# Workload Interleaving with Performance Guarantees in Data Centers

Feng Yan

Williamsburg, VA

Bachelor of Science, Northeastern University, 2008
Master of Science, College of William and Mary, 2011

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
May 2016

# APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Feng Yan

Approved by the Committee, April 2016

Committee Chair
Professor Evgenia Smirni, Computer Science
The College of William and Mary

Professor Weizhen Mao, Computer Science
The College of William and Mary

Professor Qun Li, Computer Science
The College of William and Mary

Professor Haining Wang, Electrical and Computer Engineering
University of Delaware

Dr. Alma Riska, NetApp Cooperation

# ABSTRACT

In the era of global, large scale data centers residing in clouds, many applications and users share the same pool of resources for the purposes of reducing energy and operating costs, and of improving availability and reliability. Along with the above benefits, resource sharing also introduces performance challenges: when multiple workloads access the same resources concurrently, contention may occur and introduce delays in the performance of individual workloads. Providing performance isolation to individual workloads needs effective management methodologies. The challenges of deriving effective management methodologies lie in finding accurate, robust, compact metrics and models to drive algorithms that can meet different performance objectives while achieving efficient utilization of resources. This dissertation proposes a set of methodologies aiming at solving the challenging performance isolation problem in workload interleaving in data centers, focusing on both storage components and computing components.

At the storage node level, we focus on methodologies for better interleaving user traffic with background workloads, such as tasks for improving reliability, availability, and power savings. More specifically, a scheduling policy for background workload based on the statistical characteristics of the system busy periods and a methodology that quantitatively estimates the performance impact of power savings are developed. At the storage cluster level, we consider methodologies on how to efficiently conduct work consolidation and schedule asynchronous updates without violating user performance targets. More specifically, we develop a framework that can estimate beforehand the benefits and overheads of each option in order to automate the process of reaching intelligent consolidation decisions while achieving faster eventual consistency.

At the computing node level, we focus on improving workload interleaving at off-the-shelf servers as they are the basic building blocks of large-scale data centers. We develop priority scheduling middleware that employs different policies to schedule background tasks based on the instantaneous resource requirements of the high priority applications running on the server node. Finally, at the computing cluster level, we investigate popular computing frameworks for large-scale data intensive distributed processing, such as MapReduce and its Hadoop implementation. We develop a new Hadoop scheduler called DyScale to exploit capabilities offered by heterogeneous cores in order to achieve a variety of performance objectives.

# TABLE OF CONTENTS

# ACKNOWLEDGMENTS

I would like to thank everyone who has helped with this thesis, especially:

My adviser, Professor Evgenia Smirni, who encouraged me to develop a spirit of adventure in regard to research and continuously taught me new knowledge and brought new excitement and passion all the way. Without her patient guidance and persistent help, this dissertation would not have been possible.

My committee members, Professor Weizhen Mao, Professor Qun Li, Professor Haining Wang, Dr. Alma Riska for their great support and insightful feedback and comments.

Dr. Ludmila Cherkasova for offering me the internship opportunity at HP Labs and leading me on an exciting project. Dr. Yuxiong He for offering me the internship opportunity at Microsoft Research and leading me on an exciting project.

The faculty and staff at the Computer Science Department of the College of William and Mary. Special thanks to Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes for their considerate and efficient assistance.

Professor Giuliano Casale, Professor Xenia Mountrouidou, Dr. Lei Lu, Dr. Zhuoyao Zhang, Dr. Olatunji Ruwase, Dr. Trishul Chilimbi, Shannon Hughes, Ji Xue, Jiawei Wen and many other dear friends for their support and help.

Finally, my dear parents Yunpeng Yan and Li Feng, who brought me to this world and supported my study abroad; and to everyone in my family for their encouragement and support.

# LIST OF TABLES

# LIST OF FIGURES

Workload Interleaving with Performance Guarantees in Data Centers

# Chapter 1

# Introduction

From data centers to cloud and storage clusters, most of today's systems are highly distributed serving various applications that share their resources. Such resource sharing brings a host of benefits, for example, reduced energy, power and operating cost, improved availability, enhanced reliability to name a few. Resource sharing brings also performance challenges because when multiple workloads access the same resources, the potential resource contention may cause performance interference and delay individual workloads. Providing workload performance isolation requires effective resource management policies. To derive effective management policies, workload and other system conditions need to be monitored and timely communicated. This dissertation develops a set of accurate, robust, and compact metrics, as well as models and algorithms that aim to at provide performance isolation to concurrently executing workloads and at the same time achieve efficient utilization of system resources.

Workloads in large scale distributed systems can be characterized based on the type of necessary system components to serve them. Critical data center components can be classified into storage components and computing components. Storage components support data storage and movement. A typical distributed storage system may be

composed by synchronous dynamic random access memory(SDRAMs), solid state drives (SSDs), hard disk drives (HDDs). The workload at the storage components can be further viewed at the node level or at the cluster level. The workload at the storage node level is the work received or scheduled at each single storage node (e.g., a hard disk drive) and it does not involve communication between different nodes. Typical examples are garbage collection, data integrity check, and power savings. The workload at the storage cluster level involves multiple storage nodes and may need coordination management between these nodes, e.g., consolidating work from several nodes into a single one for increasing resource utilization. Computing components support computation related functionalities and usually are composed by single or multi-core units. Similarly, the workload can be viewed from the unit perspective, i.e., the work that is directly scheduled on each compute unit; or from the cluster level perspective, e.g., a MapReduce workload that is distributed into many computing units.

## 1.1 Problem Definition

The aim of this dissertation is to develop a set of metrics, models, and algorithms to achieve performance isolation of different workloads in today's highly distributed systems. More specifically, at the storage components, the focus is on providing answers to the following questions:

- At the storage node level, how can we efficiently schedule system background tasks without impacting user traffic? If power savings is yet another metric to be optimized, how can one capture the performance impact of power savings and make smart power saving decisions?
- At the storage cluster level, how can we make cluster consolidation decisions automatically with performance guarantees and can we achieve faster eventual

consistency without impacting user traffic?

Moving on to computing components, the focus of this dissertation is to answer the following questions:

- At the computing node level, are the available priority scheduling tools that come off-the-shelf able to handle dynamic workload well? How can we achieve performance guarantees based on the off-the-shelf priority scheduling tools?
- At the computing cluster level, how can one use the capabilities offered by heterogeneous processors to benefit specific applications such as MapReduce?

The above questions represent a variety of challenging problems in today's data centers. If these problems can be solved, the performance, efficiency, consistency, and robustness of data centers can be significantly improved.

## 1.2 Dissertation Contributions

This section briefly highlights the contributions of this thesis.

### 1.2.1 Efficient background task scheduling at the storage node level

Computer systems, in general, and storage systems, in particular, rely on meeting their performance, reliability, and availability targets via scheduling of management and maintenance activities as background tasks. Such tasks may cause significant delays to user workload if scheduled extemporaneously. In this dissertation, a user traffic aware scheduling policy is proposed [129, 117, 118]. It schedules background tasks based on the statistical characteristics of busy periods and aims at completing background work expediently. Extensive trace-driven simulations show that the scheduling policy is robust and that it succeeds in completing background work faster than common practices while impacting user performance minimally.

### 1.2.2 A Performance, Power, and Reliability Framework for Storage Systems

The biggest power consumer in data centers is the storage system. Coupled with the fact that disk drives are lowly utilized, disks offer great opportunities for power savings, but any power saving action should be transparent to user traffic. Estimating correctly the performance impact of power saving becomes crucial for the effectiveness of power saving.

PREFiguRE has been developed to harvest future idle intervals for power savings while meeting strict quality constraints [128, 127]: first, it contains potential delays in serving IO requests that occur during power savings since the time to bring up the disk is not negligible and second, it ensures that the power saving mechanism is triggered a few times only, such that the disk wear out due to powering up and down does not compromise the disk's lifetime. PREFiguRE is based on an analytic methodology that uses the histogram of idle times to determine schedules for power saving modes as a function of the above constraints. PREFiguRE facilitates analysis for the evaluation of the trade-offs between power savings and quality targets for the storage workload. Extensive experimentation on a set of enterprise storage traces illustrates PREFiguRE's effectiveness to consistently achieve high power savings without undermining disk reliability and performance.

### 1.2.3 Automating storage cluster consolidation at the storage cluster level

Work consolidation has drawn great attention in today's highly distributed data centers to improve load balancing and to optimize non-traditional performance measures such as power savings, e.g., it may be desirable to shut down a lowly utilized node by moving some or all of its work to another node.

A methodology is developed for distributed work consolidation that keeps track of the

workload in the various nodes of the cluster and makes intelligent decisions on how much work to move from a sender node to a receiver node in order to minimally "affect" the performance of the receiver node or alternatively limit any performance degradation due to consolidation in a controlled way [126, 125]. The proposed methodology is based on continuously monitoring the workload on sender and receiver nodes, collecting lightweight statistics in the form of histograms of coarse granularity, and deciding when and how to initiate the work transfer. Extensive experimentation using trace-driven simulation confirms the robustness of the methodology.

### 1.2.4 Fast eventual consistency at the storage cluster level

As data and its processing increasingly becomes more critical to enterprises and consumers alike, systems have started to cross the physical boundaries of a single data center. It is very common nowadays, for service providers and corporations to span systems and data across multiple geographic locations, with the goal of reducing the chance that the data or its services become unavailable in case of network, power, or other outages. With such architectures, comes the need to facilitate achievement of data redundancy and integrity while autonomously and transparently handling the added network delay during ingesting or updating data in the system. Systems have adopted the notion of *eventual consistency* which means that the targeted redundancy of data in the system is reached *asynchronously*, i.e., outside of the critical path, so that performance of user traffic is impacted minimally.

A scheduling framework is proposed that makes decisions about when to schedule the asynchronous tasks associated with new or updated data such that they are completed as soon as possible without violating user traffic quality targets [131, 130]. At the heart of the framework lies a learning methodology that extracts the characteristics of idle periods and infers the average amount of work to be done during idle periods so that asynchronous

tasks are completed transparently to the user. Extensive trace-driven evaluation shows the effectiveness and robustness of the proposed framework when compared to common practices.

### 1.2.5 Agile priority scheduling at the computing node level

As the need for scaled-out systems increases, it is paramount to architect them as large distributed systems consisting of off-the-shelf basic computing components known as compute or data nodes. These nodes are expected to handle their work independently, and often utilize off-the-shelf management tools, like those offered by Linux, to differentiate priorities of tasks. While prioritization of background tasks in server nodes takes center stage in scaled-out systems, with many tasks associated with salient features such as eventual consistency, data analytics, and garbage collection, the standard Linux tools such as `nice` and `ionice` fail to adapt to the dynamic behavior of high priority tasks in order to achieve the best trade-off between protecting the performance of high priority workload and completing as much low priority work as possible.

A solution provided by proposing a priority scheduling middleware that employs different policies to schedule background tasks based on the instantaneous resource requirements of the high priority applications running on the server node [124, 123]. The selection of policies is based on off-line and on-line learning of the high priority workload characteristics and the imposed performance impact due to low priority work. In effect, this middleware uses a *hybrid* approach to scheduling rather than a monolithic policy. It is prototyped and evaluated via measurements on a test-bed that show robustness as it effectively and autonomically changes the relative priorities between high and low priority tasks, consistently meeting their competing performance targets.

### 1.2.6 Efficient heterogeneous resource scheduling for MapReduce processing at the computing cluster level

The functionality of modern multi-core processors is often driven by a given power budget that requires designers to evaluate different decision trade-offs, e.g., to choose between many slow, power-efficient cores, or fewer faster, power-hungry cores, or a combination of them. A typical MapReduce workload contains jobs with different performance goals: large, batch jobs that are throughput oriented, and smaller interactive jobs that are response time sensitive.

A new Hadoop scheduler called DyScale is prototyped and evaluated that exploits capabilities offered by heterogeneous cores within a single multi-core processor for achieving a variety of performance objectives [122, 121, 120]. Heterogeneous multi-core processors enable creating virtual resource pools based on "slow" and "fast" cores for multi-class priority scheduling. Since the same data can be accessed with either "slow" or "fast" slots, spare resources (slots) can be shared between different resource pools. Using measurements on an actual experimental setting and via simulation, it verifies that heterogeneous multi-core processors are in favor as they achieve "faster" (up to 40%) processing of small, interactive MapReduce jobs, while offering improved throughput (up to 40%) for large, batch jobs. The performance benefits of DyScale versus the FIFO and Capacity job schedulers that are broadly used in the Hadoop community are evaluated.

## 1.3 Organization

This dissertation is organized as follows. Chapter 2 provides some general background knowledge and discusses related work. Chapter 3 describes an efficient background task scheduling framework in data centers. Chapter 4 presents a practical power savings strategy

for storage systems. Chapter 5 gives an automated storage cluster consolidation framework. Chapter 6 presents a fast eventual consistency framework with performance guarantees in data centers. Chapter 7 introduces an agile priority scheduling in data centers. Chapter 8 presents an efficient heterogeneous resource scheduling for MapReduce processing in data centers.

# Chapter 2

# Background and Related Work

This chapter provides some necessary general background knowledge and detailed related work.

## 2.1 Workload Interleaving at the Storage Node Level

This section introduces the workload interleaving problem at the storage node level. This dissertation focuses on general node level background tasks and power savings. The next sections introduce each of them in more details.

### 2.1.1 General Node Level Background Task

Today's systems complete most of their resource management and maintenance tasks in the background. At the storage node level, there is a plethora of activities that are executed asynchronously as background tasks [9, 102] aiming at improving performance, reliability, and availability [50, 10, 125, 51]. In addition, a large body of literature points out the existence of idle periods that are interleaved with periods of high utilization [40, 88, 30]. These idle periods offer an opportunity to serve tasks of low

priority, such as data synchronization, but may lead to performance degradation if a foreground task arrives while a background task is in service. This is the case especially in storage systems, because tasks are not instantaneously preemptable. As a result, the foreground requests could be unavoidably delayed when the system executes background tasks.

Conventionally, scheduling of background tasks is done using a non-work-conserving approach by delaying the execution of an outstanding background job for a fixed time when the system becomes idle of foreground work [30]. This technique avoids using short idle intervals to serve long background jobs and averts severe degradation in foreground performance. Approaches for adaptively determining the amount of time that the system should stay idle, while there is background work to be completed, are proposed for power saving in mobile devices by spinning-down their disks [28, 48]. pClock is a framework that allows multiple workloads to share storage while achieving performance isolation [46]. This approach may also be used to allocate spare system capacity to background jobs. Storage performance insulation has been achieved by co-scheduling time slices for each workload type [111].

In [71, 72], the authors propose a framework to estimate when and for how long to utilize idle periods in a system for processing low priority background tasks without violating pre-defined foreground performance targets. This is achieved by extending the non-work-conserving nature of background scheduling as first suggested in [40, 30]. The histogram of past idle intervals can be used to determine: (1) the amount of idle wait till a background task can start and (2) the amount of the expected idle time to be used for scheduling background tasks. The consequence is that the system may remain idle while background tasks are still outstanding after the estimated time to utilize an idle interval for background scheduling elapses. Key to the methodology developed in [71, 72] are the

statistical characteristics of idle times which are used for effective background task scheduling.

### 2.1.2 Storage power savings

For the last two decades there has been a host of work on power efficiency in computer systems and more specifically in disk drives. A comprehensive comparison of power saving algorithms for disk drives on personal computers is presented in [29]. The algorithms are evaluated based on trace driven simulation for two known disk drive models. The baseline used for power savings assumes a priori knowledge of the idle interval duration. The compared algorithms vary based on when and for how long a disk is placed in power savings. A fundamental difference with the work presented in this dissertation is that these algorithms apply to personal and not enterprise systems, therefore no power, performance, or reliability guarantees are provided.

In [37] a Markov Model of a cluster of disks is used to predict disk idleness and to schedule the spin down of disks for power savings. This model is based on two states, ON and OFF, and a prediction mechanism that relies on a probability matrix. Simulations using DiskSim with synthetic and real workloads show that the Markov Model has 87.5% prediction accuracy, reduces energy by 35.5%, performs better than other multi-speed models, and has a negligible performance penalty (less than 1%). The analytical model introduced in [41] is applied to predict the idle interval duration in order to spin down a disk for power savings. The Poisson assumption used in this paper is questionable, especially given the bursty nature of real traces [88]. For this analytical model a "critical rate" is defined as the number of accesses per unit time for which it is more power efficient to leave the disk active than spin it down. The above models are useful for offline disk spin down policies but not for anticipating workload changes on the fly that are necessary for the

development of online algorithms.

An adaptive algorithm based on the idea of "sessions" is presented in [64]. A session is similar to a busy period. Different sessions are separated by intervals of inactivity of duration $\tau$. The inactivity period is defined by monitoring and adaptation of the algorithm, i.e., increase or decrease $\tau$ based on the characteristics of inactivity periods. The algorithm does not minimize energy consumption compared to other adaptive algorithms, but it reduces power while preserving performance and reliability. However, no specific guarantees are given for the performance and reliability of this power saving algorithm. A Dynamic Power Management (DPM) algorithm is introduced in [53] that extends the power savings states from idle and busy, to multiple power-saving states based on a stochastic optimization. This algorithm has the best power savings, i.e., 25% less, and best performance, i.e., 40% less, compared to other DPM algorithms. It is based on online observations and learning of the probabilistic length of an interval. The effects of power management on disk request latency for personal computers are studied in [85]. The authors find the upper bound of IO request latencies to demonstrate the worst case scenario and how to handle it with efficient system design. A simple adaptive power management algorithm is presented that predicts the duration of the next idle period based on the previous one. Immediate shutdown of disks is studied, and the authors conclude that even though it increases power savings, it may also increases latency.

## 2.2 Workload Interleaving at the Storage Cluster Level

At the storage cluster level, this dissertation focuses on consolidation and consistency. The following sections provide background and related work for them.

### 2.2.1 Storage Cluster Consolidation

Data centers, it is common to consolidate work into a smaller set of available resources so that the rest of resources can be taken off-line to reduce power consumption [139, 46]. Storage is one of the main components of a data center and it consumes about 20 to 30 percent of total power, making work consolidation in storage systems important and relevant. Since data is not all accessed simultaneously, it is common to have an underutilized or even idle storage system [40, 88, 30], making the storage component a promising one for power savings.

There is a large body of work on work consolidation to improve efficiency in a data center. pClock [46] is a framework that statistically multiplexes workloads while achieving performance isolation via scheduling. pClock guarantees deadlines for well behaved workloads and latency requirements are met as long as burst sizes and IO rates do not exceeded specified limits. HARMONY [95] is a server and storage virtualization framework based on continuous monitoring and adjustments to different workload, that aims at load balancing to improve performance and reduce resource overloading. In [137] the authors find via experimentation the time it takes to offload work from servers to Virtual Machines (VMs) can be given by measurements from a single VM, the performance degradation due to VM migration is longer than the migration time, and show that parallel migration leads to shorter times.

Efforts have been placed to use learning techniques in predicting workload and performance in storage systems. In [70], the authors use a relative fitness model to predict the performance difference between two storage devices so that the changes of I/O rate can be captured when workload moves from one device to another. In [119], the author develop a Profile Hidden Markov Models based methodology by recognizing primitive operations in a trace, aiming to identify the application I/O access patterns.

The framework proposed in this dissertation differs significantly from previous approaches in that it guarantees low performance degradation due to consolidation.

### 2.2.2 Storage Cluster Consistency

In fast growing data centers and global services, many of today's distributed storage systems need to meet several qualities simultaneously, such as performance, reliability, availability, security, and cost effectiveness. Traditionally, data redundancy is used to enhance availability, reliability, integrity, and performance. Yet because redundancy means that data (or parts of it) needs to be written multiple times, often in different locations, it is common to achieve the desired redundancy for each piece of data asynchronously rather than synchronously [36, 20, 96]. This implies that data is sometimes acknowledged to the user before it has successfully reached all its destination nodes. As a result, data consistency can be classified as follows [33]:

- *strong consistency*, where the system acknowledges the data after it has reached all nodes that hold it, and
- *weak consistency*, where the system acknowledges the data as soon as it receives and stores it locally or partially. It allows the system to complete the data distribution to its destination nodes at a later time (i.e., asynchronously). In this case, there is a temporal gap between acknowledgment of updates and distribution of updates across the system, which we call here the "inconsistency window".

Weak consistency [24, 78] favors high system performance and availability and is preferred by applications that consider liveliness more important than durability [20]. Eventual consistency [110, 26, 81, 8, 96] is a specific type of weak consistency that implies that if no new updates are made to a data object, then eventually all copies of the object

data get updated. The inconsistency window reflects data reliability since data loss may occur while the targeted redundancy is not reached immediately upon the system receiving the data.

Efforts have been placed to effective scheduling that can guarantee the foreground task. Conventionally, scheduling of non-preemptive background tasks is done using a non-work-conserving approach by delaying the execution of a background job during an idle interval [30]. This technique avoids using short idle intervals to serve long background jobs and to avoid severe degradation in foreground performance. Storage performance insulation has been achieved by co-scheduling timeslices for each striped workload in [111]. The work in this dissertation significantly differs from the above in that instead of focusing on predicting the idle period size, we concentrate on the best way to coordinate scheduling between the active and inactive nodes to achieve quick eventual consistency without further degrading the foreground traffic in the system.

The work that is most related to the work presented here is [71], where the authors propose a framework to estimate when and for how long idle periods can be used for processing low priority background tasks without violating pre-defined foreground performance target. We generalize this algorithm by introducing a new analytic formulation. The algorithm part is based on [71], but the developed extensions allow the system to serve background work as fast as possible. Experiments are driven by a set of traces collected in a distributed scenario.

## 2.3 Workload Interleaving at the Computing Node Level

This section provides background and related work for the workload interleaving problem at the computing node level. Workload interleaving at this level usually relies on off-the-shelf priority scheduling tools.

### 2.3.1 Off-the-shelf Priority Tools

Proprietary systems often have their own scheduling algorithms that allow them to maintain user workload performance while lower priority jobs execute in the background. The available off-the-shelf tools for priority scheduling in any Unix-based system are `nice`, which prioritizes access to the CPU resource, and `ionice`, which prioritizes access to the disk resource. While different distributions of Unix have different implementations of `nice` and `ionice`, they operate similarly: when enabled, they allow users to adjust the execution priority of processes.

A process that is invoked via `nice` can have a scheduling priority between -20 (the highest priority) and 19 (the lowest priority), as determined by a single parameter in the `nice` command. If the priority parameter of `nice` is set to zero or the process is invoked without the `nice` command then the process is run with the default (i.e., normal) priority. `nice` uses the priority parameter to determine the chunk of CPU time for a specific process, i.e., the higher the priority, the larger the chunk of CPU time the process gets. The exact relation between `nice`'s parameter and the amount of CPU time dedicated to a process is implementation dependent and varies between Unix/Linux distributions. The mechanism is generally simple to use and depends on fine-grained CPU consumption.

Similarly, `ionice` allows ranking the priority of a process from 0 to 3, where 3 is meant to designate a process that should be given IO resources only when the IO system is otherwise idle. A user may select to invoke both `nice` and `ionice`. Though `nice` and `ionice` static prioritization tools, dynamic amount of CPU and memory resources can be allocated to a given process via `renice`.

### 2.3.2 Related Work on Priority Scheduling

There has been a large volume of related work on priority scheduling, that can be roughly classified as scheduling that requires kernel modification, application modification, real time scheduling, or scheduling for quality of service. Yet there is no mechanism that is available at the user space that relies on automatic usage of pause/resume of background execution as well as automatic usage of the various `nice` and `ionice` options, or `renice` thereof.

Traditional work on real-time scheduling relies on strictly or semi-strictly predictable periodic tasks, such as media players, and requires kernel modification, changes to application code in order to take advantage of scheduling, and keeping track of specific deadline information for every task [93, 77, 69]. Cucinotta et. al. focus on meeting acceptable throughput for "soft real-time" applications, specifically media streaming [23]. To do this, they take a signal processing approach to characterize the activity periodicity behavior of the blackbox legacy applications they are attempting to control, and use the results to budget resources for each application. Their implementation requires kernel modification and does not explicitly stop low priority background tasks in order to better protect foreground tasks, as ours does. Meehean et. al. propose a very flexible system which requires kernel modification [68].

Scheduling that provides quality is service to individual customers has been developed in [134], which look to provide kernel support for differentiating quality of service for individual customers. Here the focus is preventing background tasks on the server from interfering with any response-time-sensitive tasks without requiring any kernel modification. Indeed, the proposed mechanism to background task management could be combined with QoS differentiation schemes by using different thresholds to protect higher QoS processes.

Recent scheduling research has often focused on the particular problems of scheduling jobs on multicore machines and computing clusters [54, 132]. When priority schedulers are considered, it is generally with the intention of improving their fairness or maintaining fairness when adapting a scheduler to more complex circumstances [132, 56]. The individual characteristics of particular tasks are often taken into account for scheduling purposes, for instance to save energy during periods of low utilization [101] or to spread out intensive tasks to prevent thermal damage to a machine [21]. In some cases the non-linear interaction of different co-located jobs is taken into account [63]. Instead, we look to use as much of the spare capacity as possible for time-insensitive background tasks, as in the case of a server handling the continuous and bursty workload of foreground user traffic while also intending to perform replication, integrity checking, data analysis, or other work [71, 130].

Virtual machines (VMs) can also be used to isolate high priority tasks [79]. VM management is not straightforward and requires significant overhead to manage, monitor, and adjust resource allocation. In contract to traditional VM managing solutions, the approach proposed in this dissertation does not require the deployment of any additional software. We provide a more precise sharing of resources since it adjusts based on percentage of total CPU usage and may allow the high and low priority processes to share cores, whereas the virtual machine approach generally assigns a whole number of cores to each virtual machine, although the exact number may change dynamically [79].

Other researchers have focused on the progress rate of applications to determine appropriate resource sharing between them [27, 34]. Ferguson et. al. describe a weighted fair-sharing system that uses the progress rate to effectively balance between jobs with specific deadlines of varying importance [34]. Douceur and Bolosky share the goal of this dissertation more clearly, identifying very low priority tasks that should not be allowed to impact the foreground task [27]. To determine whether the background task should be

run or temporarily stopped, they monitor the progress rate of the background applications, assuming that when the progress rate falls below a particular threshold, it must be because of foreground process contention for shared resources. The background tasks are then stopped for a window of time, then tried again. Inspired by the TCP congestion control mechanism, the sleep window increases exponentially as resource contention is repeatedly observed. These approaches work well, but require a way to monitor the progress rate of background applications by the foreground application.

Closely related to the work in this dissertation, Abe et. al. [6] consider distributed computing projects like SETI@home, which allow individuals to donate computing time to scientific calculations when their computer is otherwise idle. They find that built-in priority scheduling is insufficient to protect foreground performance and choose to turn off background processing when the system detects resource contention with foreground processes. They monitor the background process to detect this contention and apply an exponential back off to reduce the impact on the foreground. Instead of attempting to measure the progress of the background tasks, they monitor the share of resources given to the background process. If the share drops, they assume that the foreground processes are now demanding more resources and could benefit from the background dropping altogether. In contrast, we focus on the behavior of the foreground task, looking for the best periods in which to perform background work.

Summarize, the middleware proposed in this dissertation differs from all the above work in that it does not require changing the kernel or depend on complex software. It does not require making changes to the foreground application or its processes, it can be even deployed without interrupting the current services. To deploy it, a learning phase is required to characterize the statistical distribution of the foreground traffic's busy periods to determine the optimal periods to suspend the background job execution, and based

on this information it launches *different* background job scheduling policies that can best fit the current system conditions. Therefore, it is lightweight, portable, and flexible as it manages to take advantage of the benefits of several monolithic background scheduling policies while minimizing their respective shortcomings.

## 2.4 Workload Interleaving at the Computing Cluster Level

MapReduce and its Hadoop implementation is a very popular framework used in many of today's large distributed systems. Heterogeneous multi-core processors is the platform where MapReduce applications operate because it offers different capabilities for different workloads. The interesting question is whether heterogeneous multi-core processor can bring additional performance opportunities for workload interleaving in MapReduce workloads.

### 2.4.1 MapReduce Processing

In the MapReduce model [25], computation is expressed as two functions: map and reduce. MapReduce jobs are executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*. The map and reduce tasks are executed by *map slots* and *reduce slots*. The number of *map slots* and *reduce slots* are configurable parameters and indicate how many map and reduce tasks can be executed at one time on one node.

In the *map stage*, each map task reads a split of the input data, applies the user-defined map function, and generates the intermediate set of key/value pairs. The map task then sorts and partitions these data for different reduce tasks according to a partition function.

In the *reduce stage*, each reduce task fetches its partition of intermediate key/value pairs from all the map tasks and sorts/merges the data with the same key (it is called the

shuffle/sort phase). After that, it applies the user-defined reduce function to the merged value list to produce the aggregate results (it is called the reduce phase). The reduce outputs are written back to a distributed file system.

### 2.4.2 Heterogeneous Multi-core Processors

To offer diverse computing capabilities, the emergent modern system on a chip (SoC) may include heterogeneous cores that execute the same instruction set while exhibiting different power and performance characteristics. There is a body of work exploring power and performance trade-offs using heterogeneous multi-core processors. Some papers focus on the power savings aspect, e.g., Rakesh et al. [59, 60], while other works concentrate on the performance aspect, e.g., [103, 11, 32]. These papers exploit a similar technique, such as monitoring, evaluating threads performance, and dynamically mapping threads to different types of cores. Daniel et al. [94] propose using architecture signatures to guide thread scheduling decisions. However, this method needs to modify the applications for adding architecture signatures. Therefore, it is not practical to deploy in real world applications. The work in [42, 22] explores the per-program performance in addition to the overall chip level throughput when using heterogeneous multi-core processor.

This earlier work is designed for a single machine while Hadoop is a distributed framework and needs to manage a cluster environment. Therefore, it is difficult to apply the traditional techniques for a Hadoop framework. In addition, the aim here is to support different performance objectives for classes of Hadoop jobs, which requires an exact control of running different types of slots in different cores.

Ren et al. [87] consider a heterogeneous SoC design and demonstrates that the heterogeneity is well suited to improve performance of interactive workloads (e.g., web search, online gaming, and financial trading). This is another example of interesting

applications benefiting from the heterogeneous multi-core processors.

### 2.4.3 Heterogeneous Scheduling at the Server Level

There is a body of work that focuses on performance analysis and optimization of MapReduce processing in heterogeneous server environments. The authors in [133, 18] compute the remaining time of each task and use speculative execution to accelerate the "slow" task to reduce the heterogeneity's negative impact. This technique is applicable to the case in this dissertation as well, especially for operating with shared spare resources that are formed by different type of slots. [7, 35] use load-balancing and load re-balancing approaches in heterogeneous clusters to allow the faster node to get more data, in order for the reduce tasks to finish approximately at the same time and improve overall performance. Xie et al. [116] use data placement to optimize performance in heterogeneous environments. Faster nodes store more data and therefore run more tasks without data transfer. Lee et al. [61] propose to divide the resources into two dynamically adjustable pools and use a new metric "progress share" so that better performance and fairness can be achieved. Their approach only allocates resources based on the job storage requirement. Gupta et al. [47] use off-line profiling of job execution with respect to different heterogeneous nodes in the cluster and optimize the task placement to improve the job completion time. Polo et al. [83] modify the MapReduce scheduler to enable it to use special hardware like GPUs to accelerate the MapReduce jobs in the heterogeneous MapReduce cluster. Zhang et al. [136] explore the efficiency and performance accuracy of the *bounds-based* performance model introduced in the ARIA project [105] for predicting the MapReduce job completion times in heterogeneous Hadoop clusters and discuss factors that impact the MapReduce job performance in the Amazon EC2 environment.

