green SLA and GreenPar to improve the quantification and increase the usage of renewable energy for large scale computer systems.

Chapter 2

Background

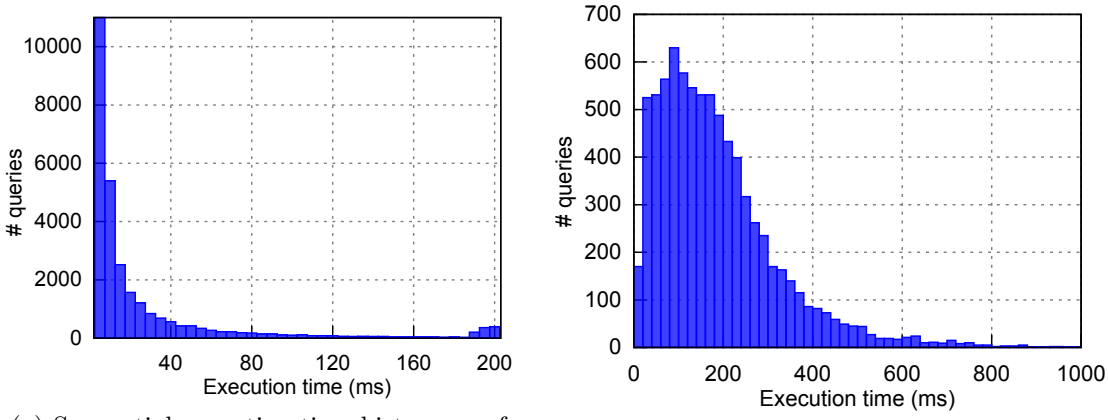
In this chapter, we provide a brief overview of the workload characteristics of interactive services and opportunities for parallel request processing. We also discuss heterogeneous processors, energy-performance trade-offs and basic scheduling mechanisms on heterogeneous processors.

2.1 Interactive Services and Workloads

Interactive services serve requests from many users. The users are actively waiting for the responses. Thus, they have stringent latency requirements that are often represented as tail latency constraints. An example tail latency requirement is that the 99th percentile response time should be less than 200ms. This means that 99% of the requests processed within an accounting time period should complete under 200ms. At the same time, many interactive services, such as search, financial trading, games, and social networking are *computationally intensive* [82, 9, 115, 129, 65]. To meet stringent latency constraints, they are carefully engineered such that (1) their working set fits in memory, since any disk access may compromise responsiveness, and (2) although they may frequently access memory, they are not memory-bandwidth constrained.

As an example, consider Web search, which divides the work among many worker servers that compute over a subset of the data, and then a few servers aggregate the responses [14, 24]. In Bing Web search, each worker server has tens of GBs of DRAM used for caching a partition of the inverted index that maps search keywords to Web documents. This cache is designed to limit disk I/O [82]: the average amount of disk I/O is less than 0.3 KB/s. To attain responsiveness and avoid queuing delay, Bing

provisions additional servers to ensure that workers operate at low to modest loads [82]. The average queuing delay at a worker is 0.35 ms even at high CPU utilization (e.g., 70%). Network I/O is also a small fraction of the overall request latency at 2.13 ms on average. CPU computation is the largest fraction of the response time at well over 70% and is even higher for long queries. Consequently, reducing tail latency requires reducing compute time.



(a) Sequential execution time histogram of 30K Bing requests

(b) Sequential execution time histogram of 10K Lucene requests

Figure 2.1: Bing and Lucene demand distributions.

2.2 Demand Distributions

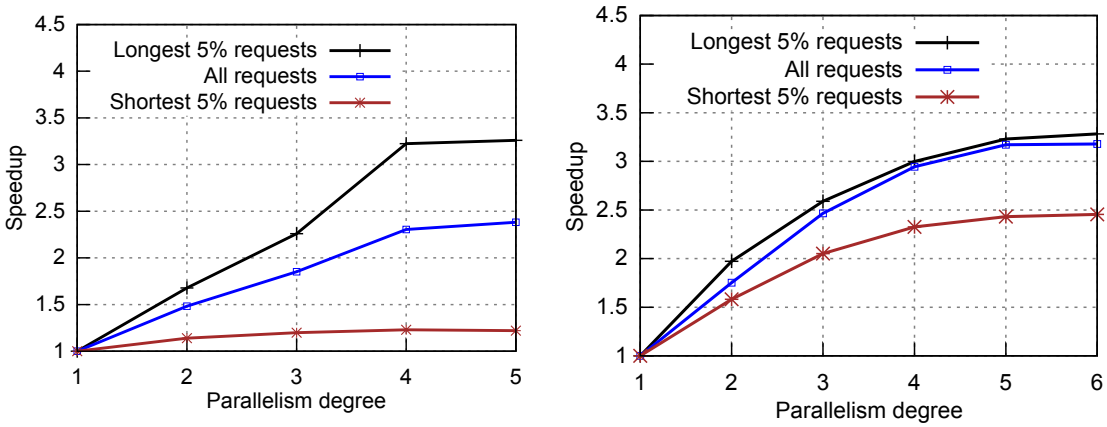
Interactive services workload shows variable demand distributions. Figure 2.1(a) shows the service demand distribution of 30K requests for the Bing Index Server Nodes (ISN). The x-axis is the execution time in 5 ms bins and y-axis is the frequency of requests in each bin. Most requests are short, with more than 85% taking below 15 ms. A few requests are very long, up to 200 ms. The gap between the median and the 99th percentile is a factor of 27×. The slight rise in frequency at 200 ms is because the server terminates any request at 200 ms and returns its partial results. We observe that these workload characteristics are fairly consistent across hundreds of ISN servers with different partitions of the index.

Similarly, Figure 2.1(b) shows the service demand histogram of 10K Wikipedia search requests for Lucene in 20 ms bins. The maximum number of requests are in the

bin around 90 ms and the median service demand is 186 ms. Since the ratio of short to long requests is high, parallelizing the long requests has the potential to reduce tail latency.

2.3 Parallelism for Interactive Services

Interactive services today commonly exploit large-scale parallelism in two ways. (1) They distribute the processing over hundreds or thousands of servers at data center scale because they must process requests over vast amounts of data that do not fit on a single server. (2) At each server, they process multiple requests concurrently. A third complementary way of parallelism is intra-request parallelism on a multicore server. In particular, individual requests run with concurrent threads on multiple cores to reduce their execution time. Since the services are already parallelized at the datacenter scale to compute over vast amounts of data that do not fit on a single server. Further dividing the data and adding parallelism at the individual server level is thus compatible with their system architectures. Prior works also demonstrate that individual requests in interactive systems, such as search, finance trading, and games are easily parallelized [82, 65]. In addition, these workloads are often amenable to dynamic parallelism, in which the scheduler can vary the number of worker threads per request during the request execution. Dynamic parallelism is supported by many parallel libraries and runtimes, such as Cilk Plus [19], TBB [28], TPL [103], and Java [53].



(a) Bing

(b) Lucene

Figure 2.2: Bing and Lucene average speedup.

2.3.1 Parallelism Speedup

Interactive Services are amenable to parallelism at request level. Similar to other workloads, the speedup decreases with increasing number of software threads. present production user requests for both Bing and Lucene and measure the sequential and parallel execution times for each request. Figure 2.2(a) presents Bing parallelism efficiency, i.e., the speedup of requests with different parallelization degrees for all requests, the longest 5%, and the shortest 5%. Long requests have over 2 times speedup with 3 threads. In contrast, short requests have limited speedup, a factor of 1.2 with 3 threads. These results show that at degrees higher than 4, additional parallelism does not lead to speed up. Figure 2.2(b) shows the parallelism efficiency. Again, we observe many short requests and few long requests. It shows that on average, requests exhibit almost linear speedup for parallelism degree 2. Parallelism is slightly less effective for 2 to 4 degrees and is not effective for 5 or more degrees.

2.4 Heterogeneous Processors

A heterogeneous processor contains cores with different architecture and performance characteristics. In this work, we focus on cores that run the same instruction set architecture (ISA) but has different power and performance characteristics. A *fast* core executes a request in less time than a *slow* core, but consumes more energy. Although slow cores can be more power efficient, interactive service providers still uses homogeneous fast cores. This is because there are several limitations in using slow cores that affect their broad adoption for large scale service providers [76]. Often the providers still worry about single threaded performance and management of large numbers of slow cores because large number of slow are required to replace small numbers of fast cores. This often makes it more difficult to deliver satisfactory response time requirements. Emerging heterogeneous multicore designs allow more flexible processor designs. Thus, by combining cores, it is possible to build more energy efficient systems than homogeneous systems that can be used to achieve the same performance goals. Example of such architectures are ARMs big.LITTLE [61] for mobile systems and Intels QuickIA [27]

for server class.

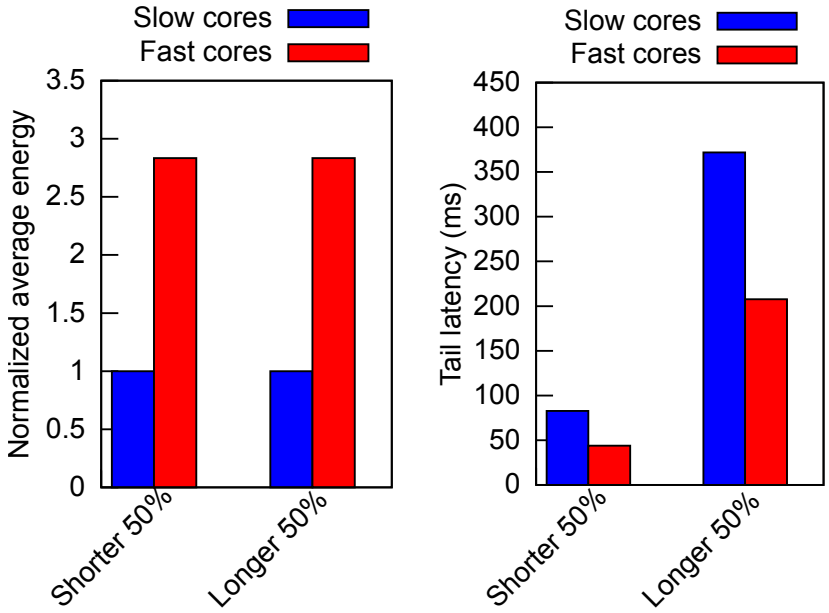


Figure 2.3: (a) Average energy consumption, and (b) 99th percentile (tail) latency of shortest and longest 50% of requests. Each executing exclusively on a slow or fast core.

2.4.1 Energy versus Tail Latency on Heterogeneous Processors

Figure 2.3(a) compares energy consumption of requests that execute exclusively on slow and fast cores using Lucene. Slow cores substantially improve energy efficiency by almost a factor of 3. Although the most energy-efficient approach uses slow cores as much as possible, the first priority for service providers is tail latency. Figure 2.3 (b) shows that slow cores deliver significantly higher tail latencies for the longest 50% of requests. To reason about these behaviors, consider a 200 ms target tail latency. Slow cores deliver a tail latency on the longest 50% of requests that is almost twice the target. On the other hand, short requests executing on slow cores deliver tail latencies of roughly 83 ms, well below the target. By using energy-efficient slow cores for short queries — and most queries are short (see Figure 2.1) — service providers have a significant opportunity to improve energy efficiency while still meeting the desired tail latency.

2.4.2 Slow-to-Fast Scheduling for Heterogeneous Processors

An ideal scheduler assigns long requests to fast cores and short requests to slow cores. However, predicting request demand when requests arrive is hard and transient factors increase the runtimes of requests. A practical approach thus migrates requests to faster cores as long requests reveal themselves [130]. [130] also shows that any arbitrary migration sequence can be mapped to *Slow-to-fast* migration policy and this would minimize energy for a given latency target. Slow-to-fast migration has the following desirable qualities:

Bound migration. A slow-to-fast scheduler limits the migrations to the number of different core types. For two core types, each request migrates at most once.

Reduce energy consumption. Using slow cores first completes short requests on slow cores and does not use fast cores at all. For a workload distribution where an individual request demand is unknown, slow-to-fast thus reduces total energy consumption.

2.4.3 Shortcomings of a Simple Slow-to-Fast Scheduler

Despite this potential, slow-to-fast has some challenges. A sophisticated policy must choose when to migrate from slow to fast cores, since poor choices will degrade tail latency or increase energy consumption. Here are a key insights that it should consider:

Exploit the workload demand distribution. We use the workload demand distribution, as it changes slowly over time and service providers already use it to provision servers. If most requests take less than 300 ms on a slow core, and the tail latency target is 350 ms, even a conservative policy can use slow cores extensively. As the target decreases making it is harder to achieve for the workload, the policy will use fast cores more aggressively.

Use fast cores just enough. The policy should use fast cores just enough to satisfy the tail latency target, sacrificing some average latency. Since slow cores are more energy efficient, this strategy satisfies the target with less energy.

Adapt to load. To handle load spikes, the policy needs to adapt, using fast cores more often as load increases.

Chapter 3

Related Work

In this chapter, we discuss relevant works on the topics of reduction in response time using parallelism, throughput and latency improvement in datacenters by parallelization, managing tail latency and energy consumption on AMPs and DVFS-enabled processors.

3.1 Parallelizing a Single Job

Many systems adapt parallelism to run-time variability and hardware characteristics [18, 31, 85, 92, 102, 126, 138, 148]. They focus on the execution of a single job to reduce execution time and improve energy efficiency. However, they do not consider a server system running concurrent jobs because of resource contention. Simply applying parallelism to minimize the execution time of every single job will not minimize the response time for concurrent jobs. Our work focuses on a server environment with many requests where parallelizing one request may affect others. We choose the degree of parallelism for a request based on both its impact on the request itself and on other requests.

3.2 Parallelization in Multiprogrammed Environment

Adaptively sharing resources among parallel jobs has been studied empirically [47, 118, 48] and theoretically [12, 73, 139]. McCann et al. study many different job schedulers and evaluated them on a set of benchmarks [118]. They also introduced the notion of dynamic equipartitioning, which gives each job a fair allotment of processors, while allowing processors that cannot be used by a job to be reallocated to other jobs. He et al. use parallelism feedback of jobs to allocate resources to minimize makespan and mean response time of jobs [73]. The prior work on adaptive job scheduling has two main

differences compared with our work. (1) It is common for jobs in a multiprogrammed environment to have different characteristics that are not known to the scheduler at scheduling time. This limits the information that a scheduler can use to make scheduling decision; thus most of the work in this area use non-clairvoyant scheduling that assumes the scheduler does not know anything about the job before executing it. While in the interactive services, there is a lot of similarity among requests because they process user queries using the same procedure. So the scheduler can use more information, such as the average execution profile of jobs, to improve scheduling decision. (2) They often consider jobs running long enough such that the scheduler learns the job characteristics along the execution and adjust the allocation to explore a good solution. However, in our search server environment, the average job length is short, only about 50ms. As a scheduling quantum size is in the range of tens of milliseconds, there is a limited chance for our scheduler to learn and adjust the job parallelism degree along the execution.

Our work considers the characteristics described in Chapter 2 and then exploit. For example, the request service demand distribution and parallelism efficiency are stable over time, and therefore we profile them and use the results to improve scheduling decisions. Moreover, as many requests complete quickly, the scheduler must act quickly. We perform offline processing so that scheduler can make efficient online decisions. Finally, this prior work focuses only on reducing mean response time, whereas our work manages tail latency, which requires different techniques.

3.3 Parallelization in Interactive Server Systems for Reducing Response Time

Adaptive resource allocation for server systems [142, 149] focuses on allocating resources dynamically to different components of the server, while still executing each request sequentially. Raman et al. proposed an API and runtime system for dynamic parallelism [128], in which developers express parallelism options and goals, such as minimizing mean response time. The runtime dynamically chooses the degree of parallelism to meet the goals, and does not change it during the execution of the requests. Jeon et al. [82] proposed a dynamic parallelization algorithm to reduce the average

response time of Web search queries. The algorithm decides the degree of parallelism for each request before the request starts, based on the system load and the average speedup of the requests. Neither approach [82, 128] targets tail latencies. Moreover, since the service demand of each request is typically unknown before the request starts, these approaches cannot differentiate long requests from short ones. Thus, they over-subscribe resources under moderate or high load, and are ineffective at reducing the tail latency.

To specifically reduce tail latency, Jeon et al. [84, 83] predict service demand of Web search requests using machine learning and parallelize the requests predicted to be long, regardless of the system load and workload characteristics. Our work considers all of these factors, as well as instantaneous load to deliver better results. We show that our policies perform better than policies with perfect predictions that don't consider load. We also compare our policy and provide a bound on how much it would benefit to have predictions for both request length and instantaneous load.

3.4 AMP Systems for Energy Efficiency

Researchers have studied AMP systems for improving energy efficiency with same performance [97, 98, 107, 25, 132, 151, 45, 145]. Lukefahr et al. showed that AMP systems can provide better energy efficiency than only voltage-frequency scaling [113]. Reddi et al. studied the effect of micro-architectural properties of processor cores on energy efficiency and performance trade-off for web search [81]. These group of works showed the potential of AMP systems for energy efficiency. Our work focuses on scheduling on these systems. Numerous work have looked at scheduling on AMP systems with batch/multiprogrammed workloads [146, 132, 26, 36]. Our work focuses on interactive services where the scheduler has to take decision in tens of milliseconds granularity.

3.5 Energy Efficiecny in Interactive Services

Exploiting the slack in interactive services between response time SLAs and actual achievable performance to improve energy efficiency is an interesting research direction.

Researchers have also targeted energy proportionality for datacenters servicing these workloads [15]. Researchers have focused on, and often attempt to achieve this goal by leveraging different power states for different components as well for the entire system [40, 119, 50, 67, 39]. Researchers have also exploited this slack to run other workloads [154, 37, 110, 115]. A few works have looked at network protocols to mark the requests so that schedulers can take benefit of the slack [144, 6]. Our work adds to the body of the previous work by considering an extra dimension which is AMP systems to exploit the slack in latency.

3.6 Tail Latency versus Energy using DVFS

A number of work manages tail latency on homogeneous multicores with DVFS [77, 109]. Lo et al. [109] propose a feedback-based DVFS controller for homogeneous servers for better energy proportionality for interactive services [109]. They exploit capacity slack at low load by reducing the processor voltage/frequency of all servers. They increase average latency, while maintaining the same tail latency. Processor-wide DVFS wastes power since the speed and power consumption of all cores must be sufficient to service the tail. By considering request demand, our fine grain policies produce greater energy savings since only some cores must be fast enough to satisfy long requests.

Hsu et al. [77] boost the speed of an individual core with fine-grain DVFS to accelerate individual requests by predicting their latency with various application-specific mechanisms and when wrong, dynamically detecting long requests [77]. Regardless of system load, they select a single core type for each request, based on the request's latency prediction. Our approach is finer grained, allowing requests to run on multiple core types, and accounts for load. In addition, accurately predicting long requests is hard and difficult to maintain as the services evolve [111, 80, 89]. Moreover, since prediction is never perfect, it is not a complete solution. Our approach does not require prediction, and can be used as a fallback mechanism for mispredicted requests.

3.7 Tail Latency Management on AMP Systems

The most closely related work optimizes tail latency on AMPs [124, 130, 129]. Ren et al. [130] prove that slow-to-fast simplifies an AMP scheduler without sacrificing optimality [130, 129]. They optimize tail and average latency by executing requests on the fastest core available and migrating the oldest requests from slow to fast cores. They show that homogeneous cores best optimize energy for *average* latency, whereas that heterogeneous cores offer significant energy advantages over homogeneous ones for multiple objectives. They do not, however, consider trading average latency for energy efficiency. Furthermore, their evaluation is limited and assumes a core of the desired speed is always available. We build on their theory, but introduce adaptive algorithms to handle practical server systems with finite number of cores.

Petrucci et al. [124] allocate either slow or fast AMP cores to an entire service [124]. Their controller selects increasingly larger configurations (type and number of cores) until it meets a target tail latency. When slow cores cannot meet the target, they cannot be used at all since their system never uses multiple core types for a single service. Our approach is more fine grain, executing individual requests on multiple core types, based on its progress and the system load, delivering tighter control over the power/performance trade-offs, and thereby higher energy savings.

Chapter 4

Reducing Tail Latency via Incremental Parallelism in Interactive Services

This chapter discusses Few-to-Many(FM) incremental parallalelism to reduce tail latency in interactive services. Recall that, FM seeks to allocate more resources (in the form of software threads) as requests get older. This ensures that extra resources are given to long requests to reduce tail latencies without overloading resources by also giving extra resources to short requests. FM has two phases, offline and online. The offline phase takes the demand distribution along with sequential and parallel runtimes to compute an interval table. In the online phase, FM uses the interval table to decide when and how many threads should be added as a particular request becomes older (that is, continues to execute without completing). We discuss the intuition and theory behind incremental parallelism and requirements for implementing it effectively in Sections 4.1–4.3. Section 4.4 discusses details of the offline and online phases. Section 4.5 discusses evaluation methodology. Next, we discuss the evaluation of FM for Lucene (Section 4.6) and Bing (Section 4.7). Finally, Section 4.8 summarizes the chapter.

4.1 Intuition Behind FM Incremental Parallelism

A simple approach to using intra-request parallelism is to use a fixed number of worker threads for each request. Depending on the number of threads per request, *fixed* parallelism would either oversubscribe resources at high systems loads or underutilize them under light loads. To see an example of oversubscription and its impact on response times, consider Figure 4.1. The figure shows the mean and 99th percentile response times of Lucene as a function of load when all requests run with 1 worker thread (SEQ)

and 4 worker threads (FIX-4). (See Section 4.6 for our methodology.) Clearly, using 4 threads for all requests reduces tail latency well with low load, but gets progressively worse with higher load. In fact, using 4 threads becomes worse than 1 thread around 42 requests per second (RPS). Henceforth, we focus on the range 30 to 48 RPS, since the latency is typically flat below 30 RPS and too high beyond 48 RPS.

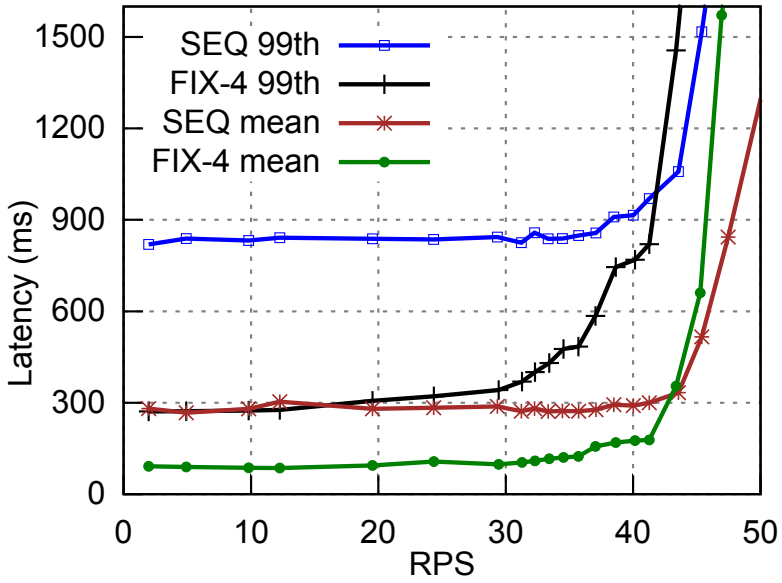


Figure 4.1: Effect of fixed parallelism on latency in Lucene.

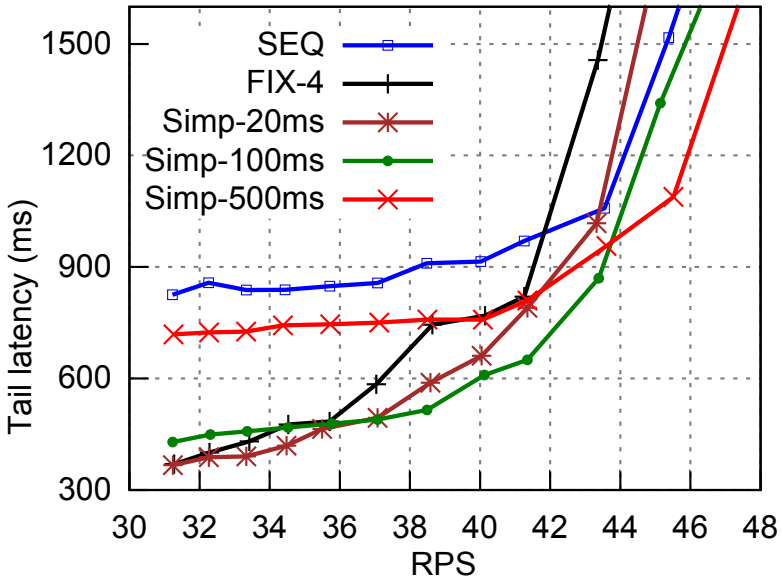


Figure 4.2: 99th percentile tail latency of sequential, degree 4, and simple fixed addition of dynamic parallelism in Lucene.

Another problem with fixed parallelism is that it targets all requests equally. However, long requests have a greater impact on tail latency than short ones. We prefer

to parallelize long requests, but it is difficult to predict if a request will be long or short [111]. Fortunately, requests in many interactive services are amenable to *incremental* parallelization. By dynamically increasing the degree of parallelism as a request executes, long requests will exploit more parallelism than short requests. This insight is the key rationale behind FM incremental parallelization.

4.2 Theoretical Foundation of FM Parallelization

Intuitively, FM parallelization increases the probability that short requests will finish with less parallelism, which saves resources, while it assigns long requests more parallelism, which reduces tail latency. This section presents the theoretical foundation behind this intuition. Theorem 1 shows that, given a tail latency constraint, the optimal policy that minimizes average resource usage assigns parallelism to requests in non-decreasing order. In other words, to minimize resource usage under a latency constraint, if a request ever changes its parallelism, it will only increase, transitioning from *few to many* degrees of parallelism. The theorem makes two assumptions. (1) We do not know if a request is long or short a priori, but we know the service demand distribution, i.e., the distribution of *sequential* request execution times (see Figure 2.1(a) for an example). (2) Each request exhibits sublinear speedup, i.e., parallelism efficiency decreases with increase in parallelism degree, as we showed in the previous section and is true for many workloads.

Theorem 1 *Given a request service demand distribution and a sublinear parallelism speedup function, to meet a tail latency constraint, an optimal policy that minimizes average resource usage assigns parallelism to requests in non-decreasing order.*

Proof.

Suppose a request needs to meet a β-th percentile latency of d. We show any optimal policy that minimizes the average resource usage assigns parallelism in non-decreasing order up to *the latency constraint at time d*. (Requests that take longer than d are in higher percentiles and, thus, may be completed with any degree of parallelism.)

We denote the service demand CDF of request sequential execution times as F and speedup with parallelism i as s_i. As requests have sublinear speedup, the parallelism efficiency decreases with increasing parallelism degree, i.e., $s_i/i > s_j/j$, if $i < j$. Using F, we find the β-th percentile service demand of requests, denoted as w, i.e., $w = F^{-1}(\beta)$. For a schedule to meet β-th percentile latency of d, we want to ensure the β-th percentile longest request can be completed by d, i.e., a work amount w is completed by d.

Let's denote $\mathcal{S}$ as a schedule that specifies how we parallelize a request. For a given piece of work at the x-th cycle where $x \in (0, w]$, and a schedule $\mathcal{S}$, if $\mathcal{S}(x) = i$, this work is parallelized using degree i. The speed to process the work is therefore $s_{\mathcal{S}(x)} = s_i$. We write the resource usage minimization problem as,

$$\min_{\mathcal{S}} \int_0^w [1 - F(x)] \times \frac{\mathcal{S}(x)}{s_{\mathcal{S}(x)}} dx \tag{4.1}$$

$$s.t. \qquad \int_0^w \frac{1}{s_{\mathcal{S}(x)}} dx \leq d \ . \tag{4.2}$$

Here, the integral in the objective function 4.1 computes the expected amount of resources a request consumes up to work w and latency d. Constraint 4.2 guarantees that the processing time of the β-th percentile request is bounded by d.

We now prove the theorem by contradiction. Suppose that there is an optimal schedule $\mathcal{S}'$ that gives higher parallelism to a request earlier and later decreases its parallelism. Thus, there exist x_1 and x_2 such that $0 \leq x_1 < x_1 + dx \leq x_2 < x_2 + dx \leq w$ and $\mathcal{S}'(x_1') > \mathcal{S}'(x_2')$, where $x_1' \in [x_1, x_1 + dx]$, $x_2' \in [x_2, x_2 + dx]$ and dx is a sufficiently small positive number.

Since we assume sublinear speedup, i.e., $s_i/i > s_j/j$ if $i < j$, the following inequality holds:

$$\begin{aligned} & [1 - F(x_1')] \times \left[\frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} - \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} \right] \\ & + [1 - F(x_2')] \times \left[\frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} - \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} \right] \\ = \ & \left[\frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} - \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} \right] \times [F(x_2') - F(x_1')] > 0 \end{aligned}$$

Thus, $[1-F(x_1')] \times \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')} + [1-F(x_2')] \times \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')}] > [1-F(x_1')] \times \frac{\mathcal{S}'(x_2')}{s_{\mathcal{S}'}(x_2')} + [1-F(x_2')] \times \frac{\mathcal{S}'(x_1')}{s_{\mathcal{S}'}(x_1')}]$. Following the optimization objective in Eqn. 4.1, the expected resource usage

is reduced by exchanging the order of parallelism degree between the x'_1-th cycle and the x'_2-th cycle, while keeping the rest of the schedule $\mathcal{S}'$ unchanged. This contradicts the assumption that $\mathcal{S}'$ minimizes Eqn. 4.1 and therefore proves Theorem 1.

Intuitively, Theorem 1 means that given an optimal schedule that first adds and then removes parallelism, there exists an equivalent schedule which only adds parallelism. We exploit this theorem to limit our offline search to finding an optimal few-to-many schedule. The dual problem of Theorem 1 also holds: given a fixed amount of resources, few-to-many minimizes latency. For a server system where each request gets a limited amount of resources, FM minimizes tail latency.

4.3 Practical and Effective FM Parallelization

The simplest approach to incremental parallelism is to simply add parallelism periodically, e.g., add one thread to each request after a fixed time interval. Unfortunately, this approach does a poor job of controlling the total parallelism (resource usage and contention), regardless of the interval length. Figure 4.2 illustrates this problem by comparing the 99th percentile tail latency of Lucene when simply adding parallelism at fixed 20, 100, and 500 ms intervals with executing each request with 1 thread and 4 threads, as a function of load. The figure shows that increasing parallelism dynamically does reduce tail latency more than fixed parallelism at medium and high loads. Short requests use fewer than 4 threads, and thus limit oversubscription of resources. However, no fixed interval is ideal across the entire load spectrum. The shorter the interval, the higher the tail latency at high load. Conversely, the longer the interval, the higher the tail latency at low load. These results suggest that FM parallelization can be effective, but to select intervals correctly FM must carefully consider the system load. Fundamentally, the main requirements for effective incremental parallelization are the following.

FM scheduling must efficiently utilize resources. When the load is low (no resource contention), FM should be aggressive and choose shorter intervals to better

utilize the hardware resources. At high load (high contention), it must be conservative and choose longer intervals to apply parallelism more selectively to just the longest requests. We observe that maintaining a fixed overall number of software threads is a good way to control hardware resource utilization as the load varies.

FM scheduling must consider workload characteristics. FM must consider the distribution of the service demand. For example, if the vast majority of requests take less than 100 ms, an interval of 100 ms will not exploit much parallelism. Moreover, FM must consider any overhead due to parallelism. With lower overhead, we choose smaller intervals, parallelizing requests more aggressively. With higher overhead, we choose larger intervals, parallelizing requests more conservatively to avoid wasting resources.

FM scheduling must consider scalability of the workload. When speedups tail off at high degrees, adding more parallelism is a less effective use of hardware resources. FM thus limits parallelism to an effective maximum.

4.4 Few-to-Many Incremental Parallelization

This section describes Few-to-Many (FM) incremental parallelization for a single server. Our goal is to reduce tail latency. The FM scheduler achieves this goal by exploiting all hardware parallelism and judiciously adding software parallelism. FM has two phases. (1) An offline analysis phase produces an *interval table*. (2) An online dynamic phase schedules requests by indexing the interval table. FM computes the interval table offline using as inputs the maximum software parallelism per request, hardware parallelism, and service demand profiles of sequential and parallel execution times. The interval table specifies when during the execution of a request to add parallelism and how much, as a function of load and request progress. At runtime, the service demand of each request is unknown. FM thus monitors the progress of each request and the total load, and then at the specified intervals, it adds software parallelism to the request. FM is decentralized and each request self-schedules. FM aggressively introduces parallelism under light load, but under high load, it makes efficient use of resources by judiciously

executing short requests sequentially and long requests in parallel. The following key insights lead to an efficient solution.

Favoring long requests

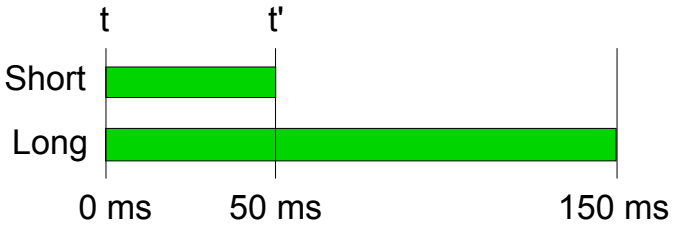
FM gives more parallelism to long requests. At moderate to high loads, FM assigns only one thread to each new request. Short requests thus execute sequentially, only ever consuming one thread. As long requests continue to execute, FM assigns them more parallelism (software threads). FM performs admission control. At moderate to high load, it may delay adding a new request in favor of adding parallelism to existing requests. Since new requests are more likely short, FM optimizes for tail latency.

Judicious use of hardware parallelism

FM explicitly controls total load, neither undersubscribing nor oversubscribing hardware resources. Undersubscribing causes resources to needlessly sit idle when they could be reducing tail latency. Oversubscribing increases contention and thus tail latency, since independent requests and parallel tasks within the same request may interfere, competing for the same resources. Thus, FM slightly oversubscribes the hardware, because threads may occasionally block for synchronization or more rarely I/O. When software parallelism (threads) on occasion exceeds the hardware parallelism (cores), we boost the oldest threads priorities, so they complete without interference from younger requests. FM thus matches software parallelism to hardware parallelism.

Judicious use of software parallelism

Since parallelism introduces overhead and has diminishing returns, the degree to which software parallelism can reduce tail latency is a function of the service, hardware, and workload. Based on workload speedup efficiency, the service provider specifies a maximum amount of software parallelism per request that will deliver a target tail latency.



q_r	t	t'
≤ 2	$0, d_3$	
3	$0, d_1$	$50, d_3$
4–6	$50, d_1$	$100, d_3$
≥ 7	e_1, d_1	$100, d_3$

Figure 4.3: Simple workload and interval table for 50 ms intervals ($t = 0, t' = 50$), number of requests (q_r), parallelism degree (d_j), for 6 cores with speedups $s(2) = 1.5$, $s(3) = 2$.

4.4.1 Offline Analysis

Our offline analysis takes as input a request demand profile, maximum software parallelism, and target hardware parallelism. It outputs an *interval table* indexed by load and each request's current processing time.

4.4.1.1 Example

Consider the simple example in Figure 4.3 that uses the notation defined in Table 4.1. Short and long requests occur with equal probability. The sequential execution time of short requests is 50 ms and long requests is 150 ms. With parallelism degree 3, both short and long requests obtain a speedup of 2. Assume 6 cores for hardware parallelism and 50 ms intervals for simplicity. The resulting interval table consists of 4 rows indexed by the number of instantaneous requests q_r. For 1 or 2 requests, the pair $t = 0, d_3$ specifies that at time $t = 0$ every request starts immediately with parallelism d_3 degree 3, resulting in tail latency of 75 ms for long requests. Average total parallelism is 3 times active requests q_r. If $q_r = 3$, then $t = 0, d_1$, so short requests run sequentially. With $t' = 50, d_3$, long requests run sequentially until 50 ms, when parallelism degree increases to 3. Long requests finish 50 ms later with a speedup of 2 and a tail latency of 100 ms. The average parallelism per request is $(1 \times 50 + 1 \times 50 + 3 \times 50)/(50 + 100) = 1.67$ since the numbers of short and long requests are equal. With 7 or more requests ($q_r \geq 7$), $t = e_1, d_1$ indicates that new requests must wait until another request exits and then start executing sequentially.

4.4.1.2 Interval table

Formally, we compute a function $f : \mathcal{R} \rightarrow \mathcal{I}$, where $\mathcal{R}$ is the set of all potential instantaneous requests in the system and $\mathcal{I}$ is a table indexed by $q_r \in \mathcal{R}$ with each q_r corresponding to one interval selection (or schedule) Φ. A schedule Φ consists of pairs (t_i, d_j), which specify that at load q_r when a request reaches time t_i, execute it with parallelism degree d_j. If $t_0 = 0$, the request immediately starts executing. If $t_0 > 0$, the interval table is specifying admission control and the request must wait to begin its execution until the specified time. If $t_0 = e_1$, the request must wait until another request exits to begin its execution. For example, $f(3) = \{(t_0 = 0, d_1), (t_1 = 50, d_3)\}$ in Figure 4.3 specifies that all requests start executing immediately with one thread and after a request executes for 50 ms $(t_1 = 50, d_3)$, all requests execute with 3 degrees of software parallelism. Load changes dynamically at runtime. A particular request will consult different entries in the interval table during its execution.

4.4.1.3 Interval selection algorithm

We formulate interval selection as an offline search problem, which takes as inputs the request demand profile, maximum software parallelism, target hardware parallelism $target_p$, and parallelism speedup. The profiled sequential and parallel request demand and parallelism speedup are collected offline. The maximum software parallelism per request is selected based on the parallelism efficiency of requests, to limit the parallelism degree to the amount effective at speeding up long requests. We select the target hardware parallelism $target_p$ to moderately oversubscribe the hardware threads through profiling. The search algorithm enumerates potential policies that satisfy $target_p$, i.e., schedules that use all available hardware resources, and then chooses ones that minimize tail latency.

To ease the presentation, we introduce an intermediate representation of a schedule as $\mathcal{S} = \{v_0, v_1, ..., v_{n-1}\}$: a request starts its first thread at time v_0, and adds parallelism from d_i to d_{i+1} after interval v_{i+1}. It is easy to see that any schedule Φ has an alternative but equivalent representation as $\mathcal{S}$. For example, for $\Phi = \{(t_0 = 0, d_1), (t_1 = 50, d_3)\}$,

Symbol	Definition
$r \in R$	Request profiles
seq_r	Sequential runtime of request r
d_n	Max degree n of software parallelism
$s_r(d_j)$	Speedup of r with d_j, $j \leq n$, $\forall r,\ s_r(1) = 1$
q_r	Instantaneous number of requests
$target_p$	Target hardware parallelism
$\Phi = \{(t_0, d_j), (t_1, d_{j+1})..., (t_k, d_n)\}$,	$t_i < t_{i+1}$, $d_j < d_{j+1}$ schedule, at t_i increase parallelism to degree d_i
$\mathcal{S} = \{v_0, v_1, ..., v_{n-1}\}$,	intermediate representation of schedule Φ start request at time v_0, and add parallelism d_i to d_{i+1} after interval v_i
$time_r(\mathcal{S})$	Execution time of r with schedule $\mathcal{S}$
$ap_r(\mathcal{S})$	Average parallelism of r with $\mathcal{S}$
$ap_R(\mathcal{S}, q_r)$	Total average parallelism of q_r of R *requests* with $\mathcal{S}$
$time_R(\mathcal{S}, mean)$	Average latency of requests from R with $\mathcal{S}$
$time_R(\mathcal{S}, \beta - tail)$	β-tail latency of R with $\mathcal{S}$ at 99th-percentile latency with $\beta = 0.99$

Table 4.1: Symbols and definitions for interval selection.

the equivalent $\mathcal{S} = \{0, 50, 0\}$ when the maximum software parallelism $n = 3$.

The interval selection algorithm has two parts. First, Figure 4.4 shows the mathematical formulation for computing parallelism and latency of requests for a given interval selection (schedule) $\mathcal{S}$ and load q_r. Equation (4.3) computes the total time a request takes given the time it spends in its sequential portion (if any) and time it takes in each parallel interval (if any). Equation (4.4) computes the request average parallelism under the schedule. Equation (4.5) computes the total parallelism of the system when there are q_r requests. Equation (4.6) and Equation (4.7) calculate the average and tail latency of the requests under the schedule.

Second, we enumerate all loads and all potential schedules, evaluate if they satisfy the parallelism target $target_p$, and compute tail and mean latency. If multiple schedules

have the same minimum tail latency, we choose the one that minimizes the mean. Algorithm 1 shows the pseudocode for this search process. For each potential system load q_r, ranging from one request to the maximum system capacity, we generate all candidate schedules $\mathcal{S}$. Each component interval of a candidate schedule $v_i \in \mathcal{S}$ will take a value from 0 to y, where y is the maximum request length in the workload. We choose the schedules whose total average parallelism of all concurrent requests does not exceed the target hardware parallelism $target_p$, so we avoid oversubscribing the system. It is also important algorithmically: the formulation in Figure 4.4 calculates the request latency assuming all software parallelism nicely maps to hardware resources, which no longer holds when the total software parallelism exceeds $target_p$. Note that we do not need a lower bound on total parallelism. While optimizing tail latency, we will find schedules with total parallelism close to $target_p$, maximizing the utilization of all resources to reduce tail latency. Moreover, if we include a lower bound of $target_p$, we may not find a feasible schedule to meet it under light load, e.g., there is only one request with maximum software parallelism 4, but $target_p = 20$.

From Algorithm 1, we can easily derive the complexity of interval table construction as

$$(y/step)^n \times req_{max} \times |R| \ .$$

This search problem is rather compute intensive. We take several steps to make it faster. First, we search in steps. If a target tail latency is in the range of 100 ms, then we limit the intervals to steps of 10 ms. Second, we only search for intervals in the range of the lifetime of the longest request. For example, when searching for intervals to increase parallelism from 1 to 4, we only search where the sum of all 3 intervals is less than the lifetime of a request. Third, if some interval does not satisfy $target_p$ for lower number of requests, it will not satisfy the $target_p$, for any higher number of requests. For accuracy, we use individual request profiles for Bing and Lucene. This process takes about four to six hours, as we process 10K - 100K requests one by one for each schedule. We may further reduce the search time by grouping requests into demand distribution bins with their frequencies, which reduces our computation time

$$time_r(\mathcal{S}) = \tag{4.3}$$

$$\begin{cases} v_0 + seq_r & \text{if } seq_r \leq v_1 \\ v_0 + v_1 + \frac{seq_r - v_1}{s_r(2)} & \text{if } v_1 < seq_r \leq v_1 + s_r(2) \times v_2 \\ ... & \\ \sum_{i=0}^{n-1} v_i + \frac{seq_r - \sum_{i=1}^{n-1} s_r(i) \times v_i}{s_r(n)} & \text{if } seq_r > \sum_{i=1}^{n-1} s_r(i) \times v_i \end{cases}$$

$$ap_r(\mathcal{S}) = \tag{4.4}$$

$$\begin{cases} \frac{0 \times v_0 + 1 \times seq_r}{time_r(\mathcal{S})} & \text{if } seq_r \leq v_1 \\ \frac{0 \times v_0 + 1 \times v_1 + 2 \times \frac{seq_r - v_1}{s_r(2)}}{time_r(\mathcal{S})} & \text{if } v_1 < seq_r \leq v_1 + s_r(2) \times v_2 \\ ... & \\ \frac{\sum_{i=0}^{n-1} i \times v_i + n \times \frac{seq_r - \sum_{i=1}^{n-1} s_r(i) \times v_i}{s_r(n)}}{time_r(\mathcal{S})} & \text{if } seq_r > \sum_{i=1}^{n-1} s_r(i) \times v_i \end{cases}$$

$$ap_R(\mathcal{S}, q_r) = \frac{\sum_{r \in R} time_r(\mathcal{S}) \times ap_r(\mathcal{S})}{\sum_{r \in R} time_r(\mathcal{S})} \times q_r \tag{4.5}$$

$$time_R(\mathcal{S}, \text{mean}) = \frac{\sum_{r \in R} time_r(\mathcal{S})}{|R|} \tag{4.6}$$

$$time_R(\mathcal{S}, \beta\text{-tail}) = L[\lceil \beta \cdot |R| \rceil] \ , \tag{4.7}$$

L is execution times $time_r(\mathcal{S})$ of all requests $r \in R$ in non-decreasing order.

Figure 4.4: Mathematical formulation of average parallelism, mean and tail latency for interval $\mathcal{S}$.

to a few minutes. The offline analysis can run daily, weekly, or at any other coarse granularity, as dictated by the characteristics of the workload.

4.4.1.4 Admission control

Optimizing for $target_p$ does not directly control the number of active requests in the system. In particular, at high load, we want to determine whether to admit a request or to increase parallelism of the existing requests. Our search algorithm explicitly explores this case by enumerating non-zero values for the first interval (v_0). Furthermore at very high load, if the search returns the maximum value of $v_0 = y$, then the schedule specifies a new request must wait for one to exit and then starts executing with parallelism degree 1. We denote this schedule as (e_1, d_1) in an interval table.

Algorithm 1: Pseudocode for interval table construction.

```
   input  : req_max, Maximum number of simultaneous requests
   input  : y, st, Maximum time interval and interval step values
   output: Core allocation for current request
 1 for q_r ← 1 to req_max do
 2     min_tl = min_ml=inf                          // Initialize minium tail and mean latency
 3     result = φ
 4     for v_0 ← 0 to y in step st do
 5         for v_1 ← 0 to y in step st do
 6             ...
 7             for v_n−1 ← 0 to y in step st do
 8                 S = (v_0, v_1, ..., v_n−1)
 9                 if ap_R(S, q_r) ≤ target_p then
10                     tail = time_R(S, β−tail)
11                     mean = time_R(S, mean)
12                     if tail < min_tl or (tail = min_tl & mean < min_ml then
13                         min_tl = tail
14                         min_ml = mean
15                         result = S
16                     end
17                 end
18             end
19         end
20     end
21 end
22 Add result to interval table entry q_r
```

4.4.2 Online Scheduling

The online FM scheduler is invoked when new requests enter the system and requests terminate. Each request self-schedules itself periodically based on a scheduling quanta, e.g., every 5 or 10 ms. If FM detects oversubscription of hardware parallelism, it boosts the priority of all the threads executing a long request to insure its quick completion.

FM tracks the load by computing the number of requests in the system in a synchronized variable and uses this number to index the interval table. This simple method has several advantages. First, the number of requests is fast and easy to compute compared to other indicators, such as CPU utilization. Second, in contrast with coarse-grained load indicators, such as RPS, it measures the instantaneous load. FM exploits instantaneous spare resources to avoid transient overloading. Third, FM self-corrects quickly. If the number of requests increases due to transient load, FM will index a higher row in the table, which has larger interval values and will introduce parallelism more conservatively. Similarly, when the number of requests decreases, FM will promptly introduce more parallelism for longer requests, as specified by a lower row in the table, which has

shorter intervals.

Each time a request enters, FM computes the load, consults the interval table, and either starts or queues the request. When a request leaves, FM computes the load and starts a queued request (if one exists). After a request starts, it self-schedules, regularly examining the current load and its progress at the periods defined by the scheduling quanta. Each self-scheduling request indexes the interval table by the instantaneous load and if it has reached the next interval, adds parallelism accordingly. We choose relatively short scheduling quanta, less than the interval size in the table, because if requests leave the system, then FM can react quickly to add more parallelism. FM self-scheduling increases scalability of the scheduler by limiting synchronization.

FM will on occasion oversubscribe the hardware resources at high load because we choose a target hardware parallelism that exceeds the number of cores. This choice ensures that FM fully utilizes hardware resources when threads are occasionally blocked on synchronization, I/O, or terminating, but under high load will degrade tail latency if long requests must share resources with short requests. For example, operating systems generally implement a round robin scheduling to give equal resources to all the threads. To mitigate this issue, we implement *selective thread boosting*. Boosting increases the priority of all threads executing a single long request. We ensure that the number of boosted threads is always less than the number of cores by using a synchronized shared variable to count the total number of boosted threads. We only boost a request when increasing its parallelism to the maximum degree and when the resulting total number of boosted threads will be less than the number of cores. This mechanism instructs the OS to schedule these threads whenever they are ready. The longer requests will thus finish faster, which improves tail latency.

4.5 Evaluation

The next two sections evaluate the FM algorithm in two settings. We implement the offline table construction algorithm using around 100 lines of Python that we use for both systems. We compare both systems to the prior state-of-the-art parallelization approaches and find that FM substantially improves tail latencies over these approaches.

We compare FM to the following schedulers.

Sequential (SEQ) Each request executes sequentially.

Fixed parallelism (FIX-N) Each request executes with a predefined fixed parallelism degree of N.

Adaptive (Adaptive) This scheduler [82] selects the parallelism degree for a request based on load when the request first enters the system. The parallelism degree remains constant.

Request Clairvoyant (RC) This scheduler is oracular, because it is given all requests' sequential execution times. It is an upper bound on predictive scheduling [84], which estimates request length. It selects a parallelism degree for long requests when they enter the system based on a threshold and executes other requests sequentially. The parallelism degree is constant.

SEQ and *FIX-N* are reference points, and *Adaptive* is the prior state-of-the-art for exploiting parallelism in interactive services. *RC* assumes perfect prediction of request length, but does not adapt to load. An algorithm for an optimal scheduler is unknown and at least NP hard. It is harder than bin packing, since it may divide jobs. It also requires knowledge of future request arrivals. We configure FM to use instantaneous load and each requests' progress to add parallelism and, when necessary, to use selective thread priority boosting. Because FM dynamically adapts to total load, it is significantly better than RC, which only considers the demand of individual requests when they enter the system.

4.6 Lucene Enterprise Search

This section presents our experimental evaluation of FM in Lucene. Apache Lucene [9] is an open-source Enterprise search engine. We configure it to execute on a single server with a corpus of 33+ million Wikipedia English Web pages [150, 112]. We use 10K search requests from the Lucene nightly regression tests as input to our offline phase and 2K search requests for running the experiments. While the nightly tests use a range of request types, we use the term requests. The client issues requests in random

order following a Poisson distribution in an open loop. We vary the system load by changing the average arrival rate expressed as RPS. The index size is 10 GB and it fits in the memory of our server. Figure 2.1 shows the service demand distribution and the speedup profile of the workload.

4.6.1 Methodology

4.6.1.1 Implementation

We execute 10K requests in isolation with different degrees of parallelism and gather their execution times. Each time is an average of at least 10 executions. For a specific parallelism degree, we compute the speedup of all requests and the average speedup across all requests. The sequential execution times and speedups of all requests constitute the input to the offline phase. Since the online module of FM implements admission control and assigns work to threads, we implement it within the Lucene request scheduler. Lucene is implemented in Java and we implement the scheduler in Lucene in roughly 1000 lines of Java code.

We make minor changes to the existing Lucene code base. Lucene arranges its index into segments. To add parallelism, we simply divide up the work for an individual request by these segments. We do *not* change how Lucene's default mechanisms create its index and maintain the segments. We note that this type of data organization is common to many services and makes implementing incremental parallelism simple. We extend Lucene to execute each request in parallel by adding a Java *ExecutorService* instance. We use the *ThreadPoolExecutor* class that implements *ExecutorService* and that configures the number of threads in the thread pool. Each main thread retrieves a request from a shared queue and processes the request. The main thread self-schedules periodically (every 5 ms) and checks the system load. As specified by the interval table, it increases parallelism of a request by adding threads. FM adds a thread by simply changing a field of *ThreadPoolExecutor*. Lucene starts a new thread that works on a new segment and synchronizes it with other worker threads.

4.6.1.2 Hardware

We use a server with two 8-core Intel 64-bit Xeon processors (2.30 GHz) and turn off hyperthreading. (Reasoning about job interference with hyperthreading together with parallelism is beyond the scope of this paper [129].) The server has 64 GB of memory and runs Windows 8. Out of the available 16 cores, we use 15 cores to run our experiments and 1 core to run the client that generates requests. We empirically set the target hardware parallelism, $target_p = 24$. We explored several values for $target_p$ and observed only small differences for $target_p \in [20, 28]$.

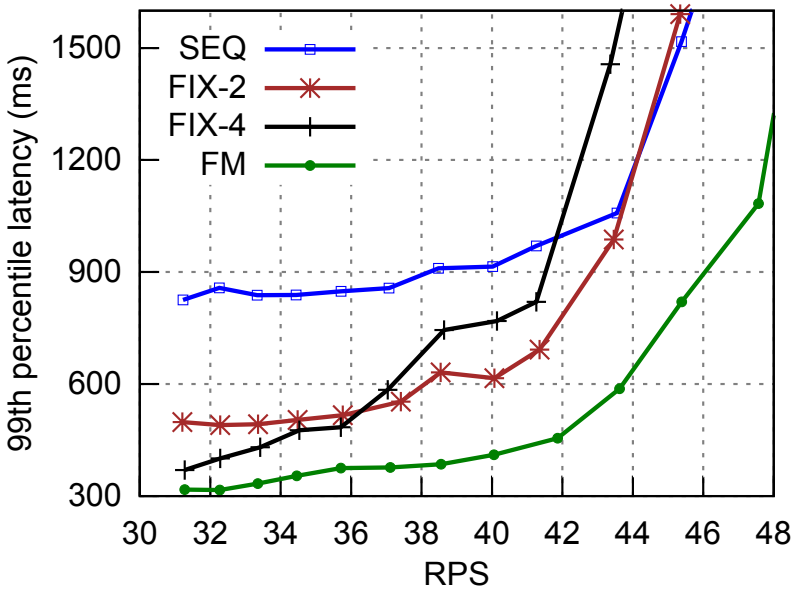
We report the 99th percentile latency, average latency, and CPU utilization of different policies. Latency includes both queuing delay and execution time. We use Java *NanoTime* for fine-grain time measurements. Our reported tail latency is the 99th percentile of the response times of all requests and the mean is the average response time measured over the 2K requests under various loads.

4.6.1.3 Interval selection for FM

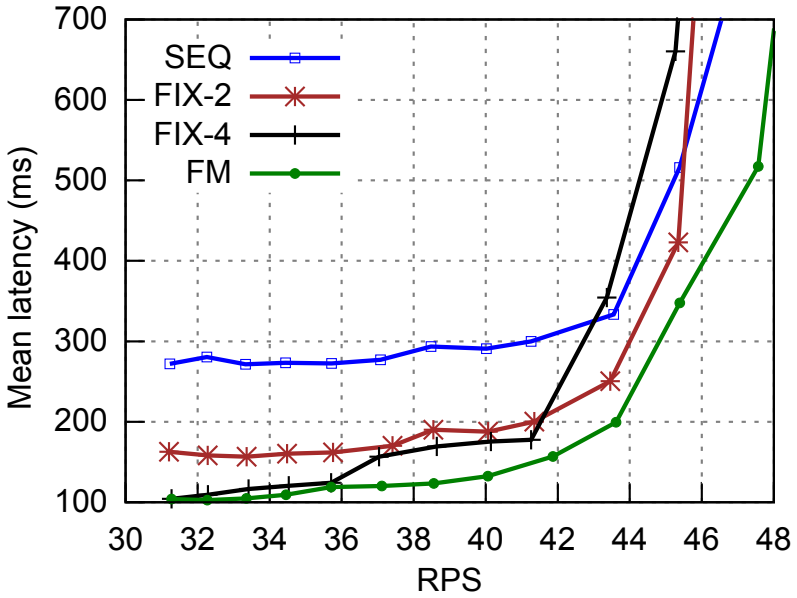
Table 4.2 shows the intervals generated by the offline phase for a target parallelism of 24. The table is indexed by the number or requests in the system, our metric for system load. Each column shows the time in ms at which FM adds parallelism. We use a step size of 5 ms to generate the interval table. As discussed in Section 4.4.1.4, the second column is for admission control. New requests must wait for this time before they start processing. When the load is very low (less than 6 requests), FM starts each request with parallelism degree 4. At other low loads (7 to 13 requests), FM starts a new request whenever it arrives and adds parallelism following the corresponding row. Finally, as load increases (more than 14 requests), FM delays new requests and uses longer intervals to add parallelism.

n_r	t_0	t_1	t_2	t_3
$\leq$6	0, d_4			
7	0, d_3	75, d_4		
8	0, d_2	25, d_3	150, d_4	
9	0, d_2	50, d_3	150, d_4	
10	0, d_1	25, d_2	100, d_3	175, d_4
11	0, d_1	25, d_2	125, d_3	175, d_4
12	0, d_1	75, d_2	150, d_3	200, d_4
13	0, d_1	100, d_2	175, d_3	225, d_4
14	10, d_1	110, d_2	210, d_3	235, d_4
15	30, d_1	130, d_2	205, d_3	255, d_4
16	40, d_1	140, d_2	240, d_3	265, d_4
17	60, d_1	160, d_2	245, d_3	285, d_4
18	60, d_1	185, d_2	260, d_3	310, d_4
19	70, d_1	195, d_2	270, d_3	320, d_4
20	70, d_1	220, d_2	295, d_3	370, d_4
21	80, d_1	255, d_2	305, d_3	375, d_4
22	80, d_1	280, d_2	330, d_3	380, d_4
23	90, d_1	290, d_2	365, d_3	415, d_4
24	90, d_1	315, d_2	390, d_3	440, d_4
$\geq$25	e_1, d_1	315, d_2	390, d_3	440, d_4

Table 4.2: Lucene interval table in milliseconds for 99th percentile latency with $target_p$ = 24 threads and maximum parallelism $n = 4$. When $t_0 = e1$, FM waits for a request to complete and then admits one waiting request. Each entry specifies execution thus far and parallelism d_k to add.

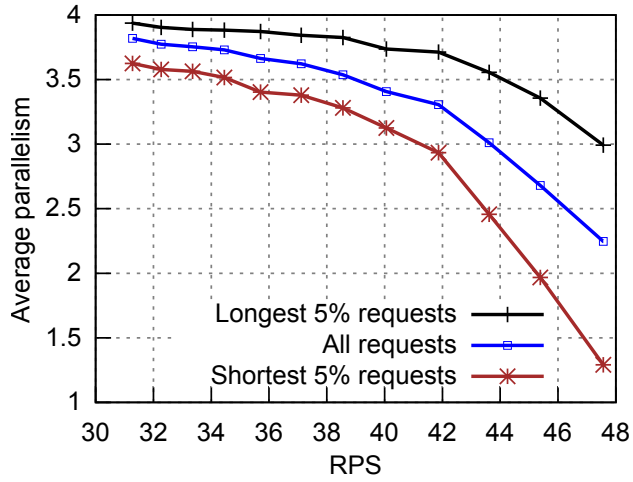


(a) 99th percentile latency

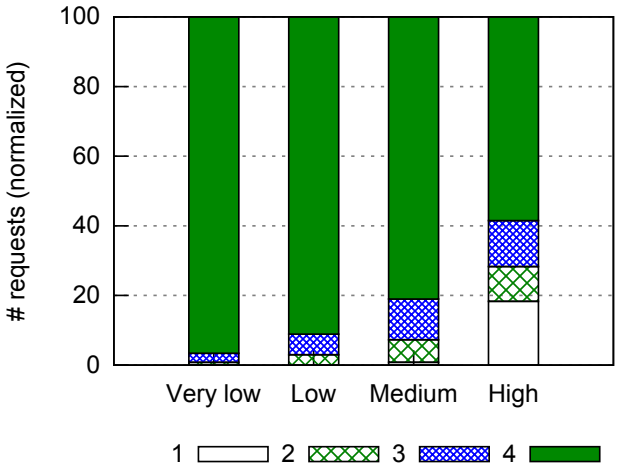


(b) Mean latency

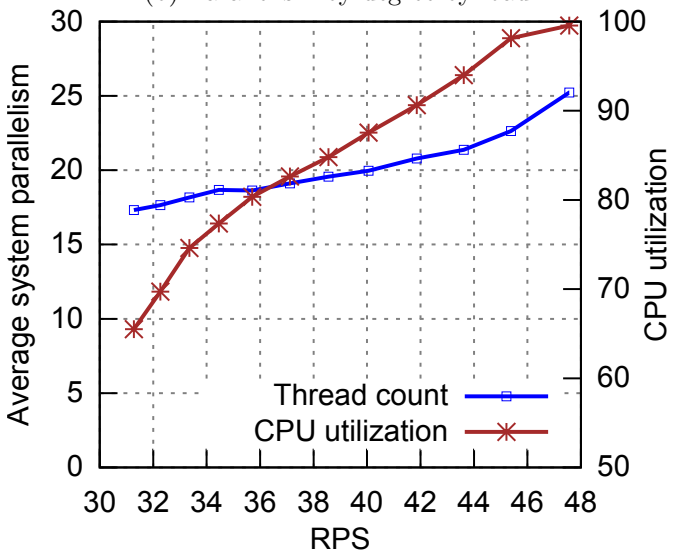
Figure 4.5: Lucene latency compared to fixed parallelism.



(a) Average request parallelism



(b) Parallelism by degree by load



(c) Threads in the system (left y-axis) and CPU utilization (right y-axis)

Figure 4.6: Lucene breakdown of parallelism degree by requests and total number of threads in the system.

4.6.2 Results

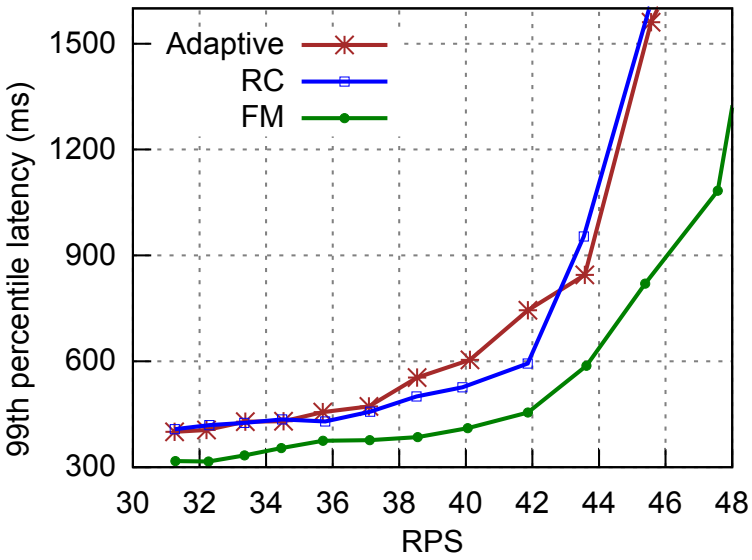
4.6.2.1 Comparison to fixed parallelism policies

We compare FM to sequential execution (SEQ) and a fixed degree of parallelism for each request (FIX-2 and FIX-4). Figures 4.5(a) and 4.5(b) show the 99th percentile and mean latency. FM has consistently lower tail and average latency than the other policies. At low load, many cores are available; FM is close to but better than FIX-4. Here FM aggressively parallelizes almost all requests with 4 threads (as shown in Table 4.2, row $n_r <= 6$). Occasional request bursts and short requests reduce intra-request parallelism on occasion even at low load. At high load, FM uses thread boosting and instantaneous load to carefully manage the degree of parallelism per request. At a medium load (40 RPS), FIX-4 is already worse than FIX-2. In contrast, FM reduces the 99th percentile latency by 33% and mean latency by 29% compared to FIX-2. At high load (43 RPS), FM reduces the 99th percentile and mean latency by 40% and 20%, respectively.

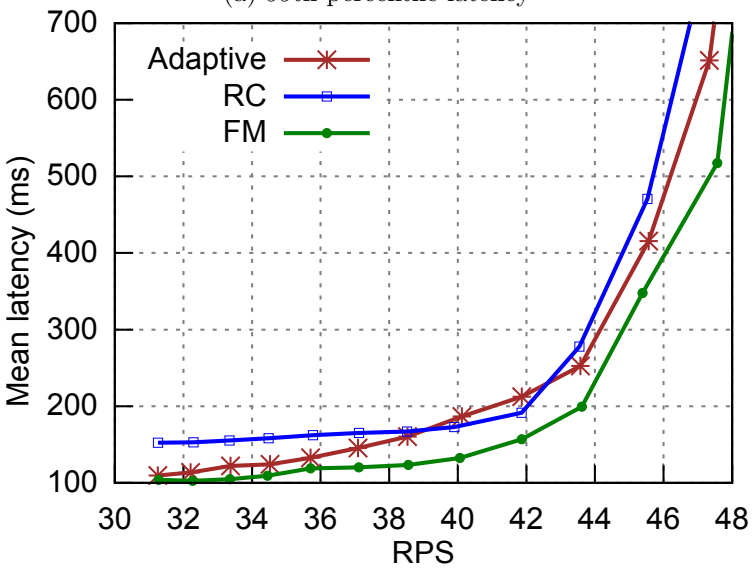
4.6.2.2 FM characteristics

We now examine the parallelism degree and number of threads in FM. Figure 4.6(a) shows the average parallelism degree for all requests, the longest 5%, and the shortest 5%. At low load, FM assigns a high parallelism degree (almost 4) to all requests. As load increases, FM becomes less aggressive and assigns requests less parallelism. On average however, long requests have a higher degree of parallelism than short requests. At high load (47 RPS), short requests mostly run sequentially with an average parallelism degree of 1.29 and long requests run with an average degree of 3. These results show that FM adapts parallelism to the system load and favors running longer requests with higher parallelism than shorter requests.

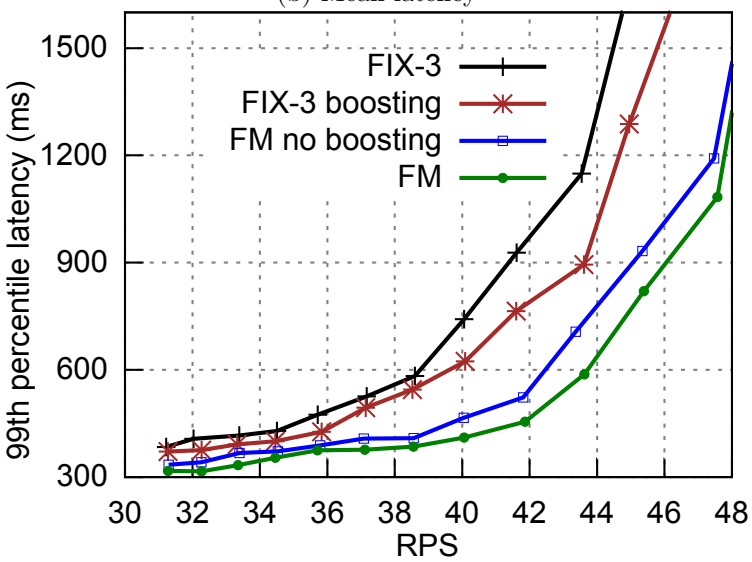
We select four RPS values, (31, 36, 40, 45) to represent ("Very low", "Low", "Medium" and "High") loads. Figure 4.6(b) shows the parallelism degree distributions over all requests. At very low load, most requests run with degree 4. At high load, 19% of the requests finish sequentially and 41% finish with degree 3 or less. At



(a) 99th percentile latency



(b) Mean latency



(c) Thread priority boosting

Figure 4.7: Lucene latency comparison with Adaptive and Predictive policies (a,b). Effect of thread priority boosting (c).

high load, a request that runs with parallelism degree 4 would have run with a lower parallelism degree for part of its execution. Short requests are thus likely to complete sequentially, which is the most efficient way to execute them, saving processing resources for parallelizing longer requests.