All these efforts focus on the server level heterogeneity in Hadoop cluster. In the case

of Hadoop deployment on heterogeneous servers, one has to deal with data locality issues and balancing of data placement according to the server capabilities. One of the biggest advantages of Hadoop deployed with heterogeneous processors is that both *fast* and *slow* slots have a similar access to the underlying HDFS (Hadoop Distributed File System) data that eliminates data locality issues.

# Chapter 3

# Efficient Scheduling of Background Tasks

Systems that support emerging computing paradigms such as cloud computing are growing distinctively larger and more complex. In order to meet the ever increasing user needs for high availability, reliability, performance, and cost-effectiveness [65, 97, 51, 14], systems are built by integrating off-the-shelf components that are managed and maintained *asynchronously*, i.e., outside the critical path of user requests. While the amount and criticality of asynchronous management is commensurate with system complexity, the expectation for such work is to remain transparent from the system users. Examples of tasks that complete asynchronously in the system, i.e., in the background, include logging of monitored resources, garbage collection, data synchronization, and data verification. Within the storage component, a significant amount of work is completed asynchronously in the background, especially because storage tasks are not instantaneously preemptable [67, 92].

While background work in storage systems may be associated with performance improvement, e.g., moving data from the low performing tier of SATA drives to the high

performing tier of SSD drives [45], it mostly targets enhancement of data availability and reliability, e.g., verification of data consistency for protection against bit-rots and replication of data in multiple storage devices or systems for added redundancy. The goal is to strike a balance between meeting user service level objectives while completing the background work as fast as possible. This goal is particularly important for background tasks that are time sensitive. Examples of time sensitive background tasks include geographically distributed data centers where data consistency is achieved only *eventually* by distributing the redundant new data asynchronously, in the background [110].

Judicious selection of scheduling *asynchronous work* vs. *user traffic* is not an easy task. The challenge lies in the fact that future user workload characteristics are seldom known a priori. If the background tasks are scheduled without consideration of the user traffic, the impact on user performance may be severe.

Common practices use simplistic measures, such as average utilization, to guide background task scheduling. Such metrics cannot describe accurately current system conditions and often yield unstable solutions because the workload, particularly in storage systems, can be fairly dynamic over short time scales. To limit the impact of background work on user performance, there exist elaborate techniques that focus on idle waiting before starting background work [30, 40]. There are techniques that even provide guarantees on the performance impact caused to user performance [71]. While some techniques that are used to schedule background work operate on fixed parameters that restrict their adaptivity to a changing workload [30, 40], others rely on monitoring a variety of complex processes, such as system idleness, delays caused by the background tasks, and user performance [71].

In this thesis, we present a simple yet adaptive solution to the problem of scheduling tasks in the background by proposing a quantitative framework that aims at monitoring,

learning, and making scheduling decisions based on a few, easy to monitor metrics. The monitored metrics capture sufficient details on the current foreground workload and the resulting available idle capacity that allow the proposed scheduling policy to complete the background work as fast as possible but with minimal impact on user performance. All scheduling decisions are based *only* on the stochastic characteristics of the length of *user busy periods* in the system. The goal is to schedule as much as possible background work when the impact on performance of user traffic is anticipated to be small (because upcoming busy periods are short) and limit delays on foreground traffic when busy periods are anticipated to be long.

Results from extensive experimentation via trace-driven simulations show that the proposed scheduling policy can maintain the same foreground performance while completing the asynchronous work up to 50% faster. The benefits of the proposed scheduling policy are particularly high when it matters most, i.e., when foreground performance imposes stringent limitations on the tolerance toward additional delays due to background work. The proposed scheduling policy enables the system to sustain its performance in the presence of background tasks, even where there are changes in the user traffic characteristics, by adapting the background scheduling parameters to current foreground characteristics. The robustness and resilience of the scheduling policy is evident especially under swift changes in user workload.

The rest of this chapter is organized as follows. In Section 3.1, we give an overview of related work. In Section 3.2, we provide a detailed characterization of a set of enterprise traces and show how this characterization can be used to develop the new scheduling strategy. In Section 3.3, we propose a dynamic scheduling framework aiming at improving the performance of background work while maintaining foreground performance. Section 3.4 presents an extensive set of trace-driven experiments that

demonstrates the effectiveness and robustness of the proposed scheduling technique. We summarize in Section 3.5.

## 3.1 State of the Art and Motivation

Systems today have to support a wide range of background tasks. These tasks should be served transparently from foreground tasks but should not starve. Avoiding starvation is the primary target to be met. In addition, if the background tasks are time-sensitive, as it is often the case in storage systems, then they should complete as soon as possible. There is an ever increasing number of time-sensitive asynchronous tasks in storage systems that are served in the background. Examples of such tasks include the asynchronous data updates in geographically distributed data centers. The data in such systems resides in multiple devices, nodes, and locations for purposes of availability and performance. New data is committed asynchronously to all designated nodes in order to avoid network and other delays that may severely impact user perceived performance. As a result the consistency of data across the distributed system is achieved *eventually* as data is committed to its destinations as a background process [110]. Note that for as long as the data is not consistent across *all* of its assigned nodes, data integrity is compromised. This is a clear case where completion of background tasks is time sensitive.

In this thesis, we strive to achieve two goals: first to complete the background work while avoiding starvation at all costs and second to reduce its response time as much as possible to better serve time sensitive tasks. Our aim is to maintain the performance of foreground tasks at the same level as common practices, e.g., the approach in [30], while serving background tasks faster. Background tasks in storage systems have similar service demands as foreground requests. This means that if a foreground request arrives to find the system serving a background task, then the delay expectation is approximately

two times the average service demand of a foreground request (i.e., accounting for the background work to complete and the storage system to get ready - positioning - to serve the next request). We consider such a delay to be "tolerable". This means that controlling foreground delay due to background work is effectively done by delaying *only* the start of a background busy period.

Deploying any "wait period" before background tasks start execution [30, 71] would result in non-work-conserving scheduling of the background tasks with low degradation on foreground performance. Such non-work-conserving scheduling we denote as "conservative". A zero "wait period" would result in work-conserving scheduling of the background tasks and better utilization of the available idleness. We denote this policy as "aggressive". To gain intuition on the simultaneous effect on the performance of both foreground and background jobs, we evaluate the aggressive and the conservative scheduling policies via a set of trace-driven simulations. We consider constant idle wait times as in [30] ranging from 0 to 100 ms. Details on the disk drive traces that are used are provided in Section 3.2. Here we simply want to highlight the advantages and disadvantages of aggressive versus conservative scheduling.

As already discussed, background tasks in a system are commonly a function of the current workload (e.g., data synchronization). Therefore, it is reasonable to assume that multiple asynchronous features generate background work out of the incoming user workload. We explore here two scenarios where the background work (BG) is 100% and 1000% of the foreground work (FG). Results are shown in Figure 3.1. From the graphs, we can see that aggressive scheduling gives the worst foreground performance while achieving the best background performance. With conservative scheduling, the foreground performance improves as the fixed idle wait (see x-axis) increases, which confirms the need to protect foreground performance via idle waiting. However, we also

observe that background performance decreases much faster when compared with the degradation caused to foreground performance. For large periods of idle waiting, foreground response time improves slightly while the performance of the background tasks degrades by orders of magnitude when compared to shorter or zero idle waiting. Since these two scheduling policies are complementary to each other, we are motivated to design a new scheduling algorithm to improve the response time of background work while preserving foreground performance.

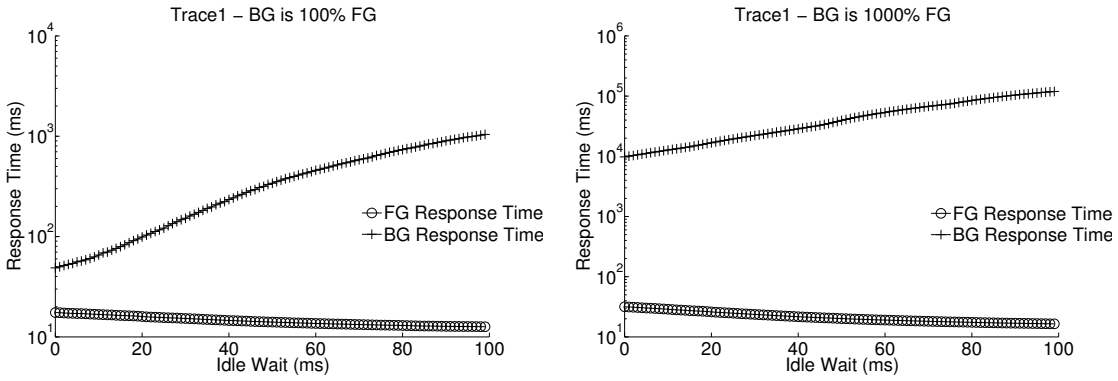



**Figure 3.1**: Performance comparison in terms of mean response time between the foreground and background tasks for a disk-level trace under conservative scheduling with fixed waiting ranging from 1ms to 100ms. The results of aggressive scheduling are also shown in the graph, i.e., the point corresponding to idle wait = 0. The response times are in log scale.

Recall that performance degradation of foreground work comes from the fact that the system needs some time to switch from serving background work before it can serve foreground requests. The *entire set* of foreground requests in the following busy period is delayed. Our key observation here is that the impact of background tasks on foreground performance is larger if the delayed foreground busy periods are long (i.e., measured in number of requests) than if they are short.

We stress that in prior work, all efforts focused on incorporating characteristics of the arrival process, service process, or idleness of the system into the scheduling of background

tasks. In this thesis, we design an intelligent scheduling mechanism by exploring and taking advantage of the stochastic characteristics of busy periods *only*.

## 3.2 Workload Characterization

In this section, we analyze the enterprise disk-level traces used in the evaluation of the scheduling policy that is devised in this thesis. We give general information about the traces but also focus on the stochastic characteristics of their busy periods.

### 3.2.1 Overview of Traces

We use three enterprise traces measured at the disk level from servers running enterprise-grade applications [88]. Although the storage subsystem of the servers consists of multiple RAID groups, we use here the user traffic seen by three individual disks located in different RAID groups. The traces are twelve hours long. Each trace contains the following information for each request: the arrival time, the departure time, the type of request (i.e., read or write), the request length in bytes, and its location on the disk.

In Table 3.1 we show a set of metrics that provide some general information on the availability of idle time at the disk level and the characteristics of foreground busy periods. The data in Table 3.1 shows that the disks are clearly underutilized and they have good potential to serve background work. The large coefficient of variation (C.V.), which is a normalized measure of the dispersion defined as the ratio of the standard deviation to the mean, and the large maximum length of idle intervals imply significant variability in the length of idle periods. This concurs with the discussion in the previous section: if the purpose is to serve background work timely, then limiting the time where background work can be served is not a good strategy. For busy period lengths, the moderate C.V. values coupled with the large value of the maximum length, suggest that there is also variability

in the length of busy periods, albeit at a less degree than in idle periods. The impact on foreground performance due to interleaving foreground with background work may be quite different from one busy period to the next.

| Trace | Util | Idle Periods in *ms* | | | Busy Periods in *IOs* | | |
|---|---|---|---|---|---|---|---|
| | (%) | Mean | Maximum | C.V. | Mean | Maximum | C.V. |
| Trace1 | 5.6 | 192.6 | 325589 | 8.4 | 2.16 | 240 | 2.1 |
| Trace2 | 1.7 | 767.5 | 186817 | 2.3 | 2.84 | 110 | 1.3 |
| Trace3 | 0.7 | 2000.2 | 364876 | 3.8 | 2.39 | 190 | 2.4 |

**Table 3.1**: General busy period and idle period characteristics of our traces.

### 3.2.2 Characteristics of Busy Periods

Because the impact of the background tasks is strongly related to the length of the upcoming foreground busy period, we now focus on the statistical features of foreground busy periods. In Figure 3.2 we plot the CDH (Cumulative Distribution Histogram) and relative frequencies using a bin size of one request. Note the log scale for the x-axis. The shape of the plots implies long tails for busy periods across all workloads, i.e., most of the busy periods are short while a few of them are quite long. One can see that across all workloads, 90% of busy periods are less or equal to 4 requests per busy period. This implies that if the background work delays a busy period, then it is with high probability that there are up to four requests to be delayed. Yet, there is also a sizable percentage of the workload with long busy periods (i.e., more than 4 requests) where the performance degradation of foreground work is going to be noticeable. The argument here is that *if* we can anticipate when these long busy periods arrive, then performance can be improved significantly by avoiding to serve background jobs during those time intervals.

Next, we plot the length of every busy period across time, measured in number of requests, see Figure 3.3. The plots show a clear repetitive cluster behavior in the sequence

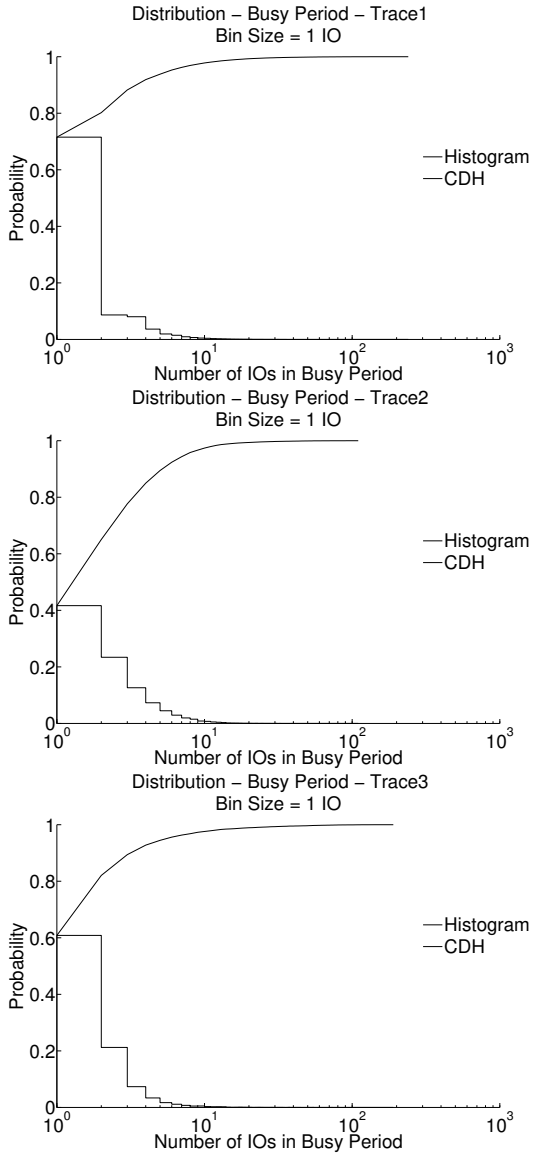



**Figure 3.2**: The distribution of the busy periods measured by number of requests.

of long busy periods (i.e., greater than 4 requests) for Trace1 and Trace2. The graphs show that the majority of busy periods are 4 to 6 requests. We conclude that this number can be used as a threshold that distinguishes busy periods as short or long.

In addition, the "clustering" of long busy periods shown in Figure 3.3 suggests that

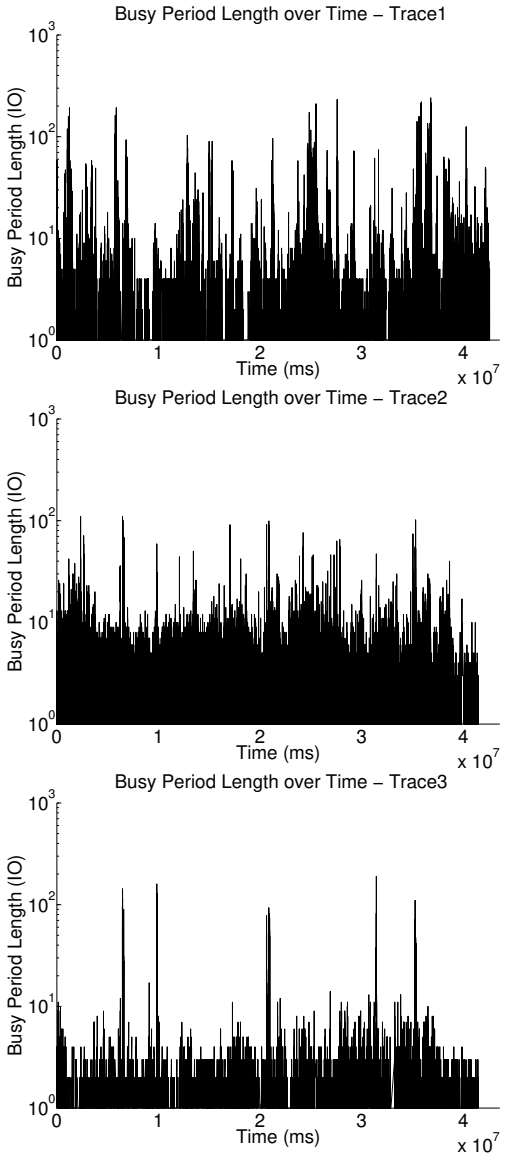



**Figure 3.3**: The across time plots of busy periods length measured by number of requests.

there is a consistent behavior across time. If we understand better how such clustering occurs, then we can use it to detect the upcoming clusters of busy periods. Once such a cluster is detected, then it would be beneficial to foreground performance if the background work is scheduled "conservatively" (i.e., the system idle waits before starting

the background tasks). Once the system predicts that the upcoming busy periods are not expected to be long, then it resumes a more "aggressive" scheduling of background tasks (i.e., schedule them immediately after the system becomes idle of foreground requests). These observations are the basic premises for the design of a scheduling policy that dynamically adapts to a changing workload.

## 3.3 Dynamic Scheduling

In this section, we propose a dynamic scheduling policy that interleaves background tasks with foreground tasks efficiently. The goal here is to improve the performance of background work, measured via its response time, while preserving foreground performance, also measured via its response time. The dynamic policy that we propose alternates between scheduling background tasks aggressively or conservatively, based on the statistical characteristics of foreground busy periods and their recent history. As future busy periods are not known a priori, the algorithm cannot always make the best decision, but it can reach to a well-informed decision based on statistics of recent workload history. The policy parameters are extracted from the most recent history of foreground busy periods.

Aggressive scheduling may result in foreground performance degradation, because if short idle periods are utilized for background work, then with high probability, it delays all requests in the upcoming foreground busy period. The idle wait ensures that only long idle periods are used for background work. For a thorough discussion on the impact of idle wait on the performance of both foreground and background work, we direct the reader to [30, 71]. If there is a large amount of background work that is time critical, then the background tasks would have to continue to run as long as the system is idle, endangering the performance of upcoming foreground tasks. We argue that rather than limiting the

amount of background work during idle periods as in [71], we limit the potential degradation on foreground performance:

- by selecting a fixed large idle wait for the periods when the system is experiencing a sequence of long foreground busy periods, with the expectation that such idle waiting would forbear the system from serving background work, and
- by canceling idle waiting if it is detected that the system is experiencing short foreground busy periods. This action would give the system the opportunity to serve a large amount of background work while delaying only a small portion of the foreground requests.

The algorithm first categorizes busy periods as long or short. Within a predefined time window, we log the information of busy period lengths and update their histogram, a process that is inexpensive, both computationally and space-wise. At the end of the time window, the count of requests that corresponds to the 90th percentile of the busy period histogram defines a *Threshold* whose value distinguishes busy periods as long or short. A new histogram is build for the next time window, which allows the algorithm to adapt well the *Threshold* parameter to changing workloads.

After categorizing the busy periods, the next step is to predict the incoming busy period length. To this end and according to the analysis in Section 3.2, we explore the clustered pattern of busy periods within each time window. This suggests that after an elapsed long busy period and based on recent history, we may be able to predict with accuracy whether the upcoming busy periods are long or short. To achieve this, we observe the conditional probability that two subsequent busy periods are long, i.e., if they are separated by one idle period with lag equal to one, as well as the conditional probabilities of busy periods that are separated by two or more idle periods, i.e., with lags equal to two or more. We define the *Cluster Window Size* ($CWS$) as the average number of consecutive long busy periods

occurring with a given high probability value. Let $P_{lag}$ be the conditional probability that the $lag$th busy period is long given that the current busy period is long (we could use any sufficiently large number here instead of twenty so that we capture enough probability mass). We define $CWS$ as the smallest $lag$ such that the sum of $P_{lag}$ is equal or over 0.8:

$$CWS = min\{lag| \sum_{lag=1}^{20} P_{lag} >= 0.8\} \tag{3.1}$$

After a long busy period is detected, then $CWS$ gives the number of upcoming busy periods that are expected to be long. During the intermittent idle intervals within those periods (which may be long or short), background work is served conservatively, i.e., deploying an idle wait period. After this number expires, background tasks are served aggressively, i.e., without any idle waiting, till the next long busy period is detected and conservative scheduling gets activated again. Note that the calculation of $CWS$ is done once for every time window, in order to reflect well changes in the process of the foreground busy periods.

Note that, according to Equation (3.1), the stronger the clustering in the foreground busy period lengths, the shorter the $CWS$, and the longer the system serves background tasks aggressively. If the long foreground busy periods in the system are distributed randomly, i.e., there is no clustering, then $CWS$ is long and the system schedules background tasks conservatively. Hence, the dynamic scheduling policy we propose here extracts the stochastic characteristics of foreground busy period lengths and reduces to the common practice of conservatively scheduling depending on the predicted foreground arrivals. Figure 3.4 gives the pseudo-code of the dynamic background scheduling policy.

In Figure 3.5 we give an example of how the aggressive, conservative, and dynamic algorithms work. We assume that there are several background tasks outstanding and the system currently operates under short foreground busy periods (the first three user busy

**1. if** in *characterization state* do
  **a.** update busy period length trace
  **b.** calculate the $Threshold$ of long busy period based on 90th percentile
  **c.** calculate the Cluster Window Size ($CWS$) based on Eqs. 3.1
**2. if** system in *decision making state* do
  **a.** initialize:
    **i.** system state ($sys\_state$) = $idle$
    **ii.** busy period state ($BP\_state$) = $short$
    **iii.** cluster count ($cluster\_count$) = 0
    **iv.** busy period length ($BP\_length$) = 0
    **v.** queue length ($QL$) = 0
  **b.** **if** $sys\_state = idle$
    **i. if** $BP\_state = long$ and $cluster\_count > 0$
      **for** no FG IO arrive do
        use *aggressive scheduling* to schedule BG work
        $cluster\_count - -$
    **ii. else**
      **for** no FG IO arrive do
        use *conservative scheduling* to schedule BG work
  **c.** upon FG IO arrive
    **i.** $sys\_state = busy$;
    **ii.** $QL$ ++
    **iii.** $BP\_length++$
    **iv.** **if** $BP\_length >= Threshold$ and ($BP\_state$) = $short$
      $BP\_state = long$
      $cluster\_count = CWS$
    **v.** go to Step **2.b**
  **d.** upon FG IO depart
    **i.** $QL$ ++
    **ii.** **if** $QL == 0$
      $sys\_state = idle$
      $BP\_length = 0$
    **iii.** go to Step **2.b**

**Figure 3.4**: Algorithm of dynamic scheduling.

periods marked with "S" in the figure) that are then followed by long user busy periods. After detecting the first long busy period (the fourth busy period), the next busy period (the fifth busy period) is marked as part of the next cluster of long busy periods. We assume that the estimations from previous observations have converged on a cluster size

of 2 (i.e., the value of $CWS$) and that the threshold to differentiate busy period lengths is 4 requests. We assume in the example that the short busy periods are 2 requests long and that the long busy periods are 6 requests long. We assume that each user request is 3 time units, with one time unit being 2 ms. The six idle intervals in the depicted scenario are 5, 8, 4, 7, 8 and 3 time units long, respectively.

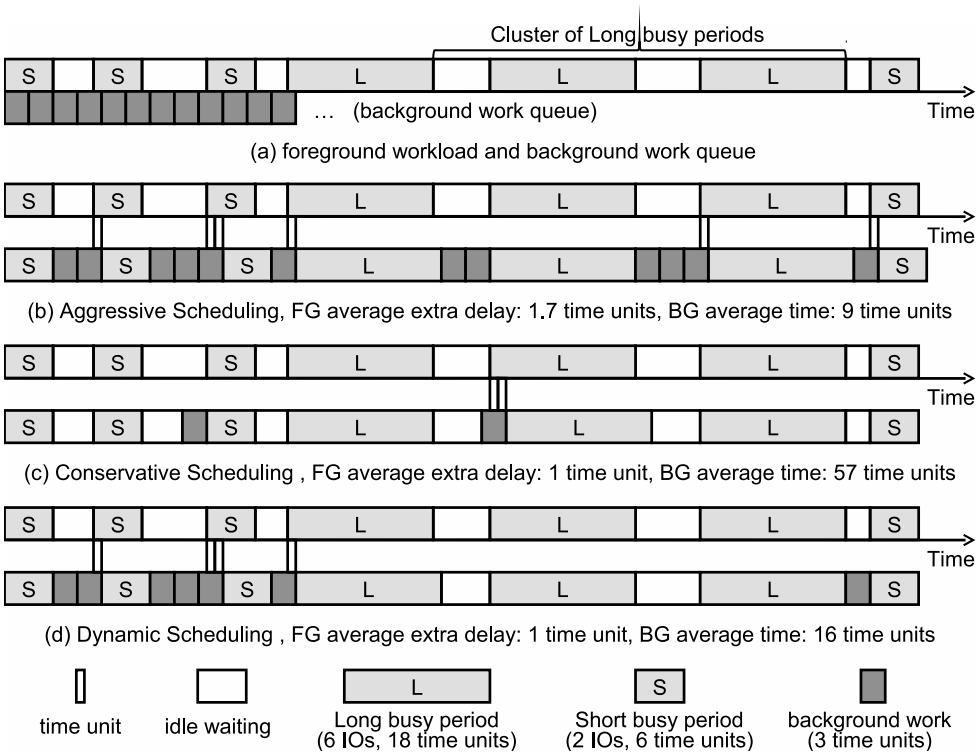



**Figure 3.5**: Example on the behavior of the three different background scheduling algorithm, aggressive, conservative, and dynamic.

Based on the discussion in this section:

- Idle waiting for the dynamic scheduling is larger than the value selected by common practices for the conservative scheduling (i.e., two times of user service demands). In the example we assume that idle wait for dynamic scheduling is 1.5 times longer than the idle time for conservative scheduling.

- Aggressive scheduling does not idle wait and serves the background tasks the fastest (i.e., uses 9 time units on the average) with the largest extra delay (e.g., 1.7 time units) per user request.

- Conservative scheduling serves the background work the slowest (i.e., 57 time units on the average) with an average extra delay per user request of 1 time unit.

- Dynamic scheduling works best because it strikes a good balance between the performance of background tasks (i.e., 16 time units on the average) and an average added delay per user request of 1 time unit only.

This high-level example shows that dynamic scheduling is expected to behave conservatively with regard to foreground performance and aggressively with regard to background work performance. In the following section we evaluate these scheduling polices in detail.

## 3.4 Experimental Evaluation

In this section, we evaluate the dynamic algorithm illustrated in Figure 3.4. The goal is to demonstrate that our algorithm can (1) effectively use the learned foreground busy period characteristics to schedule background tasks and (2) swiftly adapt its background scheduling to changing foreground traffic patterns such that both foreground and background tasks sustain the best possible performance. We evaluate two scenarios. In the first scenario, the system operates under a "stable" workload, while in the second one, the system operates under a workload that changes swiftly half-way through the experiment.

### 3.4.1 Experimental Setting

Our experimental evaluation is trace driven. The traces described in Section 3.2, are used as our foreground traffic. We use Trace1, Trace2, and Trace3 as representative of a stable operating environment. The scenario with the "swiftly changing" workload is achieved by concatenating Trace2 and Trace1, in this order.

As discussed in previous sections, our framework can be applied for scheduling of asynchronous tasks, for example, when new data arrives into a geographically distributed storage system and need to be replicated across nodes for redundancy. In such systems, the redundancy is in the form of replication (e.g., the Google File System [39] replicates data 3 times) or erasure coding (e.g., the data is split into N fragments, encoded into N+M fragments, and distributed into N+M different disks/nodes) [82]. The asynchronous tasks in such scenarios consist of reading the recently updated data, computing the codes for the case of erasure coding, and sending them to their destination via the network. Consistent with this behavior, in our evaluation the background tasks have similar demands as the foreground ones and their intensity is a function of the WRITE foreground traffic, which varies by system. The results hold across a wide range of amount of background work but here we show only two representative cases: (1) the background work is equal to the amount of foreground work (i.e., common scenario, 100% of foreground work) and (2) the background work is 10 times the amount of foreground work (i.e., an extreme scenario, 1000% of foreground work).

Switching from serving background tasks to serving user requests is not instantaneous. Upon arrival of a new user IO which finds the system serving a background task, the system must first complete the background work before re-positioning the disk head back to the location of the new request. In our evaluation, we assume that the penalty experienced by foreground requests due to background tasks is about two times the average service

time of foreground requests. Note that because both foreground and background tasks have service and response times at the millisecond (ms) level, all our metrics of interest are measured in ms. Although replicating a large file or set of files may take overall more time, they are considered tasks that are generally split into multiple smaller tasks. Serving the smaller tasks faster is the goal of our framework.

Our dynamic algorithm uses short-term history (i.e., observations during a time window to calculate the *Threshold* and $CWS$. During each time window, we build the histogram of busy periods and based on this histogram we calculate the *Threshold* and $CWS$ parameters, which are used to schedule the background tasks during the next time window. The moment the *Threshold* and $CWS$ parameters are computed, the histogram is discarded. During the next window where background scheduling is enabled, we collect data to construct a new histogram which is then used to calculate the *Threshold* and $CWS$ parameters for the next scheduling window. Note that for Trace1, Trace2, and Trace3, we specifically focus on a 5-hour window, i.e., we collect the histogram during a window defined by $[Start,\ Start+5)$ and apply the policy during $[Start+5, Start+10)$. To show the robustness of the policy irrespective of the *Start* value, we show results for three different sequences of 10-hour periods.

In our experiments, the amount of idle wait before starting the asynchronous tasks determines the aggressiveness of background scheduling. As idle wait increases, the impact on the response time of foreground requests decreases and the response time of background tasks increases. Here, we evaluate the entire range of idle wait values from 0 to 100 ms. Zero idle wait corresponds to the most aggressive background scheduling.

### 3.4.2 Evaluation Scenario One: Stable Workload

We drive our simulation using the three traces described in Table 3.1. Each trace has characteristics that change gradually over the course of its 12 hours span. Since changes are not dramatic, we consider such traces to represent stable operating environments, where our framework is expected to capture gradual changes effectively.

During the first time window, our scheduling framework monitors the system busy periods, builds their histogram, and once the time window elapses, computes the *Threshold* and $CWS$ values. Recall that *Threshold* corresponds to the value of the $90^{\text{th}}$ percentile of busy periods, while the $CWS$ is computed based on the values of the conditional probability $P_{lag}$ that two busy periods separated by $lag$ idle intervals are both long. As different histograms are collected over different windows, the changes in the workload are captured by $Threshold$ and $CWS$. Figures 3.6, 3.7, and 3.8 show the values of the conditional probabilities of long busy periods over three different 5-hour windows.

The dynamic algorithm strives to exploit any relationship that exists in the sequence of foreground busy periods. If the clustering across time is weak, as in Trace3 (see Figure 3.8), then the expectation is for the dynamic algorithm to operate more often in the conservative mode (i.e., applying some idle wait). If the clustering is non-existent, then the proposed algorithm should *always* operate in the conservative mode.