We report the average number of threads in the system and CPU utilization in Figure 4.6(c). The average number of threads is between 17 and 25, which is close to our target of 24. CPU utilization increases with load. At very high load (48 RPS), CPU utilization is almost 100% and the average number of threads is 25.

4.6.2.3 Comparison to the state-of-the-art policies

We compare FM to state-of-the-art "Adaptive" and perfect prediction "RC" polices described in Section 4.5. For RC, we execute short requests sequentially and long requests with parallelism degree 4. We experimentally search for the best threshold to divide requests between short and long requests. This threshold is 225 ms.

Figure 4.7(a) and 4.7(b) show the 99th percentile latency and mean latency. FM performs consistently better than Adaptive and RC. At low load, FM executes most requests with high degrees of parallelism. Adaptive aggressively runs all requests with high degrees, but does not differentiate between long and short requests. RC misses the opportunity to parallelize requests shorter than its threshold. At medium load (40 RPS), FM reduces the 99th percentile latency by 32% and 22% compared to Adaptive and RC, respectively. Adaptive has higher latency than RC because it executes short requests in parallel, while RC runs short requests sequentially. At high load (43 RPS), RC is worse than Adaptive because it does not adapt to high load by reducing parallelism. At this load, FM reduces the tail latency by 30% and 38% compared to Adaptive and RC, respectively.

FM reduces the tail latency more than Adaptive and RC for three reasons. First, both Adaptive and RC select the parallelism degree at the start and do not dynamically adapt. FM has an extra degree of freedom. It may dynamically increase the parallelism of each individual request. Second, FM favors longer requests with additional parallelism, and shorter requests are more likely to execute sequentially especially under high

load. In contrast, Adaptive does not differentiate between short and long requests, and RC, despite being clairvoyant of the request service demand, uses a static threshold, fixed parallelism degree, and ignores system load. Third, FM exploits thread boosting to further reduce the tail latency.

4.6.2.4 Selective thread priority boosting

To study benefits of boosting, we disable thread boosting in FM and add it to a fixed parallelism configuration. Since the average parallelism of FM ranges from 2.2 to 3.8 (Figure 4.6(a)), we select fixed degree 3 (FIX-3) and compare FM with FIX-3 with boosting. Figure 4.7(c) shows the 99th percentile latencies. Selective thread priority boosting improves the tail latency for both FM and FIX-3. In particular, thread priority boosting reduces the tail latency of FM by 12% at 40 RPS, and by 16% at 43 RPS. Selective thread priority boosting ensures that short requests do not interfere with long requests, reducing tail latency.

4.6.2.5 Load variation

To study the effect of load variation and burstiness on tail latency, we configure the client to vary the request submission rate in quick succession. The client varies load in four quanta: high (45 RPS) to low (30 RPS) to high (45 RPS) to low (30 RPS). The client submits 500 requests in each quanta. Figure 4.8 compares 99th percentile latencies of the last 100 requests in each quanta for FM, SEQ, FIX-2, and FIX-4. The performance of SEQ is usually the worst, but is slightly better than FIX-4 at the beginning of a load burst. FM adapts well to changes in load and consistently achieves the best tail latency. FIX-4 is the most aggressive and performs almost as well as FM at low load, but performs substantially worse during the load burst. Comparing FIX-2 and FIX-4, we see that FIX-2 performs better than FIX-4 during the high load quanta, but worse at low load. This experiment shows that FM is stable, responding quickly and smoothly to highly variable load. FM tunes its scheduling policy to the load to reduce tail latency.

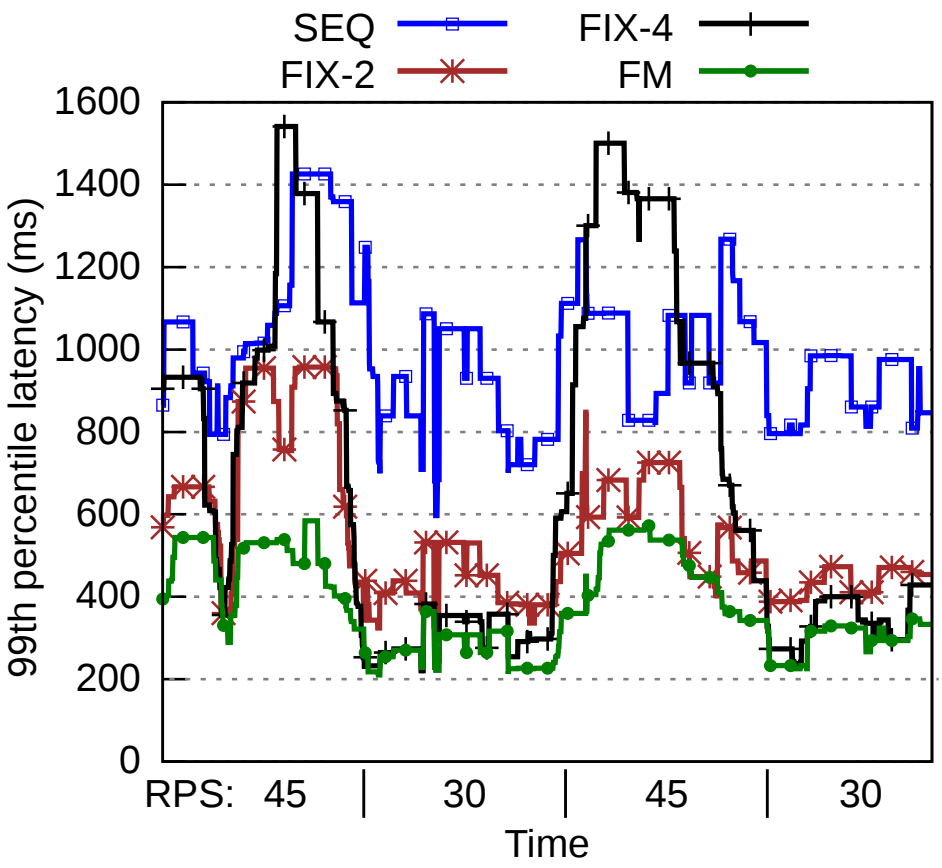


Figure 4.8: 99th percentile latency for the last 100 requests in a 500-request quanta, while varying the load in Lucene.

4.7 Bing Web Search

This section presents our evaluation of FM in the index servers of Microsoft Bing Web search with a production index and query log. The Bing index serving system consists of aggregators and index serving nodes (ISNs). An entire index, containing information about billions of Web documents, is document-sharded [14] and distributed among hundreds of ISNs. When a user sends a request and the result is not cached, the aggregator propagates the request to all ISNs hosting the index. Each ISN searches its fragment of the index and returns the top-k most relevant results to the aggregator. Because Bing is compute-bound at the ISN servers (see Chapter 2), a long latency at any ISN manifests as a slow response [33]. To reduce the total tail latency, each ISN must reduce its tail latency. For example, assuming the aggregator has 10 ISNs, if we want to process 90% of user requests within 100 ms, then each ISN needs to reply within 100 ms with probability around 0.99. In other words, for a total latency of 100

ms at the 90th-percentile response time, the response time of each ISN must be at most 100 ms at the 99th-percentile. These results motivate our evaluation of Bing on an individual server.

4.7.1 Methodology

4.7.1.1 Implementation

We implement FM in the Bing request dispatch code. To eliminate thread creation time, we use a thread pool. Bing organizes its indexes in groups and thus adding incremental parallelism is not onerous. We configure two servers. One server is the index server, which executes requests. A second server is the aggregator, which sends requests to the index server by replaying a trace containing 30K Bing production user requests from 2013. We vary system load by changing the average request arrival rate in RPS. This version of FM does not perform thread boosting. FM uses a target number of threads, $target_p = 16$, a slightly higher number than the 12 available cores (see below). As we have seen in Chapter 2, the efficiency of parallelism drops significantly at degree 4, thus we configure FM to increase the parallelism degree of a request up to 3.

4.7.1.2 Hardware and OS

For the index service, we use a server with two 2.27 GHz 6-core Intel 64-bit Xeon processors and 32 GB of main memory with Windows Server 2012. The ISN manages a 160 GB index partition on an SSD, and uses 17 GB of its memory to cache index pages from the SSD.

4.7.2 Results

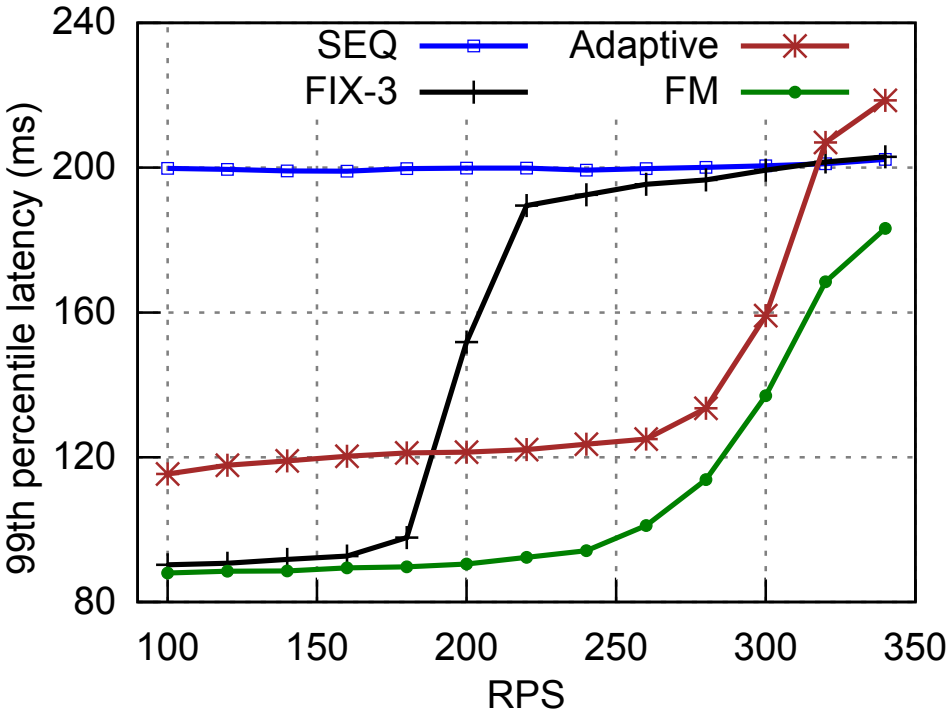
We compare FM to fixed parallelism with degree 3 (FIX-3), Adaptive, and SEQ. The production version of FIX-3 applies load protection: it parallelizes each request using degree 3 when the total number of requests in the system is less than 30. Otherwise, it runs requests sequentially.

Figure 4.9 presents the 99th-percentile latency over different loads. These experimental results show that FM consistently has the lowest tail latency. In particular, it effectively reduces the tail latency at moderate to high load. For example, up to 260 RPS, FM exhibits a 99th percentile latency of 100 ms. In contrast after 150 RPS, FIX-3 has latencies of over 200 ms because it oversubscribes the resources, parallelizing all requests regardless of system load and the request length. FM also performs better than Adaptive at all load levels. For example, at 180 RPS, the tail latency reduction over Adaptive is 26% and at 260 RPS, the improvement is 24%. FM reduces the tail latency more than Adaptive because it runs long requests with higher parallelism degrees than short requests. These results show that FM enables the provider to service the same load with 42% fewer servers compared to Adaptive for a target tail latency of 120 ms.

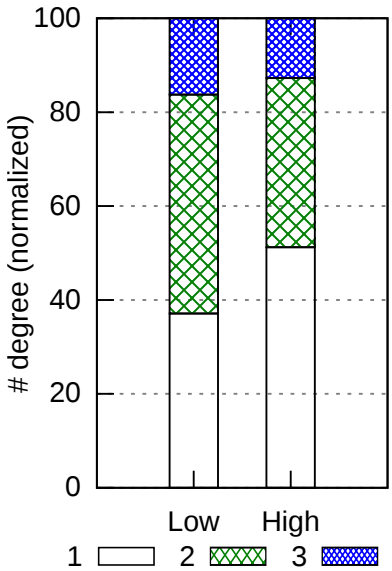
To study FM parallelism, we select two representative load values: 200 RPS for "Low" and 280 RPS for "High" load. Figure 4.9(b) shows the distribution of parallelism per request. Figure 4.9(c) shows total threads, which are underutilized at low load and match the target load of 16 at high load, as expected. Comparing low to high load, FM uses less intra-request parallelism as load (system parallelism) increases. About 35% of requests execute sequentially at low load, whereas over 50% are sequential at high load.

4.8 Summary

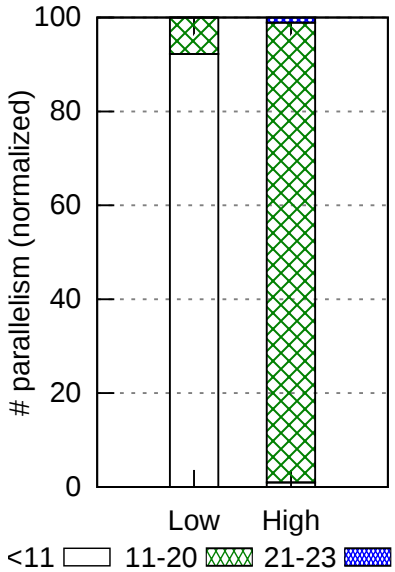
This chapter introduces a new parallelization strategy called Few-to-Many (FM) incremental parallelism for reducing high-percentile latency in interactive services. FM uses the demand distribution and a target hardware parallelism to determine dynamically when and how many software threads to add to each request. At runtime, it uses system load and request progress to add parallelism, adapting each request individually. We implement FM in two search engines, Lucene and Bing, and evaluate it using production request traces and service demand profiles. Our results show that FM can significantly reduce tail latencies, and help service providers to reduce their infrastructure costs. These results and our experience suggest that FM should be extremely useful in practice. In fact, Bing engineers have already deployed a basic version of FM



(a) Comparison to SEQ, FIX-3, and Adaptive



(b) Request parallelism



(c) Thread Distribution

Figure 4.9: Bing comparisons and parallelism.

in the production Bing system on thousands of servers.

Chapter 5

Managing Tail Latency and Energy Consumption in Interactive Services on Heterogeneous Processors

In this chapter, we discuss Adaptive slow-to-fast (Adaptive S2F) scheduling for heterogeneous processors. Recall that, Adaptive S2F seeks to use slower but more energy efficient cores to conserve energy, but migrates older requests to faster cores to meet a tail latency target. Adaptive S2F decides time thresholds for different loads at which the requests should migrate to fast cores. Requests that run for longer than the set threshold (and do not finish) will migrate to a free fast core. This ensures that long requests get more share of fast cores and, thereby, meet a tail latency target. Adaptive S2F has two components. Offline, we build a set of controllers based on the request service demand distribution, tail latency target and tail latency behavior with respect to different thresholds. In the online phase, a particular controller is selected based on offered load. The controller then determines the threshold and requests use this threshold to migrate to fast cores (tie breaking is done by request ages). We discuss the details of the controller design and online algorithm in Section 5.1. Section 5.2 presents the evaluation methodology. We present results for Lucene and Finance server in Section 5.3. Finally, Section 5.4 summarizes the chapter.

5.1 Adaptive Slow to Fast

This section presents the design of our scheduler for interactive services on heterogeneous processors. The service processes each request independently and sequentially, with multiple simultaneous requests executing on multiple cores concurrently. Requests

have variable service demands that change infrequently or slowly over time. The hardware consists of N heterogeneous core types with different power and performance characteristics, due to different microarchitectures and/or clock frequencies, but all use the same ISA. Slower cores are more energy efficient than faster cores, otherwise, why build a slower core.

Key criteria for our scheduler are that it should incur very low overhead and be scalable with the number of cores. Our solution comprises an offline controller design component, and an online component. The offline component defines feedback controllers that compute migration thresholds based on measured tail latency, target tail latency, and system load. The online component migrates requests (threads) based on the thresholds and request progress. We mainly consider two popular service provider objectives: (1) reducing tail latency to improve responsiveness and (2) reducing energy for a tail latency target to reduce operating costs. We consider an energy-only objective as a lower bound, although it is not usually the objective for interactive services. We show how to adjust the migration thresholds to meet all these objectives.

5.1.1 Scheduling Objectives

The key challenge is to determine *when* to migrate a request because the migration threshold controls tail latency and energy efficiency. A low threshold migrates requests sooner and uses fast cores more often, reducing tail latency at the expense of higher energy consumption. Higher thresholds use fast cores less often, consume less energy, and incur higher tail latencies.

5.1.1.1 EETL: Energy Efficiency with Target Tail Latency

The objective here is to meet a target tail latency while being as energy-efficient as possible. On one hand, the threshold must be short enough to meet the latency target. On the other hand, it must be long enough such that the system does not consume unnecessary energy. The threshold thus depends on the target tail latency, workload, processor configuration, and dynamic system load.

5.1.1.2 RTL: Reducing Tail Latency

If reducing tail latency is the only goal, the threshold should be zero, such that requests execute on the fastest available cores as much as possible. Furthermore, starting requests on the fastest available core, and migrating the oldest request to a faster core as soon as one becomes available optimizes tail latency. Under moderate to high load, only long requests ever migrate to the fastest cores because short requests finish on slower cores and only long old requests execute long enough to migrate to the fastest cores. A zero threshold therefore optimizes for lower tail latency, but it consumes more energy as the fastest cores are always busy.

5.1.1.3 EE: Energy Efficiency

As a lower bound, only considering an energy efficiency objective sets the threshold as high as possible (infinity), using fast cores only if the slow cores are all busy. When a new request arrives and the slow cores are busy, we migrate the oldest request to a faster core to reduce tail latency.

The next section presents the offline and online portions of AS2F, which implements RTL and EE as simple configurations. We achieve EETL tail latency and energy efficiency objectives by designing feedback-based controllers.

5.1.2 Offline Controller Design for Two Core Types

This section describes the design of a Single Input Single Output (SISO) feedback controller to determine the threshold that achieves the EETL objective for two core types. Section 5.1.4 generalizes the controller for N core types. The controller input is the difference between the current tail latency and the target, and the output is the migration threshold. To determine the effect of input changes on the output, we measure tail latency on the target processor as a function of load and migration thresholds, using a representative request trace. Given a specified tail latency target that falls in the range of measured and thus achievable tail latencies, we develop the controller to achieve the target tail latency.

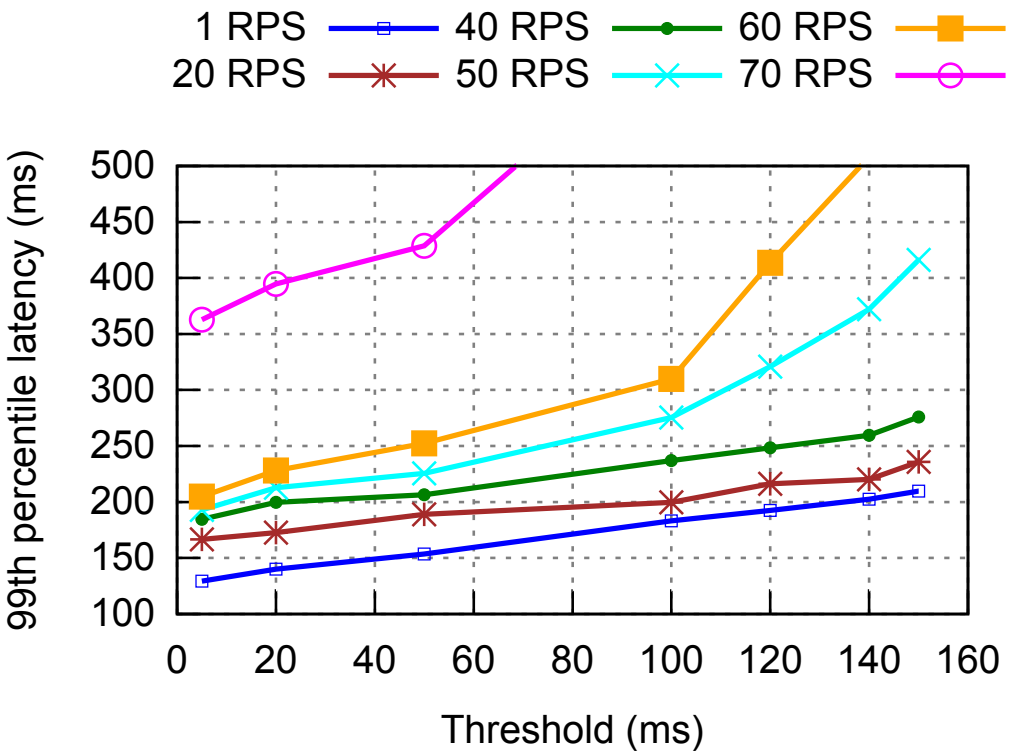


Figure 5.1: Relationship between tail latency, migration threshold, and load (RPS) on 8 slow and 1 fast cores.

Figure 5.1 depicts an example for Lucene. Unsurprisingly, it shows that the lower the threshold and load, the lower the potential tail latency. The highest load that this system can support within a target tail latency of 250 ms is a little less than 60 RPS for Lucene.

The relationship between the migration threshold and tail latency is non-linear for high load and a high threshold. However, these systems are provisioned to operate at low or moderate load, in the linear region below 60 RPS. At low load, the latency and RPS relation is linear and relatively flat. As load increases, the slope increases, but is still linear in practical ranges. Because the relationship is piece-wise linear with the slope varying based on the load, we create a controller for each load range (Gain Scheduling) [74]. We choose Proportional-Integral-Derivative (PID) control because it is effective, and has low overhead.

We divide load into three intervals and design a controller for each interval. We used a sensitivity analysis to validate this choice. Next, for each interval and target tail latency, we determine the open-loop step response with the load at the mid-point of the interval, and compute and tune the PID control gains using MATLAB PID Tuner [116]. Table 5.1 shows the generated PID gains for different intervals.

Figure 5.2 shows the resulting controller block diagram. The load estimator computes the average arrival rate from the request arrivals. The gain scheduler uses this load estimation to select one of the three PID gain parameters. The load estimator does not need to be precise as the load estimation is used only at coarse grain. The controller would still work without any load estimator or gain scheduling (albeit with higher overshoot and settling time, as we would only have one gain class). The PID controller's input is $e(t)$, the difference between the measured tail latency and the target tail latency. The output of the controller is $u(t)$, the migration threshold from slow to fast cores for the estimated load. Our online scheduler then uses the computed threshold to migrate requests from slow to fast cores.

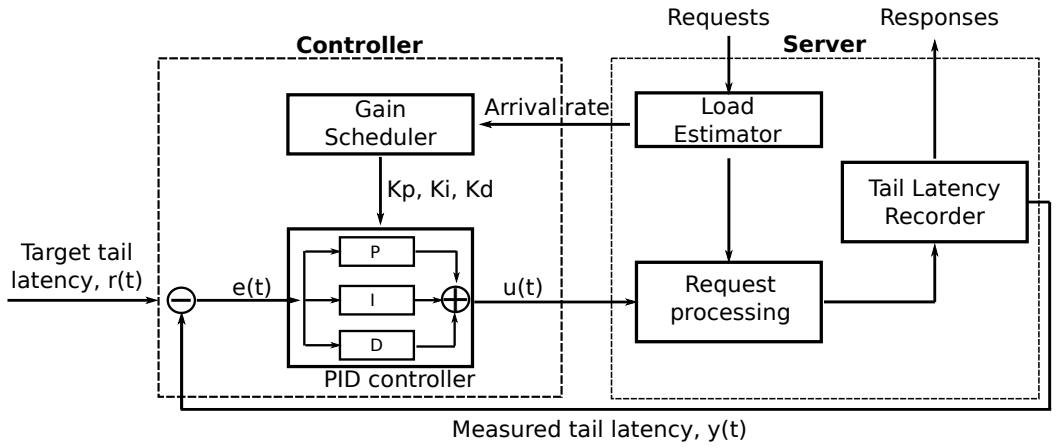


Figure 5.2: Diagram of the controller and interactive server.

Load interval	Representative load	K_p	K_i	K_d
< 20	10	0.11	0.49	0.00
20–40	30	0.15	0.25	0.00
> 40	50	0.19	0.17	0.03

Table 5.1: Gain values for the three PID controllers based on load (measured in RPS) for Lucene.

5.1.3 Online Scheduling Algorithm

At run time, each request self-schedules, periodically executing Algorithm 2. When (i) the request execution time crosses the threshold, (ii) the request is not executing on a fast core, and (iii) a fast core is available, the request migrates to the fast core (Line 5–9). A request migrates to a fast core (if one is available) before its execution time crosses the threshold if more recent requests are slotted for slow cores (Line 10–14). A new request starts running with lower priority if all cores are taken or there are no available slow cores (Lines 16–17). Otherwise, a new request starts at the regular priority on a slow core (Lines 18–20). We keep the ordering of request ages using a synchronized priority queue.

Since it is easy to generalize this online scheduling algorithm from two core types to N core types and $N - 1$ thresholds, we omit the generalization for brevity.

Algorithm 2: AS2F for two core types. Each request self-schedules independently.

```
   input  : F & S, the set of fast & slow cores
   input  : t the migration threshold
   output: core allocation for current request
 1 n = total number of active requests
 2 a = age of current request
 3 i = request position among all requests by decreasing age
 4 c = current core                                   // c is None for a new request
 5 if (i <= |F| and a >= t) then                      // migrate after crossing t
 6 |   if (c ∉ F) then
 7 |   |   f = free core from F
 8 |   |   migrate from c to f
 9 |   end
10 else if (n > |S| and i <= |F| and i <= n − |S|) then
   // migrate before crossing t because there are not enough slow cores
11 |   if (c ∉ F) then
12 |   |   f = free core from F
13 |   |   migrate from c to f
14 |   end
15 else if (c is None) then
16 |   if (i > |F| + |S| or No free core in S) then   // very high load or sync delay
17 |   |   start running with low priority
18 |   else                                           // regular request arrival
19 |   |   s = free core from S
20 |   |   start executing at s
21 |   end
22 end
```

Symbol	Definition
$r \in R$	Request profiles
$slow_r$	Runtime of request r on slowest core
s_i	Avg speedup of requests on core type c_i
p_i	Peak dynamic power on c_i
$target_l$	Target tail latency
$t = \{0, t_1, t_2, ..., t_{N-1}\}$	t_i is the migration threshold to core type c_{i+1}
$V = \{0, v_1, v_2, ..., v_{N-1}\}$	Intermediate representation of t such that $v_i = (t_i - t_{i-1})$
$e_r(V)$	Energy used by r with schedule V
$latency_R(V, \beta\text{-tail})$	βth-percentile latency of R with schedule V

Table 5.2: Definitions for computing migration thresholds for N core types.

$$\text{Min. } Objective = \frac{\sum_r e_r(V)}{|R|} \tag{5.1}$$

$$e_r(V) = \begin{cases} slow_r \times p_1 & \text{if } slow_r \leq v_1 \\ (v_1 \times p_1 + \frac{(slow_r - v_1) \times p_2}{s_2}) & \text{if } v_1 < slow_r \leq (v_1 + v_2) \\ \cdots \end{cases} \tag{5.2}$$

$$latency_R(V, \beta\text{-}tail) \leq target_l \tag{5.3}$$

$$latency_R(V, \beta\text{-}tail) = L[\lceil \beta \cdot |R| \rceil] \tag{5.4}$$

where L is the run times of all $r \in R$ in non-decreasing order.

Figure 5.3: Threshold problem for low loads. We select t to minimize the objective function.

5.1.4 Offline Controller for N Core Types

This section describes how to create a controller for AMP configurations with $N > 2$ core types. Because N core types can more finely match core capabilities to workloads, they have more potential to improve energy efficiency than two types by striking better power and performance tradeoffs [27, 75, 97, 98, 107]. We therefore expect future processors to use more than two core types. Our controller for N core types specifies $N - 1$ migration thresholds from core type c_i to c_{i+1} where cores of type i exhibit s_i performance (average speedup compared to the slowest core) with p_i power consumption. Without loss of generality, $0 < s_1 < s_2 < \cdots < s_N$ and $0 < p_1 < p_2 < \cdots < p_N$. Table 5.2 defines the symbols we use for the offline algorithm for N core types.

Determining these thresholds is a multi-dimensional search problem. To simplify

it, we introduce a decision variable that converts it into a single-dimension problem, making it amenable to an SISO controller. We divide the procedure into two steps. We first determine the thresholds for low loads based on the workload distribution, using the demand distribution and relative core speeds. We pose this optimization problem as shown in Figure 5.3. We seek to minimize the average energy used by all the requests while satisfying the tail latency target. This problem formulation only applies to low load because it assumes there is no contention for any of the core — each request may execute on a core of type $c_{(i+1)}$ whenever it is older than t_i. The optimization problem is convex, and takes 1-2 seconds with workload distributions for 4 to 8K requests on our server using Gurobi [66].

In the second step, we use the thresholds determined for low load as a starting point for finding thresholds at higher loads. If the solution of the problem in Figure 5.3 is $t^* = \{0, t_1^*, t_2^*, ..., t_{N-1}^*\}$, then we assume the thresholds for all loads take the form $t = \{0, z.t_1^*, z.t_2^*, ..., z.t_{N-1}^*\}$. For low load, $z = 1$ gives the best thresholds. For high load, $z = 0$ gives the best thresholds. Intuitively, the values of thresholds are t^* at low load and, as the load increases, the thresholds decrease until when they all become zero because all cores are occupied. The older a request is, the more likely it is executing on the fastest cores, because oldest requests are promoted first (see Section 5.1.3). Thus, $z = 1$ and $z = 0$ work well for low and high load, respectively. To select z for intermediate loads at run time, we design SISO controllers using the methodology from Section 5.1, running the request trace on the heterogeneous processor with different load and input (z) values. This results in a set of PID controllers from the Gain scheduler for various loads. This approach determines thresholds at a fixed ratio. The resulting schedule may consume unnecessary energy compared to the optimal at moderate to high loads. In practice, deviations are small across many demand distributions and core configurations. We leave the theoretical bound for future work.

5.2 Evaluation Methodology

This section describes our methodology for evaluating AS2F on emulated heterogeneous processors.

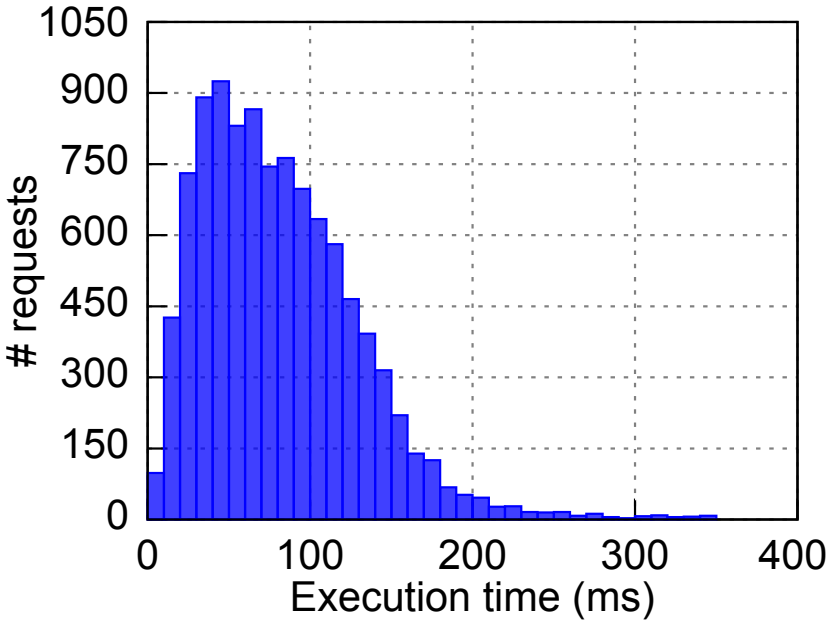
5.2.1 Workloads

We use the open-source widely used Apache Lucene [9], an Enterprise search engine and a Monte Carlo Finance Server [23, 30, 72]. Both servers use a standard pool of worker threads. A worker thread dequeues a request from the arrival queue and processes the request to completion.

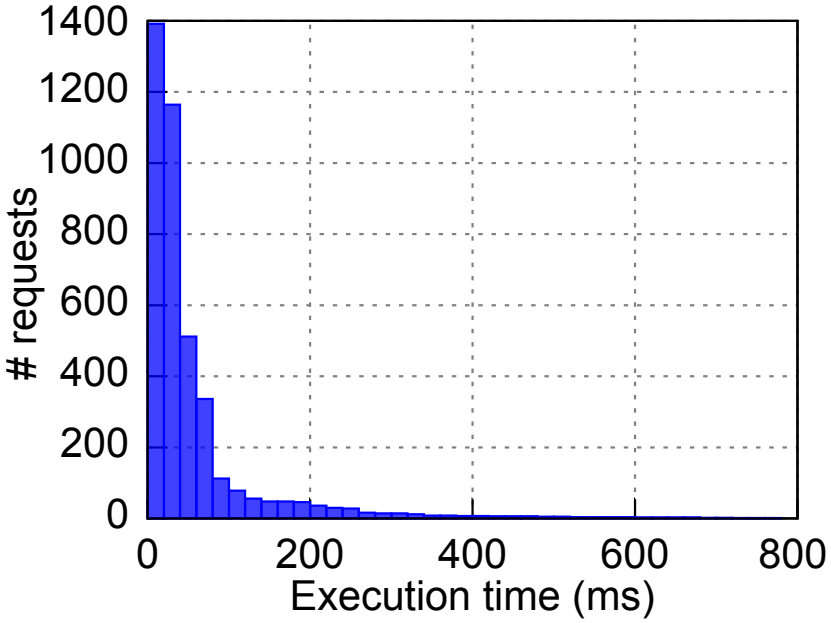
We configure Lucene to search a corpus of 33+ million Wikipedia English Web pages [150, 112]. The search index consumes 10 GB and fits entirely into the memory of our server. We use 10 K term search requests from Lucene's nightly regression tests to build the workload distribution. Each performance experiment uses a random 2 K subset of these search requests. We configure a client to issue the 2 K requests in random order, following a Poisson process in an open loop. We vary system load by changing the request arrival rate, measured as requests per second (RPS). Figure 5.4(a) shows the service demand distribution of the requests.

We add our online scheduler to Lucene, which is written in Java, with two modifications. First we add periodic calls during request processing to our scheduler using a Java JNI call.Second, we add an online profiler that tracks the tail latency and the load, consisting or about 500 line of Java. Prior work shows Lucene shares many characteristics with the Bing Search engine [69]. We implement the offline threshold computation in a Python script and invoke MATLAB for PID parameter tuning.

We use a Monte Carlo finance server from prior work, which computes financial derivatives for path-dependent Asian options and is computationally intensive [23, 30, 72]. Banks and fund management companies use such derivatives every day to make immediate interactive trading decisions. Each request estimates an option price under various economic scenarios with different interest rates, strike prices, dividend yields, and volatility. Processing time varies widely based on request parameters, for example, sampling more for trades with higher volatility or total monetary value. Figure 5.4(b) depicts the service demand distribution of 4 K synthetically generated requests. The finance server is about 150 lines of C++ and we add about 400 line of C++ code to the system to implement our scheduler.



(a) Execution time histogram of 10 K Lucene queries



(b) Execution time histogram of 4 K finance requests

Figure 5.4: Workload demand distribution in 10 ms bins.

5.2.2 Hardware

We emulate heterogeneous processors on a homogeneous multicore processor with duty cycling threads for several reasons. First, heterogeneous processors are not available yet for desktop or server workloads, and mobile processors cannot execute large complex applications such as Lucene. Existing multicore simulators are not appealing because they require sampling to produce results in a timely manner and thus cannot capture complex long running workloads. Most importantly, no multicore simulator of which we are aware is precise enough to report tail latency.

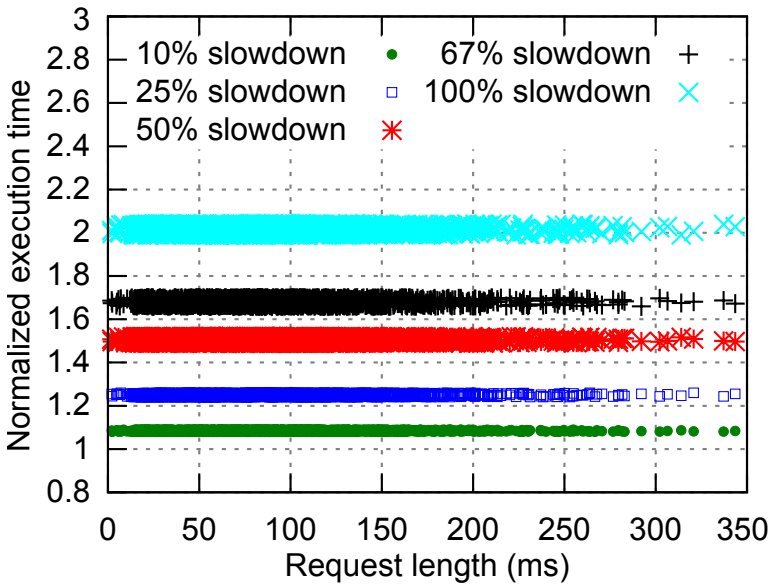


Figure 5.5: Normalized execution time of Lucene requests for different planned slowdown.

5.2.3 Heterogeneous Processor Emulation

We use a server with two 8-core Intel 64-bit Xeon processor (2.30 GHz) and 16 GB of memory running Linux kernel version 3.16. We turn off hyperthreading to avoid interference among threads sharing a physical core. We slow down cores by alternating between executing the request processing thread and a background thread. The background thread executes a CPU-bound computation and then sleeps. We control the slow down factor by a duty-cycle setting. For example, a background thread executing 50% of the time and sleeping 50% of the time consumes half of the time, and therefore

Core type	Core name	Relative Performance (s_i)	Peak power (p_i)
1	**Slow**	1	1
2	**Medium**	1.44	3
	Medium+	1.59	4
	Medium++	1.71	5
3	**Fast**	2	8

Table 5.3: Normalized performance and peak dynamic power consumption for emulated heterogeneous core types and configurations.

increasing the request processing time. We perform duty cycling in 100–500 microsecond periods. Since the shortest request processing time for both the workloads is few milliseconds, this period is short enough to affect all requests equally, and long enough to avoid excessive context switching. Figure 5.5 shows the measured and planned slowdown in processing time as a function of request length for several duty-cycling settings for Lucene requests. We observe that the emulator is quite accurate in increasing the execution time of requests, corresponding to the desired core speed. Although, the variance increases with higher slowdown settings, it is low compared to other sources of non-determinism in the system.

5.2.4 Performance Metrics

We report the tail and average latency, which includes queuing delay and execution time. We use Java *NanoTime* in Lucene and *clock_gettime* in the finance server. Since tail latency is an order statistic, we measure it over a moving window of the last 2000 requests and report the 99th percentile. Since the controllers take the tail latency as input, we set their control epoch to 2000 requests, which means that at high load, epochs are shorter than at low load. We report energy consumption based on performance measurements and the power model below.

5.2.5 Core Configurations and Power

We emulate relative core performance characteristics based on the ARM big.LITTLE cores [61]. ARM big (Cortex-A15) cores delivers roughly 2x performance over little (Cortex-A7) cores. We model core power consumption following the literature [88, 17, 39]. We assume static power is 20% of the total power of the chip. We model dynamic

power consumption as proportional to $\propto V^2 f$ when a core is running at f frequency with V voltage. Because performance and voltage are proportional to frequency, we model the dynamic power consumption of the fast core as cubic with respect to performance. Table 5.3 shows the performance and dynamic power consumption of the cores we consider normalized to slow cores. These ratios are better than current machines with DVFS [44] and are close to what todays different microarchitectures are delivering [61, 44]. Section 5.3.5 shows more potential improvements with higher ratios, but significant improvements even with more conservative ones.

Our default configuration has two core types: 1 fast (big) and 8 emulated slow (little) cores that executes the service and the client issues requests from one of the 7 spare fast cores. Section 5.1.4 shows more potential benefit with various heterogeneous three core configurations.

5.2.6 Scheduling Policies

We present results for the following scheduling policies.

Energy Efficiency with Target Tail Latency (EETL)

is Adaptive S2F configured with thresholds from the feedback controllers, for a specified target target latency. The thresholds determine the starting cores—a zero value for threshold t_0 starts the request on the slowest core and non-zero thresholds select some faster core.

Reducing Tail Latency (RTL)

is Adaptive S2F configured with the minimum zero threshold. Requests start executing on the fast available core. When a fast core c_i become available, the oldest job executing on $c_i - 1$ migrates to c_i. This configuration reduces tail and average latency.

Energy Efficient (EE)

is Adaptive S2F configured with the maximum infinite threshold. Each request starts on the slowest available core and only migrates to a faster core when the small cores

are insufficient to support the load and it is the oldest request. This configuration consumes the least amount of energy at the expense of high tail latency.

Oblivious

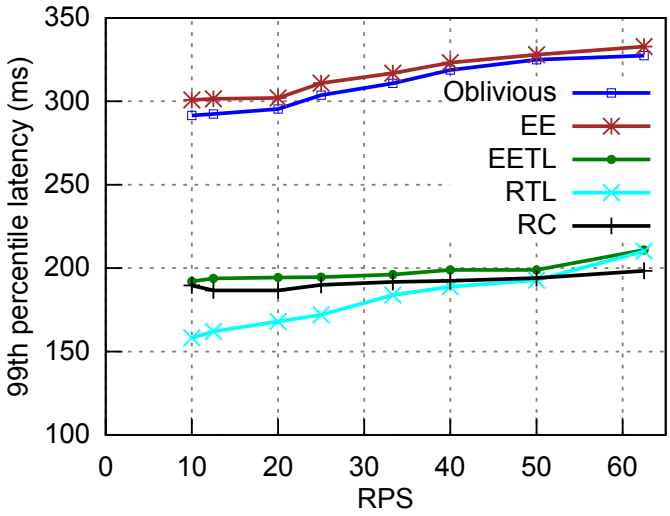
is a heterogeneity un-aware scheduler that randomly selects a core for each request and limits thread migration to facilitate cache affinity, following the current Linux defaults. Generally, the OS only performs load balancing at a context switch. Oblivious migrates a thread only when the thread is ready but the preferred core is busy. In the interactive server context, cores are usually not over subscribed, thus most threads do not migrate and complete on the core the OS initially assigns to them.

Request Clairvoyant (RC)

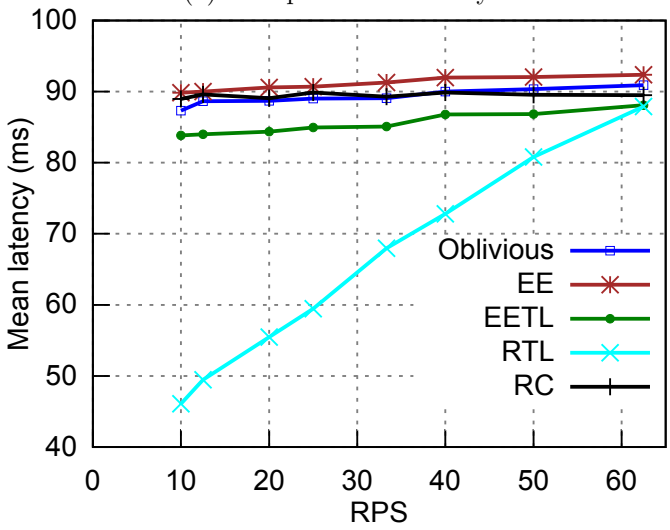
Request Clairvoyant (RC) has perfect knowledge of request demand. It executes the longest x requests on fast core such that the system satisfies the target tail latency. When a long request arrives and no fast core is available, it executes on a slow core until a fast core becomes available. All other requests run exclusively on slow cores. We experimentally select the best value of x for minimum energy at a target tail latency for every load.

5.3 Results

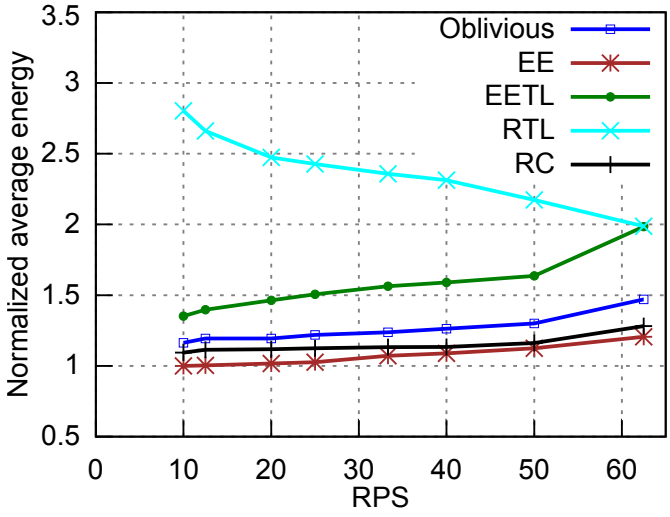
We first study tail latency, average latency, and energy results for the Lucene and Finance servers executing on the emulated AMP described in the previous section with eight slow and one fast core. We use a target tail latency at the 99th percentile of 200 ms for Lucene and 100 ms for Finance unless otherwise noted. These results show that EETL consistently delivers the tail latency target across a range of loads and targets, and offers substantial energy savings compared to RTL, the only other implementable policy that also delivers the tail latency target. We show how EETL trades off increases in latency (as allowed by the target) for energy savings, and that it dynamically handles load spikes without violating the tail latency target. We explore



(a) 99th percentile latency

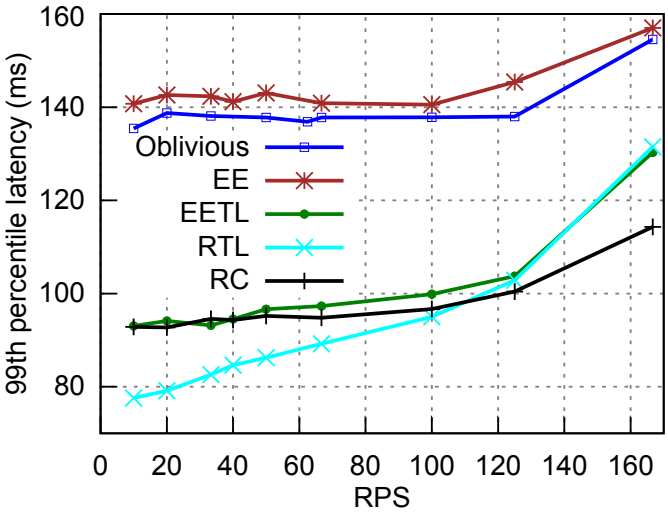


(b) Average latency

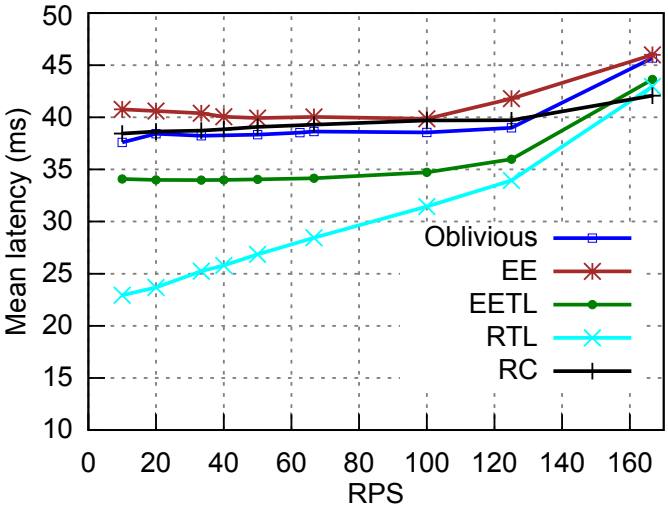


(c) Normalized energy consumption

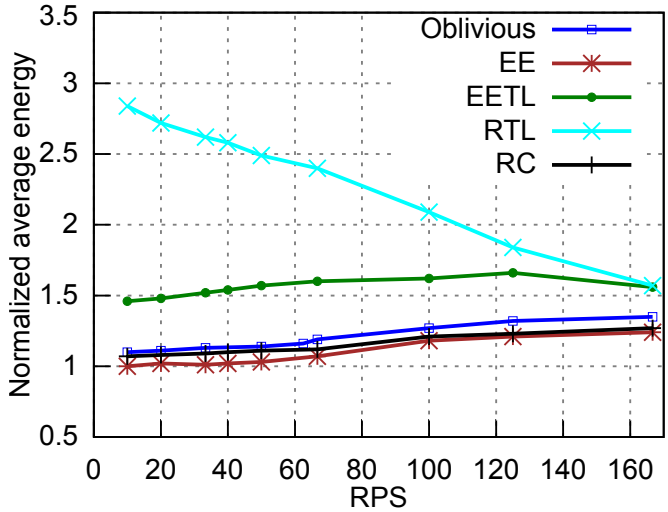
Figure 5.6: **Lucene** (target 200 ms): Tail latency, average latency, and normalized energy consumption on 8 Slow and 1 Fast. EETL delivers target latency with better energy efficiency. Reductions in average latency by RTL have high energy costs.



(a) 99th percentile latency



(b) Average latency



(c) Normalized energy consumption

Figure 5.7: **Finance** (target 100 ms): Tail latency, average latency, and normalized energy consumption on 8 Slow and 1 Fast. EETL delivers target latency with better energy efficiency. Reductions in average latency by RTL have high energy costs.

workload and hardware sensitivity, AMPs with three core types, and how to exploit our approach to choose appropriate heterogeneous hardware configurations.

5.3.1 Tail Latency and Energy Efficiency

Figures 5.6 and 5.7 plot the 99th percentile latency, average latency, and normalized average energy per request as functions of the load expressed as Requests Per Second (RPS) on the X-axis for each policy, executing Lucene and the finance server respectively. We report energy consumption per request normalized to EE at 10 RPS, the policy consuming the least energy, and use this value as the baseline for all subsequent experimental results. Because both workloads exhibit similar trends and tradeoffs, we focus on Lucene.

We observe that *leveraging heterogeneity can significantly reduce tail latency.* Both EETL and RTL use slow and fast cores effectively to meet the 200 ms target latency constraint. RTL reduces the 99th percentile latency by 46% at 10 RPS and 40% at 50 RPS compared to EE and Oblivious, which never deliver a 99th percentile latency below 291 ms. High load eventually overwhelms both EETL and RTL, but at low load RTL achieves tail latencies below the 200 ms target because it aggressively uses the fast core.

We next observe that *optimizing latency without considering energy consumption wastes a lot of energy.* Figures 5.6(c) shows that RTL consumes 2.8x more energy at 10 RPS and 1.9x at 60 RPS than EE, the lower bound on energy. RTL consumes 2.1x more energy than EETL at 10 RPS. All policies use more energy as the load increases. EETL however makes good use of this energy, since it meets the tail latency as the load increases. As load rises, EETL shortens the migration threshold until the threshold reaches 0, at which point it behaves exactly the same as RTL, with the two policies attaining the same tail latency and consuming the same amount of energy at 50 RPS (and beyond). Figures 5.6(b) and 5.7(b) show why RTL consumes more energy at low and moderate loads compared to other policies: RTL substantially reduces average latency, but surprisingly, reducing average latency has a relatively small influence on tail latency. Thus RTL wastes energy by using the fast core to process short requests

without benefiting tail latency.

The oracular RC policy meets the latency constraint with the most energy efficiency, but is not possible to implement. *A priori* knowledge of request demand improves energy efficiency while meeting the target at higher load than EETL since requests that run longer than EETL's threshold yet do not contribute to tail latency never migrate to the fast core. Even with a highly accurate predictor, which is hard to build and maintain, EETL can still reduce the impact of mis-predicted requests and requests that incur interference.

We also observe that *EETL effectively trades average latency for improvements in energy efficiency while meeting the target tail latency.* Because EETL uses thresholds, it waits to migrate long running requests to the fast core such that the request just meets the specific target. This policy increases average latency (compared to RTL), but reduces energy consumption for short requests, since most never execute on a fast core. Long requests are slightly more energy efficient as well because they execute on multiple core types. As already discussed, this energy conservation allows EETL to be much more energy efficient than RTL at low to moderate loads. EETL uses more energy than EE and Oblivious for all loads, but this consumption is necessary to meet the target tail latency — EE and Oblivious do not meet the target. These results demonstrate EETL's use of the fast core only as needed is extremely effective.

5.3.2 Effect of Varying the Load Dynamically

Figure 5.8 shows that EETL's feedback controllers dynamically adapt the migration threshold to load while meeting the target tail latency, even when load varies a lot. We subject EETL to extreme load variations between low (10 RPS) and high (50 RPS). Each point is the 99th percentile latency of the last 100 requests. EETL quickly adapts to large abrupt load changes. The conservative controller by design rarely violates the target, limiting violations by using slightly higher energy at constant and decreasing load.

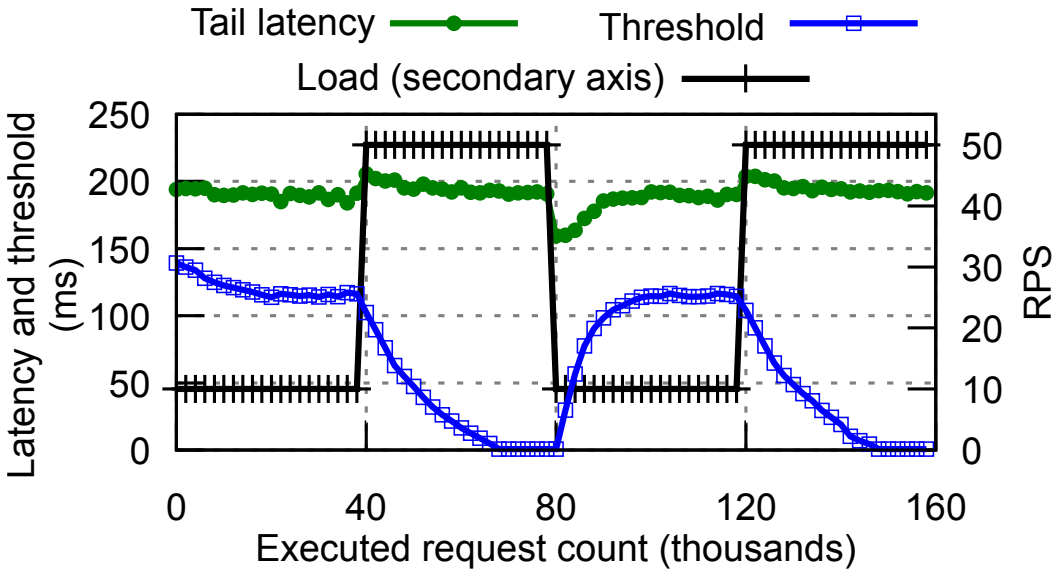


Figure 5.8: **Lucene** (target 200 ms): EETL adapts to widely varying load while delivering target tail latency.

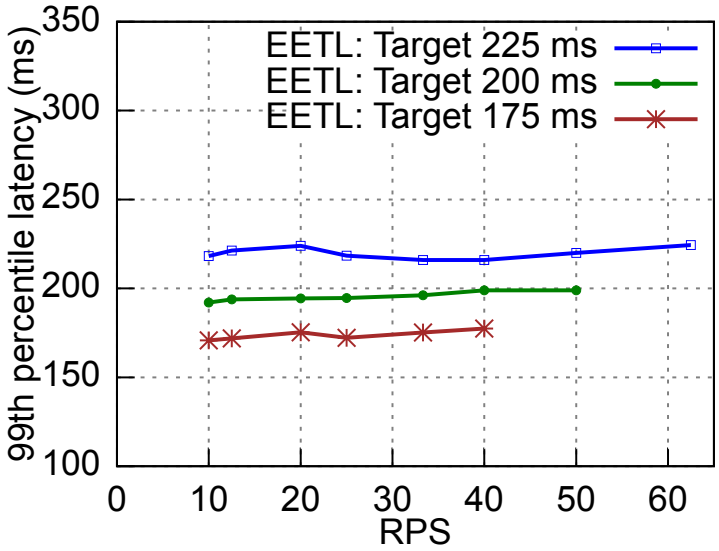
5.3.3 Algorithmic Sensitivities

We study the sensitivity of our approach to several parameters, and summarize the results for some experiments.

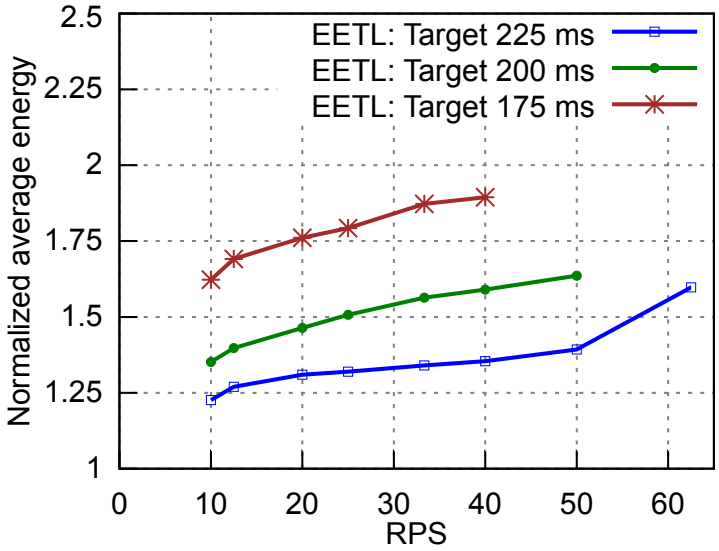
Effect of Varying Tail Latency Target

Figure 5.9 shows tail latency and normalized average energy consumed per request for different tail latency targets. These results show that EETL effectively adjusts its energy consumption based on the tail latency target. In general, EETL's performance is bounded by those of EE and RTL (Figures 5.6 and 5.7). Increasing the target tail latency above those achieved by EE does not lead to further energy savings since the use of the fast core is dictated by the load rather than by tail latency at that point. Tail latency targets below those achieved by RTL cannot be met by EETL, since RTL is EETL with the migration threshold set to 0. In particular, note that a specific core configuration can serve different maximum loads at different tail latency targets. In this case, Lucene supports up to 76, 56 and 40 RPS when the target is 225, 200, and 175 ms, respectively. Finally, as the target decreases, the difference in energy consumption becomes larger because increasingly aggressive targets require more use

of the fast core, which is less energy efficient.



(a) 99th percentile latency



(b) Normalized energy

Figure 5.9: **Lucene:** EETL delivers various target tail latencies and saves more energy for higher targets.

Effect of energy efficiency ratio of slow to fast cores

The above results are obtained using a power model where the slow core is 4x more energy efficient than the fast core (1/2 relative performance at 1/8 the power). To study the sensitivity of our approach to this model parameter, we have also compute the energy consumption when the slow core is 2x and 8x more energy efficient than the

fast core. Results show that even when slow cores are only 2x more energy efficient than the fast core, EETL saves up to 1.6x energy for a target tail latency of 200 ms. The savings increase as the slow cores become comparatively more energy efficient.

Effect of workload distribution

Previous work has shown that differentiating between short and long requests is more effective as the gap between the mean and the tail processing service demand (e.g., 99th percentile) increases. Results from the study of a modified Lucene workload with a mean of 30 ms and 99th percentile service demand of 180 ms are consistent with previous findings. That is, RTL reduces tail latencies even more compared to EE and Oblivious, and EETL conserves more energy compared to RTL because fewer short requests run on the fast core.

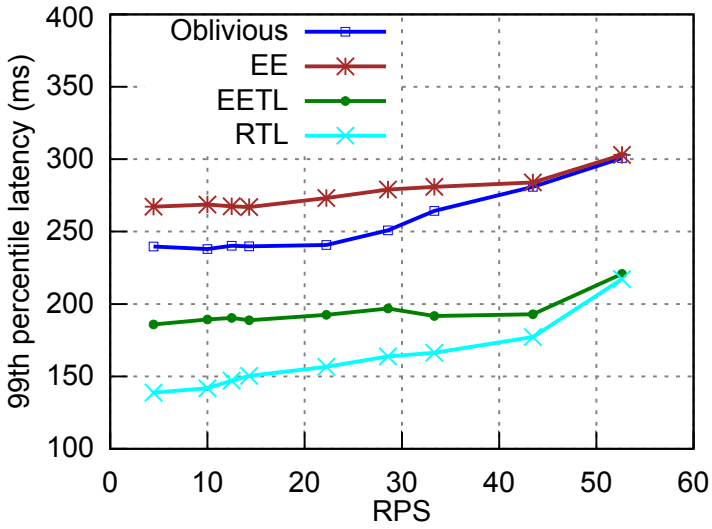
Effect of gain scheduling

The controller uses three load intervals for gain scheduling. We obtain similar results when using six load intervals. However, using only one load interval (i.e., without gain scheduling) results in significant overshoots with load increases at high loads (>40 RPS).

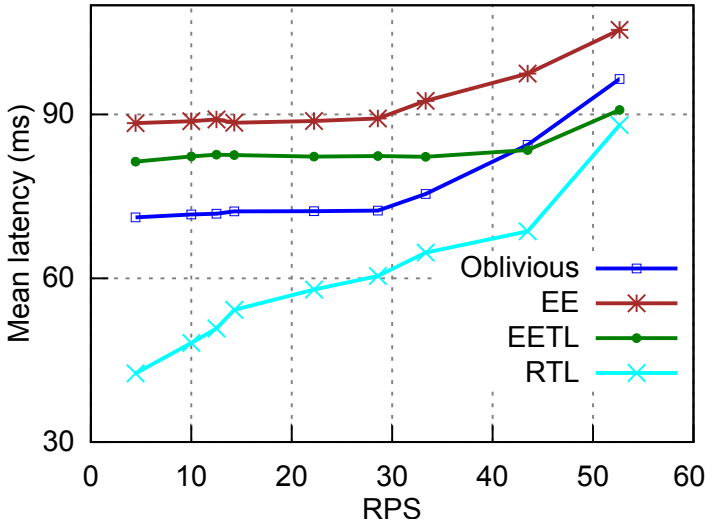
5.3.4 Heterogeneous Hardware with Three Core Types

This section explores AMPs with three core types. We choose medium cores such that their peak power is a multiple of the power of the slowest core (see Table 5.3). We use hardware configurations with a peak dynamic power budget equivalent to 14 slow cores, but with different numbers of slow, medium, and fast cores (see Table 5.4).

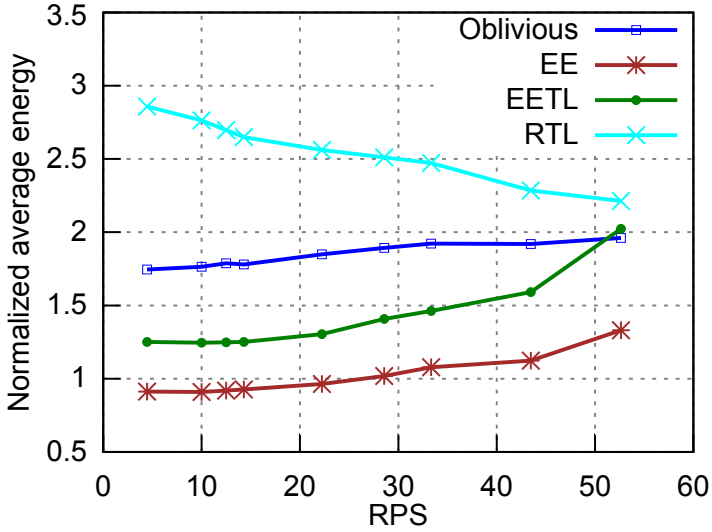
Figure 5.10 compares EETL, RTL, EE, and Oblivious running Lucene with a 200 ms 99 percentile tail latency target on the SlowMediumFast configuration. We first observe that *more core types deliver more energy savings at low to moderate loads.* We directly compare Figures 5.10(c) and 5.6(c) because energy is normalized to the same baseline (EE at 10 RPS on SlowFast). EE achieves lower energy consumption with three core types compared to two in Figure 5.6(c). As load increases, EE needs to use



(a) 99th percentile latency

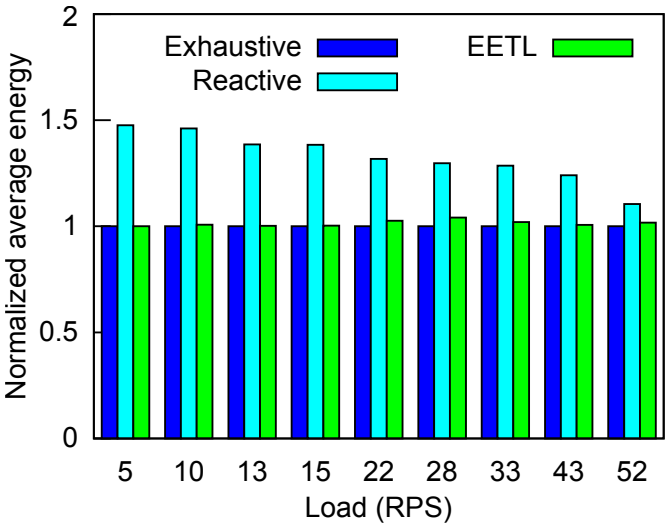


(b) Average latency

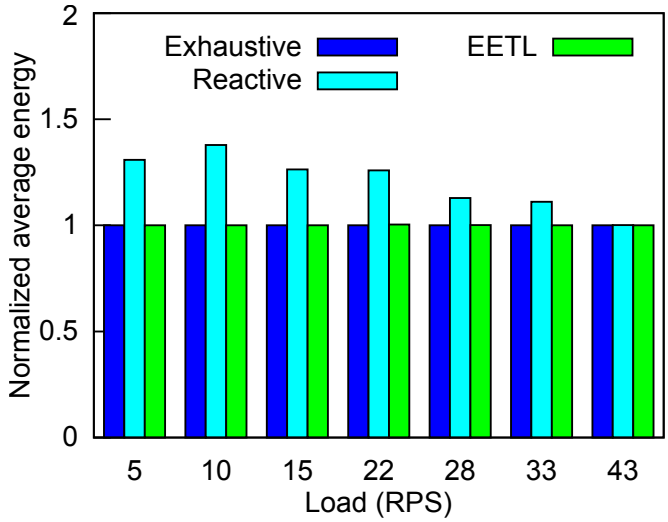


(c) Normalized average energy

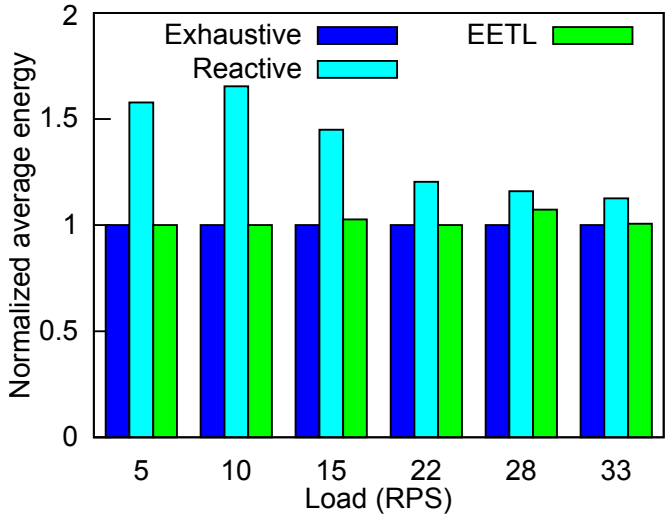
Figure 5.10: **Lucene** (target 200 ms): Tail and average Latency, and normalized energy on the SlowMediumFast configuration.