Figures 3.6, 3.7, and 3.8 clearly show a stable behavior across time within each trace. Across traces, we notice that Trace1 has long busy periods clustered together because its conditional probability values are highest among the three traces. Clustering reduces for Trace2, while Trace3 depicts the least clustering. This means that the dependence structure weakens from Trace1 to Trace3. Therefore the computed $CWS$ values increase as the dependence of long busy periods reduces from Trace1 to Trace3.

Figure 3.9 shows how long (in percentage of time) the dynamic algorithm operates

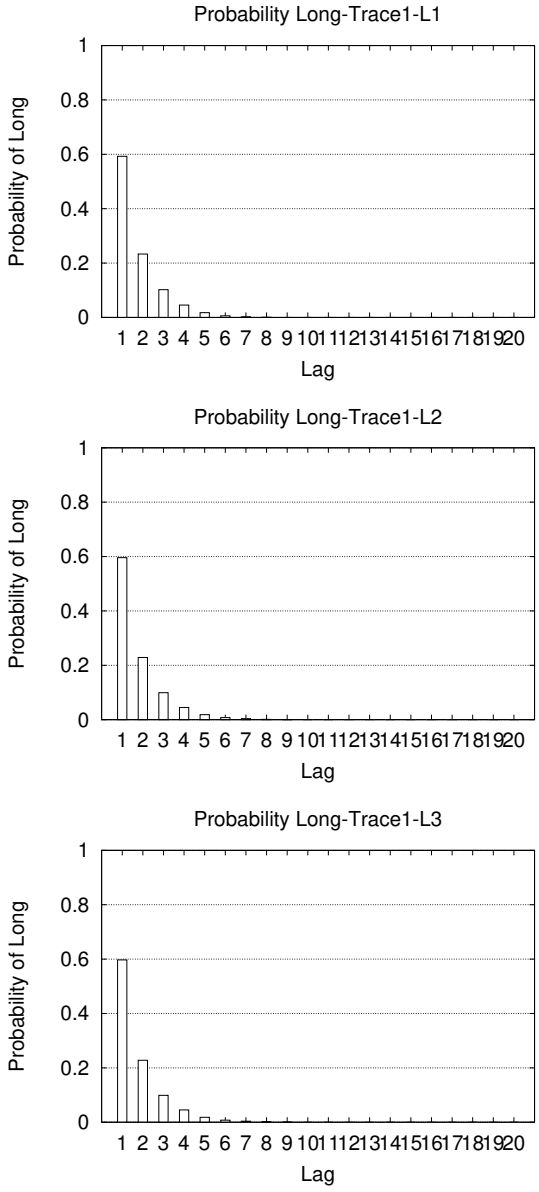



**Figure 3.6**: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace1. Three windows are considered: $Start = 0.5 \ \ hour$ (top graph), $Start = 1 \ \ hour$ (middle graph), and $Start = 1.5 hour \ \ $ (bottom graph).

in the conservative mode and how long in the aggressive mode. As expected from the discussion on the results in Figures 3.6, 3.7, and 3.8, Trace1 spends the most time in the

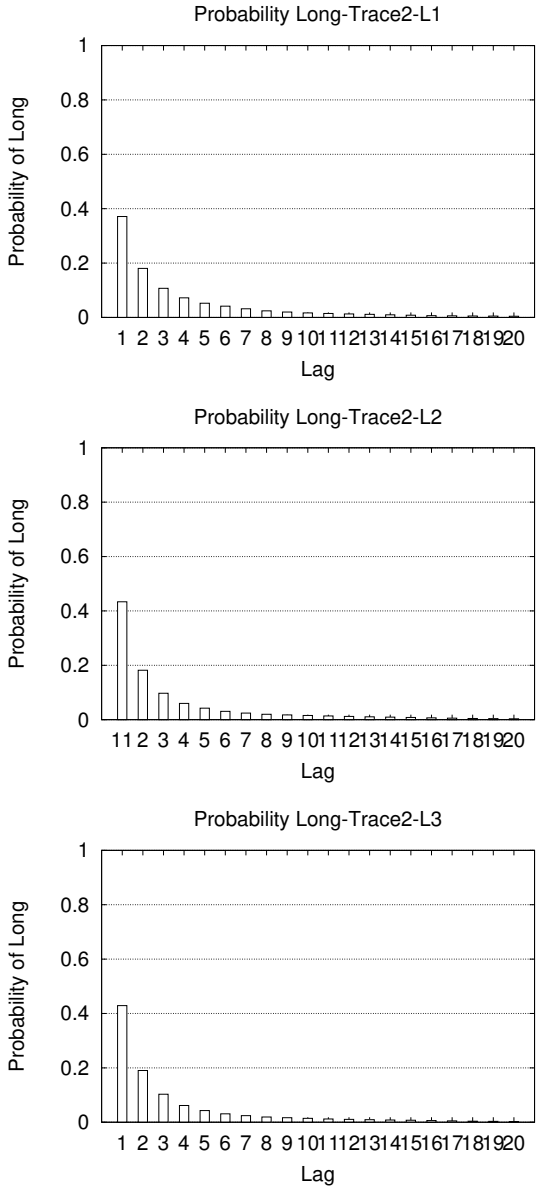



**Figure 3.7**: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace2. Three windows are considered: $Start = 0.5$ *hour* (top graph), $Start = 1$ *hour* (middle graph), and $Start = 1.5$ *hour* (bottom graph).

aggressive mode because the long busy periods in this trace are well clustered, allowing the algorithm to predict well their occurrence.

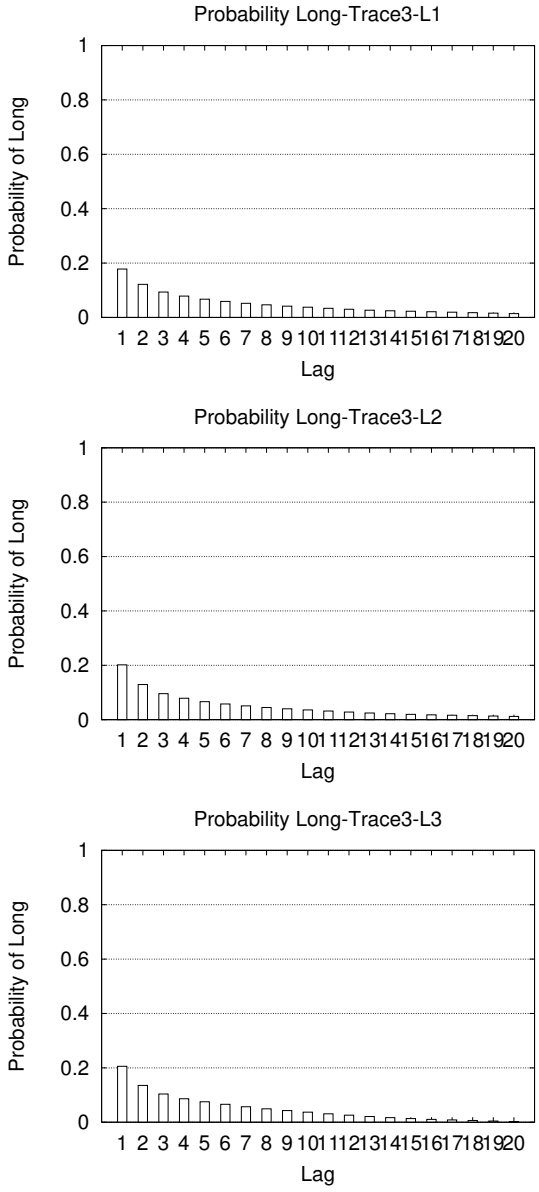



**Figure 3.8**: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace3. Three windows are considered: $Start = 0.5$ *hour* (top graph), $Start = 1$ *hour* (middle graph), and $Start = 1.5$ *hour* (bottom graph).

Because of the overhead to switch from a background task to a foreground task, the more background work served, the higher the impact on foreground performance. The goal is to

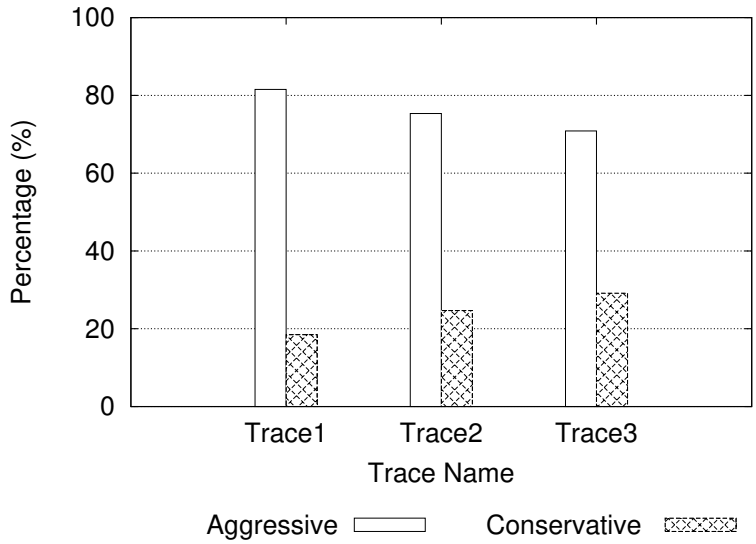



**Figure 3.9**: The percentage of time in aggressive mode and conservative mode under dynamic scheduling.

serve faster the outstanding background work, while sustaining foreground performance. Here we evaluate the effectiveness and robustness of the proposed dynamic scheduling by comparing the background mean response time for the *same* foreground mean response time under both the dynamic and conservative scheduling policies.

Figures 3.10, 3.11, and 3.12 show the average performance of background work for Trace1, Trace2 and Trace3, respectively, as a function of the achieved foreground response time. The figures are organized in a 2 by 3 grid, where each column corresponds to the performance achieved in a given window (the same ones depicted in Figures 3.6 through 3.8), and each row corresponds to the amount of background work generated in the system (i.e., 100% and 1000% of foreground work).

Recall that foreground response time is generally increased by the execution of asynchronous tasks because they arrive stochastically and the switch between asynchronous and foreground tasks is not instantaneous. This means that as long as the

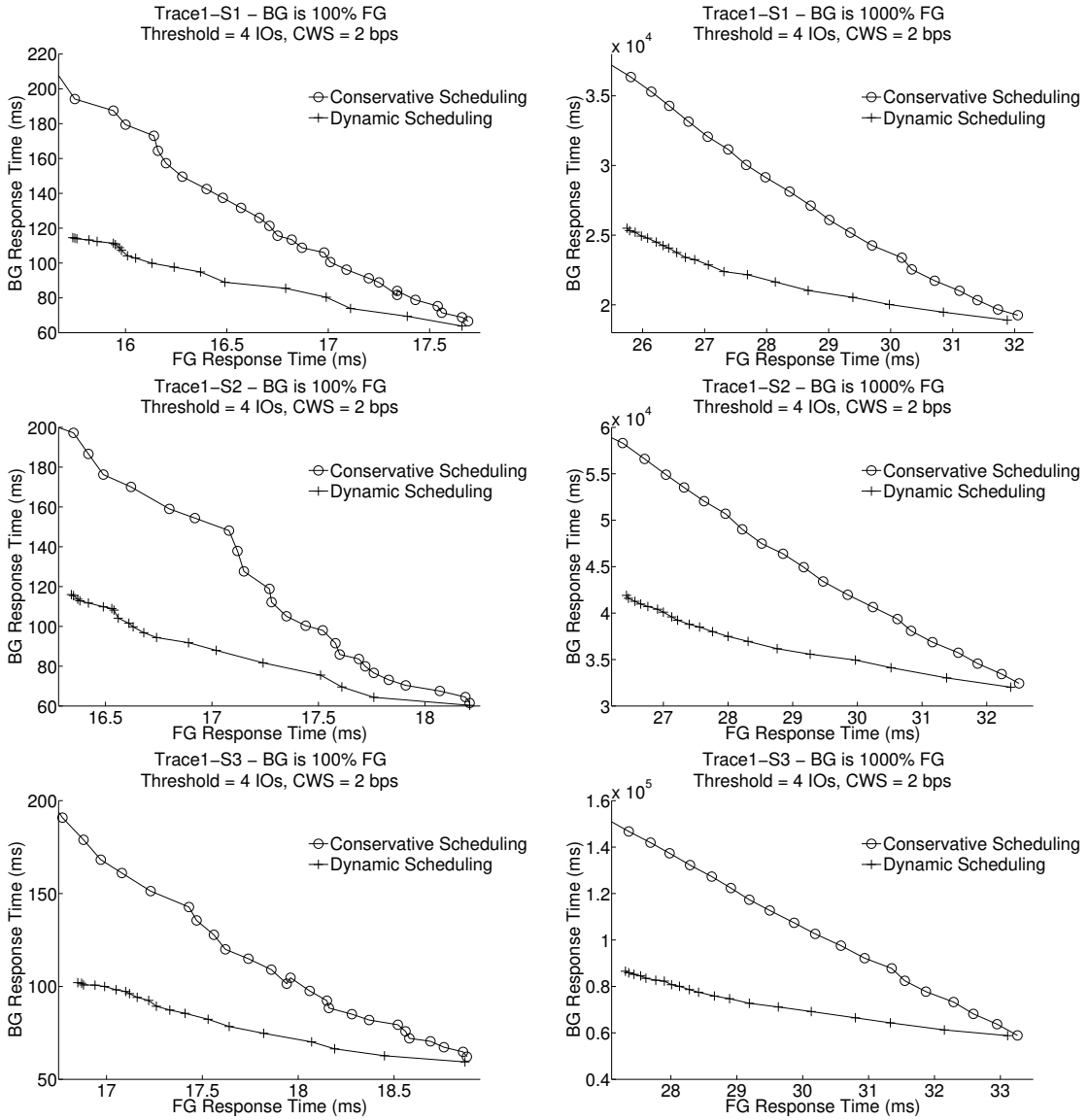



**Figure 3.10**: Scheduling comparison between dynamic and conservative scheduling for Trace1, scheduling results use the three respective periods given in Figure 3.6 (top, middle, bottom rows) to schedule in the next 5 hours.

idle wait value is smaller than the maximum idle interval length, there may be degradation in foreground performance. Idle wait is a way to control and limit performance degradation but not avoid it [72]. Our goal is to make sure we do not violate

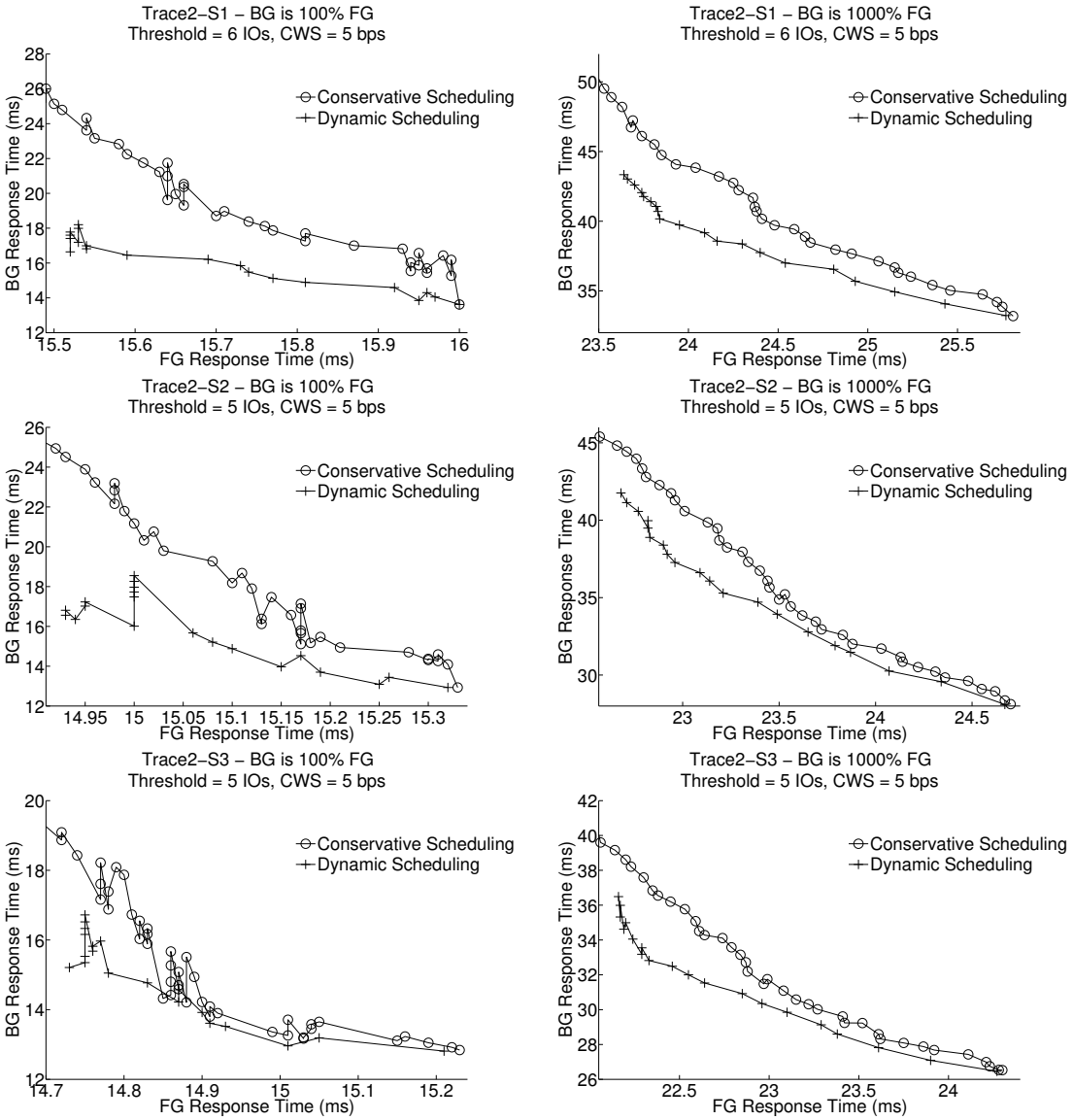



**Figure 3.11**: Scheduling comparison between dynamic and conservative scheduling for Trace2, scheduling results use the three respective periods given in Figure 3.7 (top, middle, bottom rows) to schedule in the next 5 hours.

any foreground performance targets in the system. As expected, the foreground response time increases as the value of the idle wait decreases. For an idle wait of zero (i.e., corresponding to the aggressive scheduling policy), there is almost no distinction between

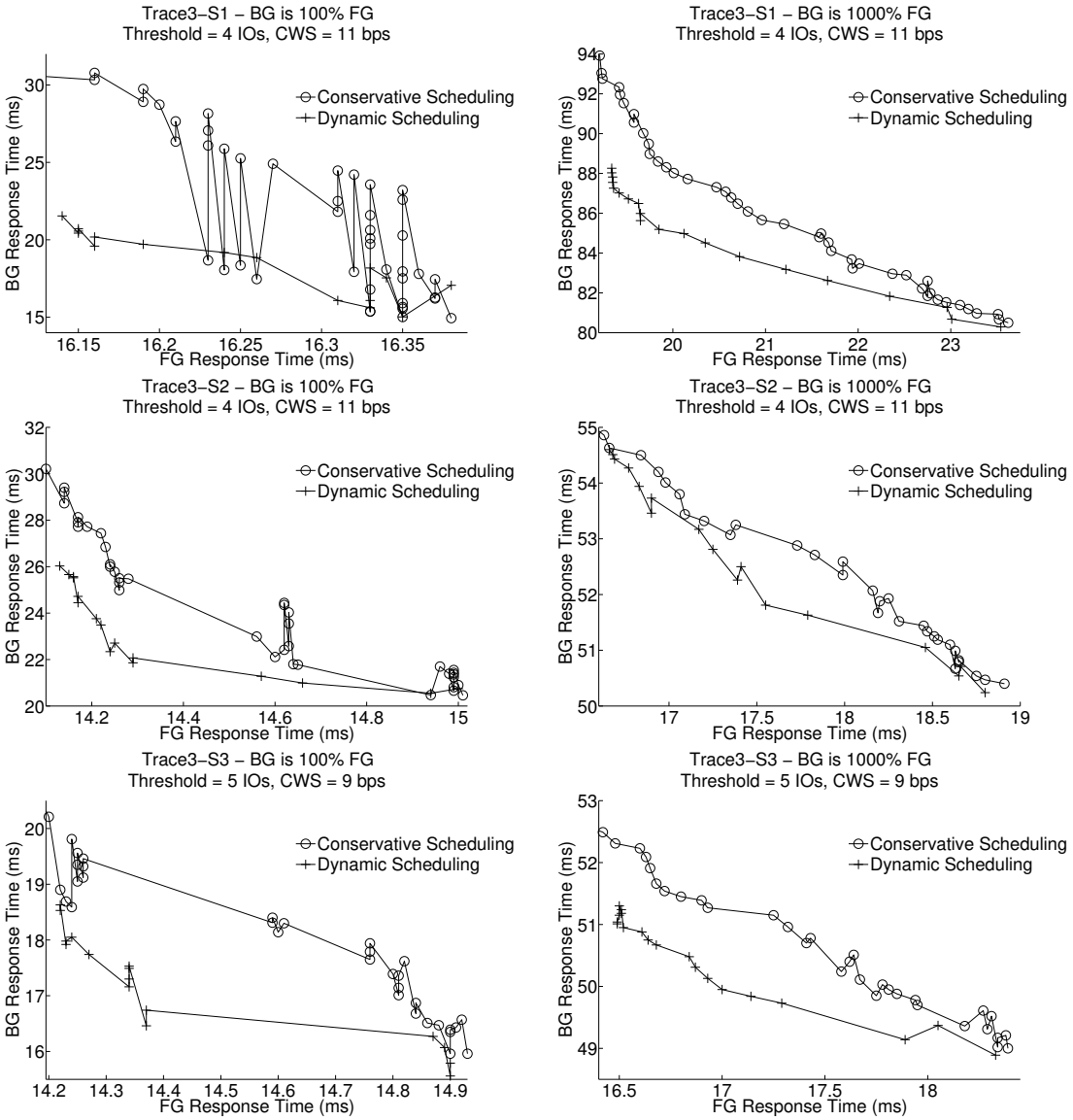



**Figure 3.12**: Scheduling comparison between dynamic and conservative scheduling for Trace3, scheduling results use the three respective periods given in Figure 3.8 (top, middle, bottom rows) to schedule in the next 5 hours.

the foreground and background work, because the background work starts executing as soon as the system becomes idle. Our scheduling always converges to this case in all plots (see the rightmost points in Figures 3.10, 3.11, and 3.12). Across all graphs, the more the

background work (see the differences between the rows of plots), the higher the foreground degradation and background response time. Results can be summarized as follows:

- **Trace1:** Figure 3.10 clearly indicates that there are consistent gains across all time periods and for all amounts of background work. The dynamic scheduling can often speed up background work by as much as 50 percent.

- **Trace2:** Figure 3.11 shows that the gains of dynamic scheduling reduce when compared with the results of Trace1 because the probabilities of a long busy period being followed by another long busy period within a certain lag reduce (compare Figure 3.6 with Figure 3.7). However, dynamic scheduling consistently outperforms conservative scheduling, particularly for large idle waits that are captured by the leftmost part of the plots.

- **Trace3:** Figure 3.12 shows that Trace3 behaves similarly to Trace2. Note that the dynamic scheduling is more robust than the conservative one, which causes fluctuation on performance of background work. This is a result of variability in both idle and busy periods.

One of the most important observations is that for longer idle wait times (left portion of each plot) where foreground performance is degraded less, the dynamic scheduling consistently outperforms the conservative one. As a result, in cases when there are stringent performance targets for foreground requests, the performance advantage of dynamic scheduling is clear. If the foreground work is less sensitive to delays, then conservative scheduling with short idle waits results to a simple and good solution. In general, aggressive scheduling is not a good practical choice because it may cause severe or unbounded delays to foreground performance.

Another characteristic of the dynamic scheduling policy that sets it apart from the conservative one, is its resilience with regard to changes in the workload and scheduling parameters. In all evaluated scenarios in Figures 3.10, 3.11, and 3.12 the results from the dynamic scheduling are gradually reflecting the change, i.e., there are no oscillations on performance as it is often the case for the conservative scheduling. This is a direct outcome of the fact that our dynamic scheduling adapts its parameters to the changes in workload characteristics while the conservative or aggressive policies are oblivious to the workload characteristics. Such gradual changing behavior as characteristics change is desirable in systems because it allows applications to run smoothly.

Of particular importance is the sensitivity of the scheduling policies toward the chosen idle wait value. Figures 3.10, 3.11, and 3.12 show that the performance of both foreground and background work under the proposed dynamic scheduling policy varies but in a significantly narrower range than under the conservative scheduling policy. This implies that for the dynamic scheduling policy, identifying the optimal idle wait value is not critical. Applying the common practices that suggest to select an idle wait as a function of foreground service demands would yield satisfactory results. Overall, we conclude that the dynamic scheduling policy is robust and consistently achieves fast service of asynchronous tasks while sustaining foreground performance.

### 3.4.3 Evaluation Scenario Two: Swiftly Changing Workload

We concatenate Trace2 and Trace1 to obtain a new trace which we name Trace4. Trace4 is used to evaluate the adaptivity of the proposed scheduling policy as the workload changes swiftly. The new trace has a 24-hour span. Because Trace2 and Trace1 have different characteristics (e.g., *Threshold* and $CWS$), we expect a significant change around the 12th hour in the characteristics of Trace4. In order to capture the behavior of the dynamic

scheduling policy, we chose to show here the following three learning windows from the 24-hour duration of Trace4.

- Period 1: learning window from the beginning up to the 8th hour; scheduling decisions apply from the start of the 9th hour through the 16th hour (i.e., learning happens before the workload change and applies during the workload change).
- Period 2: learning window from the 6th hour to the 14th hour; scheduling decisions apply from the start of the 15th hour through the 22nd hour (i.e., learning includes only a small portion of changed workload and applies over the period after the workload change).
- Period 3: learning window from the 8th hour to the 16th hour; scheduling decisions apply from the start of the 17th hour through the 24th hour (i.e., learning has equal portion before and after the workload change and applies over the period after the workload change).

Figure 3.13 shows the conditional probabilities for the three time windows and reflects the workload changes. We note also that *Threshold* changes gradually from 6 in the leftmost plot to 4 in the rightmost plot as the observed amount of Trace1 increases.

We present the scheduling results in Figure 3.14. We observe that the dynamic scheduling policy is robust and consistently performs well, even during the workload transition periods. Performance improves as the learning window includes more of Trace1 (e.g., note the differences in the foreground and the background performance in the center and rightmost columns). Overall, we conclude that the learning process incorporated in the dynamic scheduling algorithm, enables the scheduling policy to adapt well even to swift changes in workload characteristics.

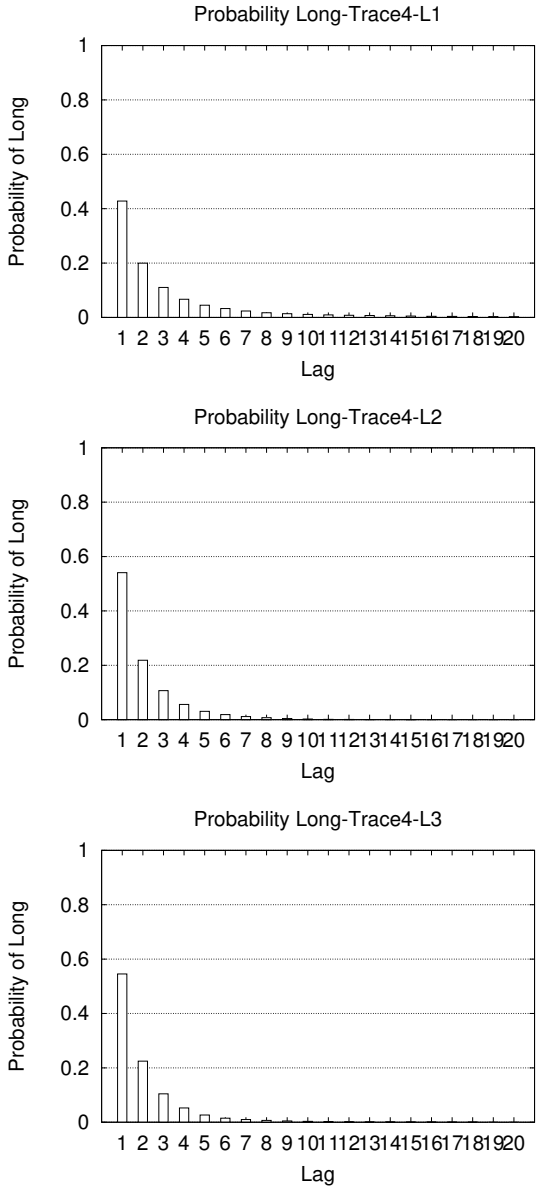



**Figure 3.13**: Probability plots that a long busy period is followed by a similar long one for lag 1 to lag 20 for different portions of Trace4. Three windows are considered: $Start = 0$, i.e., starting at the beginning of the trace (top graph), $Start = 6$ *hour* (middle graph), and $Start = 8$ *hour* (bottom graph).

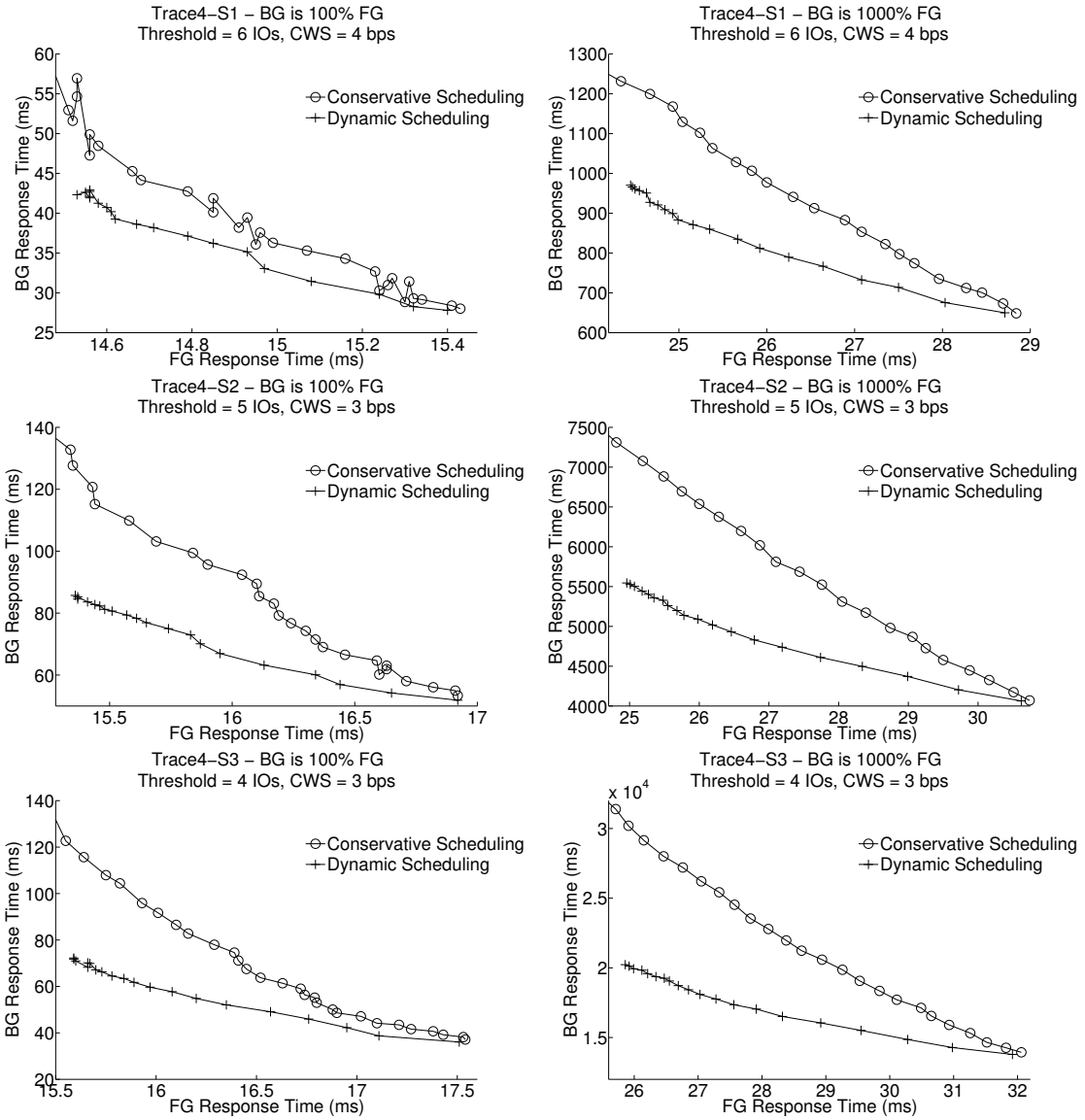



**Figure 3.14**: Scheduling comparison between dynamic and conservative scheduling for Trace4, scheduling results use the three respective windows given in Figure 3.13 (top, middle, bottom rows) to schedule in the next 8 hours.