(a) SlowMediumFast



(b) SlowMedium+Fast



(c) SlowMedium++Fast

Figure 5.11: **Lucene** (target 200 ms): EETL, Exhaustive, and Reactive on three core types configurations.

	Number of cores				
Name	**Slow**	**Medium**	**Medium+**	**Medium++**	**Fast**
All Slow	16	0	0	0	0
SlowFast	8	0	0	0	1
SlowMediumFast	5	1	0	0	1
SlowMedium+Fast	4	0	1	0	1
SlowMedium++Fast	3	0	0	1	1
All Fast	0	0	0	0	2

Table 5.4: Configurations with same peak power using core types listed in Table 5.3.

both medium and fast cores; which leads to higher energy consumption because the medium core is less energy efficient than a slow core.

We next observe that *the differential in energy consumption between EETL and EE is smaller with more core types.* EETL uses the medium core effectively to further conserve energy while meeting the target tail latency. EETL successfully completes many requests that take longer than the slow-to-fast migration threshold on the two-speed configuration on the medium core, thereby reducing energy. Requests that eventually run on the fast core are also more energy efficient because they execute on all core types.

Comparing the EETL curves in Figures 5.10(c) and 5.6(c) shows EETL with three core types is more energy efficient than two core types for low and moderate load. At high load, more cores are important for throughput. EETL cannot meet the target starting at a lower load on three core types compared to two.

To assess the effectiveness of our controller design for N core types, we compare EETL to *Exhaustive* and *Reactive* policies. For Exhaustive, we run the system exhaustively with all possible combinations of values (in 5 ms increments) for the two migration thresholds (slow-to-medium and medium-to-fast), and select the one that satisfies the target tail latency and consumes the least energy at each load.

Reactive is a simpler adaptive slow-to-fast policy based on the intuition of avoiding using the fast and most energy inefficient core until absolutely necessary. It tracks the tail latency of the last 2000 requests (as in EETL). When the tail latency exceeds the target, Reactive decreases the threshold for migrating requests to the medium core. If the threshold for the slow-to-medium migration reaches 0, Reactive starts decreasing threshold for migrating to the fast core. Reactive does the reverse when the tail latency

is lower than target.

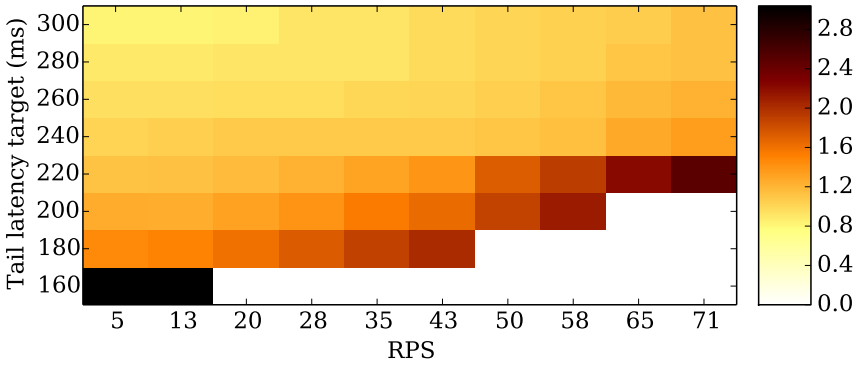
We use a simple heuristic to determine how much to adjust the threshold. If the difference between achieved tail latency and target tail latency is e and we are changing the slow-to-medium migration threshold, we change the threshold by $s_m e/(s_m - 1)$, where s_m is the speedup when moving from a slow to the medium core. For example, if the error is 1 ms and a medium core is 1.5x faster than a slow core, we decrease the threshold by 3 ms. The above heuristic converges quickly at low load, but typically needs to iterate to converge to the appropriate thresholds at high loads. (Reactive results in Figure 5.11 excludes energy during convergence.)

Figure 5.11 compares the energy consumption of EETL, Exhaustive, and Reactive on three configurations, each with a medium core of different speed, for loads when all three meet the target tail latency. These results show that there is little difference ($< 8\%$) between the energy consumption of EETL and Exhaustive. Although EETL changes all thresholds in fixed ratio, it computes a solution close to the optimal for each load. Next, Reactive uses a lot of energy comparatively at low to moderate loads because decreasing only the slow-to-medium threshold based on tail latency causes other short requests to migrate as well. This result validates the need to adjust both thresholds as load changes.

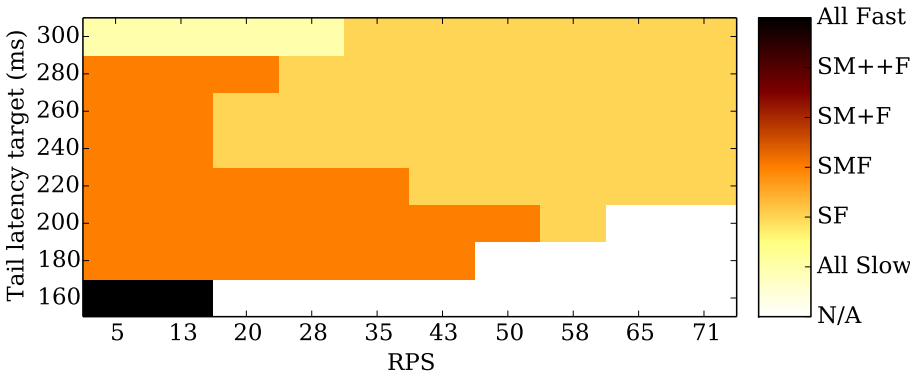
5.3.5 Choosing Core Types

This section examines the implications of interactive services on AMP core selection. We study power/performance tradeoff of the six configurations in Table 5.4, which have the same peak dynamic power. The heat map in Figure 5.12(a) depicts the minimum average energy achieved across the six configurations as a function of target tail latency and load using EETL for Lucene. A white cell means no configuration could deliver the target tail latency. EETL delivers energy proportionality — energy consumption increases non-linearly as load increases along the X-axis and as tail latency target becomes shorter (stricter) at lower Y-axis values.

Choosing good AMP configurations is challenging because it depends on target tail latency, load, and how the service will evolve. Figure 5.12(b) shows the best core



(a) Normalized minimum average energy across all configurations



(b) Best core configurations

Figure 5.12: **Lucene:** Exploring energy efficiency of AMP core configurations from Table 5.4 for.

configuration corresponding to the heat map in Figure 5.12(a). For very low load and strict target tail latencies (e.g., 160 ms), only one configuration (all fast cores) delivers the target. This server design is not attractive because it is too constrained and will miss tail latency targets often due to other random factors or if the service evolves by adding computation. At very high tail latency targets, all slow cores easily satisfy the target. At practical tail latency targets, configurations with small and medium cores perform best. Notice that three core types improves efficiency than two types. At low load, there is little or no contention, therefore, configurations with more core types will always match or improve over any result achieved by a configuration with fewer core types.

Even at medium load, configurations with three speeds are better. When target tail latency is high, slow and medium cores are sufficient, and when the target is low (strict), using both medium and fast cores becomes necessary. At high load, the slow-fast configuration performs best because it sustains higher load by having higher total number of cores. Notice that the two other medium cores configuration (SM+F and SM++F) are not in the figure. Since the figure is coarse grain, it elides some configurations where they are most energy efficient.

In summary, these results show how the workload, the target tail latency, and Adaptive S2F inform the choice of core types in heterogeneous hardware design.

5.4 Summary

This work shows that interactive services are well suited to the performance and energy efficiency potential of heterogeneous multicores. Our contributions include Adaptive S2F, a general approach to meeting performance and energy goals for interactive workloads. We show policies that effectively (i) optimize energy efficiency while meeting a target latency (EETL), (ii) optimize tail and average latency (RTL), and (iii) optimize energy (EE). These variants form a continuum — by dynamically adjusting to specific tail latency targets, EETL converges to EE as the target increases and converges to RTL as the target decreases. If EETL can meet the target tail latency using slow cores

at low load, it converges to EE, but dynamically adapts to use fast cores as load increases or other effects, such as network latency, threaten the target. EETL trades off increases in latency to improve energy efficiency quite significantly on practical AMP configurations while meeting tail latency targets that require fast cores.

Chapter 6

Conclusion and Future Work

In this dissertation, we discussed several techniques to build schedulers for managing tail latency and energy consumption within a single node of interactive services. We focused specifically on cases where there are high variability in request service time, such as a few long requests typically define the tail latencies. Then, we used the intuition that long requests reveal themselves since they stay in the system longer. Thus, our techniques either allocate more resources or resources that are more powerful (but less energy efficient) to the long requests in order to control tail latencies.

In Chapter 4, we introduced incremental parallelism to give more software threads to long requests and thereby, reduce execution time of long requests. This reduces tail latency since only long requests impact the tail latency. We information about the request processing demand distribution and speedup afforded by parallel execution to compute an interval table. The interval table is then used online to guide when and how much parallelism should be added as requests become older. Our evaluation showed that parallelism is a powerful technique to reduce execution time and judicious use of incremental parallelism can improve tail latency significantly without overloading resources.

In Chapter 5, we explored the tail latency-energy efficiency tradeoffs for asymmetric multiprocessor. We proposed a threshold based scheduler using control theory to migrate long requests. A request migrates to fast core as it age beyond the threshold. We designed a set of controllers and, online, these controllers can decide the best threshold based on system load, tail latency target and measured tail latency performance. We showed that this scheduling technique can successfully control the tradeoff between latency and energy consumption.

Overall, the dissertation has demonstrated the feasibility and importance of managing tail latency in interactive services by intelligently allocating more resources or more powerful resources to long requests. This works can be extended in a number of directions.

Simultaneous consideration of parallelism and heterogeneity: It is possible to combine dynamic parallelism (i.e., increasing or decreasing software threads) and exploiting processor heterogeneity. For example, when there are free slow cores, a long requests can add more software threads to run on slow cores as oppose to migrate to a fast core. The actual decision will depend on system load, parallelism speedup, fast core power and performance characteristics. It is also interesting to design effective processors with suitable number and types of the cores to be effective for most of the workloads.

Consideration of simultaneous multithreading (SMT) along with parallelism: In modern servers, requests can share functional units through SMT. This allows interesting tradeoff between using full core vs. running two threads using SMT. This would also require consideration of possible interference and workload dependent SMT behavior.

Co-location with batch applications: Rather than using the full server for a single interactive service workload, a provider can use the same server to run both interactive and batch workloads. Any slack in interactive service performance can translate to more resources for batch workload. A sophisticated policy would consider the performance impact on interactive services, utility of giving more resources to batch workloads and possible interference for scheduling the workloads.

Interactive service requests with phases in lifetime: Our current work assumes same parallelism speedup on fast core for the entire lifetime of a request. In some interactive services, the requests may have phases and require different temporal policy. Considering different policies for different phases will improve the opportunity and efficiency of the scheduler.

Appendix A

Providing Green SLAs in High Performance Computing Clouds

A.1 Introduction

It is well known that datacenters consume an enormous amount of electricity. This consumption translates into high carbon emissions, since most of the electricity is produced using fossil fuels. A 2008 study estimated world-wide datacenters to emit 116 million metric tons of carbon, slightly more than the entire country of Nigeria [114]. With increasing societal awareness of these emissions and climate change, there is increasing demand for cleaner products and services.

In response, enterprises have started to set explicit sustainability goals and create initiatives to reduce their carbon emissions. Such efforts have become important marketing tools. For example, in 2012, the Global Reporting Initiative [52], a non-profit organization that seeks "to make sustainability reporting standard practice for all organizations," registered 2,295 reports from companies such as AMD, Dell, and Microsoft. As enterprises and individuals shift their workloads to the cloud, this drive toward quantification and disclosure of the sustainability of business activities will lead to demand for *quantifiable green cloud services*.

A few small green cloud service providers, e.g., Green House Data [59], AISO [1], and GreenQloud [62], have already sprung up to meet the demand for clean computing. However, it is difficult to quantify how green such providers are. For example, while two of Green House Data's datacenters are 100% powered by renewable (wind) energy, a third (equipped with solar power) is not [60]. Furthermore, cloud clients may have different goals for their carbon emissions (e.g., 100% vs. 30% green energy). In fact,

since not all clients would need a 100% green service, creating fully green systems unnecessarily increases costs.

Thus, instead of a one-size-fits-all approach, we argue that Infrastructure-as-a-Service (IaaS) cloud providers should offer a new class of services, with explicit service-level agreements (SLAs) for the percentage of renewable energy used to run the clients' workloads. We refer to these renewable-energy SLAs as Green SLAs. Importantly, this class of services should not replace existing services (which are oblivious to energy sources). Rather, it would allow environmentally conscious clients to explicitly contract for use of green energy for their workloads. For example, a client interested in near-zero carbon emissions would contract for virtual machines (VMs) in its workload to run 100% on green energy. Others could contract for lower percentages of green energy. An enterprise could even contract for different SLAs over time, as their business goals and progress toward meeting those goals evolve.

Cloud providers can offer the above differentiated Green SLA service in different manners. For example, one approach is to account for the entire amount of green and brown (i.e., electrical-grid-sourced) energy consumed within an accounting period (e.g., a month) irrespective of which workload consumed how much of each type of energy. In this accounting approach, it suffices for the provider to show that enough green energy was used to meet all Green SLAs. Although simple, this approach would not suffice for some clients. Specifically, we expect that an increasing number of clients will require guarantees that their workloads are executed with specific fractions of green energy, especially if they are required to demonstrate so by legislation. For example, the UK government requires businesses consuming more than 6 GWh of brown energy per year to purchase carbon offsets from the market [141].

In this chapter, we propose and evaluate a stricter paradigm for High Performance Computing (HPC) cloud providers, where a client's Green SLA can only be satisfied by actually using green energy to run that client's job (i.e., a collection of VMs). Specifically, we assume that each job submitted by a client specifies a desired Green SLA, in the form of a minimum percentage of green energy that must be used to run

the job. The provider can accept or reject the job.[1] If it accepts a job, the provider earns a premium for the percentage of the client's job that must be run using green energy. However, the provider must pay a penalty if it violates the Green SLA. Meeting Green SLAs in the presence of intermittent sources of green energy, such as solar and wind, is a challenging proposition.

As the cloud provider needs to differentiate green energy from brown energy to satisfy Green SLAs, we assume that it operates a datacenter that either generates its own green energy (self-generation) or draws it directly from an existing nearby plant (co-location). In either scenario, the datacenter can also draw on brown energy as needed. (Importantly, note that we do *not* argue that self-generation or co-location will be the best approach for all sustainability-conscious datacenter operators. Rather, we argue that self-generation or co-location will be the approach of choice for many operators, as suggested by the many examples in [11, 120, 59, 1, 62, 42].)

However, having two distinct sources of energy (green and brown) is not enough to provide Green SLAs. The provider needs to bring the green energy all the way to the servers in the portion of the datacenter reserved to provide the Green SLA service. Thus, we propose a new power delivery infrastructure in which each rack reserved for the Green SLA service is dynamically switched between two energy sources, one entirely green, and one possibly a mixture of brown and green. The switching of the racks between the two energy sources is controlled by software running on a control module.

Next, we propose a software framework for optimization-based scheduling of client jobs and energy sources for the racks. We design two optimization-based scheduling policies using this framework. The policies seek to maximize the profit that the cloud provider can accrue by admitting and running clients' jobs. They predict the amount of green energy that will likely be produced in the future, and use these predictions together with jobs' execution information and Green SLAs to decide whether to admit jobs. They also generate schedules for executing the admitted jobs and for controlling

[1]The client can easily resubmit a rejected job with a lower Green SLA. In particular, a job with 0% Green SLA would always be accepted if there is sufficient computing capacity. The provider can also negotiate a Green SLA to reflect its commitments to other clients and the expected availability of green energy. We leave this as future work.

the energy source of each rack. We also use two greedy heuristic scheduling policies as baselines for comparison.

Finally, we evaluate our proposed infrastructure and scheduling policies using simulation. Our main results show the tradeoffs between our optimization-based scheduling policies, and their advantages over the simpler greedy heuristic policies. Importantly, policy parameters for the optimization-based policies can be adjusted to reflect different values placed on meeting Green SLAs. For example, the penalty can be set very high (even if the actual penalty the provider has to pay out to clients is lower) to force the policies to be conservative and avoid missing Green SLAs. Alternatively, the penalty can be set low to reflect a best effort environment, where Green SLA violations are not serious failures. Based on these results, we conclude that a Green SLA service that uses our policies would enable the provider to attract environmentally conscious clients, especially those who require strict guarantees on their use of green energy.

In summary, this chapter makes the following contributions: (1) we propose a new HPC cloud service that provides explicit SLAs for the use of green energy, (2) we propose a hardware infrastructure to support this new service; (3) we propose an optimization-based framework for admission control and scheduling to maximize profit while respecting green SLAs; (4) we use the framework to design two optimization-based policies; and (5) we evaluate our framework and policies extensively through simulation.

A.2 Related Work

In [91], Klingert *et al.* introduced the notion of Green SLAs. However, their work focused on identifying known hardware and software techniques for reducing energy consumption and integrating green energy, and how applications might specify preferences/requirements for these techniques. They did not propose a specific type of Green SLAs and did not explore approaches for satisfying Green SLAs.

In [38], Deng *et al.* explored the strategic placement of grid ties to concentrate green energy in areas of the datacenter used to support workloads of environmentally conscious clients. In [104], Li *et al.* proposed a static partitioning of a datacenter

into separate green and brown parts, and migration of VMs between the two parts to maximize use of green energy while minimizing performance overheads. Again, these works did not propose Green SLAs, and did not explore approaches for satisfying Green SLAs. We could have adopted a static partitioning as in [104]. However, we decided to trade off some additional hardware (i.e., the per-rack transfer switches) for increased flexibility (e.g., the possibility of achieving $x\%$ green energy consumption without requiring migration).

Many recent papers have focused on datacenters that exploit green energy [3, 56, 57, 55, 96, 95, 99, 108, 135, 137]. Of these, [56, 57, 96] studied the scheduling of deferrable batch jobs to maximize the use of renewable energy. Krioukov *et al.* proposed to adjust service quality in non-deferrable interactive workloads [95]. For datacenters that run a mix of interactive and batch workloads, [3] and [108] proposed to adapt the amount of batch processing dynamically. Goiri *et al.* considered both deferrable and non-deferrable workloads [55]. In general, these works have sought to maximize the use of green energy while maintaining performance bounds. Our work shares this goal. However, we also seek to precisely assign green energy to client jobs according to Green SLAs. We propose novel hardware and software to accomplish this assignment.

A number of prior works have also addressed VM placement to reduce energy cost and/or performance SLA violations [20, 22, 54, 101, 147]. These works either proposed migrating VMs across datacenters or carefully packing VMs within a datacenter to achieve lower energy consumption with bounded performance loss. None of these works considered green energy.

A.3 Power Distribution Infrastructure

Figure A.1 shows the power distribution and control infrastructure we propose for a module that would be used to support the Green SLA service. One or more of these modules can be housed in a larger datacenter that also houses infrastructure for providing regular services (i.e., services to clients not interested in Green SLAs). Current IaaS providers already offer many classes of services, such as the Cluster Compute and Cluster GPU service classes of Amazon EC2.

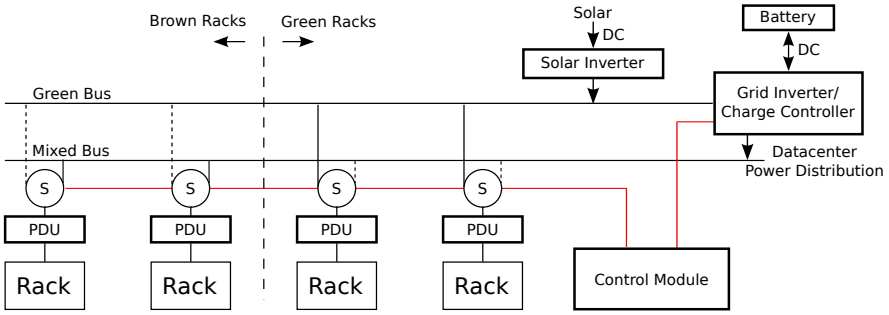


Figure A.1: Power distribution infrastructure. The vertical dashed line shows an example partitioning of the racks into a brown part (where racks are switched to the Mixed Bus) and a green part (where racks are switched to the Green bus) by a scheduling policy. This scheduling is discussed in Section A.4.

The infrastructure contains two separate power buses, with the Green Bus carrying strictly green power and the Mixed Bus carrying a mix of brown and green power. Each rack is connected to a software-controlled transfer switch, which switches the power source for the rack between the Green and Mixed buses. The Grid Inverter/Charge Controller (GICC) is configured to charge the battery when there is excess green power, and discharge it when there is insufficient green power. Once the battery is full, excess green power is routed to the Mixed Bus. The GICC is configured to never allow power flow from the Mixed bus to the Green bus or to the battery.

The control module executes software that configures the GICC and the transfer switches. We detail this software in Section A.4. The overall idea, however, is to use the solar power source, battery, and Green Bus to deliver *pure* green power to racks that can then be used to satisfy Green SLAs. This is why brown power is never allowed to enter the Green Bus and the battery. While excess green power may be routed to the Mixed Bus (so that it is not wasted), this green energy will not be counted toward any Green SLA.

Green energy is produced by a solar plant, either located on-site or at a nearby location. The solar source is supplemented by a small battery to smooth out any short-term variability in the solar power production (e.g., less energy is produced in a 15-minute interval than was predicted) as well as variability in power consumption. (Longer term matching of green energy and workload is done by the control software.)

In the proposed infrastructure, the solar inverter and GICC are both required for integrating a local green power source into the datacenter, independent of our proposal of a Green SLA service. Thus, the hardware cost of providing the Green SLA service is the extra power bus, the per-rack transfer switches, and the small battery. We comment further on the size of the battery in Section **??**.

This design is based on our experience building an actual solar-powered micro-datacenter [55]; we have direct experience with all components, except for the software-controlled transfer switches. However, similar switches are available commercially and are commonly used in current datacenters for redundant power delivery [123, 117]. In addition, while this chapter considers solar as the source of green energy, our framework can be easily applied to other sources of green energy, such as wind, as long as it is possible to predict the near-future production of the green energy.

A.4 Scheduling Framework

Given the above power distribution and control framework, our goal is to schedule client jobs, and brown and green energy consumption to maximize the cloud provider's profit while meeting the clients' Green SLAs. In this section, we first describe the proposed Green SLA service in more detail. We then overview the scheduler, formulate scheduling as an optimization problem, and propose two approaches for solving the optimization problem (leading to two different scheduling policies). We also describe two heuristic-based scheduling policies as baselines for comparison with the optimization-based policies. Finally, we describe our approach for predicting the future production of green energy, which is a needed input to three of the four scheduling policies.

A.4.1 Green SLA Service

Each arriving job includes information about the number of VMs in the job, its runtime (i.e., the contracted service time for running the job), and a Green SLA specifying the required percentage of green energy. A job with a Green SLA of $x\%$ means that at least $x\%$ of the energy used to run the job must come from the Green Bus.

The provider charges clients a rate for each resource-time unit that can be run using either type of energy (e.g., $0.24 per machine-hour). The provider charges a higher rate for each resource-time unit that should be run using only green energy (e.g., $0.29 per machine-hour). The higher rate reflects the added value that the provider places on the differentiated service provided by Green SLAs.

The provider can reject a job at arrival if it does not have sufficient processing capacity and/or expected green energy production to execute the job and meet its Green SLA. However, if the job is admitted and the provider fails to meet the job's Green SLA, the provider must pay a penalty proportional to how much of the Green SLA was missed (e.g., $0.50 per machine-hour of Green SLA not fully powered by green energy).

A.4.2 Scheduling Overview

The provider runs a scheduler that is designed to maximize profit while meeting jobs' Green SLAs, where the cost to the provider includes the cost for brown energy and any penalty that must be paid for violations of Green SLAs. Specifically, the scheduler divides time in a scheduling window (e.g., the next 48 hours) into epochs, with an epoch of time on a machine called a slot. It then predicts the green energy production, and produces a schedule that specifies for each time epoch in the scheduling window: (1) the number of VMs of each active job j that should be running on each rack r, and (2) the power bus to which each rack should be connected. A slot in a rack connected to the Green Bus is called a green slot; one connected to the Mixed Bus is called a brown slot.

While the infrastructure described in Section A.3 allows for arbitrary configurations of the power switches, the scheduler uses a more constrained configuration to simplify the scheduling problem. Specifically, the racks are divided into "left" and "right" partitions for each epoch; the vertical dashed line in Figure A.1 shows an example partitioning. The racks in the right partition will be completely powered with green energy during the epoch, while the racks in the left partition will be powered mostly with brown energy. A small amount of green power may be routed to the Mixed bus, but

only if this excess green energy is insufficient to allow another entire rack to be moved to the green (right) partition. This excess green energy is used to reduce the amount of brown energy needed, and so reduces cost, but cannot be used toward meeting Green SLAs. In this approach, the leftmost rack will be the brownest rack, i.e., it has the highest brown-to-green energy ratio over time, the rightmost rack will be the greenest rack, and the racks become "greener" from left to right.

The scheduler runs toward the end of an epoch, leaving enough time for turning servers on for the next epoch if necessary, when a new job has arrived in the current epoch and/or when the system detects that Green SLAs may be violated because green energy production is lower than previously predicted. The scheduler assumes that each VM will consume the maximum amount of energy in each slot, and meets the Green SLA of a job by scheduling its VMs on a sufficient number of green slots. If VMs running on green racks consume less than the allocated energy, the excess green energy will be automatically routed to the Mixed bus.

To minimize energy consumption and cost, unneeded servers are either turned off or put into a low-power state (e.g., S3 ACPI). For simplicity, we assume that data is stored on network-attached storage so that turning servers off does not affect data availability. We have solved this availability problem without requiring this assumption in a similar context [57].

We assume that VMs can be migrated live from one physical machine to another (i.e., a VM keeps running while it is being migrated) with no slowdown. During migration, energy is consumed on both the source and destination machines.

We handle failures in the same manner as Amazon's EC2 [7]. Specifically, if a VM is lost due to a crash or a software or hardware failure on the hosting server, the job is counted as having 1 fewer VM for its remaining runtime, and the fee for its execution and the Green SLAs are adjusted accordingly.

A.4.3 Optimization Framework

Table A.1 lists the set of parameters for our optimization framework. Using these parameters, we formulate the optimization problem shown in Figure A.2.

Table A.1: Framework parameters. Time epochs in the scheduling window are numbered from 1 to T. Racks are numbered from 1 to RA, where rack i is adjacent to the left of rack $i+1$ (left-to-right numbering of racks shown in Figure A.1).

Symbol	**Meaning**
T	The number of time epochs in the scheduling window
J	Set of active jobs
V_j	Number of VMs in job j
G_j	Number of (remaining) green slots required for job j
T_j	Number of (remaining) time epochs in job j's execution
R_b	Revenue earned per contracted execution of a VM in a brown slot
R_g	Revenue earned per contracted execution of a VM in a green slot
L	Penalty per slot of Green SLA not met
RA	Number of racks used to implement the Green SLA service
RC	Capacity of (i.e., number of machines in) each rack
EV	Amount of energy consumed by a machine running a VM in an epoch
EM	Amount of energy consumed by the migration of a VM
PB	Energy price that the cloud provider pays per brown slot
OP_r	Opportunity cost for using a slot in rack r
GP_t	Predicted green energy production in epoch t
e^b_{rt}	Amount of brown energy scheduled for rack r during epoch t
e^g_{rt}	Amount of green energy scheduled for rack r during epoch t
x_{jrt}	Number of VMs of job j scheduled on rack r in epoch t

$$Profit = \sum_{j \in J}(Revenue_j - Penalty_j - OppCost_j) - \sum_{t=1}^{T}\sum_{r=1}^{RA} Cost(r,t) \quad (A.1)$$

$$Revenue_j = G_j \times R_g + ((T_j \times V_j) - G_j) \times R_b \quad (A.2)$$

$$Penalty_j = \begin{cases} L \times (G_j - NumActGreenSlots_j) & \text{if } G_j > NumActGreenSlots_j \\ 0 & \text{otherwise} \end{cases} \quad (A.3)$$

$$NumActGreenSlots = \sum_{r=1}^{RA}\sum_{t=1}^{T}(x_{jrt} \times RackIsGreen_{rt}) \quad (A.4)$$

$$RackIsGreen_{rt} = \begin{cases} 1 & \text{if } e^g_{rt} \geq \sum_{j \in J}(x_{jrt} \times EV + NumMigrates_{jrt} \times EM) \\ 0 & \text{otherwise} \end{cases} \quad (A.5)$$

$$NumMigrates_{jrt} = \begin{cases} x_{jrt} - x_{jr(t-1)} & \text{if } x_{jrt} > x_{jr(t-1)} \\ 0 & \text{otherwise} \end{cases} \quad (A.6)$$

$$OppCost_j = \sum_{r=1}^{RA}\sum_{t=0}^{T}(x_{jrt} \times OP_r) \quad (A.7)$$

$$Cost(r,t) = e^b_{rt} \times PB \quad (A.8)$$

Figure A.2: Optimization framework.

In this formulation, the profit (Equation A.1) that the cloud provider earns is the sum of revenues earned from running admitted jobs minus the penalty incurred for missing Green SLAs and the energy cost of running the admitted jobs. An opportunity cost ($OppCost_j$), representing expected future earning that is given up when slots are assigned to current jobs, is also subtracted. We introduce this cost to ensure that VMs are not scheduled on "greener" racks than needed to satisfy their Green SLAs. Otherwise, the system may need to migrate VMs (incurring migration costs) or reject new jobs (loosing opportunities to increase profit) when they arrive in the future. The profit formulation does not include the capital and operating costs of the solar setup because all scheduling policies we seek to compare embody these same costs.

The revenue earned per job (Equation A.2) is the revenue earned for the promised number of green slots and for the number of brown slots. The penalty (Equation A.3) is proportional to the number of promised green slots replaced by brown slots because the cloud provider did not produce enough green energy while the job was running. The opportunity cost of each job (Equation A.7) is the sum of the opportunity cost of all slots assigned to the job during its run-time. The cost to the provider of running the admitted jobs includes the cost of supplying racks with brown energy (Equation A.8).

The optimization problem involves many constraints, with the main ones listed and described briefly in Figure A.3.

We use an optimization solver to instantiate, for each time epoch t in the scheduling horizon, the number of VMs from each job j to run on each rack r (x_{jrt}), the amount of brown energy supplied to each rack (e^b_{rt}), and the amount of green energy supplied to each rack (e^g_{rt}). Each time the problem is solved, T_j and G_j of each job that has already been running have to be updated to reflect how long the job has already run, and how many green slots it has already been allocated.

Note that our framework does not include the explicit management of the battery shown in Figure A.1. Although we could have included this management [55], we have chosen to exclude it for simplicity. We believe this omission has little impact (if any) on our results since we propose only a small power "smoothing" battery. We also assume that each physical server only hosts one VM; if a server can host multiple VMs, an

$$\forall_{j\in J}\forall_{0<t\leq T_j}\sum_{r=1}^{RA} x_{jrt} = V_j \Rightarrow \text{Must run all VMs of each job } j \text{ in every epoch during } j\text{'s run-time} \tag{A.9}$$

$$\forall_{0<r\leq RA}\forall_{0<t\leq T}\sum_{j\in J} x_{jrt} \leq RC \Rightarrow \text{Total number of VMs running in a rack must not exceed its capacity} \tag{A.10}$$

$$\forall_{0<r\leq RA}\forall_{0<t\leq T}(((EV \times \sum_{j\in J} x_{jrt}) + (EM \times \sum_{j\in J} NumMigrates_{jrt})) \leq (e^b_{rt} + e^g_{rt})) \Rightarrow \tag{A.11}$$

Enough energy must be scheduled for each rack to run VMs scheduled there and migrate VMs moving to another rack

$$\forall_{0<t\leq T}\ \nexists_r|(RackIsGreen_{(r-1)t} = 1) \wedge (RackIsGreen_{rt} = 0) \wedge (RackIsGreen_{(r+1)t} = 1) \Rightarrow \tag{A.12}$$

All green racks must be next to each other

$$\forall_{0<t\leq T}\ \nexists_r|(r \neq RA) \wedge (RackIsGreen_{rt} = 1) \wedge (RackIsGreen_{RAt} = 0) \Rightarrow \text{Green racks must start from right end}$$

$$\forall_{0<r\leq RA}\forall_{0<t\leq T}\sum_{r=1}^{RA} e^g_{rt} \leq GP_t \Rightarrow \text{Can schedule no more green energy than the amount of expected production} \tag{A.13}$$

Figure A.3: Optimization constraints.

additional per-rack optimizer as in [20] can be added to maximize consolidation in the presence of inter-rack VM migration. In this case, it may be necessary reserve a small amount of green energy for intra-rack migration in the green partition. We do not account for the power consumed by turning nodes on/off because these operations take little time compared to the scheduling epochs; e.g., we have measured 45 and 15 seconds to turn a server on and off, respectively, and 4 and 3 seconds to enter and exit an S3 ACPI state that consumes 5% of the peak power, compared to the 15-minute epochs used in our evaluation. We have also chosen simplicity (e.g., constant brown energy pricing and constant penalty per missed slot) whenever the added complexity would be unlikely to affect our fundamental findings. Finally, the framework currently focuses solely on the servers, assuming that cooling and network-attached storage fully rely on green energy and (larger) batteries. We will eliminate these simplifying assumptions in future work.

A.4.4 Solving the Optimization Problem

Simulated Annealing. We use Simulated Annealing (SA) [90] to solve the above non-linear formulation for maximum profit. Recall that this optimization is executed each time a new job arrives, or the system detects that one or more Green SLAs might be missed.

When a new job arrives, SA first solves a simpler, linear reformulation of the problem (see below) to schedule the new job, assuming that the current schedule for jobs already in the system cannot be changed. This produces a starting point for SA. When the system detects that Green SLAs might be violated, the current schedule is the starting point.

SA then iteratively explores new schedules, relying on randomization to avoid local maxima. To limit the size of the search space, SA does not explore completely random new schedules. Instead, it maintains three job sets: (1) an "unmet" set of jobs whose Green SLAs are expected to be violated under the current schedule, (2) an "extra" set of jobs that are expected to receive more green energy than is needed to satisfy their Green SLAs, and (3) a "migration" set of jobs whose schedules contain more than a threshold of migrations in the future. SA produces a new schedule by randomly choosing a small subset of each job set, removing them from the current schedule, and rescheduling them as if they were new arrivals using the linear reformulation.

For a new job, if SA cannot find a schedule that increases profit, the job is rejected. Otherwise, the best schedule found by SA is adopted. In the case of detecting possible Green SLA violations, the best schedule found by SA is adopted, regardless of whether one or more Green SLAs are still expected to be missed or not. Once admitted, jobs must be executed to completion. If Green SLAs are missed, the cloud provider must pay the penalty.

Linear Programming. We can reformulate the optimization problem in Figure A.2 into a simpler one that is solvable using Linear Programming (LP). In this reformulation, the optimization problem becomes one of minimizing the sum of the penalty, opportunity cost, and migration cost. We assume that the racks in the green partition

are always completely full, allowing the green/brown energy routing to be predetermined into the future. Intuitively, the minimization problem leads to a solution that tries to meet Green SLAs (minimizes the penalty) without using greener slots than needed (minimizes the opportunity cost) and without unnecessarily moving VMs (minimizes the migration cost).

The reformulated problem is a Mixed Integer Linear Programming (MILP) problem that we solve using a MILP solver. Since MILP solvers are only efficient for reasonably sized problems, LP can only be used to schedule a small number of jobs. Thus, when using just LP, we assume that once a job has been scheduled, its schedule cannot be changed in the future. When a new job arrives, it is scheduled using only resources (i.e., slots) that are not already used in the current schedule. It is rejected if the solution leads to lower profits. As already described, when used as a part of SA, LP may be used to produce a schedule for a small set of jobs, assuming the current schedule for the remaining jobs is fixed.

Greedy Heuristics. As baselines for comparison, we propose two greedy heuristic placement schemes. The first is a First-Fit (FF) scheme that is oblivious to energy type and admits all jobs as long as there is sufficient computing capacity. The second is a more sophisticated, green-energy-aware heuristic that we call Static Green-aware Placement (SGP). Specifically, when a job arrives with a Green SLA of $g\%$, SGP places the job's VMs on the rack r with expected green percentage during the job's runtime closest to but not less than $g\%$. If there are more VMs than available slots in r, the needed percentage of green is recomputed for the excess VMs, and the same placement procedure is repeated. VMs are never migrated. The job is rejected if a placement that is expected to meet or exceed the job's Green SLA cannot be found.

A.4.5 Green Energy Prediction

We use the method from [57] to predict solar energy production. This method combines the model from [136] with the approach for improving accuracy from [56]. Specifically, the model relates solar energy generation to cloud cover as $E_p(t) = B(t)(1 - CloudCover)$, where $E_p(t)$ is the amount of energy predicted for time t, $B(t)$ is the

amount of energy expected under ideal sunny conditions for time t, and $CloudCover$ is the forecasted percentage cloud cover. For the cloud cover information, we use forecasts from Intellicast.com, which predicts $CloudCover$ for each hour of the next 48 hours. This leads to prediction granularity (i.e., t) of one hour. We set $B(t)$ for each hour of the day to the amount of energy generated during that hour on the day with the highest energy generation from the previous month.

Of course, weather forecasts are sometimes wrong, which may lead to inaccurate predictions. To improve accuracy, we compute $CloudCover$ from the amount of energy generated in the previous hour. This approach then compares the accuracy of the two methods, and uses the most accurate one to predict the remainder of the horizon. For example, at the beginning of hour t, we compute $CloudCover$ for the next hour using (1) the weather forecast, and (2) the energy produced in hour $t-1$. At the beginning of hour $t+1$, we compare the accuracy of the two methods and use the best one to predict the remainder of the horizon. At hour $t+2$, the process repeats.

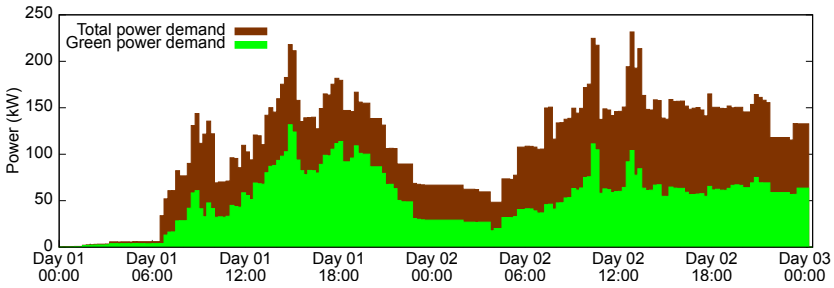
A.5 Evaluation

A.5.1 Methodology

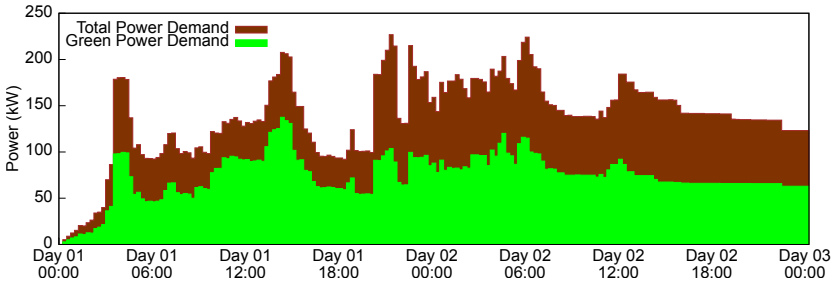
We use simulation to evaluate our framework and policies. Our simulator takes a workload trace, a solar energy production trace, a solar energy prediction trace, and the price for brown energy as inputs. Using these inputs, it simulates job arrivals and executions using a given scheduling policy, while tracking job energy consumption and whether Green SLAs are met.

Workloads. We study two real traces. The first and primary trace, called Grid5k, comes from the Grid Workload Archive [35]. The original trace was collected on the Grid'5000 system [64], a 2,218-node distributed system spread across 9 sites in France, from May 2004 to November 2006. We selected an arbitrary 48-hour portion of the trace. The chosen period has 675 jobs, with a peak processing demand of 1,654 nodes. Most of the results presented below were obtained using this workload trace.

The second trace, called Intrepid, comes from the Parallel Workload Archive [46].



(a) Grid5k



(b) Intrepid

Figure A.4: Workload power demand assuming all jobs are admitted. The green areas represent the demand for green energy given by the Green SLAs.

The original trace was collected on the Intrepid system, a 40-rack, 40,960-node Blue Gene/P system deployed at Argonne National Laboratory, from January 2009 to August 2009. We selected an arbitrary 48-hour portion of the trace, and then scaled down the processing load to match the peak demand of Grid5k (so that we can simulate the same datacenter). Specifically, we scaled down each job's node demand by a factor of 24, reducing the total system size from 40,960 to 1,680 nodes. We correspondingly scaled up job runtimes by a factor of 8 (assuming that jobs run longer when running on smaller numbers of nodes). We also filtered out 35 very large jobs to make sure the trace does not ask for more nodes than the capacity of the simulated datacenter. The final workload trace contains 446 jobs, with a peak processing demand of 1,620 nodes.

We map the workloads into our environment by assuming that each job can be split into VMs, one VM per node requested by the job. Thus, each arriving job specifies the number of VMs needed, the contracted service time for running the job, and a Green SLA. We assume discrete service time periods equal to our scheduling epochs. Thus, each job's runtime is converted into $\lceil \frac{runtime}{epochsize} \rceil$ slots. The Green SLA of each job that runs for less than 48 hours is chosen randomly according to a uniform distribution from

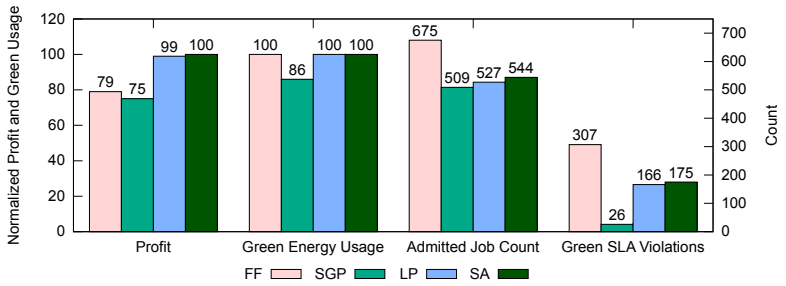


Figure A.5: Comparison of normalized profit, normalized green energy use, number of admitted jobs, and number of Green SLA violations for the Grid5k workload running over the Medium days.

the set $\{0\%, 25\%, 50\%, 75\%, 100\%\}$. For the jobs with runtimes longer than 48 hours (7 in Grid5k and 1 in Intrepid), we set the Green SLA to be 0% because we currently have predictions of green energy production only for the next 48 hours. This is done only for simplicity. We can extend our predictions further into the future using more coarsed-grained predictions (e.g., most weather services provide predictions 10 days into the future, allowing us to predict approximate daily green energy production). Figure A.4 plots the power demand over time for the two workloads assuming that all jobs are admitted.

Datacenter. We simulate a datacenter containing 42 racks, each rack containing 40 servers. This computing capacity is chosen to accommodate the Grid5k workload without requiring any scaling. Each server consumes 140W when executing a VM, giving a peak datacenter power demand of 235.2kW. The 140W value is the measured power consumption of a server equipped with a 2.4GHz 4-core Xeon CPU, 8GB of memory, 1 7200rpm disk, and a 1Gb Ethernet card.

Energy Prices. We use the average brown energy price for New Jersey: 10.55 cents/kWh [43]. We assume self-generation of solar energy, so that green energy has zero (incremental) cost. Note, however, that our framework can be easily extended to account for a non-zero green energy cost.

Service Pricing. Customers pay \$0.29 per contracted VM-hour in their Green SLAs and \$0.24 per each remaining contracted VM-hour. These prices are modeled after the green cloud provider GreenQloud's pricing [63] and Amazon EC2's pricing [8],

respectively, for a large VM instance running Linux. These prices place a premium of approximately 20% for each contracted green VM-hour.

If the cloud provider fails to meet a job's Green SLA, it pays a penalty of $0.50 per missed VM-hour. We have chosen a relatively large penalty (10 times the 5 cents premium for requiring green energy) to ensure that the cloud provider does not lightly dismiss Green SLAs to garner greater profits. We explore the sensitivity of our policies to different ratios between the service prices and penalty below.

With the above prices and penalty, if the provider admits a 1-VM job that runs for 1 hour and has a Green SLA of 100%, it earns 29 cents if it meets the job's Green SLA, earns 4 cents (29 cents fee - 25 cents penalty) if it misses the Green SLA by 50%, and pays the client 21 cents (29 cents fee - 50 cents penalty) if it does not use green energy to run the VM at all.

Opportunity Cost. We could compute the opportunity cost by scheduling a representative workload over an appropriate time frame, and computing the average profit earned per slot within each rack. However, for simplicity, we currently use an ϵ value that increases from the brownest rack to the greenest rack. This ϵ opportunity cost is sufficient to prevent unnecessary placement of VMs on green slots, which may later require migration so that the slots can be recovered to admit new jobs.

Solar Power Generation. We model the solar power generation as a scaled-down version of a solar farm at Rutgers that has a rated production capacity of 1.4MW. We scale the farm's production down to 750 solar panels capable of producing 176.2kW. After derating, the peak production can provide 75% of the peak power consumption of our simulated datacenter. We choose 75% because on days with high solar energy production, this is sufficient to admit almost the entire workload while solar energy is being produced.

We evaluate our scheduling policies using three pairs of consecutive days with different amounts of solar energy production. Specifically, we select two sunny days (5/9/11 and 5/10/11) with high solar energy production (totaling 3.23MWh), two days (6/16/11 and 6/17/11) with medium solar energy production (totaling 1.94MWh), and two days (5/15/11 and 5/16/11) with low solar energy production (totaling 626kWh). We call

these the "High", "Medium", and "Low" days, respectively.

Interestingly, the three pairs of days show different levels of accuracy for predictions of solar energy production. Predictions are mostly accurate for the High days, although there are some under-predictions for the first day. Predictions are also mostly accurate for the Low days. However, predictions for the Medium days contain significant errors, which are mostly over-predictions. We evaluated our solar energy predictions in more detail in [57].

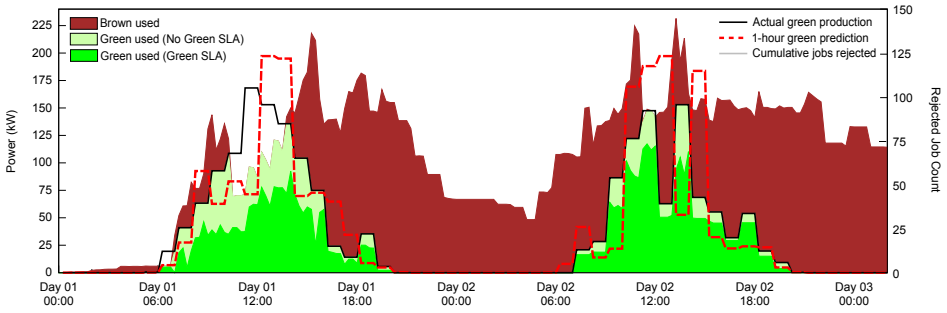
Optimization. The scheduling horizon is set to 48 hours, the extent of time into the future that we currently predict green energy production. The horizon is divided into 15-minute epochs, implying that the scheduling policy is rerun at most once every 15 minutes. We use the Gurobi solver [66] to solve the MILP optimization problem described in Section A.4.

A.5.2 Results

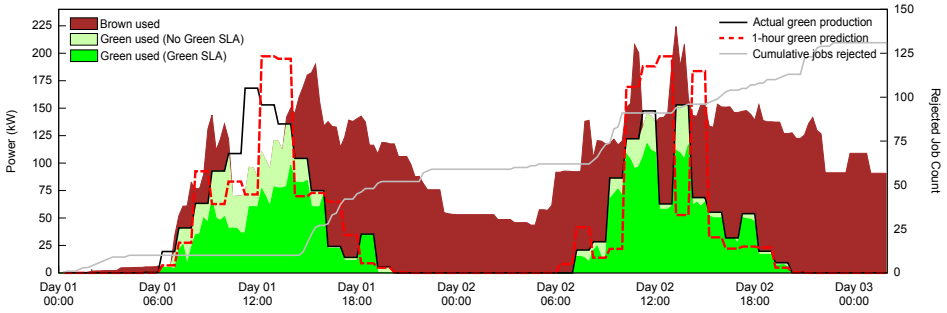
Figure A.5 compares the profit, green energy usage, number of admitted jobs, and number of missed Green SLAs achieved on the Medium days for the Grid5k workload by the four policies described in Section A.4.4. The profits and green energy usage are normalized to those of SA (left Y-axis), whereas the numbers of admitted jobs and Green SLA violations are absolute (right Y-axis). These results show that the two optimization-based policies, SA and LP, can significantly outperform the two greedy policies, with FF and SGP achieving only 79% and 75%, respectively, of SA's profit.

Interestingly, the greedy policies under-perform the optimization-based policies for opposing reasons. FF, as expected, is overly aggressive since it admits all jobs, regardless of their Green SLAs and expected green energy production. Thus, it violates many Green SLAs and incurs large penalties. However, it can use more green energy because the other policies sometimes reject jobs even when green energy is available.

In contrast, SGP is overly conservative in trying to not violate any Green SLA at all. Thus, it misses opportunities where the cloud provider can earn a profit because the penalty is relatively small compared to the revenue earned (e.g., when Green SLAs are only missed by small amounts). However, SGP violates the fewest Green SLAs among



(a) FF

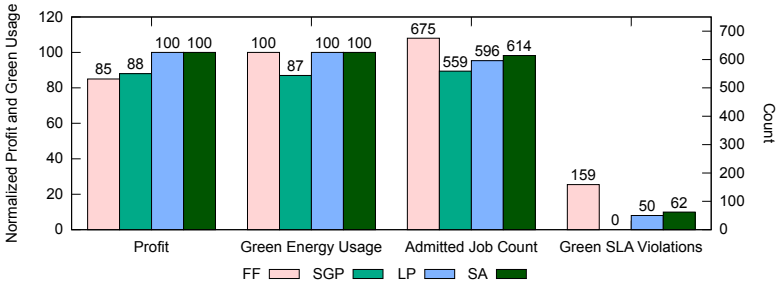


(b) SA

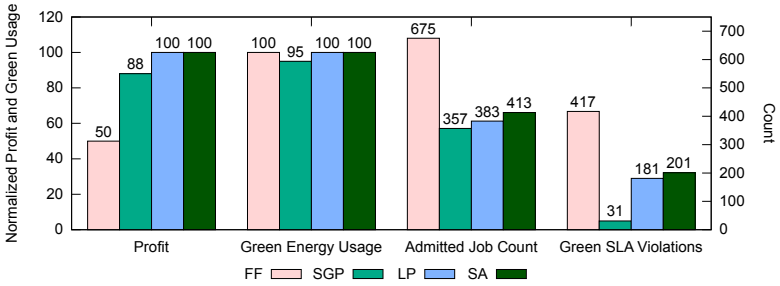
Figure A.6: Power profile of FF and SA when running Grid5k over the Medium days. The "Green used (Green SLA)" areas represent green energy used for meeting the Green SLAs of admitted jobs. The "Green used (No Green SLA)" areas represent green energy used to run jobs beyond that required to meet Green SLAs. The "Brown used" areas represent brown energy used. The dashed red lines show the 1-hour ahead predictions of green energy production. The black lines show the actual green energy production. The gray line in (b) shows the number of jobs rejected (Y-axis on the right). Note that FF does not reject any jobs.

all the policies; it only violates the Green SLAs of 26 out of 509 admitted jobs (5%). This can be advantageous if violating Green SLAs has negative implications beyond the penalty; e.g., clients seeking to reduce the carbon footprint of their computing workloads may become unhappy if too many Green SLAs are missed, despite the compensating penalties.

For the current parameters, there is little difference between the performance of LP and SA for maximizing profit. SA admits jobs more aggressively (3% more jobs than LP), but violates Green SLAs more frequently (5% more missed Green SLAs), and so pays higher penalties. SA takes substantially longer (11.3 secs on average in our experiments with a 2.4GHz Xeon server) to compute a schedule than LP (0.15 secs on



(a) High Days



(b) Low Days

Figure A.7: Comparison of normalized profit, normalized green energy use, number of admitted jobs, and number of green SLA violations for High and Low days for Grid5k workload.

average). However, both overheads are low compared to our 15-minute epochs. Also, LP runs less frequently (only when a new job arrives, i.e., 675 times) than SA (866 times) in our experiments. We explore the scalability of SA's and LP's runtimes with datacenter and workload sizes below.

For a more detailed look at the behaviors of the policies, Figures A.6(a)-(b) show the power consumption profiles of FF and SA. Overall, SA consumes less energy because it rejects jobs when it predicts that there will not be sufficient green energy to profitably admit the jobs. As already observed, this makes SA more susceptible to leaving some green energy unused, especially if green energy production is higher than predicted. In this case, both FF and SA leave some green energy unused on the first day because more green energy was produced than is required to support the entire workload for several hours. (Excess green energy can be either net-metered or stored in batteries for later use.) However, SA uses more green energy to meet Green SLAs than FF, and so earns higher profits from the produced green energy.

Impact of ratio between green revenue and penalty. The ratio between the

fee for meeting Green SLAs (green revenue) and the penalty for not meeting them significantly impacts the behaviors of LP and SA, and their performance relative to the greedy policies. As the penalty increases compared to the revenue, both LP and SA will admit fewer jobs to avoid violations of Green SLAs. Thus, their performance will become comparable to that of SGP. For example, on High days, the ratio of SGP's profit to SA's profit is 0.74:1, 0.88:1, and 0.96:1 for low ($0.20), medium ($0.50), and high ($0.80) penalty, respectively. We observe similar results for Medium and Low days. Further, FF's relative performance will worsen because its obliviousness to green energy production becomes increasingly expensive. As the penalty decreases compared to the revenue, both LP and SA will admit more jobs to increase profit, even though more Green SLAs will be missed. Thus, their performance will become more comparable to that of FF. SGP's relative performance will worsen because it does not recognize that violations of Green SLAs can be profitable.

Impact of green energy availability. We now evaluate the policies across the three pairs of days with different levels of green energy production. Figures A.7(a)-(b) show the results of running Grid5k over the High and Low days. As one might expect, the advantages of the optimization-based policies over FF become clearer when the availability of green energy is more limited. On the Low days, FF only achieves 50% of SA's profit. On the other hand, as green energy becomes more available, FF becomes more comparable, achieving 85% of SA's profit on the High days. SGP's performance compared to LP and SA is relatively insensitive to green energy production.

While not visible in the figures (the differences are less than 1%), SA outperforms LP slightly. This is because SA can migrate VMs from already admitted jobs to correct for inaccuracies in green energy prediction. If green energy production is greater than predicted, SA can migrate VMs from jobs receiving more green energy than expected to browner racks in order to admit more jobs. If production is less than predicted, SA may be able to migrate VMs to avoid or reduce the penalty from missed Green SLAs.

We have also simulated scenarios with peak green energy production providing 25% and 50% (compared to the above base case of 75%) of the peak power consumption of the datacenter. Trends in these results are consistent: the performance gap between

the optimization-based and heuristic-based policies widens as the availability of green energy decreases.

Impact of green energy prediction inaccuracies. Inaccurate green energy predictions can harm profit. Over-predictions (i.e., predicted production is greater than actual production) can be especially bad since they also lead to increased violations of Green SLAs. Thus, it may be desirable to make the green energy predictions more conservative.

To study the impact of conservative predictions, we rerun our simulations while reducing all $B(t)$ (Section A.4.5) values by 10%, 20%, and 30%. These conservative predictions lead to slightly lower profits for both SA (<3%) and LP (<3%) but significantly fewer missed Green SLAs (up to 18% and 12% fewer, respectively). We conclude that it is worthwhile to make the green energy predictions more conservative than the method described in Section A.4.5, perhaps by up to 30%.

Impact of workload characteristics. Thus far, we have focused exclusively on the Grid5k workload. We have also studied the Intrepid workload, which was collected on a distinctly different system from that of Grid5k, to validate our findings. Observations from results obtained using Intrepid are consistent with those based on Grid5k. In fact, because Intrepid exhibits a less distinct diurnal pattern than Grid5k, it has more jobs asking for green energy when none is available (i.e., at night). Thus, the difference between FF and the other policies is more significant since admission control becomes more important for profitability. For example, for the Medium days, FF's profit when running Intrepid is only 64% of SA's profit compared to 79% when running Grid5k. Also, there are more VMs per job in Intrepid than Grid5k, giving SA and LP more flexibility for scheduling. As a result, both policies migrated VMs less often.

Impact of limiting SA's search space. Recall from Section A.4.4 that we limit SA's execution time by rescheduling only a subset of the active jobs; for the results discussed above, we limited SA to reschedule at most 30 jobs. To study the impact of this heuristic on the quality of SA's solutions, we compare SA's performance to a version of SA, called SA-All, that reschedules all active jobs. Results show that SA-All achieves 3.5% higher profit than SA when running Grid5k over the Medium days, but

is 28 times slower. Thus, we conclude that our heuristic makes it practical to use SA without significantly degrading the quality of SA's solutions.

Scalability of SA and LP. We explore the scalability of SA and LP by running experiments for datacenters with 11, 21, and 84 racks. We also scale the workload proportionally (e.g., the workload of a datacenter with 84 racks is twice that of a datacenter with 42 racks). As we are increasing the problem size exponentially, the runtimes also grow exponentially. However, the overheads for a 84-rack system are still low compared to our 15-minute epochs. Average runtimes for LP are 0.02, 0.04, 0.15, 0.23 for 11, 21, 42, and 84 racks, respectively. Average runtimes for SA are 0.1, 1.0, 11.3, 58.3 secs. As systems and workloads increase further, the provider may consider utilizing more machines to run our policies, or simply increase the epoch length.

Battery size. Finally, we estimate the size of the battery needed for our simulated datacenter to validate our claim that the proposed infrastructure only requires a small battery. In particular, we study a fine-grained trace of solar energy production collected in March 2013 from our solar-powered micro-datacenter [55], scaled to match the size of the simulated datacenter (42 racks, 40 machines per rack). We choose this particular month because there were events such as snow storms that can lead to inaccurate prediction of green energy production. In this month, the green energy predictor over-predicted green energy production for 60 epochs (2%). The average over-prediction is 11.7kW with a maximum of 29.9kW. If we were to tolerate the maximum deficit for 10 seconds before switching racks from the green to the brown partition, we would need a 0.08kWh battery. For the 60 epochs, on average we need to switch 2 racks from the green to the brown partition (maximum 5 racks). The 10 seconds buffer is also more than sufficient to tolerate fine-grained variability in green energy production. This battery size is quite small compared to the battery capacity to sustain 8-10 minutes of operation at peak load already provisioned in typical datacenters [58]. For our simulated datacenter, this provisioning would translate to 31.3 to 39.2kWh.

A.6 Conclusion

In this paper, we proposed that HPC cloud service providers should offer a novel green service. In this service, each client job specifies a Green SLA, which is the minimum percentage of green energy that must be used to run the job. We then proposed a power distribution and control infrastructure, together with an optimization-based scheduling framework and policies, for providing such a Green SLA service. We also proposed two simple greedy heuristic policies for achieving the same goals. We evaluated our proposals extensively using simulations. Our evaluation results showed that the optimization-based policies can significantly outperform the greedy policies. The results also showed that the choice of optimization-based policy depends on whether the cloud provider prefers to accept more jobs or violate fewer Green SLAs. We conclude that a Green SLA service that uses our policies can be useful for the provider to attract environmentally conscious clients, especially those who require strict guarantees on their use of green energy.

Appendix B

GreenPar: Scheduling Parallel High Performance Applications in Green Datacenters

B.1 Introduction

Datacenters consume an enormous amount of electricity, and this consumption is growing rapidly. Estimates for 2010 indicate that datacenters consumed 1.5% of the total electricity used world-wide, and that this usage increased by 56% over the previous 5 years [94]. This consumption translates into high carbon emissions, since most of the electricity is produced by burning fossil fuels.

With increasing societal awareness of emissions and climate change, there is an increasing demand for cleaner products and services. As a result, there is a rising interest in powering datacenters at least partially using on-site generation of renewable ("green") energy from sources such as wind and solar. For example, Apple and McGraw-Hill have built 40MW [10] and 14MW [32] co-located solar arrays, respectively, for their datacenters. Green House Data [59], AISO [1], and GreenQloud [62] are cloud providers that operate green datacenters mostly (or entirely) powered by wind and/or solar energy.

Such "green" datacenters hold great promise for reducing the environmental impact and electricity cost of datacenter computing. For example, Goiri et al. argue that the installed capital cost for the solar energy system of a green micro-datacenter can be recovered from electricity cost savings in less than 10 years [55]. However, an important research challenge arising in these green datacenters is that generation of green energy from sources such as solar and wind is variable. For example, photovoltaic (PV) solar

energy is only available during the day and the amount produced depends on the weather and season. One approach for mitigating this variability is to store green energy in batteries or the grid via net metering. However, this approach has significant disadvantages: (1) batteries are expensive[1] and common types of batteries (e.g., lead-acid) are harmful for the environment, (2) batteries incur energy losses, (3) net metering incurs losses and is not available everywhere, and (4) where net metering is available, the power company may pay less than the retail electricity price for the green energy.

Given the above disadvantages, in this chapter, we investigate how to manage the computational workload to better match the energy demand to the energy supply. Specifically, we propose and evaluate GreenPar, a scheduler for parallel high-performance computing (HPC) workloads in green datacenters. GreenPar seeks to maximize green energy consumption without the need for energy storage, thereby increasing the benefits of on-site generation.[2] GreenPar also seeks to minimize "brown" grid energy consumption (when there is insufficient green energy) while respecting a performance service-level agreement (SLA). GreenPar increases the resource allocations of active parallel jobs to reduce runtimes when green energy is available. When there is insufficient green energy, GreenPar reduces resource allocations within the constraints imposed by the performance SLA to conserve brown energy. GreenPar relies on the jobs' speedup profiles in making its decisions. A specific performance SLA that we consider in this chapter is a maximum runtime slowdown percentage; for example, the SLA might state that a job's execution cannot be slowed down by more than 10% in order to conserve brown energy.

GreenPar can reduce brown energy consumption *even when it is not allowed to slow down the jobs*. It can do this by first allocating more resources than requested to a parallel job when green energy is available, increasing the job's rate of progress and thus letting it accumulate performance "slack." Subsequently, when running on brown energy, GreenPar can conserve energy by reducing the resource allocation of the job,

[1] Goiri et al. found that the cost of batteries is currently not amortizable when they are used as a power source in a green datacenter, unless there is enough performance slack for deferring the workload [55].

[2] The datacenter still has batteries for handling grid failures, but GreenPar does not use them because that would require extra capacity and frequent charge/discharge may shorten their lifetimes.

using the accumulated performance slack to avoid lengthening its overall completion time.

We have designed four green-energy-aware scheduling policies for GreenPar, targeting solar-powered green datacenters. Each policy assumes a different amount of knowledge about future green energy production, jobs' characteristics, and the datacenter workload. We have implemented these policies in a GreenPar prototype framework, and evaluated this implementation in Parasol, a solar-powered micro-data-center we built at Rutgers University [55]. Our evaluation uses workloads comprising jobs executing a subset of the NAS Parallel Benchmark (NBP) suite [121]. The workloads are constructed to resemble traces of real HPC workloads obtained from the Grid Workload Archive [64] and the Parallel Workload Archive [46]. The NBP applications are linked with the MPICH2 MPI implementation [21], and run inside virtual machines (VMs) executing on Xen [13]. (Prior work has demonstrated that virtualization degrades the performance of HPC applications by less than 4% [71, 79].) The virtualization layer allows GreenPar to dynamically change a job's resource allocation, i.e., the set of physical servers hosting the job's VMs, by migrating and consolidating VMs within the datacenter as appropriate, without requiring changes to the MPI implementation.

For repeatability, we also evaluate GreenPar using properly scaled-down traces of green energy production from a solar farm at Rutgers University. When allowing GreenPar to slow jobs' executions down by at most 10%, we show that GreenPar can reduce brown energy consumption by 10% while *reducing* average runtime by 13%, compared to a baseline policy not aware of on-site green energy production. In addition, GreenPar satisfied the jobs' performance SLAs in all our experiments. These results show that while GreenPar may slow down some jobs (within their performance SLAs) to increase brown energy savings, it successfully leverages green energy to speedup overall executions. If applications have good speedup profiles (e.g., close to linear speedup), GreenPar can reduce brown energy consumption by 15% while reducing average runtime by 40%.

In summary, we make the following contributions: (1) we introduce GreenPar, a scheduling framework for parallel HPC workloads in datacenters partially powered by

solar energy; (2) we introduce four scheduling policies for GreenPar, each assuming a different amount of knowledge about future green energy production, job characteristics, and datacenter workloads; (3) we implement GreenPar and evaluate it on Parasol, a real solar-powered micro-datacenter; and (4) we present extensive evaluation results for GreenPar, isolating the impact of the different amounts of knowledge in the four policies, and exploring its sensitivity to various parameters.

B.2 Related Work

As far as we know, GreenPar is the first resource scheduler for green datacenters running parallel HPC applications. Though prior works have considered moldability, malleability, and folding in these applications (e.g., [41, 143]), none of them considered green energy and using it to reduce brown energy consumption and speed up executions. Other works [56, 57, 55, 96, 105] considered green energy in the context of scheduling batch applications, but did not dynamically change server allocations of running jobs or attempted to speed up executions using green energy while considering application speedup characteristics.

We divide the other works that relate to GreenPar into two areas: brown power management in parallel applications, and green energy management in datacenters.

Brown power management in parallel applications. Researchers have used multiple techniques to minimize energy consumption or to keep power consumption within a budget when running parallel applications. A popular technique is Dynamic Voltage and Frequency Scaling [131, 78]. Researchers also addressed power measurement and/or modeling of parallel applications [51, 106, 125] in power-aware clusters. Other researchers [122, 29] focused on scheduling for greater efficiency, which indirectly reduces energy consumption. Unlike GreenPar, these works did not consider on-site generation of green energy, where sometimes consuming more (green) energy is a good idea. Leveraging green energy, GreenPar can conserve brown energy even when it is not allowed to trade off performance (i.e., when the SLA does not allow for any performance slowdown).