## 3.5 Summary

In this chapter, we propose a dynamic framework for scheduling background tasks, often associated with eventual consistency in geographically distributed storage systems. The

framework ensures that the performance of foreground traffic is sustained while data consistency is achieved as fast as possible. We define a metric that measures the likelihood of busy periods of similar length arriving in a clustered way. This metric allows us to identify patterns in the length of busy periods and their probabilistic arrival. The reasoning behind the proposed scheduling framework is that if there is a sequence of short busy periods, then the system schedules aggressively the background work without much impact on foreground performance. If the sequence of long busy periods is detected, then scheduling of background tasks is done conservatively during the anticipated duration of long busy periods, i.e., only long idle intervals are used for serving background work. Extensive trace-driven experimentation shows that the framework is effective and robust. It achieves better response time for the background work without degrading performance of foreground traffic. The main findings of this chapter are also reported in [129].

# Chapter 4

# Performance, Power, and Reliability Framework

Storage systems in data centers host thousands of disk drives. Despite the emerging new storage technologies such as solid state drives (SSDs), it is the hard disk drives (HDDs) that continue to store the overwhelming majority of corporate data [104, 43, 76]. Specifically, hard disk drives are expected to store aging data (from a few weeks old to several years old) which are expected to grow in size over the years. Given the characteristic of data stored in HDDs, it is expected that not all data in a vast data center is accessed simultaneously. Consequently, a compelling approach for reducing power consumption in data centers is to spin down idle hard disk drives. This approach is routinely deployed in storage systems that serve as archival or backup systems [19] and is being exploited even in high-end computing environments [74].

Spinning down disk drives to save energy in a high-end environment *transparently* to the end user and *reliably* to the disk drive's lifetime is a challenging open problem for a host of reasons. First, in enterprise environments, requests that arrive while the drive is in a power saving mode are to be inevitably delayed during the time it takes for the disk drive

to reactivate, e.g., to be physically ready to serve jobs again. Second, idle times can be highly fragmented while the overall drive utilization is very low, therefore idle periods that are long enough to be used effectively for power savings may be very few [89]. Third, every power up/down wears out the disk drive, which implies strict limitations on the number of times a disk drive can be placed into a power savings mode without affecting its reliability.

Common practice methods try to address these challenges by idle waiting for a fixed amount of time or use the past utilization to guide future scheduling decisions. However, these common practice methods cannot provide performance guarantees nor take into consideration disk reliability. In order to overcome these short-comings, we develop PREFiguRE, a framework that uses as input user- or system-level constraints such as the number of allowable power ups/downs of a disk within a time period (strict constraint) and the user acceptable potential performance degradation of future IOs (soft constraint), and estimate the projected power savings as well as provide a strategy on *how* these power savings should occur. PREFiguRE uses as a basic tool the histogram of past idle times and projects future power savings based on statistical information that is monitored or extracted from this histogram. Probabilistic interpretation of all of the above information leads PREFiguRE to define robust schedules for power saving modes. As the workload changes in the system, the histogram of idle times as well as information about the sequence of idle times, are updated. Such updates enable the adjustment of the schedules of power saving activation to the workload dynamics.

The core of PREFiguRE is a robust, accurate, and computationally efficient analytic model that enables the identification of effective, user-transparent schedules of power saving modes in disk drives. Most importantly, the analytic model that is encapsulated in PREFiguRE encompasses a strong reliability component to comply with the restrictions on the number of times a hard disk can go into a specific power saving mode during its

lifetime [55]. In addition, thanks to the excellent prediction accuracy of the model, it is possible to answer a wide range of questions regarding the power saving capabilities of the current disk workload i.e., if the power gains are projected to be marginal then it may not be worth engaging the system in any power savings mode or it may signal that part of the workload should be offloaded (to a buffer or to another disk) such that idle times, and consequently, power savings, are increased.

Although the main contribution of our framework lies in its theoretical aspect, we also conduct trace driven simulations to verify its practical benefit. We drive the evaluation of PREFiguRE via a set of enterprise disk drive traces with a wide range of idleness characteristics. The excellent agreement between the results from PREFiguRE's analytic estimations and trace driven simulations suggests that our analytic methodology can achieve good accuracy and robustness even under the real world workloads.

The rest of this chapter is organized as follows. Section 9.1.2 summarizes the power savings opportunities in disk drives and storage systems. In Section 8.3, we present the methodology that we propose to identify and estimate the power savings opportunities in a system under a given workload. We validate the effectiveness of the approach and illustrate its robustness in Section 8.4 using trace-driven analysis and simulations. We summarize this chapter in Section 9.

## 4.1 Power Saving Modes in Disk Drives

Disk drives represent the overwhelming majority of the storage devices deployed in large data centers where power conservation is a priority. Individual disk drives consume moderate amount of power when compared with other components in a computer system. However, disk drives tend to be more idle than other system components. This is particularly true in large data centers that deploy thousands of disk drives and host

terabytes and petabytes of data, which are not all accessed simultaneously.

Disk drives are complex hardware devices that consist of both mechanical and electronic components. The mechanical components, such as the platters that rotate at high speeds, or the positioning arm that is kept at a specific distance away from the platters, continue to consume power even when not accessing data. Similarly, the electronics in a disk drive consume power even during periods of idleness. Overall disk drives consume less power when they are idle than when they serve IOs.

Beyond the moderate power savings when an active disk is idle (i.e., at the "active idle" state), additional power can be saved by slowing down components in a disk drive, such as platter rotation, or by unloading and parking the heads (and the positioning arm) on the side instead of flying them at constant height over the platters. Finally, completely shutting down the disk drive eliminates almost the entire power consumption from the disk drive. Slowing or shutting down the disk comes with a performance cost to user IOs, because bringing the disk back to its active state requires time which ranges from hundreds of milliseconds to tens of seconds. The required time period to reactivate a disk drive can be viewed as an unavoidable performance *penalty* paid by those IOs that by arrival find the disk drive that stores their data in an inactive (i.e., power saving) mode.

There are several levels of power consumption depending on the state of the disk's mechanical and electronic components. Each power consumption level or mode is characterized by the amount of *power* it consumes and the amount of *time* it takes to get out of the power saving mode and become ready to serve IOs. The exact amount of power saved in a given power saving mode or the amount of time it takes to become ready again, differs between disk drive families and manufacturers. Table 4.1 presents a coarse description of the possible power saving modes focusing on the components that are slowed down or shut off, and the penalties associated with each power saving mode.

| Operation Mode | Description | Power savings | Penalty (sec) |
|---|---|---|---|
| Level 1 | Serving IOs | 0% | 0.0 |
| Level 2 | Active (but) idle | 40% | 0.0 |
| Level 3 | Unloaded heads | 48% | 0.5 |
| Level 4 | Slowed platters | 60% | 1 |
| Level 5 | Stopped platters | 70% | 8 |
| Level 6 | Shut down | 95% | 25 |

**Table 4.1**: Characteristics of power saving modes.

The reported penalty values are within representative ranges published by disk drive manufacturers [91, 90, 49]. For example, the penalty (in seconds) for Level 6 is between 23 (typical) and 30 (max) [91] page 7.

Note that during the process of bringing a disk drive out of a power saving mode, the consumed power surges before settling to a normal consumption level. As with the power savings in Table 4.1, this power surge during reactivation depends on the drive family and manufacturer.

The time it takes a disk to become active following a power saving mode make obvious the need to account for the performance penalty before deciding on a disk operation mode for power savings. One could argue that putting the disk into an idle mode immediately after any idleness is detected could maximize power savings. Given the stochastic nature of the length of idle times and the penalty to bring the disk up to active mode, it is important to use idle intervals that are sufficiently large (i.e., longer than the reactivation time) for power savings. In storage systems it is very common to not put the system automatically in a power saving mode when an idle interval is observed. Instead the system waits for a time period in anticipation of future IO arrivals.

In addition to the performance penalty associated with reactivating a disk drive that is put in a power saving mode, there is a reliability penalty as well. The later is not

straightforward to quantify, because it is associated with the wear out of the disk drives during power ups (i.e., in the spin-up phase) or reactivation of individual components. In disk drives, the spin-up/down (Level 5 and 6 in Table 4.1) involves certainly more components than loading/unloading heads (i.e., Level 3 in Table 4.1) or spinning platters slower while heads are parked on the side (i.e., Level 4 in Table 4.1). While spin-up and spin-down have been analyzed for years as part of the disk drive wear out process [62], the cost of head load/unloading in disk drives is more recent and is introduced solely for the purpose of power savings [55]. As it is discussed in the following sections, in an enterprise environment, loads/unloads (Level 3) are expected to occur more often because the penalty to bring the HDD into the active state is smaller than the other power saving levels. During its lifetime a disk drive is expected to survive well beyond 300,000 loads/unloads [55], which is used as a threshold in the methodology in this work.

In the following section, we present a framework that determines when and for how long a disk drive should be put into a power saving mode without violating a pre-defined quality of service target. The framework takes into consideration both the performance and reliability penalties associated with disk drive power saving modes.

## 4.2 Algorithmic Framework

Here, we develop an algorithmic framework that determines the schedule of the periods when a disk drive is placed in power saving modes, such that pre-defined targets of system quality metrics are met. There are three system quality metrics used in the framework. They include the performance degradation $D$, the portion of time the disk is placed in power saving modes $S$, and the reliability constraint $X$. A definition of these metrics, as well as other notations used in the framework are given in Table 4.2. Note that it is not necessary to have all three system quality metrics set. For example, if only the performance

target $D$ and the reliability target $X$ are set, then the framework can meet those targets while the third one, i.e., power saving $S$, is maximized. It is also possible to set all three metrics, but whether all targets can be met depends on the viability of the workloads. Note also that the application performance can be impacted by many factors (e.g., CPU, memory, networking), thus for an unbiased analysis, we focus only on the disk performance itself, which is measured by the average response time of IO requests.

In addition to the system quality targets, our framework bases its calculations on a set of monitored (or pre-defined) input metrics. In particular, it uses the time penalty $P$ that is necessary to bring a disk drive out of a specific power saving mode. Recall that different power saving modes have different penalties $P$. However, because $P$ depends on the disk drive model, the correct $P$s for a given disk drive can be either received from the manufacturer or measured in off-line testing. Note that $P$ is the *extra* delay due to power saving. This delay is in addition to any queuing delays that requests may experience due to bursty or heavy arrivals. Throughout this chapter, the focus is on estimating and reducing the delay due to power saving. The set of monitored metrics used in our framework include the *cumulative data histogram (CDH)* of idle times observed in the system and the *average response time RT* of IOs (excluding any slow down effect previous power saving modes may have had on average IO response time). The CDH is a list of tuples (at most a few thousands of them). We stress that this representation is very efficient both memory-wise and computation-wise. As we show later in this section, the estimation of scheduling parameters to meet the required targets only requires a few scans of the CDH, which can be executed almost instantaneously. Each tuple contains a range of idle interval lengths and their corresponding empirical cumulative probability. Note that the CDH of idle times is used to capture the characteristics of the overall workload in our framework. As a result, the granularity of the CDH bins determines the accuracy of the estimations and

| Input parameters | |
|---|---|
| $D$ | *Quality metric - performance:* relative average response time increase due to power savings (in %). |
| $S$ | *Quality metric - power savings:* portion of time in power savings (in %). |
| $X$ | *Quality metric - reliability:* the number of reactivations per time unit a disk can have without impacting its lifetime. |
| $P$ | Penalty due to power savings (i.e., time to reactivate a disk from a specific power saving mode). |
| **Monitored metrics** | |
| $p(j)$ | Probability of idle interval of length $j$. |
| $CDH(j)$ | Cumulative probability of an idle interval of length at most $j$. |
| $E[idle]$ | Average idle interval length. |
| $RT$ | Average IO request response time. |
| **Intermediate metrics** | |
| $W$ | Average additional wait time IO requests experience due to the disk in a power saving mode. |
| $w_i$ | Additional waiting time affecting IOs in the $i^{th}$ busy period following a power saving mode. |
| $Prob_i(w)$ | Probability of $w$ waiting time for the IOs in the $i^{th}$ busy period following a power saving mode. |
| $i_j$ | Length of the $j^{th}$ idle interval following a power saving mode. |
| $Prob(LL_l)$ | Probability of two idle intervals of at least length $L$ to be $l$ lags apart. |
| **Output parameters and estimated metrics** | |
| $I$ | Amount of time that should elapse in an idle disk before it is put into a power saving mode. |
| $T$ | Maximum amount of time that a disk is kept in a power saving mode. |
| $D_{(I,T)}$ | Achieved average degradation of response time due to power savings. |
| $S_{(I,T)}$ | Achieved time in power savings. |

**Table 4.2**: Notation used in Section 8.3. All time units are in ms.

calculations. The coarser the CDH, the less accurate our solution is.

The monitored metrics can be easily obtained from the arrival and departure times of IO requests in the system, which are generally monitored or can be monitored without complex instrumentation. The framework adapts its decisions to changes in workload (captured via the histogram of idle times, system utilization, and average IO response time) and other

inputs. As a result, the output of the framework, i.e., the schedule of the power saving modes, changes if the workload that arrives in the disk drive changes or if the system quality targets change. For example, in an enterprise storage system the performance quality target $D$ can be adjusted to be more stringent during the day (i.e., business-hours) and less stringent during the night (i.e., non-business hours). Another example is that the framework can estimate for a given performance target $D$ and reliability target $X$ the time in power saving if the Level 3 is used or if Level 4 is used. Comparing the resulting time in power saving $S$ allows the system to decide which power saving mode to use (if any) for the current workload.

In our framework, power saving modes *always* take advantage of only the *idle periods* in a disk drive and are not purposely scheduled if user requests are waiting for service in the system. This condition must be satisfied even if the target power saving $S$ is set and not met. Here, we assume that user workload has always a higher priority, although our framework can be adapted to a situation where power savings have the same priority or higher than the performance of user workload.

Given this consideration, we model the power saving modes as *low priority tasks* that need $P$ units of time to be preempted. The IO requests arriving in the system are modeled as *high priority tasks*. Because the penalty $P$ to preempt the low priority work, i.e., the time to reactivate the disk, is orders of magnitude higher than the expected service and response time of user IOs, the performance impact that power saving modes could have on user IOs may be significant. Our framework schedules power saving modes in disk drives proactively, i.e., average IO slowdown is limited to the performance target $D$. The framework achieves its targets by scheduling power saving modes according to parameters $I$ and $T$, where

- $I$ represents the amount of time the system remains idle before a power saving mode

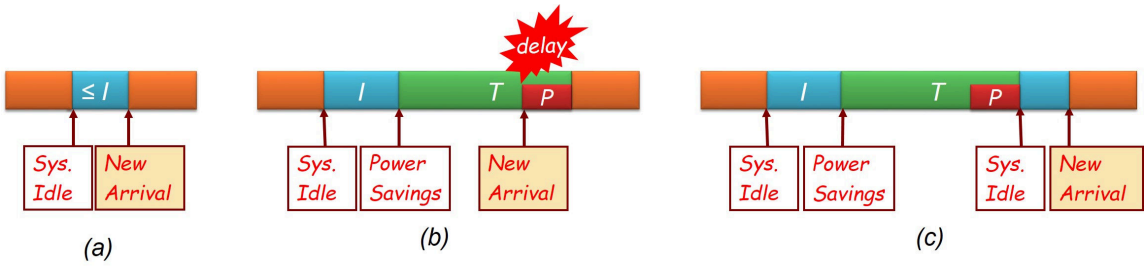



**Figure 4.1**: Examples of relationship between idle periods length, requests arrival and parameters $I$, $T$ and $P$. The orange represents busy times, blue represents idle times, green represents power saving mode.

starts, and

- $T$ represents the maximum amount of time the disk remains in a power saving mode (i.e., if an IO arrives before $T$ elapses, the power saving mode is interrupted). $T$ includes the penalty $P$ which implies that $T > P$.

The scheduling pair $(I, T)$ is recalculated every time the monitored metrics are updated or the system quality target changes adapting the scheduling of power saving modes to the dynamics in the storage system.

Figure 4.1 demonstrates three examples of the relationship between idle periods length, arriving requests, and parameters $I$, $T$, and $P$. Figure 4.1(a) shows the idle period being smaller than $I$; Figure 4.1(b) shows when the idle period being larger than $I$ but smaller than $I + T$; Figure 4.1(c) is an example of idle period that is larger than $I + T$.

### 4.2.1 Modeling Waiting Times Due to Power Saving Modes

In our framework, the scheduling pair $(I, T)$ is calculated such that it guarantees the quality targets (reliability, performance, and/or amount of power savings). In order to meet the performance or power saving target, it is critical to estimate correctly the *waiting time* (or delay) caused to IOs arriving during or after a power saving mode. Without loss of generality, we measure the idle interval length as well as the wait within the 1 ms

granularity. The coarser the granularity, the less the accuracy, but the monitoring overhead is expected to reduce.

Assume that $W$ is the average IO waiting due to the power savings, i.e., $W$ =// $RT_{w\ power\ saving} - RT_{wo\ power\ saving}$. Because a disk is loaded upon an IO arrival, $W$ can be at most $P$, i.e., the time it takes the disk to become active. By denoting a possible delay by $w$ and its respective probability by $Prob(w)$ then

$$W = \sum_{w=1}^{P} w \cdot Prob(w). \tag{4.1}$$

We define a busy period as the time period that there are one or several IO requests being served without idle time between requests. The power saving mode preemption time $P$ may be longer than the average idle interval. As a result, the delay due to a power saving mode may not be absorbed by the immediate following idle period and may *propagate* to impact multiple user busy periods. Figure 4.2 shows the example of no delay propagation and with delay that propagates two busy periods. As shown in Figure 4.2(a), the idle period following the second busy period is longer than the delay caused by power savings, therefore, the delay is absorbed and does not propagates further. Figure 4.2(b) is an example of delay propagates two busy periods. The idle period after the second busy period is very short and the delay caused by power savings propagating into the third busy period. Therefore, IO requests in both the second and third busy periods are delayed. Although all IOs in one busy period get delayed by the same amount, the delay propagates to multiple busy periods and different delays may be caused to IOs in the future busy periods because of the activation of a single power saving mode.

To estimate $Prob(w)$ of a delay $w$, we identify the events that happen during disk reactivation that result in a delay $w$ and estimate their corresponding probabilities. These events are the basis for the estimation of the average waiting $W$ due to power savings.

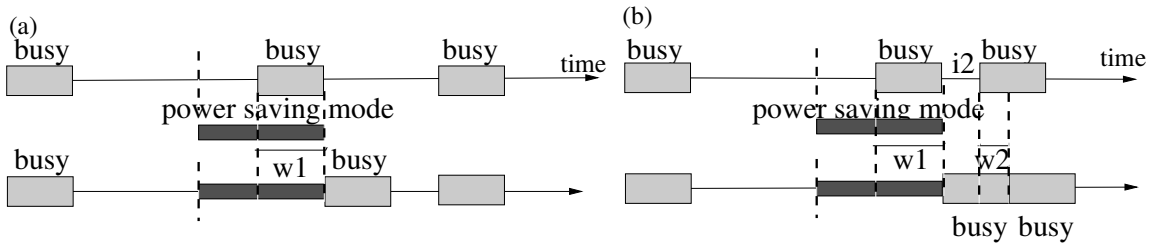



**Figure 4.2**: (a) No delay propagation. (b) Delay propagates two busy periods.

Without loss of generality, we assume that a disk reactivation affects at most $K$ consecutive user busy periods. The larger the $K$ the more accurate our framework is. In general, the larger $P$, the larger the value of $K$ should be for better estimation accuracy. In our estimations, $K$ is set to be equal to $P$, which represents the largest practical value that $K$ should take. During disk reactivation, the delay propagates as follows:

- *First delay*: User IOs arrive during a power saving mode or disk reactivation and find an empty queue and a disk that is not ready for service. These IOs would have made up the *first* user busy period if the disk would have been ready. Their waiting due to power saving is $w_1$ ms (where index $i = 1$ indicates the first busy period and $1 \leq w_1 \leq P$).

- *Second delay*: User IOs in the "would-be" second busy period in the absence of the power saving mode, could also be delayed if the above wait $w_1$ is longer than the idle interval $i_2$ that would have followed the above first busy period. The waiting time experienced by the IOs of the second busy period following a power saving mode is $w_2 = (w_1 - i_2)$.

- *Further propagation*: In general, the delay propagates through multiple consecutive user busy periods until all the intermediate idle periods absorb the initial delay $w_1$. Specifically, the delay propagates for $K$ consecutive user busy periods if $(i_2 + i_3 + ... + i_K) < w_1 < (i_2 + i_3 + ... + i_K + i_{K+1})$. The waiting times experienced by the IOs due to this power saving mode are $w_j$ for $1 \leq j \leq K$.

Denoting with $Prob_k(w)$ the probability that wait $w$ occurs to the IOs of the $k^{\text{th}}$ delayed

busy period, we estimate the probability of delay $w$ as $Prob(w) = \sum_{k=1}^{K} Prob_k(w)$. The delay $P$ may occur only to IOs of the first delayed busy period, because for the IOs of the second (or higher) delayed busy period the intermediate idle interval would absorb some of the delay and would therefore reduce it. The same argument can be used to claim that the delay of $P-1$ can occur to only IOs of the first and second delayed busy periods. In general, it is true that the delay $w = P - k$ may occur only to the IOs of the first $k+1$ delayed busy periods ($0 \leq k \leq K$).

The fact above is used as the base for our recursion that computes $Prob(w)$ for $1 \leq w \leq P$. The base is $w = P$ and $Prob(w = P) = Prob_1(P)$ because the delay $P$ is caused *only* to the IOs of the first delayed busy period. For a scheduling pair $(I, T)$, the delay to the first busy period following a power saving mode is $P$ for all idle intervals whose length falls between $I$ and $I + T - P$. The probability of this event is given as CDH($I + T - P$) - CDH($I$), where CDH(.) indicates the cumulative probability value of an idle interval in the monitored histogram.

The delay $w$ caused to the IOs in the first busy period following a power saving mode may be any value between 1 and $P$. This delay can not exceed $P$ since $P$ is the time that the disk required to revert from power saving mode to serving mode. Using the CDH of idle times, the probability of any delay $w$ caused to the IOs of the first busy period are given by the equation below

$$Prob_1(w) = \begin{cases} \text{CDH}(I + T - w + 1) - \text{CDH}(I + T - w), & \text{for } 1 \leq w < P, \\ \text{CDH}(I + T - P) - \text{CDH}(I), & \text{for } w = P, \end{cases} \tag{4.2}$$

If the length $i_2$ of the idle interval following the first delayed busy period is less than $w$, then the IOs of the second busy period may be delayed too by $w - i_2$. The IOs of the second busy period are delayed by $w - i_2$ if (1) the idle interval following the first delayed busy

period is $i_2$, which happens with probability $Porb_(i_2)$ and (2) the first delay was $w + i_2$, which happens with probability $Porb_1(w + i_2)$. Since there is the independence between the arrival and service processes, the delay propagation is also independent of the process of idle lengths. Therefore, the probability $Prob_2(w)$ is given by the equation

$$Prob_2(w) = \sum_{j=1}^{P-w} Prob_1(w + j) \cdot p(j), \tag{4.3}$$

where $Prob_1(w+j)$ for $1 \leq j \leq P-w-1$ is defined in Eq. (6.2) and $p(j)$ is the probability of an idle interval of length $j$.

The delay $P - 1$ can occur only to the IOs of the first busy period with probability $Prob_1(P-1)$ and to the second busy period with probability $Prob_2(P-1)$. Using Eqs. (6.2) and (4.3), we get

$$Prob(P - 1) = Prob_1(P - 1) + Prob(P) \cdot p(1). \tag{4.4}$$

This implies that $Prob(P-1)$ depends only on $Prob_1(.)$ and $Prob(P)$ which are both defined in Eq. (6.2) and represents how the base $Prob(P)$ of our recursion is used to compute the next probability $Prob(P - 1)$.

Similarly, we determine the probabilities of delays propagated to the IOs of the busy periods following the power saving mode and establish recursion for all $1 \leq w \leq P$. For clarity, we show how we develop the next step recursion and then generalize. Specifically, delay $w$ is caused to the IOs of the third delayed busy period and $w$ takes values from 1 to at most $P - 2$ (recall that the granularity of the idle interval length is 1 ms).

$$Prob_3(w) = \sum_{j=1}^{P-w} Prob_1(w + j) \sum_{j_2=1}^{j-1} Prob_2(j - j_2) \cdot p(j_2). \tag{4.5}$$

The delay of $P - 2$ does not propagate beyond the third delayed busy period and its

probability is given as the sum of probabilities of its occurrence to IOs of the first delayed busy period, $Prob_1(P-2)$, second delayed busy period, $Prob_2(P-2)$, and third delayed busy period, $Prob_3(P-2)$. Using Eqs. (6.2), (4.3), and (4.5) we obtain

$$\begin{aligned} Prob(P-2) &= Prob_1(P-2) + \\ &\quad Prob_1(P-1)\cdot p(1) + Prob_1(P)\cdot p(2) + \\ &\quad Prob_1(P)\cdot p(1)\cdot p(1) \end{aligned} \tag{4.6}$$

Substituting $Prob_1(P-1) + Prob(P)\cdot p(1)$ with $Prob(P-1)$ from Eq. (4.4) we get

$$\begin{aligned} Prob(P-2) &= Prob_1(P-2) + \\ &\quad Prob(P-1)\cdot p(1) + Prob(P)\cdot p(2). \end{aligned} \tag{4.7}$$

In general, for the $k^{\text{th}}$ delayed busy period, delay $w$ occurs with probability $Prob_k(w)$ given by the equation

$$\begin{aligned} Prob_k(w) &= \sum_{j=1}^{P-w} Prob_1(w+j)\cdot \\ &\quad \sum_{o_2=1}^{j-1} Prob_2(j-o_2)\cdot \\ &\quad \sum_{o_3=1}^{o_2-1} Prob_3(o_2-o_3)\cdot \ldots \cdot \\ &\quad \sum_{o_{k-1}=1}^{o_{k-2}-1} Prob_{k-1}(o_{k-2}-o_{k-1})\cdot p(o_{k-1}). \end{aligned} \tag{4.8}$$

Recursion in Eq. (4.7) is generalized using probabilities defined in Eq. (4.8) as follows

$$Prob(w) = Prob_1(w) + \sum_{j=w+1}^{P} Prob(j) \cdot p(j-w). \tag{4.9}$$

To estimate the average delay $W$, first all $Prob_1(w)$ for $1 \leq w \leq P$ can be estimated using Eq. (6.2). Then starting from $w = P$, all probabilities $Prob(w)$ for $1 \leq w \leq P$ are computed using the recursion in Eq. (4.9). Note that the granularity of the CDH bins determines the granularity of the recursion step. In the above presentation, we assumed, without loss of generality, that each bin is 1 ms.

We stress that only $Prob_1(w)$ for $1 \leq w \leq P$ in Eq. (6.2) depends on the scheduling pair $(I, T)$. The rest depends on the probabilities of the monitored CDH of idle times (as depicted in Figure 4.3). This is important to the computational complexity of the framework because the majority of components in the recursion of Eq. (4.9) are computed only once.

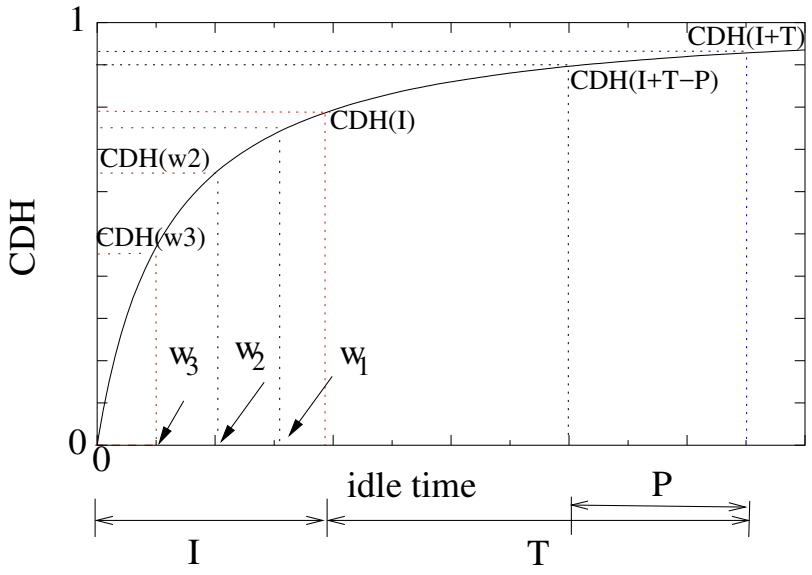



**Figure 4.3**: Estimation of probabilities for propagation delay.

### 4.2.2 Meeting Performance Target $D$

Here, we develop the method to determine the pair $(I, T)$ for scheduling the power saving modes such that performance does not degrade more than the target percentage $D$ on the average. Because we want to control performance degradation, $T$, the time that the disk stays in a power saving mode includes the penalty $P$ (i.e., $T > P$) for reactivating the disk and represents a proactive measure to control performance degradation.

To find the best scheduling pair $(I, T)$, we scan the CDH of idle times for $(I_l, T_j)$ pairs that would not violate the target $D$. Note that $I_l$ and $I_l + T_j$ correspond to successive histogram bins. A pair $(I_l, T_j)$ guarantees the performance target $D$ if

$$D \geq \frac{W_{(I_l,T_j)}}{RT_{w/o\ power\ saving}}, \tag{4.10}$$

where $RT_{w/o\ power\ saving}$ is monitored and $W_{(I_l,T_j)}$ is computed using Eq. (6.1) and Eq. (4.9).

If $(I_l, T_j)$ satisfies the performance target $D$, then the corresponding "time in power savings" $S_{l,j}$ can also be computed. Because $T_j$ includes $P$, for all idle intervals longer than $(I_l + T_j - P)$, the time in power saving is $(T_j - P)$. For all idle intervals with length $i$ between $I_l$ and $I_l + T_j - P$, the time in power saving $i - I_l$ becomes

$$S_{l,j} = \frac{\sum_{o=I_l}^{I_l+T_j-P} p(o) \cdot (o - I_l)}{E[idle]} + \frac{\sum_{o=I_l+T_j-P}^{max} p(o) \cdot (T - P)}{E[idle]}, \tag{4.11}$$

where $max$ is the value of the last bin in the CDH, and $E[idle]$ is the average idle interval length.

We choose the scheduling pair $(I, T)$ to be the pair $(I_l, T_j)$ that results in highest time in power saving $S_{l,j}$. Recall that the estimation of $S_{l,j}$ is done only for these pairs $(I_l, T_j)$ that meet the performance degradation target $D$ of Eq. (6.3).

The computational complexity of the procedure to choose the scheduling pair $(I, T)$ is $O(n^2)$, where $n$ is the number of CDH bins. Note that the recursion for estimating $W$ has a time complexity of $O(n)$.

### 4.2.3 Meeting Power Target $S$

In this scenario, the system quality target is the time in power savings $S$, which means that the scheduling pair $(I, T)$ should achieve a time in power saving of at least $S\%$. The scheduling pair $(I, T)$ should satisfy the targeted time in power saving $S$ and degrade performance at the lowest possible minimum, i.e., in this scenario there is no $D$ defined. Note that if $S$ is larger than the idleness in the system, then our procedure does not estimate an $(I, T)$ pair, because power savings should not be scheduled when there are user requests outstanding.

Here, we need to find the scheduling pair $(I, T)$ that meets the target $S$ and causes the smallest performance degradation $D$. If every idle interval would be used for power saving, then $S$ can be expressed as the time in power savings per idle interval $\overline{S}$ and would relate to the average idle interval length $E[idle]$ and utilization $U$ according to the equation

$$\overline{S} = \frac{S \cdot E[idle]}{1 - U}. \tag{4.12}$$

However, for an $(I_l, T_j)$ pair only $(1 - \text{CDH}(I_l))$ idle intervals can be used for power savings. It follows that the target $S$ can be met only if the time in power saving $T_j - P$ for the idle intervals that to be used for power saving is such that if normalized over all idle intervals, then it is at least $\overline{S}$, as shown by the following equation

$$\overline{S} = \frac{S \cdot E[idle]}{1 - U} \leq (T_j - P)(1 - \text{CDH}(I_l)). \tag{4.13}$$

All possible pairs $(I_l, T_j)$ as defined by the bin values of the CDH of idle times are evaluated against Eq. (4.13) (a scan that requires $O(n^2)$ steps). Those pairs $(I_l, T_j)$ that satisfy Eq. (4.13) meet the power saving target $S$. Among these pairs, we select the one with the smallest performance degradation $D_{I_l,T_j}$ that is estimated according to Eq. (6.3). The actual anticipated time in power savings for a pair $(I_l, T_j)$ is $S_{I_l,T_j}$ and is estimated using Eq. (4.11).

### 4.2.4 Meeting Reliability Target $X$

The reliability target $X$ is another quality target in our framework and is measured as the rate of power saving modes (measured usually at coarse granularity, e.g., one day) that the disk can have without impacting its lifetime. This rate is equal to the rate of spin ups that a disk can tolerate without premature wear-out.

Let us denote utilization as $U = E[busy]/(E[idle] + E[busy])$, where $E[busy]$ is the average busy interval and $E[idle]$ is the average idle interval. Let us denote $\hat{X}$ as the rate of opportunities for power savings, and $\hat{X} = 1/(E[idle] + E[busy]) = U/E[busy] = (1 - U)/E[idle]$. If $X$ is smaller than $\hat{X}$, then an idle interval should be used for power savings with probability $X/\hat{X}$. Otherwise, all idle intervals are to be utilized for power savings. Denote $\overline{X}$ as

$$\overline{X} = \begin{cases} \frac{X}{\hat{X}}, & \text{for} X < \hat{X} \\ 1, & \text{otherwise.} \end{cases} \tag{4.14}$$

Because a scheduling pair $(I, T)$ uses only (1-CDH($I$)) of idle intervals for power savings, the reliability target $X$ is violated only if (1-CDH($I$)) is larger than $\overline{X}$. In this case fewer idle intervals than (1-CDH($I$)) should be used for power savings. As a result, the delay $W$ should reflect the potential fewer power saving modes and the resulting lower delay. For this, we redefine Eq. (6.2) to reflect that the delay caused to the IOs of the *first* busy

period following a power saving mode happens with probability $\overline{X}$ / (1-CDH($I$)). Note that if $\overline{X} > 1 - \text{CDH}(I)$, then no correction needs to take place, because $X$ is not violated.

Reflecting the reliability target $X$ in Eq. (6.2) results in:

$$Prob_1(w) = C \cdot \begin{cases} \text{CDH}(I+T-w+1) - \text{CDH}(I+T-w), & \text{for } 1 \leq w < P, \\ \text{CDH}(I+T-P) - \text{CDH}(I), & \text{for } w = P, \end{cases} \tag{4.15}$$

where $C$ is defined as

$$C = \begin{cases} \frac{\overline{X}}{1-\text{CDH}(I)}, & \text{for} \overline{X} < 1 - \text{CDH}(I) \\ 1, & \text{otherwise.} \end{cases} \tag{4.16}$$

Using Eq. (4.15) to estimate the first delay ensures that the average delay $W$ is estimated accurately based on Eq. (6.1) and the recursion of Eq. (4.9). As a result, the framework meets both reliability and performance targets. The reliability target is reflected similarly in the estimation of power savings achieved by a scheduling pair $(I, T)$. Eq. (4.11) is updated to account for the reliability target as follows

$$S_{l,j} \quad = \quad C \cdot \frac{\sum_{o=I_l}^{I_l+T_j-P} p(o) \cdot (o - I_l)}{E[idle]} + C \cdot \frac{\sum_{o=I_l+T_j-P}^{max} p(o) \cdot (T-P)}{E[idle]}, \tag{4.17}$$

where $C$ is defined in Eq. (4.16). By using these improved formulas, we can achieve the reliability target.

### 4.2.5 Correlation-Based Enhancement

So far, the scheduling pair $(I, T)$ is computed using heavily the CDH of idle times. As a result, the decisions are made on the probability of an idle interval length assuming that

the sequence of idle intervals is a renewal process. However, the utilization of idle time would improve further if the length of idle intervals is predicted more accurately than by using only the marginal distribution (i.e., CDH). Here we show how to exploit any existing short-term correlation in idle interval lengths.

For this, we define the category of *long idle intervals* as all idle intervals longer than $L$, where $L$ is defined such that idle intervals of at least length $L$ are observed at a rate close to the reliability target $X$. We compute on-line, similar to the CDH of idle times, the probabilities that two consecutive idle intervals, up to $G$ lags apart, are both long. We denote these probabilities as $Prob(LL_l)$, i.e., two idle intervals of at least length $L$ that are $l$ lags apart.

The lag $l$ with the highest $Prob(LL_l)$ is selected for prediction. Although any $Prob(LL_l)$ value can be used in the framework, only if $Prob(LL_l)$ above 0.5 is recommend for good power savings effect because when $Prob(LL_l)$ is above 0.5, the correlation structure is considered as strong and yields a good prediction accuracy. Therefore, once a long idle interval is observed, the upcoming idle interval $l$ lags in the future are also to be used for power savings. This correlation-based prediction is used to enhance the performance of our framework *in addition* to the regularly estimated scheduling pair $(I, T)$.

We argue that if a long idle interval is predicted, then the probability of causing a delay is less than when the regular probabilities in the CDH are used. As a result, we propose to use a shorter $I$ and a longer $T$ without violating the performance target $D$. Specifically, we denote the scheduling pair that results from such prediction as $(I_L, T_L)$, where $I_L$ is defined such that $\text{CDH}(I_L) = 0.5$ and $T_L$ is defined such that corresponds to the length of the long idle interval $L$, i.e., $T_L = L - I_L$. Although we define $L$ such that the occurrence of idle intervals of at least length $L$ is at most $X$, it is expected that for most enterprise workloads the number of idle intervals of length at least equal to $L$ to be less than $X$

within a specified time period. For this reason, we generate two scheduling pairs $(I, T)$ and $(I_L, T_L)$, where the first one is estimated as a regular scheduling pair using the CDH of idle times and it is used to "fill up" the quota $X$ left unused by the second pair.

The most important characteristic in our framework is the ability to estimate accurately performance of a scheduling pair $(I, T)$. In the case when two scheduling pairs are used, we combine the estimations of delay $W$ and power savings $S$ for both scheduling pairs. We define

$$W = (1 - Y) \cdot W_L + Y \cdot W_R, \quad S = (1 - Y) \cdot S_L + Y \cdot S_R, \tag{4.18}$$

where $W_R$ and $S_R$ are the delay and power savings yield by the regular scheduling pair $(I, T)$, and $W_L$ and $S_L$ are the delay and power savings yield by the predictive scheduling pair $(I_L, T_L)$. The coefficient $Y$ captures the portion of $X$ that is contributed by $(I, T)$. This coefficient is zero if the probability of having long idle intervals is larger than the allowance $A(X)$. We define $Y$ as

$$Y = \begin{cases} A(X) - (1 - \text{CDH}(L)), & \text{for } A(X) > 1 - \text{CDH}(L) \\ 0, & \text{otherwise.} \end{cases} \tag{4.19}$$

While $W_R$ and $S_R$ are defined in the previous sections, we need to define $W_L$ and $S_L$. From the conditional probability $Prob(LL_l)$, we know that we need to have $Prob(LL_l)$ true positives in prediction of idle intervals longer than $L$ and $1 - Prob(LL_l)$ false positives (i.e., the predicted long idle interval is in fact shorter than $L$). Because this prediction occurs only if a long idle interval is observed, with probability $1 - \text{CDH}(L)$, the $(I_L, T_L)$ scheduling pair causes a power saving mode with probability $(1 - \text{CDH}(L))(1 - \text{CDH}(I_L))$. This means that a delay $P$ is caused with probability $(1 - \text{CDH}(L))(1 - \text{CDH}(I_L))(1 - Prob(LL_l))$ while the savings of $T_L - P$ units of time

occur with probability $(1 - \text{CDH}(L))(1 - \text{CDH}(I_L))Prob(LL_l)$. We have

$$\begin{aligned} Prob(P)_L &= (1 - \text{CDH}(L)) \cdot (1 - \text{CDH}(I_L)) \cdot (1 - Prob(LL_l)), \\ S_L &= (1 - \text{CDH}(L)) \cdot (1 - \text{CDH}(I_L)) \cdot Prob(LL_l)(L - I_L - P), \end{aligned} \tag{4.20}$$

where $Prob(P)_L$ is used as the basis for the recursion to compute $W_L$ as given by Eq. (4.9), Eq. (6.3), and Eq. (4.15).

## 4.3 Experimental Evaluation

In this section, we evaluate PREFiguRE with regard to accuracy, robustness, flexibility, and adaptivity in estimating schedules for power saving modes while meeting system quality targets, including the performance slowdown target $D$, the reliability target $X$, as well as the power savings target $S$. One of the most important aspects of PREFiguRE is making decisions based *only* on metrics that are monitored on real-time and it does not depend on static models or knowledge of the underlying disk drive characteristics. As a result, for the evaluation of PREFiguRE we use trace-driven simulations as long as they allow for the calculation of the PREFiguRE input parameters like the histogram of idle times. Recall that PREFiguRE does not interfere with disk request service or scheduling and as a result we do not need a full-disk simulator. PREFiguRE is computationally lightweight as it only scans the CDH of idle times, which is at most a few thousand entries, at a frequency of every few hours. PREFiguRE computes a nearly optimal (as our experiments show) scheduling pair almost instantaneously. In this section, we show the proximity of the scheduling pair $(I, T)$ given by PREFiguRE to the *optimal* pair that is found by exhaustive search, i.e., by simulating and evaluating *all* possible pairs for scheduling power saving modes. In addition, we show how one could use workload patterns in the time series of idle intervals, to further

improve on power savings without deviating from the preset reliability and performance constraints.

### 4.3.1 Performance of PREFiguRE

Our evaluation is driven by a set of disk-level enterprise traces collected at mid-size enterprise storage systems hosting dedicated server applications such as a development server ("Code") and a file server ("File") [88]. Each trace corresponds to a single drive in a RAID. For an unbiased treatment, we focus on the performance requirement of each disk. We monitor the workload of each disk drive and determine whether to put it to sleep or not. Storage systems that deploy advanced redundancy schemes may schedule a request such that it avoids the disks that are in power saving modes. However, our method is orthogonal to such solutions, because we monitor the disk workload *after* those policies have been applied. In addition, our framework works with a lower priority compared to the upper level policies. Therefore, our framework can be applied at individual storage nodes (e.g., single disk drive) without interfering with upper level power saving policies.

The traces are collected at the disk level and measured using a SCSI or IDE analyzer that intercepts the IO bus electrical signals and stores them. The final traces are produced by decoding the electrical signals. This trace collection method does not require modifying the software stack of the targeted system and does not affect system performance. We stress that our framework only requires knowledge of idleness and is completely independent of the complexity of the arrival and service processes, as well as complex scheduling behavior in the various levels of the storage stack (e.g., the RAID set up). More importantly, they record the arrival and departure time of each disk-level request allowing for exact calculation of the histogram of idle times.

The traces that we use to evaluate PREFiguRE have varying characteristics, see Table 4.3 for an overview. From this table we notice that these traces are characterized by very low utilization, yet their idleness is highly fragmented. Notice the differences in the mean idle intervals and their coefficients of variation (C.V.s). The columns labeled "Time in Power Savings" include the percentage of time relative to the duration of the entire trace that is used for power savings if *all* idle intervals that can be used for Level 3 or Level 4 savings are indeed used, and if *perfect* knowledge of future workload is available. This is of course not practical, but this value represents an absolute upper bound on power savings. The table shows that the eight traces are quite diverse, thus constitute an excellent set to evaluate PREFiguRE's ability to estimate the best scheduling pair $(I, T)$ for any workload. We stress that our traces are measured in enterprise systems with idle intervals that yield power savings only for Level 3 and 4 whose penalty $P$ is up to 1 sec but not Levels 5 and 6 whose penalty $P$ is several seconds. Consequently, we do not show results from Levels 5 and 6 of power saving modes and do not discuss wear out because of spin ups/downs. The reliability aspect of power savings is evaluated in association with load/unload cycles that occur when Level 3 and 4 of power saving modes apply on a disk drive.

We use the first half of each trace as the "training period" during which we construct the CDH of idle times, and determine other monitored metrics. PREFiguRE computes the scheduling pair $(I, T)$ using the metrics collected during the training period using the analytic methodology presented in Section 8.3. The second half of each trace is used as the "testing period" during which we run a simulation that uses the computed $(I, T)$ pair to schedule power saving modes. The testing period validates the accuracy of the PREFiguRE scheduling decision. Specifically in the trace-driven simulation, the power saving modes are activated only after $I$ idle time units elapse. The disk remains in a power saving mode

| Trace: Entries | Util (%) | Idle Length | | Potential Time in Power Savings (%) | |
|---|---|---|---|---|---|
| | | Mean (in *ms.*) | CV | Lev. 3 | Lev. 4 |
| Code 1: 379,490 | 5.6 | 192.6 | 8.4 | 55 | 48 |
| Code 2: 56,631 | 0.5 | 1681.6 | 2.3 | 92 | 87 |
| Code 3: 286,612 | 4.8 | 233.95 | 22.5 | 66 | 55 |
| Code 4: 18,865 | 0.1 | 8293.67 | 7.8 | 97 | 94 |
| File 1: 135,629 | 1.7 | 767.5 | 2.3 | 70 | 53 |
| File 2: 44,607 | 0.7 | 2000.2 | 3.8 | 94 | 90 |
| File 3: 44,607 | 0.1 | 2046.51 | 9.1 | 87 | 79 |
| File 4: 14,160 | 0.1 | 2615.74 | 11.3 | 95 | 92 |

**Table 4.3**: General Trace characteristics. All traces have a duration of 12 hours.

for at most $T$ time units. A new IO arrival *always* preempts a power savings mode and reactivates the disk drive, which takes $P$ units of time.

Table 4.4 gives an overview of the effectiveness of PREFiguRE. All columns labeled "Estim." represent values estimated by PREFiguRE and the ones labeled "Actual" are obtained via trace driven simulation. The "Target $D$" column is the performance target input to PREFiguRE. Performance target $D$ is not violated if columns labeled "Performance Degradation" are less or equal to "Target $D$". Finally, $S_{max}$ corresponds to the optimal value found by exhaustive search of all possible $(I, T)$ pairs to identify the one that offers best savings with performance degradation equal or under the target $D$. The penalty to reactivate the drive is set to $P = 500$ ms (Level 3) [90, 49]. The reliability target $X$ is set to 200 for Level 3 or Level 4 power saving modes per day [55] assuming a lifetime of 4 years.

The main observations from this table are:

- The performance $D$ is *never* violated by the scheduling pair computed by PREFiguRE as validated by multiple simulation experiments.

- PREFiguRE consistently estimates excellent scheduling parameters for maximum

| **"Code 1"** | | | | | | **"Code 3"** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving |
| Target $D$ | Estim. | Actual | Estim. | Actual | $S_{max}$ | Estim. | Actual | Estim. | Actual | $S_{max}$ |
| 1 | 1 | 0.0 | 1.68 | 1.37 | 2.06 | 1 | 0.0 | 12.54 | 10.87 | 17.24 |
| 5 | 3 | 0.0 | 2.22 | 1.94 | 2.06 | 2 | 0.0 | 15.93 | 11.69 | 17.99 |
| 10 | 3 | 0.0 | 2.22 | 1.94 | 2.06 | 2 | 0.0 | 15.93 | 11.69 | 17.24 |
| 20 | 3 | 0.0 | 2.22 | 1.94 | 2.06 | 2 | 0.0 | 15.93 | 11.69 | 17.99 |
| 100 | 3 | 0.0 | 2.22 | 1.94 | 2.06 | 2 | 0.0 | 15.93 | 11.69 | 17.99 |
| **"Code 2"** | | | | | | **"Code 4"** | | | | |
| | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving |
| Target $D$ | Estim. | Actual | Estim. | Actual | $S_{max}$ | Estim. | Actual | Estim. | Actual | $S_{max}$ |
| 1 | 1 | 0.0 | 0.09 | 0.09 | 0.33 | 1.0 | 1.0 | 8.18 | 4.99 | 12.57 |
| 5 | 5 | 0.0 | 0.28 | 0.32 | 0.33 | 4.0 | 1.0 | 13.68 | 8.03 | 13.07 |
| 10 | 10 | 2.0 | 0.29 | 0.33 | 0.33 | 9.0 | 3.0 | 21.47 | 18.89 | 18.89 |
| 20 | 20 | 20.0 | 0.31 | 0.35 | 0.35 | 20.0 | 10.0 | 35.73 | 35.35 | 35.35 |
| 100 | 22 | 21.0 | 0.31 | 0.35 | 0.37 | 31.0 | 25.0 | 37.79 | 37.51 | 37.57 |
| **"File 1"** | | | | | | **"File 3 "** | | | | |
| | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving |
| Target $D$ | Estim. | Actual | Estim. | Actual | $S_{max}$ | Estim. | Actual | Estim. | Actual | $S_{max}$ |
| 1 | 1.00 | 0.00 | 0.50 | 0.39 | 0.39 | 1.00 | 0.00 | 2.69 | 1.77 | 5.76 |
| 5 | 5.00 | 3.00 | 0.73 | 0.69 | 0.70 | 4.00 | 2.00 | 6.32 | 4.42 | 5.76 |
| 10 | 7.00 | 4.00 | 0.75 | 0.71 | 0.71 | 10.00 | 4.00 | 8.47 | 6.98 | 6.98 |
| 20 | 7.00 | 4.00 | 0.73 | 0.71 | 0.71 | 20.00 | 6.00 | 12.02 | 10.79 | 10.80 |
| 100 | 7.00 | 4.00 | 0.73 | 0.71 | 0.71 | 28.00 | 21.00 | 13.45 | 11.17 | 11.17 |
| **"File 2"** | | | | | | **"File 4 "** | | | | |
| | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving | Performance Degradation | | Time in Power Saving | | Max Time in Power Saving |
| Target $D$ | Estim. | Actual | Estim. | Actual | $S_{max}$ | Estim. | Actual | Estim. | Actual | $S_{max}$ |
| 1 | 1.00 | 0.00 | 0.31 | 0.30 | 0.87 | 1.00 | 1.00 | 0.44 | 0.36 | 2.60 |
| 5 | 5.00 | 5.00 | 1.59 | 1.37 | 1.55 | 4.00 | 3.00 | 11.78 | 8.75 | 8.75 |
| 10 | 9.00 | 6.00 | 1.90 | 1.69 | 1.87 | 8.00 | 4.00 | 14.67 | 12.70 | 12.70 |
| 20 | 19.00 | 10.00 | 1.92 | 1.72 | 1.75 | 19.00 | 19.00 | 17.38 | 15.86 | 15.86 |
| 100 | 18.00 | 12.00 | 1.92 | 1.72 | 1.75 | 44.00 | 44.00 | 27.08 | 26.33 | 26.34 |

**Table 4.4**: Power savings and performance degradation estimated using PREFiguRE (columns "Estim.") and simulation (columns "Actual."). Level 3 savings are used. All values are in (%) (for the columns of time, it means % time compared to the entire trace duration).

power saving while limiting the number of load/unloads per day.

- The time in power savings $S_{(I,T)}$ estimated analytically by PREFiguRE is accurate

most of the time, see its proximity to the actual values given by simulation. The errors come from two sources: first, the estimation method relies on past information to predict the future. Consequently, its accuracy depends on the change in workload characteristics used by the framework between future and past. Second, the estimation method is a statistical approach, which relies on the granularity and accuracy of characterization measurements, e.g., finer granularity of CDH of idle periods yields better prediction accuracy than coarse granularity.

- High accuracy of PREFiguRE and its ability to estimate the scheduling outcome in the form of $D_{(I,T)}$ and $S_{(I,T)}$ is critical because it suggests that PREFiguRE can be used to drive analysis in the system.

- Monitoring of metrics in the short past ("training period" of several hours) yields good and robust predictions for the near future ("testing period" of several hours).

- For $D > 5\%$, the accuracy of estimations is consistently high. For $D = 1\%$, the accuracy reduces, as it becomes difficult for PREFiguRE to capture the very small variations in performance. Recall that estimation of delays is the most critical aspect of the framework and its accuracy depends on the CDH bin granularity. As a result, discrepancies become noticeable for very small performance targets such as $D = 1\%$.

A phenomenon worth discussing is that PREFiguRE estimates for various target $D$'s are the same for "Code 1" and "Code 3". This happens because PREFiguRE calculates the same $(I, T)$ pair for $D \geq 5\%$. The CDHs of "Code 1" and "Code 3" reveal that these two workloads have many small idle intervals but only a few long ones. Indeed, 95% of "Code 1" idle intervals are smaller than the Level 3 penalty (500 ms), thus they are excluded from PREFiguRE as a scheduling choice. As a consequence, a large idle waiting time $I$ is used, to prevent small idle intervals from being used for power savings. Therefore, $W$ in Eq. (6.3) is small and results in the same $D$ that is always less than the target $D$s we set

in Table 4.4. This results in selecting the same $(I, T)$ pairs for $D \geq 5\%$.

Overall, the table shows that PREFiguRE is robust across all workloads and ranges of performance targets, with excellent accuracy for both power and average delay estimation, without compromising on the reliability constraint $X$. This makes the case that PREFiguRE can be also used very effectively in analysis to select among power saving options as shown in the following section.

### 4.3.2 "What-if" Analysis

In system design and on-line resource management it is critical to be able to know the outcome of features and enable them *only* if beneficial. Specifically, because power savings in disk drives impact both performance and reliability then the disk should be put into power saving modes only if the savings are significant for the system. Because of its analytic core, PREFiguRE has the ability to compute schedules and estimate their outcome. As such it facilitates the automation of on-line decisions on disk power savings by giving answers to a wide range of "What-if" questions.

Table 4.5 lists a set of "what-if" questions that could be answered using the PREFiguRE framework. Table 4.5 shows how PREFiguRE predicts for a given workload if a specific power saving target can be met. For example, in a cluster with the four disks (and workloads) a target of 10% time in power saving can be achieved by Code 1 with a performance degradation of 33.0%.

Similarly, the system can also estimate beforehand if it is worth increasing the performance target $D$ for higher power savings. In this table we can clearly see that it is not beneficial to increase the performance degradation to 20% since it does not offer additional savings for any of the workloads in the four disks in the storage cluster. It is obvious in this table that for most workloads when the penalty due to power savings is

| What-If Question | "Code 1" | "Code 2" | "File 1" | "File 2" |
|---|---|---|---|---|
| How much should I slow down the user traffic to get power savings of 10% | 33.0% | 59.0% | 195.0% | 27.0% |
| How much should I slow down the user traffic to get power savings of 20% | 61.0% | 104.0% | 458.0% | 140.0% |
| How much power savings do I get if I slow the user traffic with 10% | 1.94% | 0.33% | 0.71% | 1.69% |
| How much power savings do I get if I slow the user traffic with 20% | 1.94% | 0.35% | 0.71% | 1.72% |
| Which power saving level should I use Level 3 or Level 4 if I slow the user traffic with 10% | Level 3 | Level 3 | Level 4 | Level 3 |
| Which power saving level should I use Level 3 or Level 4 to get power savings 10% | Level 3 | Level 3 | Level 3 | Level 3 |
| If I relax the X condition for the next 12 hours and slow the user traffic with 10%, how much additional savings will I get and by how much is X violated. | 6.59% (50) | 3.36% (19) | 0.73% (23) | 8.15% (285) |

**Table 4.5**: Various what-if scenarios that can be answered using the estimation engine in PREFiguRE to assist with making power saving decisions in a storage system.

low, i.e., Level 3, the power savings are better. Finally we can estimate beforehand whether it is worth relaxing the reliability condition to achieve better power savings. The last what-if scenario presented in this table illustrates the power savings when we relax the reliability target $X$. Given that $X$ captures the wear-out effect that power savings have on disk drives over their lifetime, $X$ can be set higher at times and lower at other times. Specifically for "Code 2" the savings are considerable and the compromise in reliability is small compared to the original reliability constraint. The system may decide to relax $X$ for that disk for a while and account for it at a later time when the workload has changed and savings are limited.

### 4.3.3 PREFiguRE's Adaptivity and Estimation Capabilities

PREFiguRE is a framework that monitors current workload and updates its scheduling decisions, i.e., the $(I, T)$ pair accordingly. So far in this section, the learning (or training) has occurred for six hours and the computed $(I, T)$ pair is used for the following six hours.

However, there are various ways to learn the workload characteristics (i.e., the histogram of idle times) and update the corresponding scheduling parameters. Here we evaluate the robustness of PREFiguRE against the length of the learning window and the granularity of updates in the computed $(I, T)$ pair.

We experiment with two additional learning window sizes (i.e., three and five hours), and scheduling parameters updates every half an hour or only at the end of a learning window. Specifically, we evaluate the following variations in learning a CDH:

- *Learning1* - learning windows are non-overlapping and $(I, T)$ is computed only at the end of a learning window,

- *Learning2* - CDH of idle times is accumulated from the beginning and $(I, T)$ is computed every half an hour,

- *Learning3* - learning window slides with half an hour granularity and $(I, T)$ is computed every half an hour.

- *Baseline* - similar to Learning1 above, but the CDH is built with the knowledge of idleness in the current learning period, not the previous one. It is included only for comparative purposes as a best case.

We present the results from our trace-driven simulation in Figure 4.4, where the left column plots the performance degradation in the system validating the accuracy of the framework with regard to performance slowdown target of $D\%$. The right column of plots in Figure 4.4 captures the power savings resulting from the scheduling framework.

It is clear that different learning methods and granularity achieve different accuracy. We observe that it is important to learn over longer rather than shorter periods of time (compare first row of results corresponding to five hour learning with the second row of results corresponding to three hour learning in Figure 4.4). Another important observation is that updating the $(I, T)$ pair every half an hour yields better robustness to the Learning

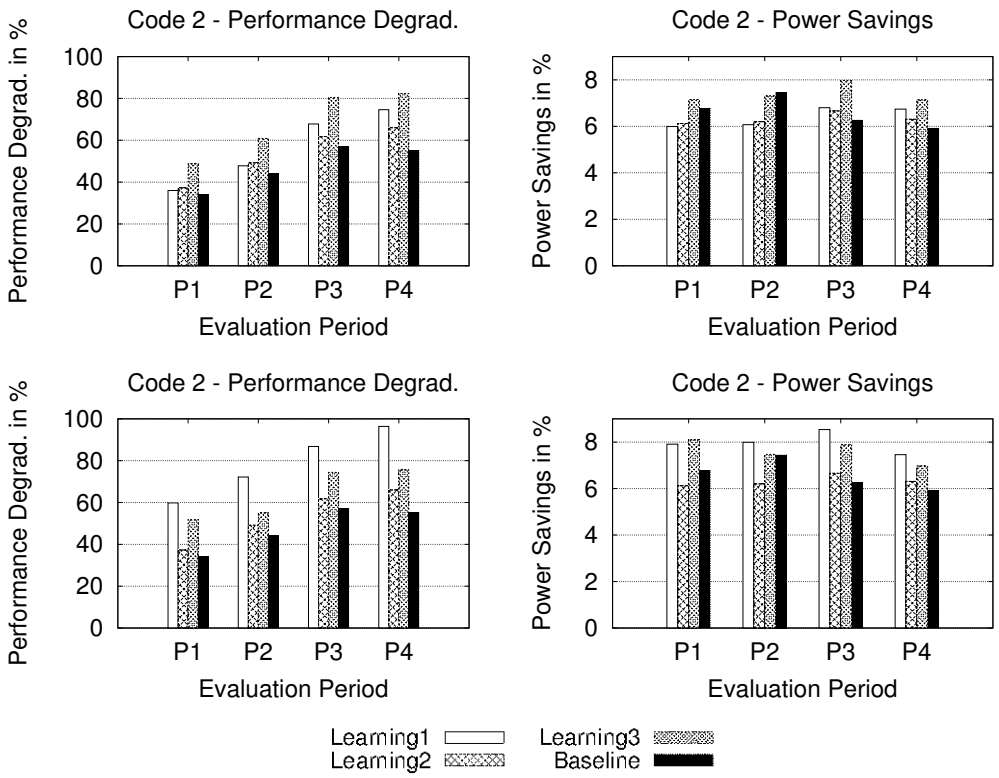



**Figure 4.4**: Performance degradation and time in power savings over time for Code 2, three different learning methods and 2 different lengths of learning (the first row of plots corresponds to 5 hours of learning and second row to 3 hours of learning). The performance degradation target is 50%. P1 is evaluation period starts at the 4th hour, P2 starts at the 5th hour, P3 starts at the 6th hour and P4 starts at the 7th hour. For fair comparison, the evaluation lasts for 5 hours in each evaluation period for both learning length cases.

Window Size changing than updating it less frequently because it can reduce the impact caused by the changing workload. Recall that the computation complexity of computing the $(I, T)$ pair is minimal and a frequency of every half an hour that we suggest here is expected to have an equally minimal impact on the overall system performance.

### 4.3.4 Comparison with Common Practice Methods

The efficiency of PREFiguRE is shown by comparing its performance to common practices used for power savings in storage systems. The most widespread approach is to idle wait for a fixed amount of time before putting a disk into a power saving mode. Usually the fixed amount of time is set to be a multiple of the penalty $P$ to bring back the disk into operational state. Here we show results obtained when the idle wait $I$ is set to $2P$ [30]. A second approach is to guide power savings by the current utilization levels in the storage node (i.e., disk drive). Here, we apply the first approach of fixed idle wait only if the utilization in the last 10 min is below a pre-defined threshold (set to the average utilization in the trace).