Green energy management in datacenters. Stewart and Shen [137] and Le et al. [100] have explored request distribution policies for interactive online services that span multiple datacenters to explicitly manage the consumption of green and brown energy. Aksanli et al. [4] have considered the scheduling of mixed batch and service workloads in green datacenters. Akoush et al. [2] proposed workload distribution in virtualized systems. Several efforts have investigated load migration within a datacenter to better leverage on-site green energy production [87, 104, 70, 108].

Unlike these works, GreenPar manages the green energy usage to reduce the brown energy consumption and improve the performance of parallel HPC applications. Moreover, GreenPar dynamically re-allocates resources across running jobs while considering their speedup characteristics to maximize the benefit of green energy.

B.3 GreenPar

We propose GreenPar, a job scheduler for parallel HPC workloads running in green datacenters partially powered by solar energy (see Figure B.1). Note that, except for the prediction of near-future production of solar energy, *GreenPar is directly applicable to wind-powered green datacenters.* In this section, we first overview GreenPar, and then discuss four specific green-energy-aware policies for it. We also present a simple green-energy-unaware policy that we use as a basis for comparison in Section B.5.

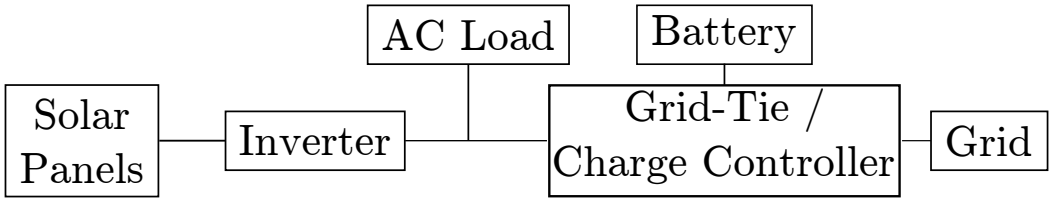


Figure B.1: A possible setup for a datacenter partially powered by solar energy. The datacenter is connected to the electrical grid, which can meet the datacenter's peak power demand even when no solar energy is being produced. Batteries are only used as backup for grid outages.

B.3.1 Overview

GreenPar seeks to maximize green energy consumption and minimize brown energy consumption, while respecting a performance SLA. GreenPar achieves its goals by dynamically changing the resource allocations of jobs based on the availability of green energy, jobs' characteristics, and jobs' accumulated resource allocations.

We assume that each parallel job executes inside a set of VMs (one process per VM), as has been done in previous works (e.g., [71, 79]). GreenPar can dynamically change the number of physical servers assigned to run a job, migrating and consolidating VMs within the datacenter as necessary. Unused servers are put into a low-power state (e.g., ACPI S3) to conserve energy. Note that *GreenPar does not require a virtualized environment.* Rather, it only requires that jobs can adapt to dynamically changing numbers of servers. Bag-of-tasks applications, for example, are amenable to changes in the number of servers without any need for virtualization. Moreover, several works (e.g., [41, 143]) have demonstrated moldable and malleable HPC frameworks based on MPI, again without using virtualization.

Each parallel job j is submitted to GreenPar with a requested number of servers (N_j) and a maximum acceptable slowdown factor (F_j). Three of the GreenPar policies also ask the user to supply a speedup profile $SP_j(n)$, defined as $SP_j(n) = RT_j(1)/RT_j(n)$, where $RT_j(n)$ is j's runtime when running its VMs on a constant allocation of n servers. (Two policies actually require two speedup profiles, as we explain below.) F_j is a factor ≥ 1 such as 1.1, which would allow GreenPar to slow down job execution by at most 10%. For simplicity, we assume that F_j is the same for all jobs, although our policies can easily be extended to handle different slowdown factors for different jobs.

GreenPar avoids excessively degrading execution times (i.e., degrading them by more than a factor of F) as a result of trying to conserve brown energy. To avoid potentially increasing waiting times due to brown energy conservation, GreenPar reverts back to the user-requested resource allocations when utilization becomes so high that jobs cannot be started immediately upon arrival. As Section B.5 demonstrates, GreenPar tends to reduce average runtimes compared to green-energy-oblivious schedulers, which makes

it less likely that jobs have to wait at all.

In our current implementation, three GreenPar policies start either twice or four times as many VMs as the requested number of servers. (The other GreenPar policy and the baseline policy always start two VMs per requested server.) The choice between these two options depends on the expected amount of green energy. For example, these policies typically start 16 VMs for $N_j = 8$ in our evaluation environment, where each server has a dual-core processor and so can efficiently host 2 VMs. But they may start 32 VMs, if they are expecting plentiful green energy, so that they are likely to allocate more than N_j servers to job j. Two of these policies use speedup profiles to make decisions, and so require two speedup profiles–one for 16 VMs and another for 32 VMs in our example–for each job, because the job's speedup depends on its number of VMs.

GreenPar then periodically computes the resource allocations for the active jobs. It dynamically increases the numbers of servers allocated to jobs that can achieve further speedups when expecting excess green energy. It dynamically decreases resource allocations of jobs that have accumulated or can be expected to accumulate performance slack (via larger resource allocations than requested when green energy was/will be available) to conserve energy when brown energy is needed.

GreenPar considers the speedup profiles of active jobs when making resource allocation decisions. For example, suppose two jobs, A and B, are each running on 10 servers, and A will not speedup if given more than 10 servers but B will until its allocation exceeds 25 servers. Then, if GreenPar expects that enough green energy will be produced to support 30 servers for a period of time, it will use the speedup profiles to increase B's server allocation to 20, while leaving A's at 10. Note that if B finishes before the green energy production decreases, GreenPar will have successfully used the excess green energy to decrease B's runtime.

However, if both A and B continue to run until a time period when no green energy is available, GreenPar will reduce both their allocations to below 10 servers to conserve brown energy. Of course, GreenPar cannot reduce the allocations so much that either A or B would take more than F times longer to complete compared to when each is running on a constant allocation of 10 servers. In this example, GreenPar may be able

to reduce the allocation of B more than A, since B had accumulated some performance slack while running on 20 servers.

While the use of jobs' speedup profiles allows GreenPar to make intelligent decisions about resource allocation, it does make GreenPar reliant on having this information. Speedup information can be gathered across multiple runs of an application, and/or constructed using profiling (e.g., [51, 134]). These approaches can produce very accurate speedup profiles, because HPC applications are often run many times with similar inputs. It is also possible to measure speedup at runtime with sufficient hardware and system software support [122]. Regardless, *to assess the impact on GreenPar if this information was not available, one of the GreenPar policies only uses a hint from the user on whether the submitted job would speedup if given more resources than requested.* Users that are unsure about their applications' behaviors can just supply a "no, the job will not speedup" hint.

B.3.2 Policies

We now present the four GreenPar policies and the baseline policy. Each GreenPar policy assumes a different amount of knowledge about future green energy production, jobs' characteristics, and future load. All policies divide time into discrete epochs for computing resource allocation schedules. All policies respond to job arrivals and completions within a time epoch.

Reactive. As its name suggests, Reactive does not assume any knowledge or prediction of future job arrivals and runtimes. Instead, it tracks the resource allocation for each job, predicts the green energy production for the next time epoch at the end of every time epoch, and reacts to the expected availability of green energy. When there is excess green energy, Reactive tries to increase the server allocation of active jobs to decrease runtimes. It greedily tries to increase the allocation of jobs with the best marginal speedups first, seeking maximum reduction in job runtimes for each additional allocated server. Subsequently, when there is insufficient green energy, Reactive tries to shrink the server allocations of active jobs that have accumulated some performance slack to conserve brown energy. It again greedily tries to reduce the allocations of jobs

```
0.  For each job j, the user specifies the desired number of servers N_j and speedup profile S_j(n)

1.  For each event in {Job Arrival, Job Finish, Change in Green Energy, Reallocation Needed}
2.      For each job j,
3.          Calculate the minimum server allocation n_j for j (lines 14-20 below)
4.      Let C = sum of all n_j
5.      Let G = number of nodes that can be powered by the expected available green energy
6.      For each job j, allocate n_j servers to j
7.      While G > C:
8.          For each job j,
9.              Calculate the decrease in efficiency for further node allocation δ_j
10.         Sort the jobs in ascending order of δ_j
11.         Increase n_j by a threshold number of servers m_j to the first job j in the list
12.         C = C + m_j and update δ_j
13.         Resort the jobs in ascending order of δ_j

14. Calculate Minimum number of nodes for job j:
15.     if j is new, average speedup as_j = 0 and achieved runtime rt_j = 0
16.     Assume that j will run for another threshold time period t
18.     n_j = the minimum number of nodes such that as_j ≥ F · S_j(N_j)
19.     Schedule a Reallocation Needed event t time into the future
20.     Return n_j
```

Figure B.2: Pseudocode for the Reactive scheduling algorithm.

with the smallest marginal speedups first, seeking a minimum increase in runtimes for each removed server.

Reactive does not need to know jobs' runtimes to respect the SLA. Instead, knowing the requested number of servers and jobs' speedup profiles is sufficient since:

$$MaxRunTime = F \cdot RunTime_{j,1}/SP_j(N), \text{ and}$$
$$RunTime = RunTime_{j,1}/AverageSpeedup_j$$

implying that we can meet the performance SLA if:

$$AverageSpeedup_j \geq SP_j(N)/F$$

In our current implementation, Reactive always starts a job with twice as many VMs as the number of requested servers, so it requires only one speedup profile for each job. Figure B.2 shows the pseudo-code for Reactive.

Offline optimization. For comparison, we develop an optimization-based policy with oracular knowledge of job arrivals, job speedups and runtimes, and green energy production. We call this policy Offline because GreenPar is unlikely to have the assumed information.

Table B.1: Framework parameters. Time epochs in the scheduling horizon are numbered from 1 to T.

Symbol	Meaning
T	Set of time epochs comprising scheduling horizon
J	Set of jobs arriving in the scheduling horizon
β	Relative weight of the average runtime vs. the brown energy cost in the objective function
F	Maximum acceptable slowdown factor
C	Number of servers in the datacenter
$GE(t)$	Predicted green energy production in epoch t
$PB(t)$	Price of the brown energy during epoch t
A_j	Epoch that job j arrives and starts running
E_j	Epoch that job j completes
N_j	Number of servers requested for job j
$RT_{j,1}$	Runtime of job j when running on 1 server
$SP_j(n,v)$	Job j's speedup when running on n servers with v VMs
$ME_j(n)$	Migration energy when job j's server allocation changes by n
$Pow_j(n)$	Power demand of job j when running on n servers
V_j	Number of VMs used to run job j
$S_j(t)$	Number of servers allocated to job j in epoch t

Table B.1 lists the parameters of our optimization framework. Using them, we formulate the optimization problem shown in Figure B.3. The optimization works on a scheduling horizon (e.g., 24 hours into the future) that is divided into discrete time epochs (t). Solving the optimization problem produces V_j (the number of VMs to run job j) and an epoch-based schedule of the number of servers ($S_j(t)$) assigned to each job j arriving in the scheduling horizon. Our current implementation considers starting each new job with twice or four times as many VMs as the user-requested number of servers for the job. The optimization is re-solved periodically to account for new knowledge (e.g., jobs arriving after the current 24-hour scheduling horizon).

This optimization minimizes *Objective*, a weighted sum of the brown energy cost (*BrownCost*) and average job runtime (*AvgRuntime*).[3] The β factor allows us to

[3]Previous studies have used different optimization objectives, e.g., $Energy \times Delay$ or $Energy \times Delay^2$. We

adjust the relative importance of conserving brown energy versus increasing job runtimes. $BrownCost$ is a function of the brown energy use ($BrownEn(t)$) over time and the brown energy price. Brown energy is consumed only when more energy ($En(t)$) is needed than the amount of green energy being produced. Energy is consumed both by jobs executing and migrations required to adapt to dynamically changing resource allocations ($MiEn_j(t)$). We use the v parameter in $SP_j(n, v)$ to represent the two user-provided speedup curves.

$$
\begin{aligned}
Objective &= BrownCost + \beta \cdot AvgRuntime && \text{(B.1)}\\
BrownCost &= \sum_{t \in T} BrownEn(t) \cdot PB(t) && \text{(B.2)}\\
BrownEn(t) &= \begin{cases} En(t) - GE(t) & \text{if } En(t) > GE(t) \\ 0 & \text{otherwise} \end{cases} && \text{(B.3)}\\
En(t) &= \sum_{j \in J} (Pow_j(S_j(t)) \cdot |t| + MiEn_j(t)) && \text{(B.4)}\\
MiEn_j(t) &= ME_j(|S_j(t) - S_j(t-1)|) && \text{(B.5)}\\
AvgRuntime &= \frac{1}{|J|} \cdot \sum_{j \in J} \sum_{t \in T} Running_j(t) \cdot |t| && \text{(B.6)}\\
Running_j(t) &= \begin{cases} 1 & \text{if } S_j(t) > 0 \\ 0 & \text{otherwise} \end{cases} && \text{(B.7)}\\
E_j &= A_j + \sum_{t \in T} Running_j(t) && \text{(B.8)}\\
AvgSpeedup_j &= \sum_{t \in T} SP_j(S_j(t), V_j)/(E_j - A_j + 1) && \text{(B.9)}
\end{aligned}
$$

Figure B.3: Optimization framework.

GreenPar solves the optimization problem under constraints formulated to ensure that the jobs' performance SLAs are met, jobs complete their executions, jobs are not preempted once they start running, and the allocated servers do not exceed the datacenter capacity.

The optimization problem can be transformed into a linear problem except for: (a) the migration energy $ME_j(n)$ and the power demand of a job $Pow_j(n)$ may be non-linear functions, and (b) the speedup functions $SP_j(n, v)$ may be non-linear. We address

choose this linear combination because it has been used successfully [5], and it allows us to formulate a linear optimization problem that can be solved efficiently.

the first issue by experimentally measuring and using approximate linear functions for both functions; in fact, we use only a single $Pow(n)$ and $ME(n)$ function for all jobs. In our evaluations, we found that this assumption introduced only small inaccuracies. We address the second issue by instantiating the speedup functions of each job as a table look-up using Integer Programming. Thus, the result is a Mixed Integer Linear Programming (MILP) problem that we can efficiently solve (at the sizes we consider) with standard solvers (e.g., [66]).

Aggressive. This policy is essentially the same as Offline except that it only assumes knowledge of jobs' runtimes and speedup profiles. It predicts future green energy production, implying that $GE(t)$ includes prediction errors. We discuss green energy predictions further in Section B.4. It also solves the optimization problem only for active jobs, i.e., J only contains the jobs that have already arrived, and thus does not rely on knowledge of future arrivals.

Aggressive re-solves the optimization whenever a new job arrives. The optimization generates the number of VMs (V_j) for the new job and an epoch-based schedule of the number of servers ($S_j(t)$) assigned to each job j. Aggressive must also re-solve the optimization when there are significant changes to the predictions of green energy production. In our experiments, solving the Aggressive optimization problem takes less than 2 seconds.

Nebulous. Finally, we develop and study Nebulous, a policy that does not assume accurate knowledge of speedup profiles. Rather, it uses a user-provided hint about job speedup (excellent, good, fair, poor) and a rough estimate of job runtime. With these hints, it is impossible for Nebulous to support the same performance SLA as the previous policies. Thus, Nebulous instead seeks to ensure that the average resource allocation to each job is never less than $1/F$ times the number of requested servers.

Nebulous predicts green energy production, and, when a job arrives, it starts the job with more VMs if it expects that there will be enough green energy to run the larger number of VMs for the estimated duration of the job. This approach is conservative in that the runtime estimate is for the smaller number of VMs, so it is likely to be longer than the runtime when using the larger number of VMs.

Nebulous is then somewhat similar to Reactive. When there is excess green energy, it greedily allocates more servers to jobs with the best speedup hints and smallest average allocations thus far vs. their requested numbers of servers. Using the intuition that marginal speedup typically decreases with increased resource allocations, it only gives the smallest number of extra servers possible to each job before considering the next job. When green energy production decreases, Nebulous reduces server allocations to jobs in the reverse order (i.e., take away servers from jobs with poor speedup hints and most accumulated slack first).

Finally, since Nebulous only uses the runtime estimate to decide whether to run a job with more VMs, it can use statistics from previous runs of the application. When an application is run for the first time, Nebulous conservatively assumes that the job will run for a long time, implying that it does not start the job with more VMs. In our evaluations, we use the average of the previous runs of each application and correct speedup hints.

Baseline. We use a Baseline policy as a basis for comparison in our evaluation of the green-energy-aware policies. Baseline runs each job as it arrives, starting each job with twice the number of VMs as requested servers.

B.4 Evaluation Methodology

Prototype implementation and evaluation platform. We have implemented a prototype of GreenPar comprising roughly 2200 lines of python code. The prototype includes all four policies Reactive, Aggressive, Offline, and Nebulous.

We evaluate the prototype running on Parasol [55]. Our evaluation uses 55 servers, where each server is equipped with a dual-core 1.6GHz Atom processor, 4GB of memory, one 250GB hard disk, and one 64GB solid-state drive. This cluster uses 5 machines as NFS servers to hold VM images and application data, and the remaining 50 machines for running jobs executing inside VMs. Each server consumes between 22W to 30W, giving a peak power consumption of 1.5kW for the computing servers. The servers are interconnected with a 1Gbps Ethernet network.

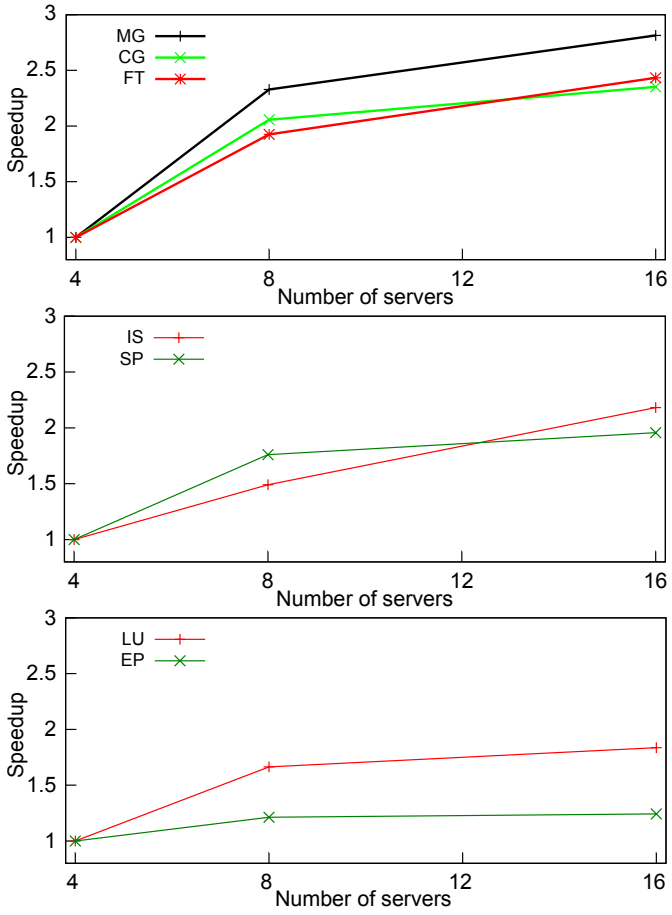


Figure B.4: Speedup profiles of NPB applications running with 16 VMs on different numbers of physical servers. The speedup profile of BT is not shown because it is exactly same as SP.

The 50 compute servers run Xen 4.3 [13], using the Linux 3.11 kernel for both the host and guest OSes. Each VM image is configured with 1 virtual CPU, 384MB of memory, and a 4GB disk. Baseline always assigns two VMs to each server, giving each VM a dedicated core. The other policies can assign 1 VM to each server when a job is given extra resources, or up to 4 VMs per server when the job is consolidated to save energy. Although running a VM with just one virtual CPU on a dual-core server may seem somewhat inefficient, it can speedup execution by 66% over running 2 VMs on the server for communication-intensive applications (e.g., NAS IS). We use Xen's live migration capability to migrate VMs without interrupting job executions. The virtual disks are stored on the NFS servers, and so do not have to be migrated. Measurements show that live migrations had very little impact on job runtimes. Idle servers are placed into the ACPI S3 state to conserve energy.

Parasol allows us to measure the energy consumption of individual servers. Thus, all results below are from actual measurements of energy consumed by executing workloads.

Workloads. We study two workloads created to emulate two real traces. The first workload, called Grid5k, emulates a trace from the Grid Workload Archive [35]. The original trace was collected on the Grid'5000 system [64], a 2,218-node distributed system spread across 9 sites in France, from May 2004 to November 2006. We chose an arbitrary 24-hour period from this trace and filtered out short running jobs that ask for just one processor core, assuming that they are small test runs, to scale the workload to our datacenter. After filtering, the chosen sub-trace has 26 jobs, with a peak processing demand of 96 cores. Most of the results presented below were obtained using this workload trace.

The second workload, called Intrepid, emulates a trace from the Parallel Workload Archive [46]. The original trace was collected on the Intrepid system, a 40-rack, 40,960-node Blue Gene/P system deployed at Argonne National Laboratory, from January 2009 to August 2009. We selected an arbitrary 24-hour portion of the trace, and then scaled down job sizes as explained in the next paragraph to fit our datacenter. The selected period has 113 jobs, with a peak processing demand of 86 cores.

Each of the workload traces includes information on job arrival time, job runtime,

and the number of cores requested. However, they do not include information about the applications nor the input data. Thus, we use a subset of applications from the NAS Parallel Benchmark (NPB) [121] to instantiate the workloads. NPB contains 8 applications with different input sizes. We use a mix of applications and input sizes to emulate the two different traces. Figure B.4 shows the speedup curves for all the applications used in our workload.

Most NPB jobs require a power-of-2 number of VMs. We map each job in the Grid5k trace to the closest NPB application in terms of core count (assuming 1 VM per core) and runtime, with 2 VMs running on each dual-core server. With this mapping, the resulting Grid5k workload includes 12 FT, 6 LU, 3 MG, 2 CG, and 1 each of SP, EP and IS. In contrast, Intrepid jobs ask for large numbers of cores, with the minimum being 256. To scale these jobs to our datacenter, we map the Intrepid jobs following Table B.2. The resulting workload contains 73 IS, 24 MG, 14 CG, 1 FT and 1 SP applications. Figure B.5 shows the workload demand (in number of cores) of Grid5k and Intrepid, as a function of time. The workloads never require more servers than are available (50 servers with 100 cores) in our setup.

Table B.2: Core count mapping for Intrepid.

Intrepid core count	256, 512	1024, 2048	4096	Others
Mapped core count	4	8	16	32

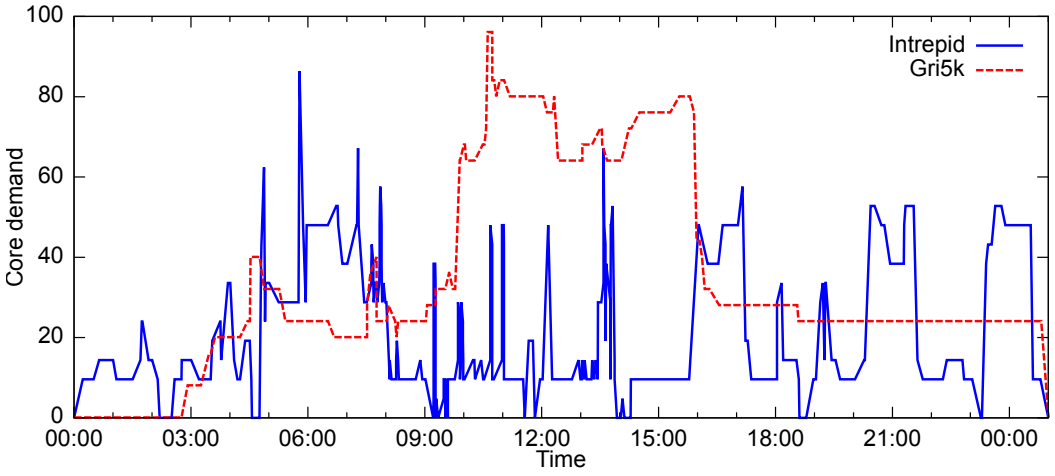


Figure B.5: Workload demand in number of cores, as a function of time.

Energy prices. We use the average brown energy price for New Jersey: 10.55

cents/kWh [43]. The use of a constant price equates saving brown energy with reducing brown energy cost. However, in general, both Aggressive and Offline can leverage dynamically changing energy prices to reduce the brown energy cost. We assume self-generation of solar energy, so that green energy has zero (marginal) cost. Our framework can be easily extended to account for a non-zero green energy cost, if desired.

Optimization weighting factor. The outcome of the Aggressive and Offline policies is affected by the value of β. β controls the relative importance of runtime compared to electricity cost. To gauge this relationship, we first convert runtime to a cost figure (using the cost of computation in the cloud) and then compare it to the electricity cost of the same computation and for the same runtime. Specifically, we compare the per-hour cost of renting a large VM instance on Amazon's EC2 [8] ($0.24) with the per-hour electricity cost of a server with the same specification as the instance (roughly 0.3kWh × $0.1055/kWh). So, our chosen value for β is 8 ($= \frac{\$0.24}{0.3kWh \times \$0.1055/kWh}$).

We also ran experiments with β ranging from 0.8 to 80. We omit these results as they were as expected, and do not affect the observations in our evaluation.

Estimating migration overheads. We measured the migration time for all the applications in our environment. The measured migration time (and thus energy overhead) does not vary much among the applications. So, we use the average migration time for every application when solving the optimization problem in Aggressive and Offline.

Solar energy production and prediction. Although Parasol is partially powered by a set of solar panels, for repeatability, we used traces of solar energy production in our experiments. However, we have run many validation runs on Parasol, using real predictions and production of solar energy. We show results from one of these runs below to show that GreenPar behaves as expected under live execution.

In our experiments, we model the solar power generation as a scaled-down version of a solar farm at Rutgers that has a rated production capacity of 1.4MW. We scale the farm's production down to 8 solar panels capable of producing 1.88kW. After derating, the peak solar power production matches the peak power consumption of our datacenter.

We evaluate our scheduling policies using three days with different amounts of solar energy production: one sunny day (5/9/11) with high solar energy production (totaling

12kWh), one day (6/16/11) with medium solar energy production (totaling 8kWh), and one day (5/15/11) with low solar energy production (totaling 2.84kWh). We call these the "High", "Medium", and "Low" days, respectively.

We use the method from [57] (which is based on [136]) to predict solar energy production for a horizon of 48 hours. Briefly, the model relates solar energy generation to cloud cover as $E_p(t) = B(t)(1 - CloudCover)$, where $E_p(t)$ is the amount of energy predicted for time period t, $B(t)$ is the amount of energy expected under ideal sunny conditions, and $CloudCover$ is the forecasted percentage cloud cover. We use Intellicast.com's weather forecasts, which predicts $CloudCover$ for each hour of the next 48 hours. This leads to prediction granularity (i.e., t) of one hour. We set $B(t)$ for each hour of the day to the amount of energy generated during that hour on the day with the highest energy generation from the previous month.

Of course, weather forecasts are sometimes wrong. Thus, the prediction includes a technique for using recent energy production to predict future production when predictions based on weather forecasts are inaccurate [56]. The evaluation in [57] shows that one-hour ahead predictions using this method produce inaccuracies of around 11%.

Our prediction method produces different accuracies for the High, Medium, and Low days. Predictions are mostly accurate for the High day, although there are some under-predictions. They are also mostly accurate for the Low day. However, those for the Medium day contain significant errors, with periods of over- and under-predictions.

Accelerated experiments. To make it possible to run a large number of experiments, each experiment is an accelerated run of the workload. Specifically, the time frame is accelerated by a factor of 60, making 1 minute of experiment execution represent 1 hour of real time. We accelerate jobs' execution times by reducing the size of the inputs. We also constrained the system to not migrate VMs more often than once a minute (1 hour in real time). Note that this is pessimistic in two ways: (a) the migration time did not scale down linearly; thus, migration overheads would have been smaller in un-accelerated execution; and (b) lower migration overheads would have allowed our policies to adjust allocations more often (e.g., every 15 minutes). The validation experiment shown in Figure B.6 was run in real time (that is, it was not accelerated).

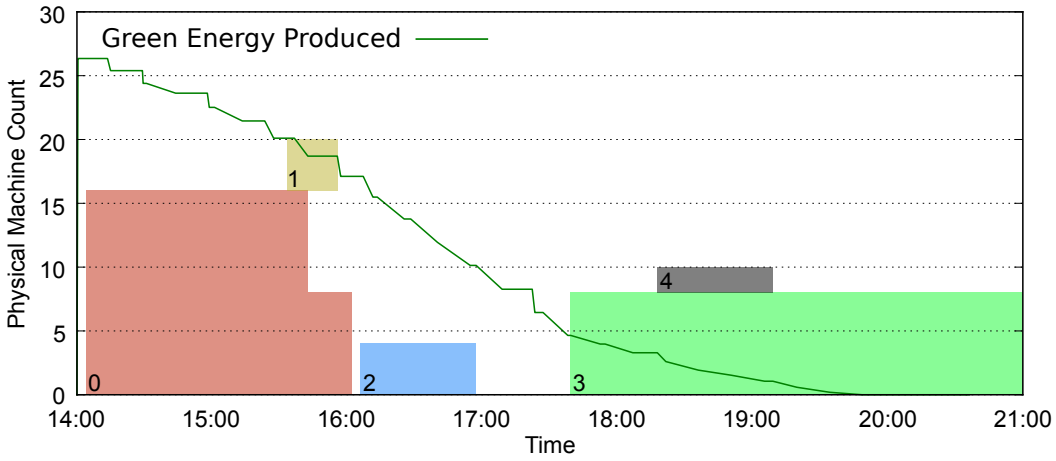


Figure B.6: Green energy production and server allocations during a 7-hour validation run of Grid5k.

Experimental setup. We set the scheduling horizon to 24 hours, and divide this horizon into 15-minute epochs.[4] We use the Gurobi solver [66] to solve the optimization problems for Aggressive and Offline. In our experiments, solutions of Aggressive take less than 2 seconds to execute. Offline runs completely offline to determine the best case for comparison, so we let it execute for 30 minutes in our experiments.

We use the same amount of maximum acceptable runtime slowdown for all jobs for Reactive, Offline, and Aggressive. The default is 10%, although we also study settings in the 0%–50% range. For Nebulous, we use a maximum reduced average resource allocation of 10%. In our experiments, *GreenPar is able to satisfy all performance SLAs.*

Finally, to make efficient use of resources, GreenPar only allocates powers-of-2 servers to each job, so that the number of VMs per server for a job is always the same. Also, GreenPar cannot allocate more servers to a job than the number of VMs in that job, and cannot consolidate more than 4 VMs onto a single server. All of these limit the flexibility with which GreenPar can adjust jobs' server allocations, but are limitations of our experimental setup rather than fundamental to GreenPar itself. GreenPar would perform better if jobs could dynamically adapt to arbitrary server allocations.

[4]The epoch duration can involve tradeoffs between scheduling overhead and faster response to changing conditions. However, our evaluation is insensitive to this parameter, since migration overheads are small and near-future prediction of green-energy is relatively accurate.

B.5 Validation and Evaluation

In this section, we first present a validation experiment of GreenPar running on Parasol with real energy production and prediction. We then present our evaluation of the four GreenPar policies using our scaled-down solar traces.

Validation. Figure B.6 shows the results from a validation experiment using Aggressive and the Grid5k workload. The experiment ran over 7 hours on Parasol, using real solar energy production and live predictions of solar energy production. In Figure B.6, each box shows the server allocation of a job over its lifetime. The green line depicts the green energy available over time. The numbers in the colored boxes are the job identifiers. In this execution, jobs 0, 1, and 2 requested 8, 2, and 2 servers, respectively. However, because of the high green energy availability, GreenPar doubled the allocations of each of these jobs to reduce their runtimes. When the production of green energy decreased toward the end of job 0's execution, GreenPar adjusted by reducing job 0's server allocation to 8 servers. GreenPar was able to reduce the runtimes of these jobs by an average of 23.8%.

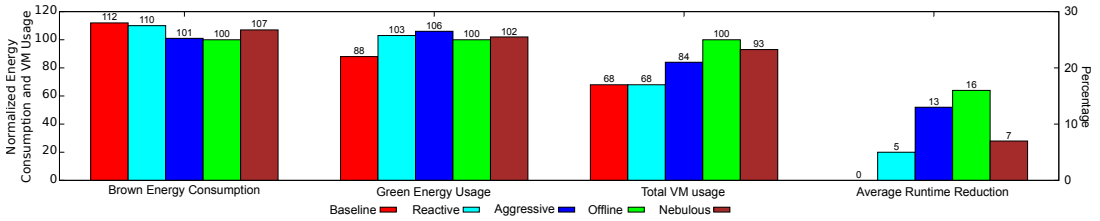


Figure B.7: Comparison of policies for Grid5k workload running on the High day.

Evaluation. Figure B.7 compares the brown energy consumption, green energy usage, total number of VMs, and average runtime reduction achieved by Baseline and the four GreenPar policies when running Grid5k for the High day. The total number of VMs is the sum of all VMs used by all the jobs; a larger number of VMs means that jobs were executed with greater parallelism. The brown energy consumption, green energy usage, and total number of VMs are normalized compared to those of Offline (Y-axis on the left). Average runtime reduction is the percentage reduction of jobs compared to when they are scheduled by Baseline (Y-axis on the right).