In Figure 4.5, we plot the performance degradation and power saving results of PREFiguRE and the above two common practice methods. For PREFiguRE, three performance targets, i.e., 10%, 50% and 100%, are evaluated. For the two common practice methods, the performance target cannot be set beforehand and the slowdown may be unbounded. Often in practice in order to limit the performance slowdown, the fixed idle wait and/or the utilization threshold are set such that the system goes into power savings only occasionally.

In Figure 4.5, the y-axis is in log-scale and the absolute values are shown above each bar. The fixed idle wait method for $I = 2P$ results in a slowdown of 5662%, i.e., several orders of magnitude more than PREFiguRE for less than 10 times the power savings. The utilization-guided method reduces performance degradation of the fixed idle wait method, but its power savings are 10 times lower than PREFiguRE for similar performance slowdowns.

The results in Figure 4.5 clearly illustrate that PREFiguRE outperforms common practice methods. By taking into consideration the idleness, which in a way confines in a

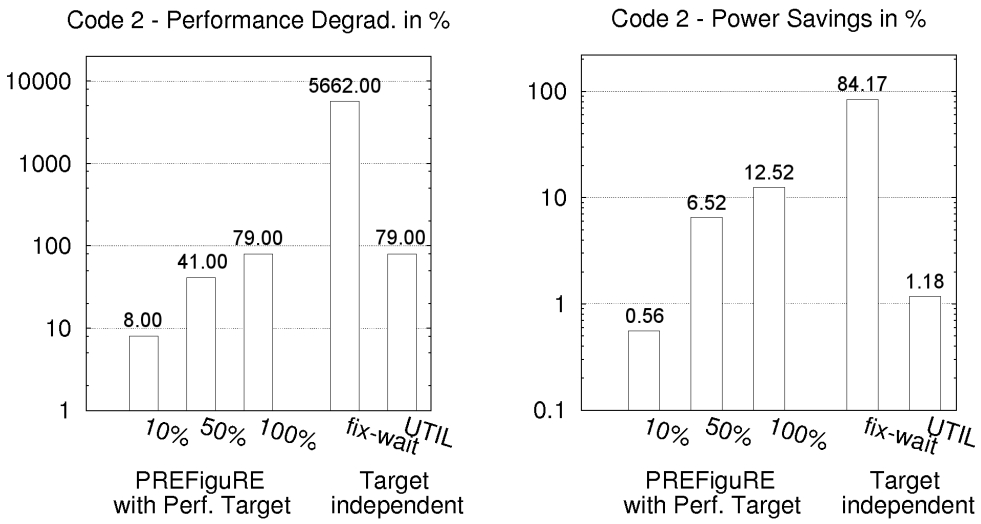



**Figure 4.5**: Performance degradation and time in power savings for Code 2 under PREFiguRE and other common practices, i.e., fixed idle wait and utilization-guided. Because y-axis is in log scale, the y-axis values are shown for each bar.

compact measure the complex interaction of the arrival and service processes, PREFiguRE meets performance targets while achieving high power savings.

### 4.3.5 Correlation-based Enhancement: PREFiguRE-LL

To further extend power savings without violating the performance degradation target, we enhance PREFiguRE with the predictive capabilities of the conditional probabilities of successive idle intervals, see Figure 4.6. We construct conditional probabilities of two idle intervals up to $G = 10$ lags apart being at least of length $L$, where $L$ represents long idle intervals observed in the system such that the number of such intervals is close to $X$, i.e., the reliability target. The length $L$ of long idle intervals depends on the workload characteristics, i.e., the average, maximum, and variability in the distribution of idle intervals as captured by the CDH, which means that for more idle workloads this value is higher than for the busier ones.

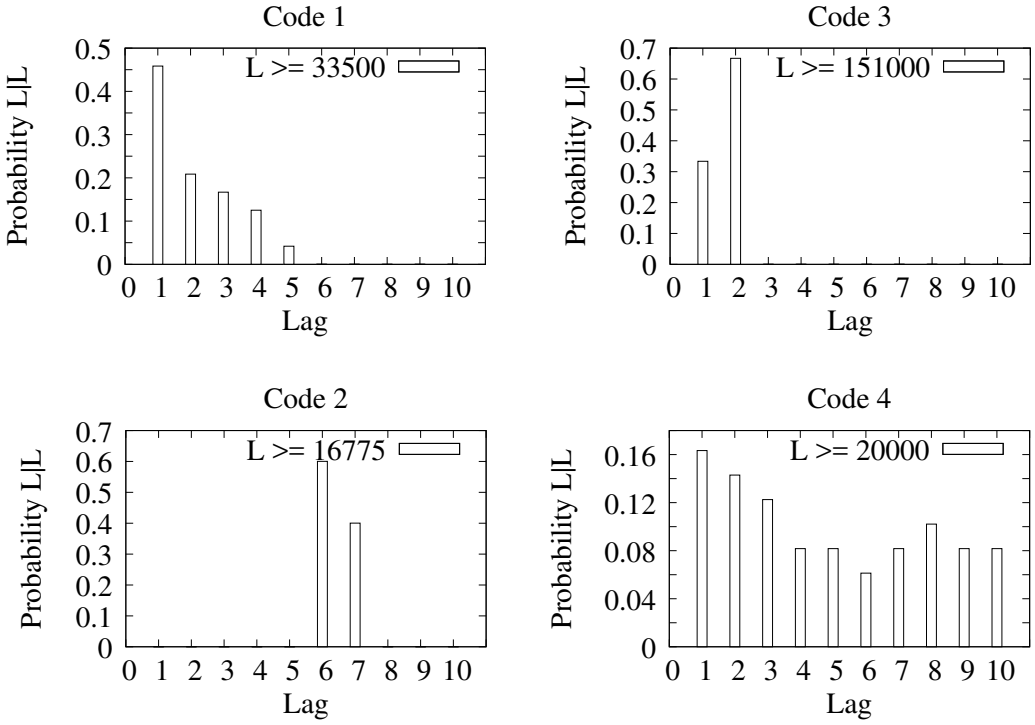



**Figure 4.6**: Conditional probability values of a long interval being followed by another long that are $k$ lags apart. The long interval length $L$ is defined in the legend.

In our evaluation of this enhancement, which we call PREFiguRE-LL, we focus on workloads "Code 1", "Code 2", "Code 3", and "Code 4". Figure 4.6 shows the probability that successive idle intervals of at most 10 lags apart are at least of length $L$. We observe that for the "Code 2" and "Code 3" workloads, these conditional probabilities are higher than 0.5 for at least one lag. This suggests that the enhanced PREFiguRE-LL could benefit from the prediction capabilities embedded in these probabilities and harvest these long idle intervals to extend power savings according to the discussion in Section 4.2.5. For "Code 1" and "Code 4" the conditional probabilities have small values, therefore PREFiguRE-LL is expected to not increase power savings. We stress that in these cases PREFiguRE-LL reduces seamlessly to PREFiguRE and still meets both reliability and performance

targets. For PREFiguRE-LL, we pick among all 10 evaluated lags the one with the highest conditional probability to predict the future long idle interval and define the scheduling parameters as explained in Section 4.2.5.

Power savings with PREFiguRE and PREFiguRE-LL are shown in Figure 4.7, while the corresponding performance degradations are given in Table 4.6. Consistently with the expectations set from the probability values in Figure 4.6, we observe that PREFiguRE-LL extends power savings for "Code 2" and "Code 3" workloads. The high correlation between successive long idle intervals enables PREFiguRE-LL to start early and stay longer in a power saving mode and almost double the overall power savings for several of the performance targets $D$. For "Code 1" and "Code 4", however, such information does not exist and, as expected, PREFiguRE-LL performs the same as PREFiguRE. As supported by the results in Figure 4.7, the gains of as much as double in power savings come at *no* cost in performance degradation. PREFiguRE-LL does not violate the performance target $D$ for the entire spectrum of evaluated slowdowns. Note there are cases when higher performance degradation targets are set, but the actual performance degradation and power savings stays the same. This is because of the reliability targets in the framework. In addition, the policy remains robust as stochastic information on the sequence of idle times is incorporated in the framework.

### 4.3.6 Caveats and Limitations

The interplay between device driver decisions and upper level policies is an important factor to consider when implementing PREFiguRE. When the framework is implemented at lower levels, e.g., at the device driver level, the interplay between device driver decisions and higher level scheduling is less likely to happen as the lower levels are transparent to upper levels. For example, during the periods that the disk is in power saving mode,

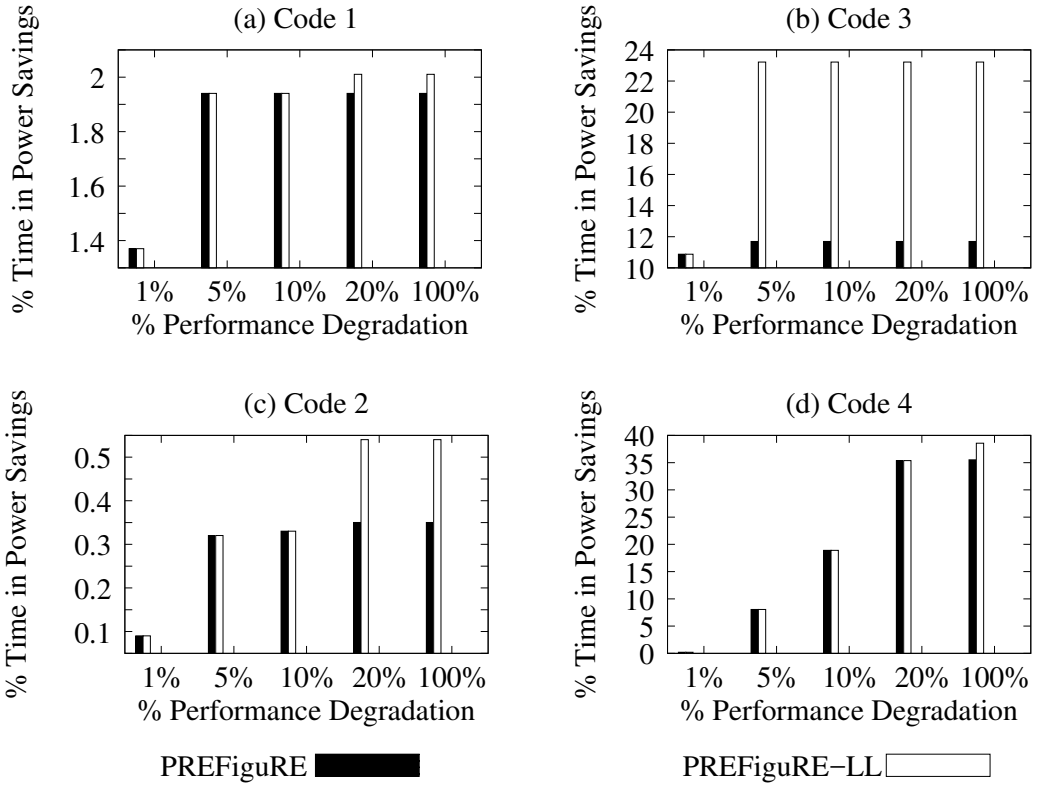



**Figure 4.7**: Power savings (Level 3) for performance degradation target 1%, 5%, 10%, 20% and 100% for "Code 1" and "Code 2", "Code 3" and "Code 4".

upper level policies see that the disk is available and idle. We propose PREFiguRE to be implemented at the lower levels, i.e., at the storage controller or HDDs rather than other levels of the IO hierarchy to avoid the potential interference with upper-level non-FCFS schedulers. If PREFiguRE needs to be deployed in the same level as other non-FCFS schedulers, interference is likely to happen, but such interference is usually harmless for performance. This is because for a non-FCFS disk scheduler, the more the requests to choose from, the better the performance. Extra delays caused by the waking-up process become even smaller than the values we estimate as we consider the propagated delay affected up to k consecutive busy periods in our estimation (see Eq. 4.8). In addition, there

| | "Code 1" | | "Code 2" | |
|---|---|---|---|---|
| | Performance Degradation | | Performance Degradation | |
| Target $D$ | PREFiguRE | PREFiguRE-LL | PREFiguRE | PREFiguRE-LL |
| 1 | 0.0 | 0.0 | 0.0 | 0.0 |
| 5 | 0.0 | 3.0 | 0.0 | 0.0 |
| 10 | 0.0 | 3.0 | 2.0 | 2.0 |
| 20 | 0.0 | 3.0 | 20.0 | 17.0 |
| 100 | 0.0 | 3.0 | 21.0 | 17.0 |
| | "Code 3" | | "Code 4" | |
| | Performance Degradation | | Performance Degradation | |
| Target $D$ | PREFiguRE | PREFiguRE-LL | PREFiguRE | PREFiguRE-LL |
| 1 | 0.0 | 0.0 | 1.0 | 0.0 |
| 5 | 0.0 | 2.0 | 1.0 | 1.0 |
| 10 | 0.0 | 2.0 | 3.0 | 3.0 |
| 20 | 0.0 | 2.0 | 10.0 | 12.0 |
| 100 | 0.0 | 2.0 | 25.0 | 25.0 |

**Table 4.6**: Actual performance degradation under PREFiguRE and PREFiguRE-LL for Level 3 savings. Values are in (%).

are many activities that occur in the path that add variability on measurements more than what we are adding by controlling the sleep times. Sources of variability on higher-level scheduling include multiple interrupts of the communication protocols to communicate with each other: application-OS-client-driver-RAID controller-SAS/PCIcable-HDD. Sources of the HDD variability could be missed rotation, failure in seeks, failure in rotation. All these happen regularly and only increase latency at the HDD level. We leave the rigorous estimations (via measurement data and/or simulation) of the exact indirect affect as our future work.

## 4.4 Summary

In this chapter, we presented a compact analytic model and its integration into an algorithmic framework that provides the following: given performance and reliability

targets, it provides answers to the following difficult questions: "when" and for "how long" idle periods in disk drives can be utilized for putting the system in a specific power saving mode such that performance/power/reliability targets are met. A detailed analytic model is also developed that determines quite precisely the respective amount of power savings that is possible to save. The effectiveness of the proposed heuristics of PREFiguRE are demonstrated using a set of traces from enterprise storage systems. The main findings of this chapter are also reported in [128].

# Chapter 5

# Toward Automating Work Consolidation

The volume of digitally stored data has grown rapidly and continues to grow with a tremendous pace with the expectation of reaching a total of 35 zetabytes by 2020 [52]. Data centers host most of these data and efficient operation requires meeting power consumption, performance, data reliability, availability, and integrity targets that may change during the lifetime of the system. As workload demands and system targets vary, there is a need to make data server resources available on demand in a robust and transparent way. For example, if the aim is to consume as little power as possible while maintaining specific service level objectives, when work intensity decreases it is natural to aim at consolidating work into a smaller set of available resources in order to allow the rest to be taken off-line such that power consumption is reduced [139, 46].

Storage is one of the main components of a data center and it consumes about 20 to 30 percent of the total power, making work consolidation in storage systems important and relevant. Since data is not all accessed simultaneously, it is common to have an underutilized or even idle storage system [40, 88, 30], making the storage component a

promising one for power savings. To this end, there have been efforts in developing techniques that exploit idleness in a storage system and its devices by taking off-line portions of it without impacting data availability or violating performance targets [89, 19, 114, 44]. Even more, there have been efforts that aim at *increasing* idleness in selective storage devices or systems by redirecting portions of the workload [73, 74] or the entire working set from a set of storage devices to another [109] in order to open more opportunities for power savings.

Consolidation of a storage workload to a limited set of devices is undoubtedly beneficial for power consumption but may come at a dear cost: performance of the storage system may suffer. Striking a balance between consistently meeting service level objectives and power savings is difficult given that future workload demands are seldom known a priori. Judicious selection of *which resources* to consolidate and on *which nodes* to initiate power savings is not an easy task. Using simplistic measures such as average node utilization to guide consolidation can result in lamentable system operation, as we show later in this chapter. The question now becomes, where to shift data, how much data to shift, and from which senders to which receivers.

In this chapter, we present a solution to the above problem by using a quantitative framework that estimates (i.e., predicts) the performance of consolidated storage workloads in the available nodes in the cluster. The predicted consolidated node performance is paired with projected power savings at each node [89] to enable a cluster-wide identification of the nodes that if put off-line can bring the highest power savings, while performance of the storage devices that serve their redirected workload does not degrade beyond a pre-defined response time target.

By pairing the framework that predicts performance of a storage node serving consolidated workload with the framework that estimates power saving capabilities in a

storage node, we enable the storage system to determine whether consolidation can be effective. The framework is based on data monitoring at the device level that is routinely done on storage systems and its estimations are lightweight, i.e., they are based on simple histogram scanning that is used as input to simple equations. A critical component is the use of a look-up table that couples arrival intensities and service demands on each node with average expected service time. This look-up table is continuously changing as workload evolves, by continuously "learning" how performance depends on the changing workload. Finally, the framework can be deployed on every node, making this approach highly distributed, thus highly scalable.

The rest of this chapter is organized as follows. In Section 5.1 we give an overview of the framework that estimates power saving capabilities in a storage device such as a disk drive. In Section 5.2 we develop the methodology that learns the dependencies between the arrival, service, and response times in a storage device. Section 5.3 presents an extensive set of trace-driven experiments that demonstrates the robustness of the framework. We summarize in Section 5.4.

## 5.1 Background

In this section, we give an overview of a methodology that demonstrates how a storage device such as the disk drive can identify its own capabilities for power savings [89] as well as how this methodology is used to define how to increase power saving capabilities via workload shaping [73], i.e., redirecting part of workload (or the entire workload) to other nodes in the cluster.

The above methodologies take into consideration the fact that power saving modes in disk drives impact performance. Performance degrades because if a request comes and finds the disk in a power saving mode, it needs to wait for the disk to come back up, which

may take up to 20 seconds. This waiting time is several orders of magnitude higher than the average service time in a disk drive. As a result, for high-end systems with stringent performance targets, if scheduled naively, power saving modes may cause a significant drop in performance.

### 5.1.1 Estimating Power Savings Capabilities in Disk Drives

In [89], an analytic framework is proposed to estimate the capabilities for power savings in a disk drive. The framework uses the histogram of idle times to capture the important stochastic characteristics of idleness in the device. The histogram is used to probabilistically assess the occurrence of long idle intervals that can be used to schedule power saving modes in the drive, i.e., slow down the disk or shut it off completely.

Deciding power saving capabilities based on the histogram of idle times for the current workload makes this framework versatile, because it eliminates the need to monitor and make decisions based on device utilization or request arrivals, whose relations to power savings are more complex to capture. Furthermore, the framework incorporates the relation between workload characteristics (such as sequentiality or randomness) and the impact of the periods of power saving on performance via a single *penalty* parameter. Such abstraction of the workload parameters results in an analytic framework that can estimate the power saving capabilities (in portion of time in power savings) for a range of power saving modes and penalties on system performance.

The framework treats power savings as a low priority work that takes place during idle intervals. To this end, it takes as input the histogram of idle times, the penalty during power savings, the response time without savings and an acceptable performance degradation. The output of the framework is a "schedule" that defines when and for how long to put the disk in power savings. The framework calculations are based on

scans of the histogram of idle times, which may have several hundred entries only. As a result, scanning is nearly instantaneous making the framework lightweight and compact. An important feature in the framework is that it allows the storage node itself to sort out several power saving options (resulting from combinations of power saving modes and degradation on performance) internally and present to the cluster management module the most effective one. To summarize, the evaluation is lightweight and with minimal overhead which is mostly related to monitoring rather than computation.

### 5.1.2 Workload Shaping for Power Savings

A way to enhance power savings in a single storage device (or a set of devices) is to increase the length of idle intervals. This can be done by identifying portions of the workload that can be redirected somewhere else in the system. In this chapter, we refer to this activity as workload shaping [73, 74, 109, 138]. Workload shaping may require to copy some of the data from the storage device that are to be placed in the power saving mode at the *new* destination node [73, 109]. However, the main feature in all these techniques is that the storage devices are consolidated and the ones that remain active take over the load of the storage devices that are to be placed in a low power mode (or even off-line).

These methods define clearly what part of the workload to redirect out of a storage device. [74] proposes to redirect the entire WRITE traffic arriving at the storage device. [109] proposes to redirect the entire active working set. [73] proposes to remove the most frequent busy periods (statistically or according to a probabilistic weighting scheme).

We stress that if the system monitors the idle periods and the busy periods in a system, then it is possible to estimate the power saving capabilities of these workload shaping techniques via the framework in [89]. This requires that the histogram of idle times is updated to reflect the changes in the workload, and this can be done by monitoring the

requests within busy periods [73]. For example, for the workload shaping in [74], the histogram of idle times can be updated to reflect the removal of WRITEs. In general, the capabilities of these workload shaping techniques to improve power savings can be quantified [89] and for a given workload, the storage device can determine which workload shaping technique to use and how much load it can off-load to other nodes.

Once the workload to be redirected is removed, the next question is to determine the receiver of this extra load. Consolidation of the workload over a smaller set of storage devices is effective only if the cluster continues to perform without violating performance targets in the system. This chapter focuses on this problem and in the following sections we present a method that predicts performance, measured via the response times, in a storage device that serves consolidated work.

## 5.2 Performance Degradation Estimation on a Storage Device due to Consolidation

Here, we present a quantitative framework that enables a storage system to intelligently identify which storage devices should be put into power saving modes and which storage devices should serve the consolidated workload such that the performance target is not violated while power savings are the highest possible. In this framework, we assume the performance target is the response time which, on the average, should not be higher than a predefined limit.

Our goal is to construct a lightweight and accurate framework that predicts the response time in a potential *receiver* storage device knowing

- its workload in terms of average arrival rate, average service rate, and average response time;

- the portion of the load in the *sender* device that may potentially go to the *receiver* device and its characteristics such as average arrival rate and average service rate.

We expect average workload statistics such as arrival, service, and response time to be available in the logs that monitor the storage system operation. The workload shaping input is expected to be available from monitoring that is used to facilitate the various workload shaping techniques such as those introduced in Section 5.1.

For the framework to reach a consolidation decision, the following two steps are required. First, estimate the arrival and service rate of the consolidated workload at the receiver storage device. Second, predict the response time of the consolidated workload at the receiver end.

### 5.2.1 Estimating arrival and service rates of the consolidated workloads

In order to be able to predict the arrival and service rates at the consolidated storage node, we assume prior knowledge of the arrival rate $\lambda_s$ and the service rate $\mu_s$ of the portion of the workload from the sender node that will be consolidated at the receiver storage node. It is also assumed that the arrival rate $\lambda_r$ and the service rate $\mu_r$ at the receiver storage node are known. The characteristics of the merged workload, i.e., its arrival rate and its service rate, if a certain pair of nodes is selected for consolidation, is

$$\lambda_c = \lambda_s + \lambda_r \tag{5.1}$$

and

$$\mu_c = \frac{\lambda_r}{\lambda_s + \lambda_r}\mu_r + \frac{\lambda_s}{\lambda_s + \lambda_r}(\rho \cdot \mu_s) \tag{5.2}$$

where $\lambda_c$ and $\mu_c$ are the arrival and service rate after consolidation, while $\rho$ is a correction factor that accounts for the change in the service rate from the sender storage node to the receiver storage node. The parameter $\rho$ is meant to capture *only* the effect of the differences on disk positioning time on service time.

Estimating average arrival rate at the receiver storage node, is less complex than estimation of the service rate, given that the load is known on both sender and receiver nodes. The average service rate is more complex, particularly since in storage devices the service process depends highly on workload characteristics (such as randomness and the mix of READs and WRITEs) and the physical characteristics of the device (e.g., rotation speed for a disk drive). The parameter $\rho$ is used to capture the effect of the physical characteristics of the device. As we introduce a new set of data into the active data set on a disk drive, it is possible to break any existing characteristics in the workload such as sequentiality. This can impact the expected service rate. However, if the sender load is expected to be much smaller than the current load at the receiver, then the workload characteristics of the receiver (before consolidation) should continue to dominate the workload. In addition, workload characteristics of the receiver storage node may be preserved if the placement of the new data is done such that the locality of the receiver's working set (if it exists) is preserved. This is possible especially since in many cases the amount of data to be moved may be small, perhaps only a few GBytes in size. Placement choices and how they can change the disk layout are outside the scope of this thesis.

### 5.2.2 Predicting response time of the consolidated workload

In storage devices, it is not straightforward to determine the response time for a workload, given its arrival and service rates, because of all the idiosyncrasies that determine the device service rate, such as workload access patterns and disk characteristics, as well as the

complex features of the arrival process, such as variability and burstiness. Our goal is to have an approximate, yet accurate prediction of the response time in the receiver storage device under the consolidated workload with parameters defined in Subsection 5.2.1. For this, we propose to predict the response time, by "learning" the response time patterns over time for the storage nodes in the cluster. The goal is for individual storage devices to monitor and record the observed response times for pairs of arrival and service rates.

We propose to construct on-line (i.e., as the system operates) *look-up tables* that contain tuples of observed average arrival rate, average service time, and average response time over periods of time. The averaging can be at different granularities from 15 minutes intervals to a few hours. As the system operates, the goal is to update the look-up table at each storage node by adding new tuples that have not been there before, but also by avoiding repetition, such that the size of the table remains relative small. We show in the evaluation section that a look-up table of several hundred entries provides good prediction accuracy and is small enough to facilitate fast searching through it. Constructing the look-up table is part of the "workload learning" process in our framework.

The expectation is that the workload that the receiver storage node sees as a result of consolidation has been, at some point in time, already observed and recorded and can be used in the future for approximate, yet fairly accurate, predictions. As a result, as the estimation of the consolidated arrival rate and service time is done using Eqs. (5.1) and (5.2), the look-up table at the receiver is searched and the closest tuple that matches *both* arrival and service rates is selected as the tuple whose response time value is chosen as the approximate prediction of the performance at the receiver storage node.

An exact match of the estimated arrival and service pair with the tuples available in the look-up table is not expected. However, while constructing the look-up table, the density of the tuples can be controlled. In the look-up tables that we have constructed, we have

followed the rules that the differences in the arrival and service rates of the neighboring tuples should be at 10%. While this condition can easily be satisfied for the common cases and low to medium arrival and service rates, the rare events of high arrival rates or very slow service rates may be more difficult to obtain. For such cases, there is a possibility to run off-line benchmarks to populate the look-up tables with rare events. Here, we focus on building the look-up tables on-line. How to populate the tables off-line is not considered in this thesis.

The prediction of the response time is only an approximation but as we show in the evaluation section, it serves as an excellent way to quantify performance in a consolidated cluster and select the right pair or pairs of nodes that should consolidate their workload for an overall reduction in the number of active storage devices. We stress again that the average arrival and service rate here is mainly served as the "index" to find the response times measured in the real system environment in the look-up table. The performance effects of workload characteristics such as sequentiality and burstiness are already captured in the look-up tables.

One source of error in our predictions laid out in Sections 5.2.1 and 5.2.2, is associated with the fact that we expect the arrival and service rates observed for the past several hours to hold in the next several hours. While this is expected to be the case in clusters where changes in the workload happen gradually, there may be cases when the recent past is very different from the close future. The accuracy of the framework presented here naturally suffers in cases of abrupt and unexpected temporal workload changes. There are methods that can complement our workload prediction framework to account for abrupt changes. A feedback-loop monitoring could be used to check at small time intervals (every several minutes) if the observed average arrival rate and service rate are close (up to a threshold) to the observations of previous monitoring period. If there is violation, then

the estimations should be recalculated around the new observations. The threshold should be large enough to avoid unnecessary oscillations. Another approach is to detect at coarse granularity (i.e., several hours) any obvious regular workload changes, such as the ones that may be associated with daily and weekly business cycles. If such cycles are learned in advance, then they can be predicted and the corresponding actions taken to ensure that the decisions are made on accurate current workload characteristics. There are various aspects to be evaluated and analyzed before such methods can be incorporated into our prediction framework. They are not discussed in this thesis. Here we work under the assumption that the short-term past predicts well the short-term future (i.e., the error associated with the differences in the workload between the past and the future is acceptable).

## 5.3 Experimental Evaluation

In this section, we evaluate the effectiveness of the consolidation framework having as target to maximize power saving capabilities while controlling performance degradation after the consolidation of resources. We first describe the traces that we use to assess the effectiveness of the consolidation framework. The accuracy of our predictions and estimations are validated by trace-driven simulation. Then, we provide detailed workload characterization that supports the assumptions and decisions used in the development of the estimation and prediction components of the framework. Finally, we use the prediction framework to decide which storage nodes to consolidate. We focus here on a small cluster of four nodes, to facilitate a clear presentation since the results we present are for each node in the cluster.

| Trace | Util (%) | Mean Arrival Rate | Mean Service Rate | C.V. Arrival Rate | C.V. Service Rate | R/W Ratio |
|---|---|---|---|---|---|---|
| CODE1 | 5.6 | 0.0089 | 0.1596 | 1.56 | 0.22 | 5.48 |
| CODE2 | 0.7 | 0.0013 | 0.1859 | 1.34 | 0.06 | 1.39 |
| FILE1 | 1.7 | 0.0033 | 0.1938 | 1.07 | 0.27 | 8.28 |
| FILE2 | 0.7 | 0.0011 | 0.1596 | 2.93 | 0.20 | 3.63 |

**Table 5.1**: General Trace characteristics.

### 5.3.1 Traces and Simulation environment

The validation of the proposed consolidation framework is done via trace-driven analysis and simulation. We use a set of enterprise traces measured at the disk level from an application development server ("CODE") and a file server ("FILE") [88]. In each of the measured storage systems, there are tens of storage devices (disks) organized in several RAID groups. The traces do not provide information on the RAID groups. However we have observed that disks of small sets have identical workloads, which allows us to infer with high confidence that those disks belong on the same RAID groups. Although we could have used representative disks of each RAID group to simplify presentation, we have selected traces that correspond to *only* 4 representative RAID groups, two from each storage subsystem. The total duration of each trace is twelve hours. Each trace record consists of: the arrival time, the departure time, the type of the request (i.e., read or write), the request length in bytes, and the location on the disk. This information allows to calculate exactly a rich set of metrics that we can use in the evaluation process of our framework. In Table 5.1, we show a subset of these metrics of interest.

The data in Table 5.1 show that the disks are clearly underutilized, implying that here are opportunities to temporary consolidate data and obtain power savings. Similarly, the low arrival rates and relatively much higher service rates indicate that temporary consolidation of work in a few disks only may be effective without taking a toll on system

performance.

The consolidation framework is based on the assumption that short-term past predicts well the short-term future. In the twelve hour traces, we use the first 6 hours ("short-term" past) to collect statistics with regard to arrival, service, and response time, as well as idleness, and apply the learning on the second 6 hours ("short-term" future) of the trace.

In Figure 5.1, we plot for each of the four traces, the arrival rate and the service rate for 5 minute intervals as a function of time. In each of the plots we separate with a vertical dashed line the learning period (the first 6 hours) from the testing period (the next 6 hours). The main observation is that both arrival and service rates are fairly stationary, as shown by both the average and the coefficient of variation of the measured metrics.

Figure 5.1 further confirms that the service rate exhibits almost no change throughout the duration of the traces. This observation supports the argument that the workload characteristics for each of the traces remain fairly stable, resulting in an almost deterministic service process.

The arrival rate in each of the traces is not as deterministic as the service rate (i.e., its coefficient of variation is between 1 and 3) but the average, that is used in our predictive framework, changes only slightly. The highest change we notice is for the FILE2 trace, for which, as we show later, also the prediction errors are higher.

Another interesting observation from Figure 5.1 is that often higher arrival rates (more work) correspond to higher service rate (i.e., faster service). This is attributed to the specifics and optimizations of disk scheduling that always aims at minimizing seeks in drives. For our specific target of consolidated workloads, only slight increases in the load at some receiver node may actually result in lower service times because of optimization of the disk service process. Consequently, response times are expected to suffer minimally from the additional load.

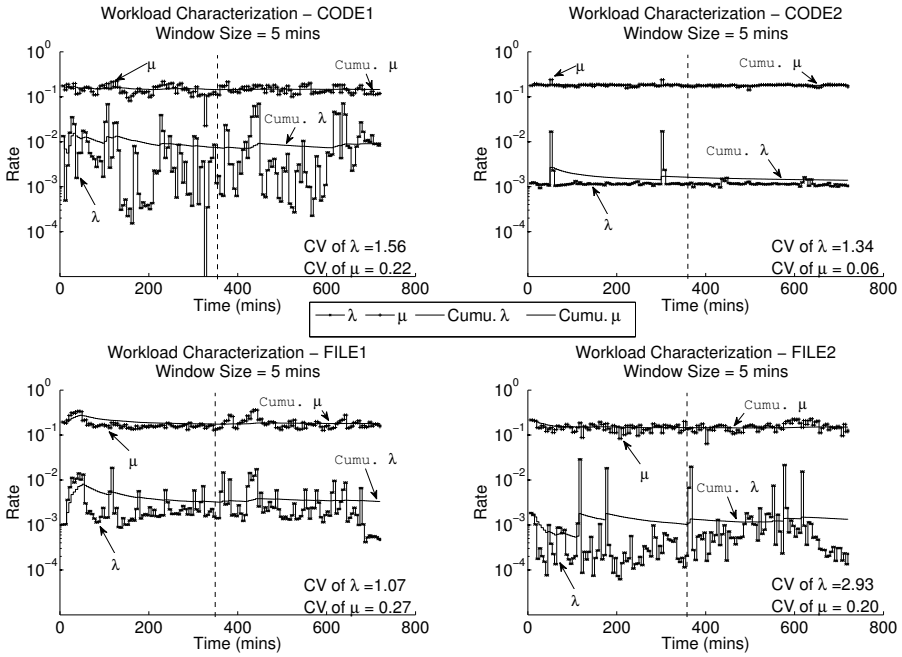



**Figure 5.1**: The workload characterization of the four traces. $\lambda$ is the Mean Arrival Rate of each window (5 mins), $\mu$ is the Mean Service Rate of each window, Cumu. $\lambda$ is the Cumulative Mean Arrival Rate across the time, Cumu. $\mu$ is the Cumulative Mean Service Rate across the time. The rate is measured in msec and plotted in log scale. The vertical dash line in Time = 360 mins separate the first part (left side, which we use as learning period) and second part (right side, which we use as testing period) of the entire trace.

In a cluster, there is a high chance that the disks or storage devices are heterogeneous. This is the reason why in Eq. (2), we introduce the correcting factor $\rho$, that captures on the average captures the differences in physical capabilities (such as rotation speed) between different disks. While there may be ways to determine $\rho$ off-line, here we estimate $\rho$ by analyzing the service rates for requests of the same or similar sizes. We group requests based on their sizes, in a effort to separate the random portion of the workload (short requests) from the sequential portion of the workload (long requests), because the differences in service rates can mostly be observed in the random rather than in the sequential portion

of the workload.

In our four nodes, the type of disk and rotation speed are not known. In addition, we do not know the sequentiality/randomness of the workloads. We do know that both CODE traces are from the same storage system as are both FILE traces. The first inclination is to set $\rho = 1$ for the pair of CODE traces and the pair of FILE traces. Figure 5.2 left shows for the FILE1 and FILE2 traces the service rates (measured by MBytes/ms) as a function of the request size for each IO request. In order to eliminate the effect of seek optimization for queued requests on the service times, the plots show service rates only for the requests within a busy period in our traces.

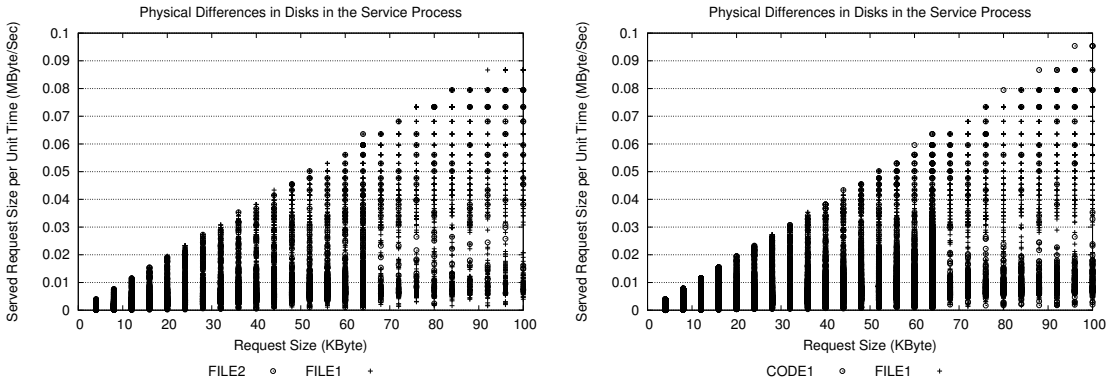



**Figure 5.2**: Identifying effects of physical differences in disks in the service process.

Clearly, because of sequentiality, service rates increase as request sizes increase. The service rates of short IOs (left part of the plot) are almost indistinguishable between the two disks (points overlap with each other in the figure). Even the rates of large IOs are also similar. This suggests that both short and long IOs behave similarly on both FILE1 and FILE2, suggesting the same "random" (and "sequential") behavior in the two traces. This justifies our choice of $\rho = 1$. The right graph of Figure 5.2 plots the same metrics but now for CODE1 and FILE1. The behavior captured by the right graph is very similar as the behavior of the left graph in Figure 5.2, suggesting the same "random" behavior across

FILE and CODE traces too,[1] The plots in Figure 5.2 justify setting $\rho = 1$ in Eq. (2) even when evaluating pairings of FILE and CODE traces. This process for estimating $\rho$ can be automated by allowing each storage node to maintain together with other metrics a small histogram of requests sizes and the service rates observed for each of them.

### 5.3.2 Response Time Prediction

We predict performance (measured via average response time) on the device that serves the consolidated workload using a look-up table, which records the observed response time for a pair of observed arrival and service rate. The prediction of the arrival and service rate at the receiver node is done using Eqs. (1) and (2). For each node, we store the average arrival rate (observed in the short-past which serves as the prediction of its near-future arrival rate). In the experiments presented here "short-past" and "near-future" are intervals of 6 hours each, that correspond to the first and the second part of the trace in Figure 5.1.

An additional information needed is the amount of work to be shifted to the receiver server. We use the workload shaping techniques outlined in [73] that also calculate which requests are to be moved provided that the intermediate buffer (i.e., the total data to be transferred) is equal to 1 GB, 5 GB, or 10 GB. Note that the size of these buffer sizes is relatively small. Moving just a small amount of data rather than the entire working set (e.g., as proposed in [109]) can be very beneficial for consolidation purposes. The data that we move is based on workload characterization of the most frequently accessed blocks or groups of blocks [73]. As a result this intelligent data copy may relieve the disk from the most highly accessed blocks and increase the idleness used for power savings considerably.

[1] We have plotted the service rates as a function of request size for all combinations of FILE and CODE traces. The behavior is very similar to that reported in Figure 5.2 and is not shown here in the interest of space.

The small amount of data movement can also relieve the concern of power consumption during copying and make the data placement problem easier. In addition, placing only a small amount on data on the receiver end minimizes the odds of dramatically increasing the service process due to the additional work. Note here the redirected workload would have to be served someplace anyway, so by making another disk serve it does not increase the overall power consumption. Figure 5.3 illustrates the IO load to be redirected from the sender node to the receiver node for two workload shaping techniques: Busy Period Offloading (BP-Offload.) and Probabilistic Offloading (Prob.-Offload.) [73]. BP-Offload. offloads the most frequently accessed busy periods, i.e., groups of blocks between two idle intervals, until the predefined buffer is filled. Prob.-Offload. removes a number of busy periods based on the correlation of the length of idle intervals succeeding the busy period (e.g., a long idle interval following another long one), and aims at concatenating long idle intervals. The figure clearly illustrates that irrespective of the sender or the workload shaping technique used, the redirected workload is only a fraction of the overall workload. Therefore, the overall load at the receiver may only increase a little. Consequently the expectation is that the performance at the consolidated nodes degrades only slightly, if it degrades at all.

An integral part of the workload prediction is to approximate the response time at the receiver storage node after work is consolidated. We achieve this by building look-up tables that hold observations of different pairs of arrival and service rates and the corresponding response times, see Section 5.2. The look-up tables are constructed from the traces of Table 5.1. Since the length of our traces is relatively limited, we varied observation lengths from 15 minutes to one hour that result in look-up tables of 512 entries each. Here, because the physical capabilities of all storage devices are very similar (see Figure 5.2), we merge the tables constructed by all traces into a single table.

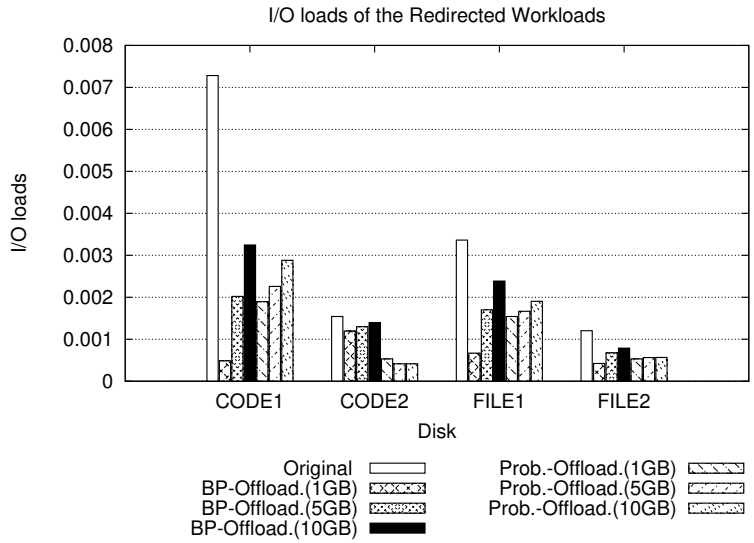



**Figure 5.3**: IO load to be redirected from the node that would be set off-line to the node that would remain active.

Selecting the best matching tuple in the look-up table is an integral part of our prediction. Naturally, all possible arrival/service exact values are not going to be found, so we approximate by exploring values that are within 10% of the anticipated arrival and service rates. While we expect the tables to be dense enough to allow for 10% match, in case no matching pair is found, we rescan the look-up table with 5% higher difference in matching. Among the set of pairs that meets these criteria, we select the one that would best continue to maintain the "trend" of the observed response time. For example, if the current response time in both storage nodes under consideration are higher than the one predicted from the look-up table, then that prediction is not possible. This is supported by our assumptions that the service rates change only based on the differentiator $\rho$ and consequently the response time should be at the minimum and not better than either of them. If there are still multiple tuples that have matched our criteria, then we go for the one that minimizes the sum of the differences between the observed rates and the

anticipated ones.

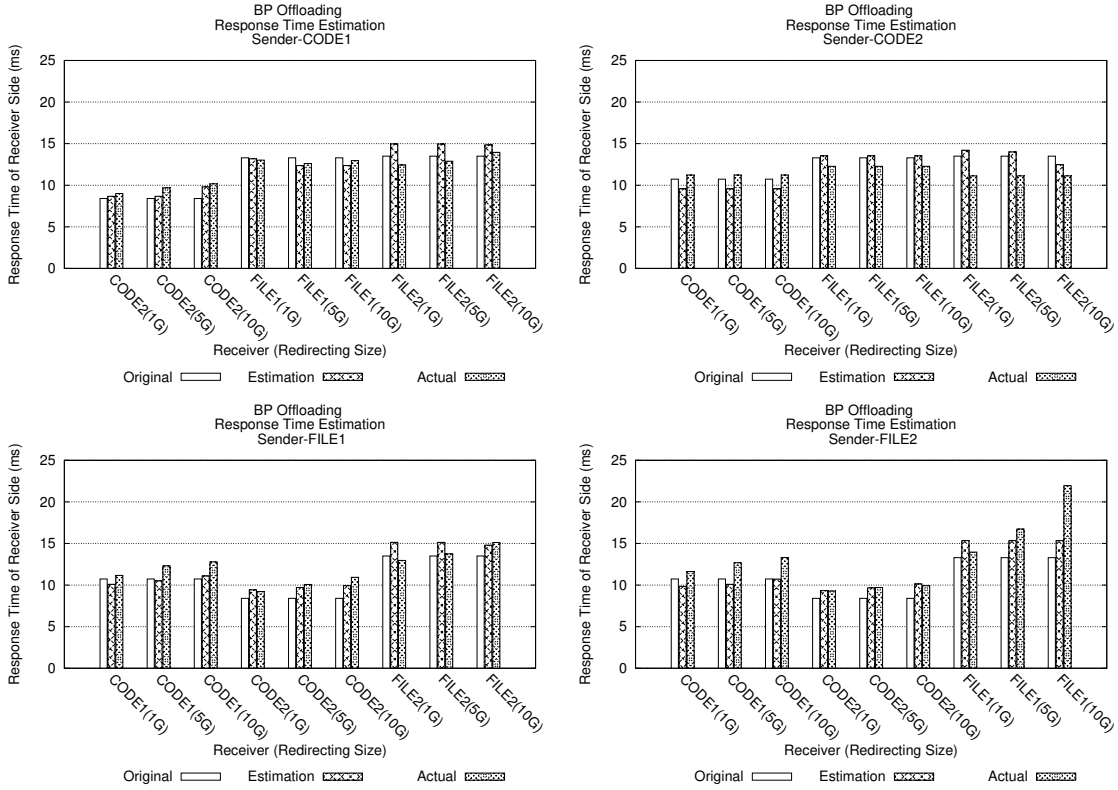



**Figure 5.4**: Performance measured via response time for BP-Offload..

The response time estimations for all cases under consideration are given in Figures 5.4 and 5.5. Note that except for the bar that is labeled "Original" for each receiver (and that corresponds to the response time without any consolidated workload), the graph also shows the framework's prediction (labeled "Estimation") and the actual response time after consolidation (labeled "Actual"). Recall that all framework estimations are done using the first half of the trace for all nodes. The simulation validates the accuracy of the estimations on the second half of the traces. The consolidated workloads in the second half of the trace maintain the same service process as measured in the second half of the trace, since the assumption is that the small areas that will hold the replicated data can be

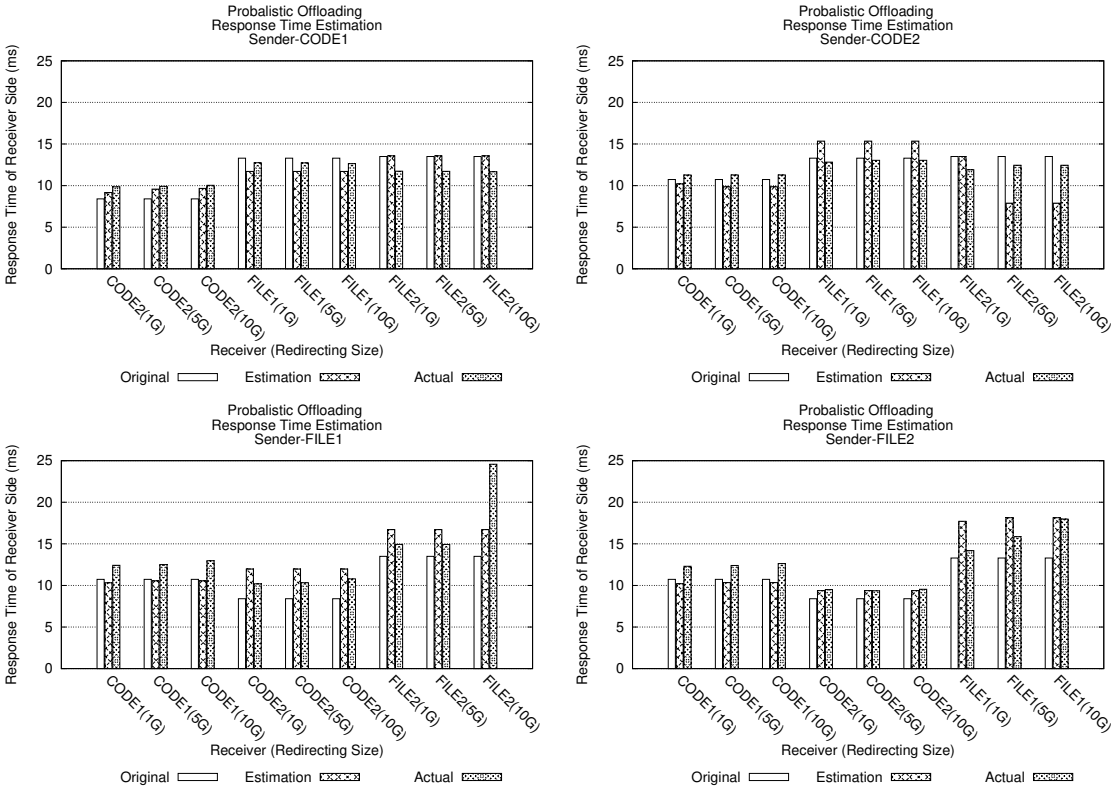



**Figure 5.5**: Performance measured via response time for Prob.-Offload..

placed such that the service process does not degrade. In most of the cases the framework overestimates response time, so the chances for wrong suggestions are small. Overall, the framework is successful in identifying pairs of sender-receiver nodes given certain power and performance targets.

### 5.3.3 Consolidation Decisions

In order to make decisions on how to consolidate the storage devices in a cluster such that the power savings are as high as possible without violating performance targets, each node should know "how much power it can save" and "how to achieve such power savings". We use the workload shaping proposed in [73] and the framework proposed in [89] to predict

power saving benefits for a given workload shaping method as well as the amount of data and the load that needs to be redirected for that purpose. In Figure 5.6, we show power savings for each of the four storage nodes in our cluster for BP-Offload. and Prob.-Offload. workload shaping schemes. The figure shows the percentage of time that the sender (disk) can be placed in low power mode. As with all estimations in this thesis, the workload shaping estimations are done based on monitored metrics during the first half of the traces that are applied (i.e., tested) the second half of the traces.

The "Original" bar corresponds to the time that the system can be in power savings *if* there is no workload shaping and only the observed idleness in the storage node is taken into consideration. The graph also reports savings for the two workload shaping methods and three different sizes of data to be moved (i.e., buffers equal to 1, 5, and 10 GBytes). The content of the buffers (i.e., "what" is going to be replicated in the storage node) depends on workload shaping. For details on the shaping methods and their performance, we refer the reader to [73].

Figure 5.6 shows that if we assume a cluster of 4 storage nodes serving the workload of CODE1, CODE2, FILE1 and FILE2, then CODE1 and CODE2 have the highest potential for power savings. While for some workloads the buffer size matters, a medium buffer size of 5 GBytes performs overall well.

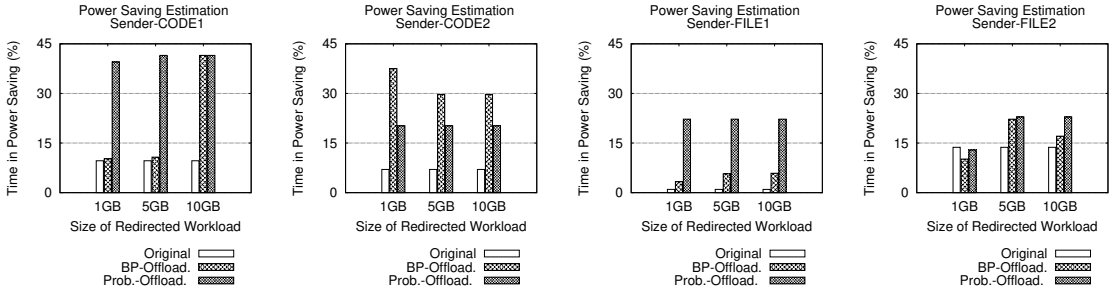



**Figure 5.6**: Power Saving by different offloading methods.

Replicating the data from each storage node as defined in Figure 5.6 somewhere in the cluster and redirecting the IO load accessing that data to the consolidated node, would result an additional load for each cluster, see Figure 5.3. The values shown in the plots are estimations of how the accesses on the data replicated to other nodes would look in the near future based on the observations from the first half of each trace.

Figure 5.6 suggests the best disks to offload work elsewhere but does not tell us "where" to move this work. Figure 5.4 and Figure 5.5 give the answers to this question. Here, we present the estimation and validation of performance degradation (in terms of average response time at the receiver end) for each of the possible (sender, receiver) pairs in the storage cluster. Each plot represents the possible pairings of a sender storage node and the potential receivers. The inaccuracies in the prediction of the consolidated response time come from the changes in the workload between the learning period and the testing period, as well as the density in the look-up table and their ability to provide a close enough match to the predicted arrival and service rate pairs that are used to locate in the look-up table the expected response time.

The decision on which pairs of storage nodes to choose for possible consolidation is done based on the storage performance target: average response time has to be always below a certain value. For example, if the receiver must have an average response time less than 10 ms, then if either BP-Offload. or Prob.-Offload. are used at the sender, pairing CODE1 as sender with CODE2 as receiver is a good idea, ditto for FILE2 as a sender and CODE2 as receiver. However, once CODE2 receives the load from CODE1 (buffer size of 10 GB), then no other pairings in the cluster would satisfy the performance condition. With such performance targets, power savings can be initiated only 40% of time on a single storage node. Also note that our estimation (and validation) points to a counterintuitive choice of a node to be put off-line: CODE1 is the one with the highest utilization in the cluster

of 5.6 (see Table 5.1), i.e., an unlikely choice for any scheme that makes decisions based solely on utilization levels.

If the performance target is response time of 15 ms, then most pairs can be selected except the following two cases: FILE2 as a sender and FILE1 as a receiver when sender using BP-Offload., and FILE1 as a sender and FILE2 as a receiver when the sender uses either of the offloading methods. Then, the best choice is to select two nodes to be turned off and two nodes to serve the consolidated workloads. Since CODE1 and CODE2 provide the highest power savings, then they can be selected as senders and FILE1 and FILE2 as receivers. For a 1 GByte buffer (the smallest buffer size), CODE1 achieves 40% time in power saving under Prob.-Offload. while CODE2 achieves almost 40% time in power saving under BP-Offload., for a total of two devices providing 40% time in power savings for each.

Such decisions are not obvious, since all traces have low utilizations and any pairing of nodes would represent an opportunity for schemes that make decisions based on utilization only may result in detrimental savings. For example, FILE1 does not have high power savings compared to other traces, and FILE1 would perform poorly if FILE2 offloads its work on it.

## 5.4 Summary

We present a method for consolidating workloads in a storage cluster while meeting performance targets such that the transparency to the end users is preserved. Our estimation provides criteria for the work consolidation between sender and receiver storage nodes in the cluster that aims at maximizing metrics such as power savings. At the center of the framework is a learning method that enables prediction of performance in presence of workload consolidation. The framework offers great flexibility in proposing

alternatives that can maintain performance guarantees that remain below pre-advertised values. The main findings of this chapter are also reported in [126].

# Chapter 6

# Toward Fast Eventual Consistency

The majority of computer systems today are faced with the need to scale because the services they provide and the data they store increase at a significant pace. In order to provide uninterrupted computing services and access to the related data, it is necessary for systems to extend their boundaries. In designing such scaled-out distributed systems [26, 39, 16], it is necessary to balance cost, performance, reliability, and availability. Specifically, in distributed storage systems, it is expected that data is spread across multiple nodes and geographic locations such that a wide range of network, power, and other failures do not cause data unavailability [96, 58, 100]. Yet, as new data arrives in the system, from the performance perspective, it is not as efficient for the system to propagate the data to the various locations in real time, because the impact on end user performance (now including also WAN transfers) may be significant. A solution is for the system to distribute the data across the locations asynchronously [36, 20, 96]. As a result, the data reaches its expected locations *eventually* and the systems strive to achieve *eventual data consistency* [110, 8, 112]. Systems that aim to achieve eventual data consistency often provide cloud services [112, 57], which require that data reliability is not compromised. For example, data is protected via RAID [17] locally. However, the location failure tolerance is achieved only eventually.

Eventual data consistency is a "loose" term. It means that data can eventually reach its distributed locations, but it just does not quantify how fast. This depends largely on the supporting infrastructure, e.g., the network bandwidth, the distance between the data centers, as well as the scale of the system and its quality goals, e.g., performance, reliability, and availability. It also depends on how aggressively the system schedules the asynchronous tasks [33], given that they may interfere with the normal user traffic and impact its performance. Commonly, these tasks are scheduled based on the current utilization levels of each node, i.e., asynchronous tasks are scheduled mostly during periods with low node utilization.

In this chapter, we focus on how to schedule these asynchronous tasks that distribute data across different locations such that the performance in the sending and receiving nodes meets predefined quality of service goals. The scheduling parameters for the asynchronous tasks are determined and updated continuously at the individual node level as they learn the characteristics of the workload they are serving. Such parameters are exchanged between the nodes in order for them to synchronize the speed of data transfer. It is expected that different pairs of nodes in a geographically distributed system have different communication speeds. As a result, it is critical to synchronize the speed of data transfer so that failed attempts are reduced and eventual consistency is achieved faster.

The learning in our scheduling policy consists of understanding the available idleness that can be used to serve the asynchronous updates. We utilize the histogram of idle periods as it is done in [71] to determine when to start and stop serving tasks without violating performance goals.

In this chapter, we illustrate the robustness of the proposed solution visa extensive experimentation with simulations driven by traces collected in real storage systems. The proposed solution is orders of magnitude faster than the common practice of utilization-

based scheduling and is comparable to an aggressive policy that schedules asynchronous tasks as soon as the system becomes idle. We note that our solution provides guarantees on the performance of each node in the system and reduces the time to reach consistency for newly added data, something that none of the alternative policies can achieve.

The rest of this chapter is organized as follows. In Section 6.1, we provide some background on consistency issues and state-of-art scheduling strategies in storage systems. In Section 6.2, we provide a detailed analysis of a set of enterprise traces and show how the characteristics of workload can help us to develop our scheduling framework. In Section 6.3, we propose an analytic framework that computes the scheduling parameters based on the learned characterization of idleness and other system information. Section 6.4 presents an extensive set of trace-driven experiments that demonstrates the effectiveness and robustness of the proposed solution. We summarize in Section 6.5.

## 6.1 Motivation

In this section, we introduce the asynchronous data redundancy scheme and aspects of data reliability and integrity when handled asynchronously. We also summarize the state of the art in scheduling techniques, which also motivates the work presented here.

### 6.1.1 Asynchronous Data Redundancy Scheme

We denote the node in the distributed system that receives the new data as the "active" node and the nodes that would receive replicas (or parts of the data) asynchronously as "inactive nodes". In our exposition, data may arrive in any node in the system, which means that any node can be an active node for some data and an inactive node for other pieces of data.

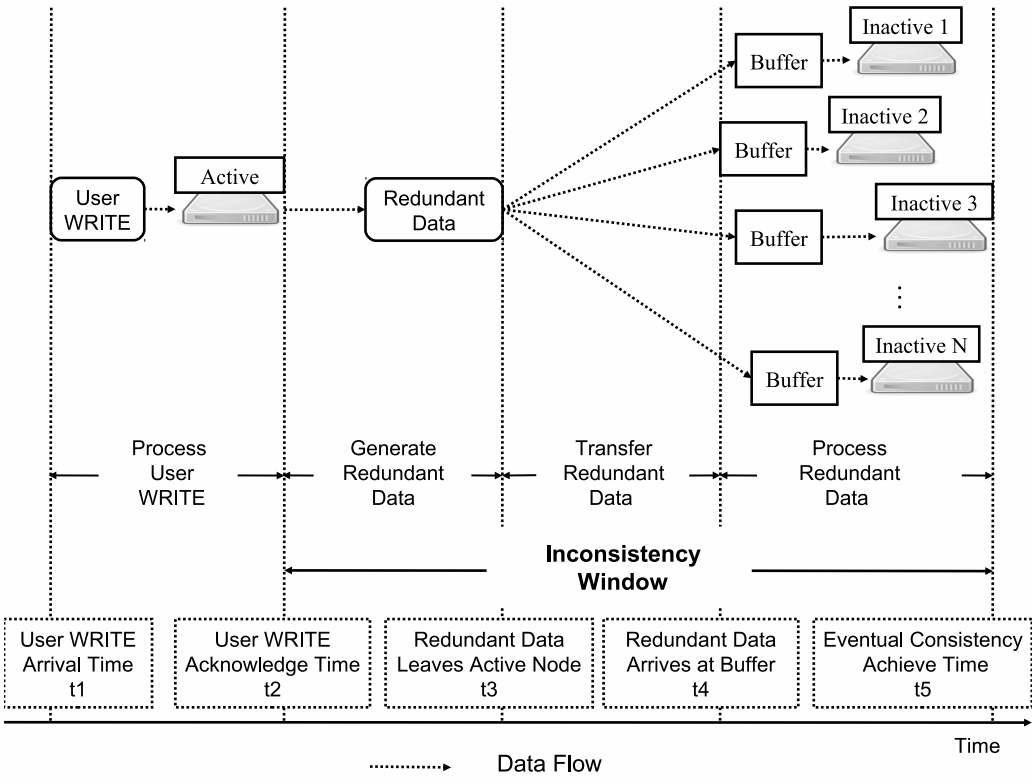



**Figure 6.1**: Schematic view of the system with asynchronous replication and eventual consistency.

A schematic view of how data is stored redundantly and asynchronously is shown in Figure 6.1. When new data arrives, it is acknowledged and processed by its active node and then spreads across the system (i.e., to the nodes that should receive it). In large scale storage systems,data is either replicated in multiple locations or it is stripped, coded, and distributed in different locations (erasure coding). Independent of the specifics on how the data is redundantly stored, the fact is that the targeted redundancy is achieved asynchronously as background tasks (BG), which is outside the critical path of serving the user traffic.

When the redundant data is sent out from the active node over the network, it can be delayed depending on the distance between the nodes and the amount of data being

transferred. The inactive node that receives the data, buffers it in cache before committing it to a storage device. The buffer is required because the inactive node may be serving its own user traffic, and the goal is not to impact its performance according to system quality targets. If the inactive node processes such data upon arrival, then its user performance impact may be severe. It is clear that the inconsistency window has three parts: the time it takes to send out the data from the active node, the time to transfer the data over the network, and the time to commit the data on the storage devices of the inactive node. The eventual consistency for each piece of data is achieved when all inactive nodes that should have received a copy or fragment of it have done so successfully. This means that from the perspective of modeling the duration of the inconsistency window, the problem can be simplified to having one active node and one inactive (slowest) node, without loss of generality.

The issue with the asynchronous traffic is that it impacts system performance regardless of how carefully it is scheduled because often IO tasks are not *instantaneously* preemptable [67, 92, 84]. Judicious scheduling of asynchronous tasks that is done as quickly as possible so that data durability is high, while user traffic is not affected, is a challenging task.

### 6.1.2 State of the Art in Scheduling of Background Jobs

In this section we quickly review three scheduling methods that are widely used to schedule background work in storage systems:

- *Aggressive* scheduling schedules replication work immediately and without any consideration of foreground user traffic. Such scheduling reduces the inconsistency window but may result in very high and unpredictable user performance degradation.

- *Utilization-guided (Aggressive)* scheduling takes the user traffic into consideration by monitoring utilization. If the system utilization is below a threshold, then it schedules replications immediately. When utilization is high, it stops scheduling any replication work.