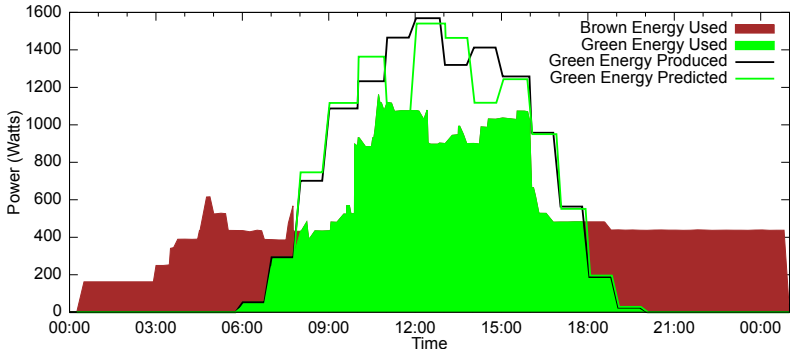


Figure B.8: Behavior of Baseline, Grid5k, High day.

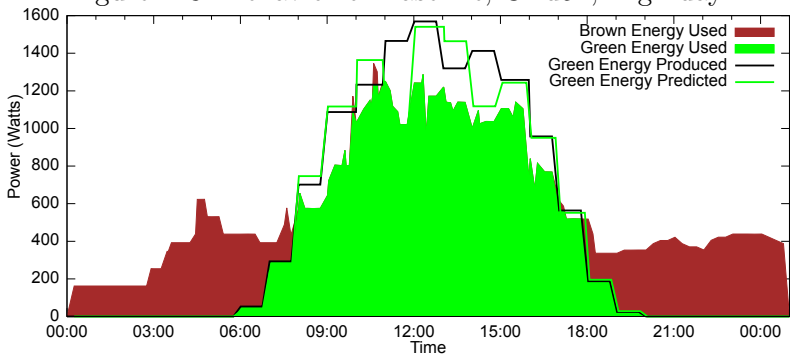


Figure B.9: Behavior of Reactive, Grid5k, High day.

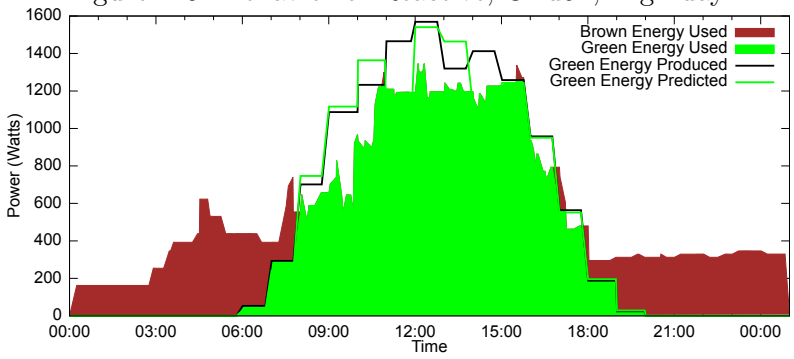


Figure B.10: Behavior of Aggressive, Grid5k, High day.

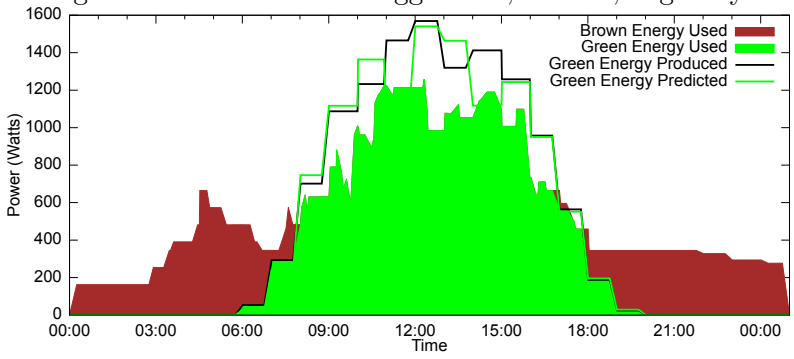


Figure B.11: Behavior of Offline, Grid5k, High day.

These results show that all four policies can reduce brown energy consumption *and* reduce job runtimes. They achieve these savings by slowing down job execution when only brown energy is available, and speeding up job execution when green energy is available. While Reactive only reduces brown energy consumption by a small amount (∼2% reduction), it successfully uses more green energy to reduce average job runtimes by 5%. This means that even when the policy has no knowledge of job arrivals and runtimes, it can leverage green energy to improve runtimes.

Aggressive is better than Reactive because it uses knowledge of job runtimes and predictions of green energy availability in the near future to more aggressively start jobs with higher levels of parallelism; the same set of jobs is run with 84 VMs under Aggressive compared to 68 under Reactive. This allows Aggressive to successfully use more green energy, less brown energy, and achieve shorter job runtimes.

In fact, Aggressive behaves almost as well as Offline. It uses only slightly more brown energy (1% more), and almost matches Offline in the average reduction of runtimes (13% vs. 16%). The advantages of Offline arise from its oracle knowledge of the future. Aggressive immediately runs an arriving job with a larger number of VMs if it expects green energy to be available. However, this may prevent it from running the next arriving job with a large number of VMs–there may be insufficient resources left–even though the subsequent job has better speedup characteristics. Offline can make the correct choice given its knowledge of the future. Interestingly, Offline uses less green energy because it uses it more efficiently; it allocates the energy to the jobs with the best speedup, allowing these jobs to finish more quickly and stop consuming energy.

Finally, Nebulous behaves slightly better than Reactive even though it is given only hints about speedup. This is because it aggressively starts jobs with more VMs when expecting excess green energy in the future. Thus, Nebulous uses more green energy to speedup jobs than Reactive.

On the other hand, Nebulous cannot match Aggressive because its heuristic for choosing the number of VMs is less accurate. In our implementation, Nebulous only chooses the larger number of VMs (per requested server) if it expects to have excess green energy for the entire estimated runtime of a job. However, the job may finish

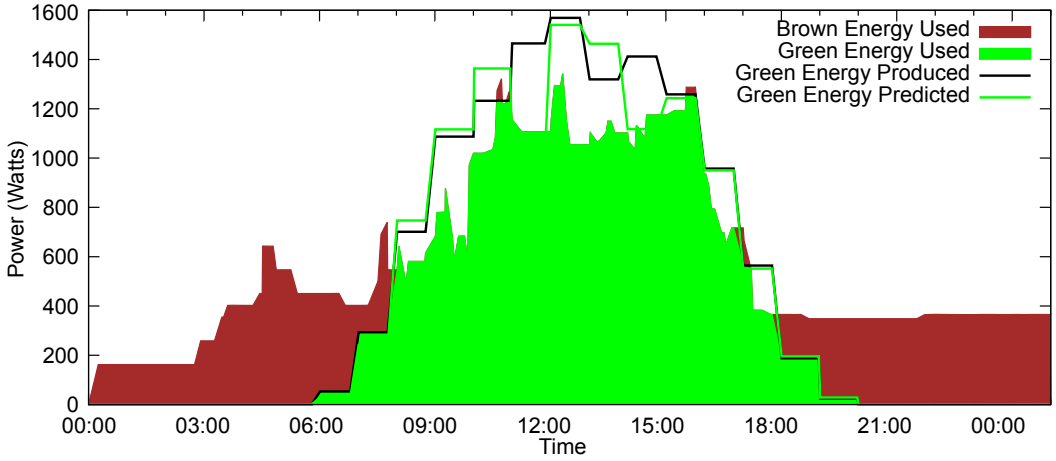


Figure B.12: Behavior of Nebulous, Grid5k, High day.

much faster when it is given extra resources. Thus, Nebulous's decisions are often overly conservative compared to Aggressive. Also, Nebulous's allocations of extra resources between multiple jobs are often not as good as Aggressive's allocations, because it does not have accurate speedup information.

Detailed power demand profiles for the five policies when running Grid5k for the High day are shown in Figures B.8–B.12. The first 6 hours are similar for all policies because there is no green energy production, and the jobs can only be slowed down by a maximum of 10%. In the daytime, when green energy becomes available, the green-energy-aware policies increase power consumption by either increasing the server allocations of active jobs (all four policies) and/or starting new jobs with more VMs, allowing larger than requested server allocations (Aggressive, Offline, and Nebulous). As mentioned earlier, Offline consumes less green energy even though it runs jobs with more parallelism (greater number of VMs) because it allocates the green energy to the most efficient jobs, leading to the greatest reduction in job runtimes. Finally, Aggressive and Offline can reduce brown energy consumption at night (right portion of the figures) either because jobs finished faster (they were run with more parallelism when green energy was available) or enough performance slack has been accumulated so that some jobs can be consolidated onto fewer physical machines compared to Baseline. Reactive and Nebulous achieve smaller savings.

For a more detailed look at the behavior of a green-energy-aware policy, Figure B.13

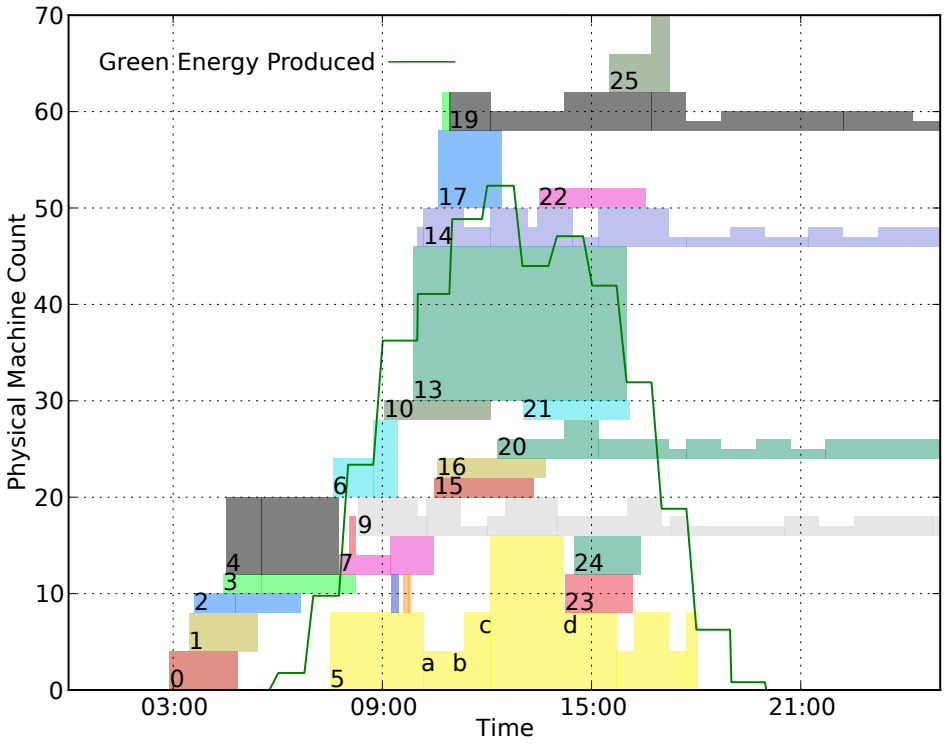


Figure B.13: Aggressive's server allocation for Grid5k.

shows the allocation of physical servers to jobs for Aggressive over time. The green line, boxes, and numerical labels are the same as for Figure B.6. A closer look at job 5 reveals several interesting points. First, Aggressive starts job 5, which requested 4 servers, with 16 VMs because it is expecting to have excess green energy. This allows Aggressive to run job 5 on more servers, consuming green energy to speedup job execution. Initially, Aggressive allocates 8 servers to the job (2 VMs per dual-core machine). At point 'a', job 14 arrives with better speedup characteristics. Thus, Aggressive shrinks job 5's server allocation to 4. Later, as more green energy becomes available, Aggressive increases job 5's allocation to 8 (point 'b') and then to 16 (point 'c'). Note that, with 16 servers, job 5 is being run with only 1 VM per server. This allocation achieves limited speedup since the second core on each server is frequently idle. However, since the green energy is "free" in this case, it is beneficial to use this energy whenever some speedup is still possible. As green energy production decreases over the day, and other jobs enter the system, Aggressive again decreases job 5's allocation (point 'd' and beyond). In comparison, job 5 ran for a much longer time with a static allocation of 4 servers under

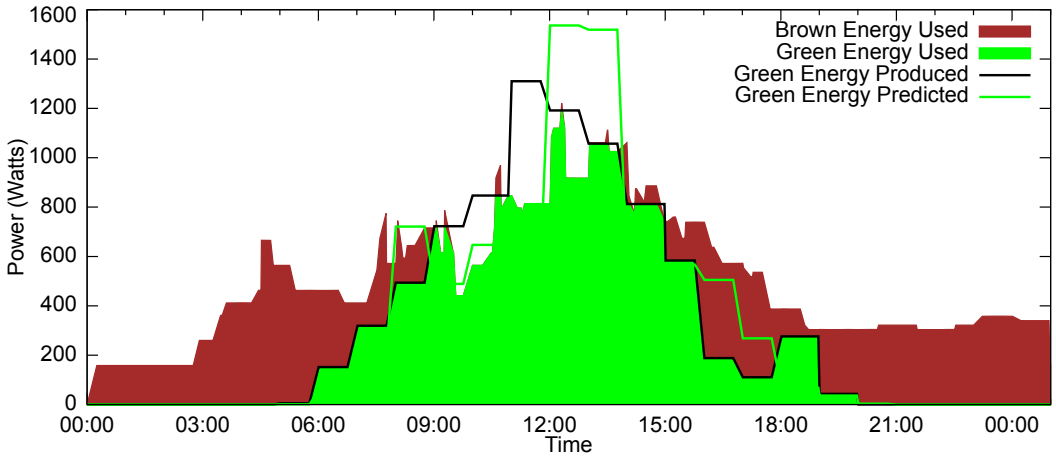


Figure B.14: Behavior of Aggressive, Grid5k, Medium day.

Baseline (not shown here).

Impact of green energy availability. We also run Baseline and Aggressive for the Medium and Low days. Figure B.14 shows the behavior of Aggressive for the Medium day. As previously mentioned, this day has significant mis-predictions of the solar energy production, including both under- and over-predictions. Over-predictions can lead Aggressive to use brown energy unnecessarily (and less efficiently). On the other hand, under-predictions can lead it to waste green energy. Overall, Aggressive still reduces the average runtime by 27% despite the mis-predictions. However, it uses 5% more brown energy than Baseline. On the Low day, Aggressive and Baseline behave the same because there was little green energy to leverage.

We also study Aggressive and Baseline for different green energy production capacities. Scaling the solar power system does not change the overall behaviors. On the High day, a peak solar production capacity of 125% of the peak datacenter power consumption leads to an 11.3% reduction in the brown energy consumption (compared to 10% above), and a 12.9% reduction in the average runtime (compared to 12% above). Scaling down the peak solar energy production to 75% of the peak datacenter power consumption leads to an 8.6% reduction in the brown energy consumption and a 10.3% reduction in the average runtime.

Impact of application speedup. GreenPar's efficacy can be strongly affected by the speedup characteristics of jobs in the workloads. To see the impact of having jobs with

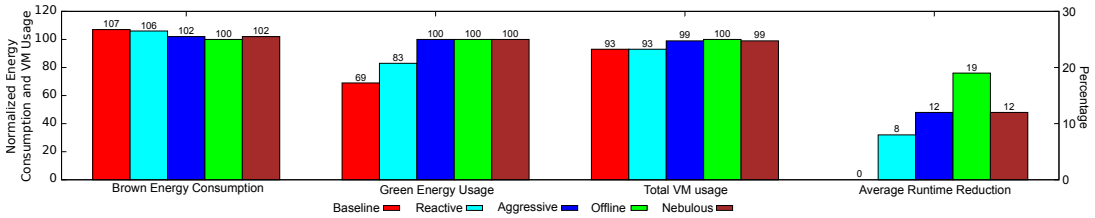


Figure B.15: Comparison of policies for the Intrepid workload running on the High day.

better speedups, we constructed a Grid5k workload using only EP jobs. When an EP job is run with 16 VMs, it achieves much better speedup (much closer to linear) than that shown in Figure B.4. When running this workload, Aggressive reduces the brown energy consumption by 15% while reducing the average runtime by 40% compared to Baseline.

Impact of performance SLA. When we disallow GreenPar to slow down job execution, i.e., set F to 1, Aggressive can still reduce the brown energy consumption by 8% and reduce the average runtime by 12%. If we instead allow a 50% slow down factor, Aggressive consumes 16% less brown energy than Baseline while reducing the average runtime by 9%. In this case, Aggressive's lack of knowledge about future job arrivals limits its performance. Specifically, before the green energy production starts to ramp up, Aggressive (aggressively) slows down the execution of active jobs, planning to use the green energy produced later on to recoup the performance loss. However, new jobs arriving later in the day will consume some of the green energy, leading Aggressive to consume brown energy instead to catch up; the catching up is typically done at lower efficiency since execution at larger server allocations is typically less efficient.

Impact of runtime mis-estimations. Aggressive outperforms Baseline but relies on knowing jobs' runtimes. We have studied their relative performance where the user provided job runtimes to Aggressive are inaccurate by ±25%. These inaccuracies did not significantly affect Aggressive since it recomputes a new schedule whenever a job completes anyways. Thus, the performance differences between Aggressive and Baseline were close to those presented above.

Impact of datacenter utilization. Under Baseline, Grid5k produces a datacenter utilization of approximately 33%. The traces we use to construct our workloads exhibit

similar utilizations. Nevertheless, we also run experiments with higher utilizations to gauge GreenPar's behavior under heavy load. Specifically, we doubled (2x) and tripled (3x) Grid5k, causing lengthy queue build up over time in Baseline. At 2x load, 12% more jobs complete under Aggressive than Baseline. This is because Aggressive is still occasionally able to give some jobs extra resources, allowing them to complete faster and reduce contention. At 3x load, the two policies perform the same. Thus, we conclude that GreenPar's benefits tail off as utilization becomes very high. *However, GreenPar does not harm performance even at very high loads.*

Impact of workload characteristics. Figure B.15 compares the brown energy consumption, green energy usage, total number of VMs, and average runtime reduction achieved by the policies when running Intrepid for the High day. These results show the same trends as those observed for Grid5k. Interestingly however, Nebulous's performance almost matches Aggressive. This is because most jobs are short so that they easily fit within periods of high solar energy. Thus, both policies make similar decisions about the number of VMs to start per job.

B.6 Conclusions

In this chapter, we proposed GreenPar, a scheduler for parallel HPC workloads running in datacenters partially powered by solar energy. GreenPar seeks to maximize the use of green energy to reduce job runtimes and brown energy consumption. We have implemented and evaluated it using realistic workloads, running on a real solar-powered datacenter. Our results show that GreenPar can increase green energy consumption and decrease brown energy consumption, while reducing average runtime at the same time. The results also show that an online policy using information about job speedups, runtimes, and predictions of solar energy production can come close to matching an offline policy that additionally has perfect information about future job arrivals and green energy production. Finally, GreenPar still provides benefits (albeit smaller ones) when it only has rough hints for speedup and runtime.

We conclude that GreenPar can become an important software component in green

datacenters that run HPC workloads, helping to improve the sustainability of our Information Technology ecosystem.

Bibliography

[1] AISO.net. Web Hosting as Nature Intended, 2012. `http://www.aiso.net`.

[2] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper. Free Lunch: Exploiting Renewable Energy for Computing. In *Workshop on hot topics in operating systems (HotOS)*, 2011.

[3] B. Aksanli et al. Utilizing Green Energy Prediction to Schedule Mixed Batch and Service Jobs in Data Centers. In *HotPower*, 2011.

[4] B. Aksanli, J. Venkatesh, L. Zhang, and T. Rosing. Utilizing Green Energy Prediction to Schedule Mixed Batch and Service Jobs in Data Centers. In *Workshop on Power-Aware Computing and Systems (HotPower)*, 2011.

[5] S. Albers and H. Fujiwara. Energy-Efficient Algorithms for Flow Time Minimization. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2006.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (DCTCP). In *ACM SIGCOMM*, pages 63–74, 2010.

[7] Amazon EC2. `http://aws.amazon.com/ec2`, Retrieved on May 2013.

[8] Amazon EC2 Pricing. `http://aws.amazon.com/ec2`, Retrieved on February 2013.

[9] Apache Lucene. `http://lucene.apache.org/`. Retreived July 2014.

[10] Apple Inc. Apple Environmental Responsibility Report. `https://www.apple.com/environment/reports/docs/apple_environmental_responsibility_report_0714.pdf`, 2014.

[11] Apple Inc. Apple and the Environment. `http://www.apple.com/environment/renewable-energy`, Retrived in February 2013.

[12] N. Bansal, K. Dhamdhere, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.

[13] 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 *Symposium on Operating Systems Principles (SOSP)*, 2003.

[14] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[15] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12), December 2007.

[16] L. A. Barroso and U. Hlzle. *The Datacenter as a Computer*. Morgan & Claypool, 2009.

[17] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive system. In *SIGOPS European Workshop*, 2000.

[18] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, C. D. Antonopoulos, and M. Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 33(10-11):700–719, 2007.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[20] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[21] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In *Supercomputing (SC)*, 2003.

[22] D. Breitgand and A. Epstein. SLA-Aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds. In *International Symposium on Integrated Network Management*, 2011.

[23] M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Management Science*, 42(2):269–285, 1993.

[24] B. Cahoon, K. S. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems (TOIS)*, 18(1):1–43, 2000.

[25] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 225–236, 2012.

[26] J. Chen and L. K. John. Efficient Program Scheduling for Heterogeneous Multi-core Processors. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 927–930, New York, NY, USA, 2009. ACM.

[27] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[28] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 57–66, 2008.

[29] J. Corbalan and J. Labarta. Improving Processor Allocation Through Run-time Measured Efficiency. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[30] G. Cortazar, M. Gravet, and J. Urzua. The valuation of multidimensional american real options using the LSM simulation. *Computers and Operations Research*, pages 113–129, 2006.

[31] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *ACM International Conference on Supercomputing (ICS)*, pages 157–166, 2006.

[32] Data Center Knowledge. Data Centers Scale Up Their Solar Power, 2012. `http://www.datacenterknowledge.com/archives/2012/05/14/data-centers-scale-up-their-solarpower`.

[33] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[35] Delft University of Technology. The Grid Workloads Archive. `http://gwa.ewi.tudelft.nl/pmwiki/pmwiki.php?n=Workloads.Gwa-t-2`.

[36] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[37] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[38] N. Deng, C. Stewart, and J. Li. Concentrating Renewable Energy in Grid-Tied Datacenters. In *Proceedings of the International Symposium on Sustainable Systems and Technology*, 2011.

[39] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[40] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-power Modes for Main Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 225–238, New York, NY, USA, 2011. ACM.

[41] T. Desell, K. El Maghraoui, and C. A. Varela. Malleable Applications for Scalable High Performance Computing. *Cluster Computing*, 10(3), 2007.

[42] EcobusinessLinks. Green Hosting - Sustainable Solar & Wind Energy Web Hosting. `http://www.ecobusinesslinks.com/green_webhosts/`, Retrieved on February 2013.

[43] Energy Information Administration. Average Retail Price of Electricity to Ultimate Customers by End-Use Sector, by State. `http://www.eia.gov/electricity/monthly/epm_table_grapher.cfm?t=epmt_5_6_b`, Retrieved on January 2013.

[44] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2011.

[45] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Communications of the ACM*, 52(12):48–57, 2009.

[46] D. Feitelson. Parallel Workload Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/l_anl_int/index.html`.

[47] D. Feitelson. *A Survey of Scheduling in Multiprogrammed Parallel Systems*. Research report. IBM T.J. Watson Research Center, 1994.

[48] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 261–272, April 2013.

[49] B. Forrest. Bing and Google Agree: Slow Pages Lose Users, 2009.

[50] A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Are sleep states effective in data centers? In *Green Computing Conference (IGCC), 2012 International*, pages 1–10, June 2012.

[51] R. Ge and K. Cameron. Power-aware Speedup. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[52] Global Reporting Initiative. `https://www.globalreporting.org`.

[53] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[54] I. Goiri, F. Julià, R. Nou, J. L. Berral, J. Guitart, and J. Torres. Energy-Aware Scheduling in Virtualized Datacenters. In *IEEE International Conference on Cluster Computing*, September 2010.

[55] I. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini. Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy. In *International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 2013.

[56] I. Goiri, K. Le, M. E. Haque, R. Beauchea, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *Supercomputing (SC)*, 2011.

[57] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *European Conference on Computer Systems (EuroSys)*, 2012.

[58] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *Proceedings of the International Sympoisum on Computer Architecture*, June 2011.

[59] Green House Data. An Economically Responsible Data Center, 2012. `http://www.greenhousedata.com`.

[60] Green House Data. `http://www.greenhousedata.com/about-us/green-data-center`, Retrived in February 2013.

[61] P. Greenhalgh. Big. little processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[62] GreenQloud, 2013. `http://greenqloud.com`.

[63] GreenQloud, 2013. `http://greenqloud.com/pricing`.

[64] Grid'5000. Grid'5000 Experimentation Platform. `www.grid5000.fr`.

[65] R. Guida. Parallelizing a computationally intensive financial R application with zircon technology. In *The R User Conference*, 2010.

[66] Gurobi Optimization Inc. Gurobi Optimization. `http://www.gurobi.com`.

[67] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[68] J. Hamilton. The cost of latency, 2009. `http://per\discretionary{-}{}{}spect\discretionary{-}{}{}ives\discretionary{-}{}{}.mvdirona.com\discretionary{-}{}{}/2009/10/31\discretionary{-}{}{}/\discretionary{-}{}{}TheCostOfLatency.aspx`.

[69] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 161–175, 2015.

[70] M. E. Haque, K. Le, I. Goiri, R. Bianchini, and T. D. Nguyen. Providing Green SLAs in High Performance Computing Clouds. In *International Green Computing Conference (IGCC)*, 2013.

[71] S. Hazelhurst. Scientific Computing Using Virtual High-performance Computing: A Case Study Using the Amazon Elastic Computing Cloud. In *Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)*, 2008.

[72] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[73] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1263–1279, 2008.

[74] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[75] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[76] U. Holzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 2010.

[77] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[78] Q. Huang, S. Su, J. Li, P. Xu, K. Shuang, and X. Huang. Enhanced Energy-Efficient Scheduling for Parallel Applications in Cloud. In *Symposium on Cluster, Cloud, and Grid Computing (CCGRID)*, 2012.

[79] W. Huang, J. Liu, B. Abali, and D. K. Panda. A Case for High Performance Computing with Virtual Machines. In *International Conference on Supercomputing (ICS)*, 2006.

[80] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM '13*, 2013.

[81] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[82] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *ACM European Conference on Computer Systems (EuroSys)*, pages 155–168, 2013.

[83] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *ASPLOS*, April 2016.

[84] M. Jeon, S. Kim, S.-W. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: taming tail latencies in web search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 253–262, 2014.

[85] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 236–246, 2005.

[86] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and

C. Bouras. Platform for distributed 3D gaming. *International Journal of Computer Games Technology*, 2009:1:1–1:15, 2009.

[87] K. Kant, M. Murugan, and D. H.C.Du. Willow: A Control System for Energy and Thermal Adaptive Computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[88] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan & Claypool, 2007.

[89] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2015.

[90] S. Kirkpatrick. Optimization by Simulated Annealing: Quantitative Studies. *Journal of Statistical Physics*, 34(5), 1984.

[91] S. Klingert, T. Schulze, and C. Bunse. GreenSLAs for the Energy-efficient Management of Data Centres. In *International Conference on Energy-Efficient Computing and Networking*, 2011.

[92] W. Ko, M. N. Yankelevsky, D. S. Nikolopoulos, and C. D. Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 130, 2002.

[93] J. Koomey. *Growth In Data Center Electricity Use 2005 To 2010*. Oakland, CA, 2011.

[94] J. Koomey. Growth in Data Center Electricity Use 2005 to 2010, 2011. Analytic Press.

[95] A. Krioukov, S. Alspaugh, P. Mohan, S. Dawson-Haggerty, D. E. Culler, and R. H. Katz. Design and Evaluation of an Energy Agile Computing Cluster. Technical Report EECS-2012-13, University of California at Berkeley, January 2012.

[96] A. Krioukov, C. Goebel, S. Alspaugh, Y. Chen, D. Culler, and R. Katz. Integrating Renewable Energy Using Data Analytics Systems: Challenges and Opportunities. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, March 2011.

[97] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[98] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 64–75, 2004.

[99] K. Le, R. Bianchini, M. Martonosi, and T. D. Nguyen. Cost- And Energy-Aware Load Distribution Across Data Centers. In *Workshop on Power Aware Computing and Systems*, October 2009.

[100] K. Le, O. Bilgir, R. Bianchini, M. Martonosi, and T. D. Nguyen. Capping the Brown Energy Consumption of Internet Services at Low Cost. In *International Green Computing Conference (IGCC)*, 2010.

[101] K. Le, J. Zhang, J. Meng, Y. Jaluria, T. D. Nguyen, and R. Bianchini. Reducing Electricity Cost Through Virtual Machine Placement in High Performance Computing Clouds. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2011.

[102] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *International Symposium on Computer Architecture (ISCA)*, pages 270–279, 2010.

[103] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 227–242, 2009.

[104] C. Li, A. Qouneh, and T. Li. iSwitch: Coordinating and Optimizing Renewable Energy Powered Server Clusters. In *International Symposium on Computer Architectur (ISCA)*, 2012.

[105] C. Li, R. Wang, T. Li, D. Qian, and J. Yuan. Managing green datacenters powered by hybrid renewable energy systems. In *International Conference on Autonomic Computing (ICAC)*, 2014.

[106] D. Li, B. De Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[107] T. Li, D. P. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *ACM/IEEE Conference on Supercomputing*, pages 53:1–11, 2007.

[108] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[109] D. Lo, L. Cheng, R. Govindaraju, L. A. d. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *In Proceeding of the 41st Annual International Symposium on Comput er Architecuture*, 2014.

[110] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[111] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 50–61, 2001.

[112] Lucene Nightly Benchmarks. http://people.apache.org/~mikemccand-/lucenebench. Retreived June 2014.

[113] A. Lukefahr, S. Padmanabha, R. Das, R. Dreslinski, Jr., T. F. Wenisch, and S. Mahlke. Heterogeneous Microarchitectures Trump Voltage Scaling for Low-power Cores. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 237–250, 2014.

[114] J. Mankoff, R. Kravets, and E. Blevis. Some Computer Science Issues in Creating a Sustainable World. *Computer*, 41(8), 2008.

[115] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[116] Mathworks. `http://www.mathworks.com/discovery/pid-tuning.html`, 2015. Retreived on July 2015.

[117] Maxim Integrated. Switching Between Battery and External Power Sources. `http://www.maximintegrated.com/app-notes/index.mvp/id/1136`, Retrieved on February 2013.

[118] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[119] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, 2011.

[120] R. Miller. Data Centers Scale Up Their Solar Power, May 2012.

[121] NASA Advanced Supercomputing Division, Retrieved on December 2013. `www.nas.nasa.gov/publications/npb.html`.

[122] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1996.

[123] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, and J. Underwood. Power Routing: Dynamic Power Provisioning in the Data Center. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[124] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang. Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicore in Warehouse Scale Computers. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[125] A. Porterfield, S. Olivier, S. Bhalachandra, and J. Prins. Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs. In *International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2013.

[126] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125, 2011.

[127] Qualcomm. Snapdragon 810 processors, 2014.

[128] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, 2011.

[129] S. Ren, Y. He, S. Elnikety, and K. McKinley. Exploiting processor heterogeneity in interactive services. In *USENIX International Conference on Autonomic Computing (ICAC)*, pages 45–58, 2013.

[130] S. Ren, Y. He, and K. McKinley. A theoretical foundation for scheduling and designing heterogeneous processors for interactive applications. *The International Symposuium on Distributed Computing (DISC)*, LNCS 8784:152–166, 2014.

[131] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding Energy Consumption in Large-Scale MPI Programs. In *Supercomputing (SC)*, 2007.

[132] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems*, 30(2):6:1–38, 2012.

[133] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Conference*, 2009.

[134] M. Shantharam, Y. Youn, and P. Raghavan. Speedup-Aware Co-Schedules for Efficient Workload Management. *Parallel Processing Letters*, 23(02), 2013.

[135] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing Server Clusters on Intermittent Power. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[136] N. Sharma, J. Gummeson, D. Irwin, and P. Shenoy. Cloudy Computing: Leveraging Weather Forecasts in Energy Harvesting Sensor Systems. In *International Conference on Sensor Mesh and Ad Hoc Communications and Networks (SECON)*, 2010.

[137] C. Stewart and K. Shen. Some Joules Are More Precious Than Others: Managing Renewable Energy in the Datacenter. In *Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[138] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on

CMPs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, 2008.

[139] H. Sun, Y. Cao, and W. J. Hsu. Efficient Adaptive Scheduling of Multiprocessors with Stable Parallelism Feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):594–607, April 2011.

[140] N. Tizon, C. Moreno, M. Cernea, and M. Preda. MPEG-4-based adaptive remote rendering for video games. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, 2011.

[141] UK Government. Carbon Reduction Commitment. `http://www.carbonreductioncommitment.info/`.

[142] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with aspen. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–23, 2007.

[143] G. Utrera, J. Corbalan, and J. Labarta. Implementing Malleability on MPI Jobs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[144] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[145] K. Van Craeynest and L. Eeckhout. Understanding Fundamental Design Choices in single-ISA Heterogeneous Multicore Architectures. *ACM Trans. Archit. Code Optim.*, 9(4):32:1–32:23, Jan. 2013.

[146] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In

Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[147] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Middleware*, 2008.

[148] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 75–84, 2009.

[149] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[150] Wikipedia: Database download. http://en.wikipedia.org/-wiki/wikipedia:database_download#english-language_wikipedia/. Retreived May 2014.

[151] D. Wong and M. Annavaram. KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 119–130, Dec 2012.

[152] T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, 2006.

[153] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

[154] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.