- *Utilization-guided (Conservative)* scheduling uses system utilization as guidance and schedules the replication work only when the system utilization is low. Before scheduling any replication job during a low utilization interval, the system idle waits for a certain amount of time [30] to avoid using small idle intervals, which have a higher chance to cause extra delays to user traffic.

From the above policies, only the third one strives to reduce the performance impact of the inactive node traffic, although still without performance guarantees. Note that utilization-based policies depend on the characteristics of system utilization that may be very different across different time scales (e.g., minutes versus days). To illustrate this, we plot in Figure 6.2 the average utilization of a representative trace from Microsoft Research, and this trace is described in detail in the following section. The plot shows a large variance in utilization when looking in 10 minute, 1 hour, and 1 day windows and suggests that utilization, as a steady-state metric, is not suitable for scheduling purposes. If utilization is monitored in a too long interval, then it cannot capture well the unpredictability of user traffic. If it is monitored in a too short interval, it may not be able to predict the near future correctly based on current and past information because utilization changes swiftly at such scale. This observation motivates us to devise a more sophisticated yet simple learning-based scheduling framework to overcome the above shortcomings.

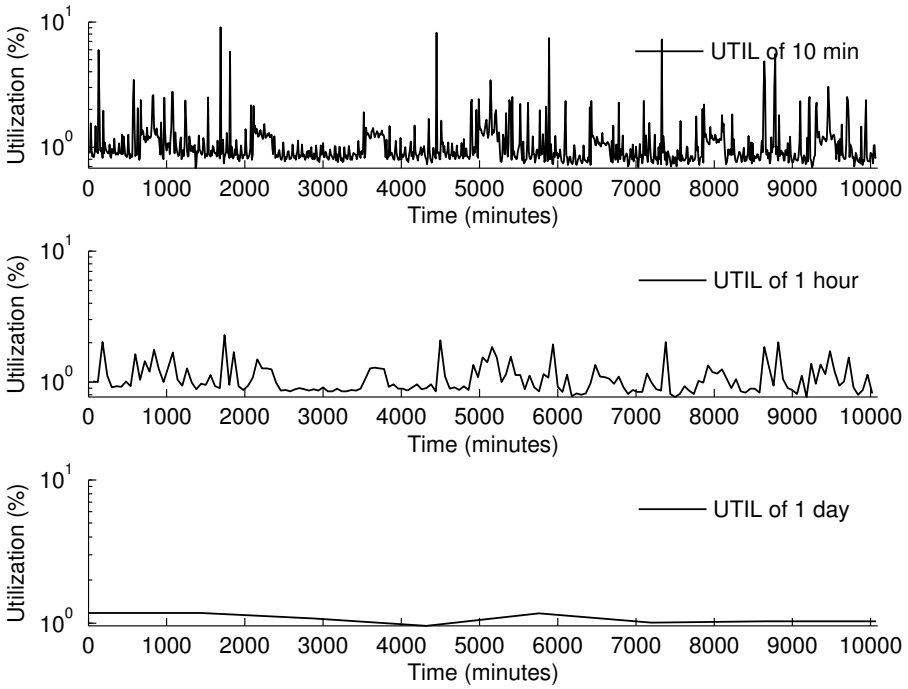



**Figure 6.2**: The utilization over time plots, the bin size for the top plot is 10 mins, for the middle one is 1 hour and for the bottom one is 1 day. Note y-axis is in log scale.

## 6.2 Workload Characterization

In this section, we analyze the set of traces used in our evaluation. First, we give some general information about these traces. Then we further characterize the idle periods length in more details and give some intuitions on how we take advantage of such characterization for the purpose of running fast and with performance guarantees the asynchronous tasks that aim at achieving eventual consistency in a distributed storage system.

### 6.2.1 Overview of Traces

We use storage system traces made available through the SNIA IOTTA repository [2] collected by Microsoft from its servers in data centers and published by Microsoft

Research Cambridge (MSR) [75]. Each trace records information about a set of attributes for each I/O request. Specifically, for each IO, we have the arrival time stamp, request type (write/read), offset from the start of logical disk, request size, and response time. In addition, other storage features such as simultaneous IO requests are reflected in these traces.

Table 6.1 presents an overview of various statistical measures for four traces[1]. The usr0 trace is obtained from a user file server, the mds0 trace comes from a media server, the ts0 trace is collected from a terminal server, and the web0 trace is captured in the Web/SQL server. Each trace has a duration of one week (168 hours) and represents a wide range of common traffic patterns. From the table, we can see that these systems show very low utilization, which suggests that good opportunities exist for serving background work, such as asynchronous tasks. The relatively substantial Coefficient of Variation (C.V., which is a normalized measure of dispersion, defined as the ratio of the standard deviation to the mean) suggests that using idleness may be challenging because scheduling too much background work during small idle periods may cause performance degradation while during large idle periods, scheduling too little background work may waste idleness and slow down the synchronization speed. We also note these traces are WRITE dominant workloads for which the asynchronous tasks of propagating the data through the system nodes play a very important role.

### 6.2.2 Characteristics of Idle Periods

We further evaluate the characteristics of the idle periods because asynchronous tasks are to be scheduled during these intervals. Figure 6.3 shows the Cumulative Distribution Histogram (CDH) of the idle period lengths. The figure indicates that more than half

[1]The Microsoft IOTTA repository has a larger number of traces than what we show here. We have selected only these four traces as representatives.

| Trace | Dura. (hour) | Util. (%) | Average Arrival Rate (1/ms) | Average Service Rate (1/ms) | Average Response Time (ms) | Idle Length | | R/W ratio |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Average (ms) | CV | |
| usr0 | 168 | 1.07 | 0.0012 | 0.1203 | 8.94 | 805.36 | 1.74 | 0.11 |
| mds0 | 168 | 0.52 | 0.0007 | 0.1412 | 7.21 | 1404.16 | 1.93 | 0.03 |
| ts0 | 168 | 0.61 | 0.0008 | 0.1455 | 7.06 | 1150.20 | 1.74 | 0.04 |
| web0 | 168 | 0.72 | 0.0010 | 0.1468 | 7.12 | 959.72 | 2.11 | 0.13 |

**Table 6.1**: General trace information. ms stands for millisecond.

of the idle periods are very small (note the log scale in the x-axis), which means that if we schedule the asynchronous tasks during these short intervals then it is highly possible that user requests may be delayed as a result of arriving to a system that is serving the asynchronous tasks, which cannot be preempted *instantly*. The goal is to incorporate the learning of characteristics of idle periods in a policy that schedules the asynchronous tasks such that they are served as fast as possible while user requests are impacted at a minimum.

Figure 6.4 plots the idle time intervals across time. The plots clearly show that there is a daily cyclic pattern which suggests that if we characterize well these idle periods within such a cycle, then we may be able to accurately predict the next cycle. Comparing to utilization, idleness depicts more of a cyclic behavior, making it more reliable as a metric to guide the scheduling policy. In addition, we expect that using the information from the CDH of idle intervals rather than a simple average value of idle interval lengths would result in more reliable predictions and robust scheduling.

## 6.3 Asynchronous Update Scheduling Framework

In this section, we propose a learning-based framework for scheduling asynchronous updates. We first introduce the basic premise of the learning-based scheduling of background work. Then we explain in more details how to estimate the amount of replication work so that the framework can compute correct scheduling parameters.

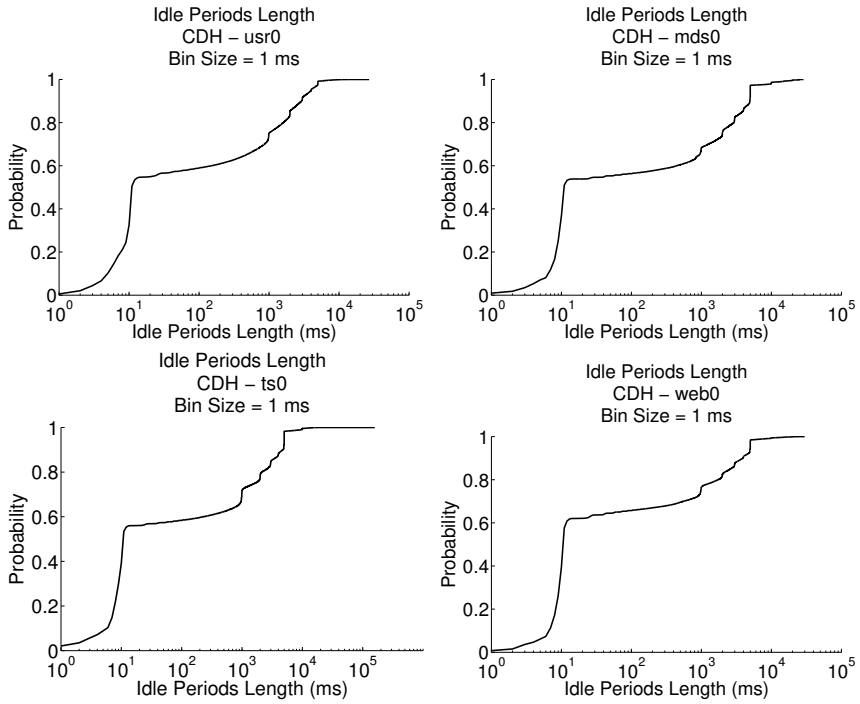



**Figure 6.3**: The CDH of idle period lengths measured in ms. Note that the x-axis is in log scale.

### 6.3.1 Learning-based Scheduling with Performance Guarantees

We first describe an algorithmic framework that schedules background work, e.g. asynchronous tasks, with performance guarantees for the foreground traffic. This algorithmic framework is used to estimate the performance impact of background work and determine the most effective schedule for it by determining when and for how long to schedule background tasks in storage devices, such that the trade-off between performance degradation and how fast background tasks can be scheduled meets system performance targets.

One could argue that starting a background task immediately after the storage subsystem becomes idle would be most efficient. However, because of the stochastic nature of idle periods and the *non-instantaneously* preemptive nature of tasks in storage

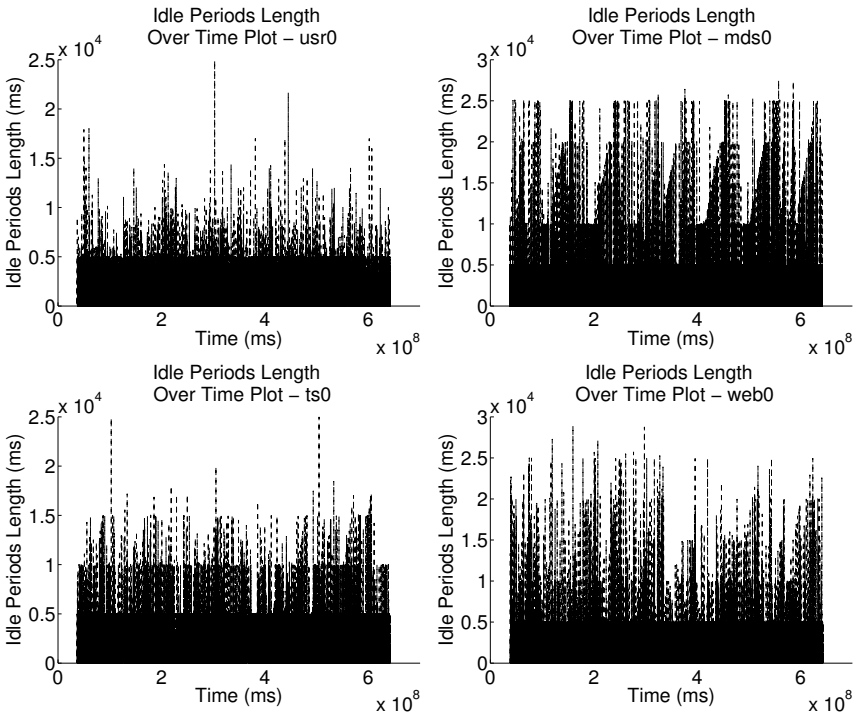



**Figure 6.4**: The idle periods length overtime plots.

devices, user performance may suffer significantly. In storage systems, it is very common to idle wait for some time before starting a background task, as to avoid utilizing the very short idle periods for any background activities [30]. In addition to that, [40] suggests that limiting the amount of time that the system serves background tasks further limits the performance impact on foreground jobs. The framework in [71] computes both the *idle wait* $I$ and the duration $T$ of the time to serve background jobs as a function of past workload (i.e., the stochastic characteristics of past idle periods). Note the $T$ is introduced here so that the disk can be proactively ready to serve user traffic and therefore avoid user performance degradation in all the idle periods used for scheduling background work. We use here this $(I, T)$ tuple to compute the schedules of the asynchronous updates in distributed storage systems, while meeting predefined

performance targets.

Central to the calculation of $I$ and $T$ is the CDH of idle intervals. In addition to the CDH, the framework also uses the user-provided average performance (degradation) target $D$, which is defined as the allowed average relative delay of an IO operation due to the background tasks and can be computed from the $(I, T)$ scheduling pair and other statistical information such as average response time.

Let's assume that $W$ is the average IO waiting due to serving background tasks. Without loss of generality, we measure the idle interval length as well as the wait within the 1 ms granularity. Because a disk is activated upon an IO arrival, $W$ can be at most $P$, which is the time penalty that a foreground IO request may suffer, if it arrives while the disk is still serving the asynchronous tasks that result from propagating new data throughout the distributed system. We assume that the data to be redundantly stored in distributed nodes is already stored in the local storage. As a result, the penalty can be estimated from the average service time of an IO request done to the local storage, because when a new user request comes, it needs to wait until the asynchronous task completes. By "local storage" we mean any storage device that can be used, from memory to SSD to local disk. Consequently, the penalty is different for different storage devices and we reflect this in our computations. Yet, we argue that in scaled-out systems, where efficiency is key, utilizing memory or SSDs to store the data that is asynchronously distributed across the nodes, may not be the most cost-effective choice because such background work should be off the critical path for better user traffic performance. In our evaluations we assume that disk IO penalty of several ms depends on the specific characteristics of the IO workload.

By denoting a possible delay by $w$ and its respective probability by $Prob(w)$, we define

$$W = \sum_{w=1}^{P} w \cdot Prob(w). \tag{6.1}$$

where $w$ is the delay caused to the IOs of the busy period following the scheduling of background tasks and may be any value between 1 and $P$. Using the probabilities in the CDH of the lengths of idle periods, the probability of any delay $w$ caused to the IOs of the following busy period is given by the equation below

$$Prob(w) = \begin{cases} \text{CDH}(I+T-w+1) - \text{CDH}(I+T-w), \\ \qquad\qquad\qquad\qquad \text{for } 1 \leq w < P \\ \text{CDH}(I+T-P) - \text{CDH}(I), \text{for } w = P, \end{cases} \tag{6.2}$$

where CDH(.) indicates the cumulative probability value of an idle interval in the monitored histogram. The intuition behind this equation is that for a scheduling pair $(I, T)$, the delay to the busy period following the scheduling of background tasks is $w$ $(1 <= w < P)$ if the idle interval length is larger than $I + T - P$ and the probability is given as $CDH(I + T - w+1) - CDH(I+T-w)$. For all idle intervals whose length falls between $I$ and $I+T-P$, the delay is $P$ and the probability of this event is given as $CDH(I + T - P) - CDH(I)$.

To find the qualified scheduling pair $(I, T)$, we scan the CDH of idle periods length for $(I, T)$ pairs that would not violate the target $D$. Note that $I$ and $I + T$ correspond to successive histogram bins. A pair $(I, T)$ guarantees the performance target $D$ if

$$D \geq \frac{W_{(I,T)}}{RT_{w/o\ BG}}, \tag{6.3}$$

where $RT_{w/o\ BG}$ is monitored and $W_{(I,T)}$ is computed using Eq. (6.1). Larger $D$ ensures

faster background work completion.

### 6.3.2 Calculation of Scheduling Parameters

The replication work should be transparent to user performance. We measure transparency in terms of the performance degradation $D$ as introduced earlier. The first scheduling target is to complete all replication work without violating any performance target. The algorithmic framework in 6.3.1 can be used to schedule asynchronous updates (e.g., replica WRITEs in disk IOs) during appropriate idle periods at both active and inactive nodes. The framework uses the histogram of the lengths of idle periods to generate a "schedule" for replication work and estimate the amount of completed work for each idle interval so that it is higher than the average amount of replica WRITEs. This is necessary to prevent uncontrolled replica backlogs. We estimate the average WRITE work amount $B_W$ measured in units of time as

$$B_W = \frac{\rho_W * E[idle]}{1 - \rho_{FG}} \tag{6.4}$$

where $\rho_W$ is the average utilization contributed to WRITE requests, $\rho_{FG}$ is the average utilization of all user requests, and $E[idle]$ is the average idle interval length. The term $\frac{E[idle]}{1-\rho_{FG}}$ corresponds to the average length of one busy period plus one following idle period, and if multiplied by $\rho_W$, it represents the average amount of time WRITE requests need to be served during one busy plus one following idle periods.

As a second step, we use the framework introduced in 6.3.1 to compute all valid scheduling pairs $(I, T)$ given the performance target $D$. Each scheduling pair schedules in average $B_{BG}$ amount of background task measured in units of time in idle intervals at the storage nodes. We calculate $B_{BG}$ as follows:

$$B_{BG} = \sum_{o=I}^{I+T-P} p(o) \cdot (o - I) + \sum_{o=I+T-P}^{max} p(o) \cdot (T - P) \tag{6.5}$$

where $p(o)$ is the probability that an idle interval is of length $o$ and $max$ is the maximum length of the idle intervals in the CDH. Intuitively, $B_{BG}$ is comprised of two kinds of idle intervals that are larger than idle wait time $I$ (intervals smaller than $I$ are not used for replication work). The first type of idle intervals are of length $o$ that falls between $I$ and $I + T - P$. Because the replication work in this kind of intervals terminates at the end of each idle interval, which is before the limiting time $T$, their contribution to the overall $B_{BG}$ is only $o - I$. The second type of idle intervals are of length $o$ that at least $I + T - P$. In this case, the replication mode stays for $T$ time units, so their contribution to the overall $B_{BG}$ is $T - P$. Then we multiply them by the probability of each used interval and sum them together to get the average amount of replication work $B_{BG}$. Among all the valid scheduling pairs $(I, T)$, we only choose the one with $B_{BG} >= B_W$ so that there is never replication work that is never served (i.e., there is no starvation). There may be multiple pairs that qualify for meeting both the target $D$ and $B_W$. From those, we select the one with smallest $I$. If still multiple pairs qualify, we select the one with largest $T$ so that it is possible to schedule as aggressively as possible to ensure that replication work also finishes as fast as possible and there is no backlog.

### 6.3.3 Learning-based+ Scheduling

We also provide a more aggressive variation of the scheduling mechanism described above. The standard approach above only schedules for a $T$ period of time for each idle interval longer than $I$. If there are still asynchronous tasks to complete upon $T$ elapsing, the system does not schedule them even if it is still idle. For this reason we consider the above scheduling policy as being strictly non-conserving, guided by both $I$ and $T$.

Here we are proposing a more aggressive policy by relaxing the condition on $T$. Specifically, after scheduling asynchronous tasks for $T$ time units and the system remains

idle with additional asynchronous work outstanding, then the policy is changed to wait another $I$ units and re-start scheduling for another $T$ time units. This is done repeatedly until there is no more asynchronous work to be served or when the system becomes busy.

This extension to our framework ensures that the very long idle intervals are utilized more if there are asynchronous tasks waiting for completion. It does not change the behavior for the short idle intervals, where the potential for delays to user traffic is higher. However, since the goal is to serve as fast as possible all asynchronous tasks, then by allowing the long idle intervals (that are only a few) to be utilized more if there is work to be done, we achieve a faster response time for asynchronous tasks without the additional delay on user performance.

## 6.4 Experimental Evaluation

In this section, we evaluate the proposed scheduling framework via an extensive set of experiments. We use the traces described in Section 6.2 to drive a set of simulations. The experiments that we present in this section validate the robustness and efficiency of our solution proposed in this chapter with regard to

- the time it takes to achieve the eventual consistency,

- the impact on user performance, and

- the amount of buffer space required to store all incoming data updates at the destination nodes before committing them on persistent storage.

### 6.4.1 Experiment Scenarios

The set of simulations that we developed to evaluate the framework proposed in Section 6.3 as well as the other baseline alternatives are driven by the Microsoft Research traces. Recall that the node that receives the new data is the "active node" and the node that does the

same updates in the background (asynchronously) as the "inactive" one. The inconsistency window is composed by three parts: active node delay, network delay, and inactive node delay as introduced in Section 6.1.1.

We apply our scheduling framework for both active and inactive nodes and focus on minimizing the delays experienced at these nodes. We do not limit the buffer space, contending that the faster we complete the synchronization of data, the less buffer is needed. We also assume that there is no packet loss in the network and that the network delay is exponentially distributed with an average of 100 ms (i.e., the average delay for intercontinental round trip communication).

In our experiments, we use four different pairs of traces to evaluate the proposed solution and the alternatives under 4 different workload combinations. These pairings are given in Table 6.2. For each workload combination, we divide the available traces into seven portions or time windows, each corresponding to a full day workload (i.e., recall that the traces are 7 days long). Recall that during learning we update the histogram of idle periods length, the average arrival and service rate of WRITE, the average arrival and service rate of all IO. Our solution uses these monitored parameters to compute the scheduling parameters, i.e., when and for how long during the idle interval, the asynchronous tasks are executed. The learning procedure occurs during one full time window and the learning results apply on the next time window. This means that we run our framework once a day and update the scheduling parameters accordingly. We run the experiments across all six time windows (the first day/time window is used only for learning), but due to the limited space, we only show results for a subset of time windows.

| Pairs | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Active | usr0 | mds0 | ts0 | web0 |
| Inactiv | mds0 | ts0 | web0 | usr0 |

**Table 6.2**: The traces used for pairing active and inactive nodes during experiments.

We evaluate the following solutions for achieving eventual consistency: the fully work-conservative approach (we label it as "Aggressive") that starts to serve the asynchronous tasks as soon as the node becomes idle. The "Utilization-based" policy monitors the utilization of the system for the past 10 minutes, and if it increases above a threshold (the threshold is chosen as the average utilization during a long period, e.g., one day), then no asynchronous tasks are scheduled. If utilization drops below the threshold, then asynchronous tasks are scheduled aggressively, i.e., as soon as the node becomes idle. Note we use 10 minutes as the measurement window for a utilization-based approach because the utilization is a statistical parameter and if set too small (e.g. 1 min), it is statistically meaningless. Such swift changes are difficult to be used for predicting the near future. If set too large (e.g., 1 hour), the synchronization speed is too slow and there is always backlog. The above two policies are evaluated as baseline versions to compare with the two scheduling versions of proposed here; the basic "Learning-based" non-work-conserving version and the "Leaning-Based+" work-conserving variant introduced in Section 6.3.1.

Note that the "Utilization-based" approach is not work-conserving but is widely used in systems today, in an effort to limit the unpredictable performance impact that an "Aggressive" approach would have during periods of high utilization. Our experiments show that the impact of *all* alternative methodologies have an unpredictable impact on node performance and that only our "Learning-based" methods provide a solution that can maintain user-performance guarantees.

### 6.4.2 Delay on Achieving Eventual Consistency

Our initial experiments evaluate the total time that it takes, on the average, to propagate the new data or updates (e.g. WRITE in disk IOs) from the active node to the inactive node. Obviously, the faster the propagation of WRITEs, i.e., the smaller the inconsistency

window, the more robust and resilient the system is because during the inconsistency window, the system has staled data in inactive nodes, which may cause various problems, e.g., impact the back-order rate in TPC-W system [36] or break the application's contract with the user as the classic example discussed in [20]. We provide the results of the experiments on the duration of the inconsistency window in Figure 6.5, each row of plots in the figure corresponding to the node pairs described in Table 6.2. Since the learning-based solutions rely on the knowledge of various scheduling parameters including the CDH of idle intervals, we compute the $(I, T)$ scheduling pair based on system measurements in the previous time interval (an entire day). The columns of Figure 6.5 correspond to results for three different days. Results are plotted for different user performance targets (in %) (captured in the x-axis). For different performance targets (captured in the x-axis) there are different scheduling parameters for our framework and consequently, different results. The results for the baseline approaches are independent of such goals and their corresponding results do not change across the x-axis.

The Aggressive approach performs best with regard to how fast the WRITEs propagate through the distributed system, because it represents the *only* work-conserving policy. As we show in the next subsection, it also causes the largest, possibly unbounded (e.g. the delay can propagate and accumulate) delays in user performance because. As a result, in systems today, it is rarely used, but we include it here to use its performance with regard to the length of the inconsistency window as a baseline of the possible minimum. The closer other policies come to this approach without sacrificing performance, the more resilient they are.

On the other hand the Utilization-based policy makes scheduling decisions based on the monitored utilization levels in the immediate past. Because of the strong oscillations in the short-term utilization, it behaves as a very conservative policy that does not take

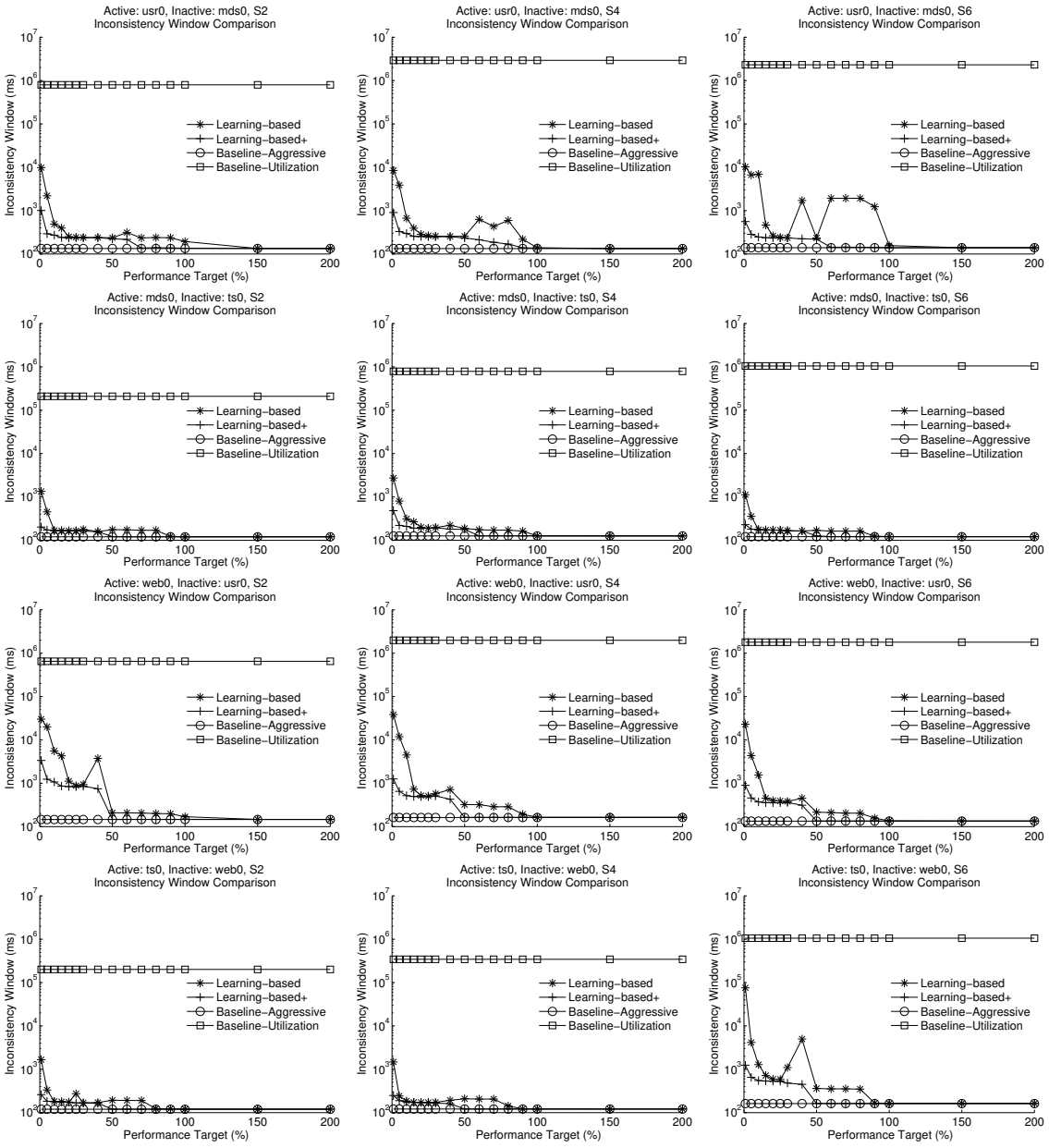



**Figure 6.5**: Inconsistency Window comparison between different scheduling for various active-inactive pairs (first row: usr0 - mds0, second row: mds0 - ts0, third row: web0 - usr0, fourth row: ts0 - web0. Three learning windows are considered: $Start = first\ day$ (left column), $Start = third\ day$ (center column), and $Start = fifth\ day$ (right column).

into consideration the available idleness in the system. Observe that the inconsistency window is orders of magnitude higher than the other alternative policies. Similar policies are common practices in systems today.

The curves corresponding to our framework, dynamically change as the target performance goals change. As expected, for systems that are more sensitive to performance and where the target is low, the eventual consistency is achieved at a slower pace than when the performance target is less stringent. Our scheduling converges to the Aggressive scheduling as the performance target increases to the performance degradation caused by the Aggressive approach. Note that the higher the performance target, the smaller the value of $I$, which indicates how non-work-conserving the policy is (i.e., $I = 0$ and large $T$ corresponds to a work-conserving policy). As expected, Learning-based+ achieves eventual consistency faster than the basic Learning-based approach and converges faster to the Aggressive scheduling. The few fluctuations in our scheduling results is due to the fact that we use the learning of a previous day, which obviously can result in some errors on the predicted workload characteristics.

The main observation from Figure 6.5 is that the proposed solution performs comparable to the Aggressive policy for any performance target (excluding the very small and impractical ones 1-5%). The Utilization-based approach is orders of magnitude worse for several times higher performance degradation.

### 6.4.3 Impact on User Performance

As discussed above, the time it takes to propagate the WRITE traffic and achieve eventual consistency is highly dependent on how much the user performance is degraded. Recall that serving the IO replicas as background work delays foreground user requests that arrive while the system serves replica updates because IO tasks are not *instantaneously* preemptable.

Here, we focus on how the various approaches perform with respect to foreground task degradation, measured as the percentage of the average user response time increase in presence of asynchronous tasks. We show the results in Figure 6.6, each row corresponding to different active-inactive pairs, and each column corresponding to different days in the trace. We still use the performance target (in %) as index of the x-axis and plot the *actual* performance degradation measured in simulations (in %) in the y-axis.

As expected, the Aggressive policy performs very poorly with regard to the actual user degradation in the system. The average user response time increases well beyond 50%, despite the fact that the asynchronous replica work is modest. The Utilization-based policy proves to be really ineffective, because although it results in very slow eventual consistency, it still penalizes user performance significantly, which attests to the inefficiency of making decisions based on short-term learning. We believe that not only the short-term learning is ineffective, but also that the metric of utilization itself as a guide to scheduling asynchronous tasks is also inefficient, despite the fact that it is widely used in practice.

Our framework, on the other hand, adapts its decisions to the system quality targets striking a good balance between system user performance and replica completion speed with the goal of achieving eventual consistency quickly without significant performance loss. The results in Figure 6.6 confirm the robustness of periods of long learning (we update our learning once a day, see results per column) as being more robust and effective than shorter learning periods as used in the Utilization-based policy.

### 6.4.4 Buffer Space Requirements

Since there cannot be a perfect synchronization between the speed that the active node sends its updates with the speed that the inactive node processes them, there is a clear need for buffering at the inactive node to temporarily store the incoming replica WRITEs.

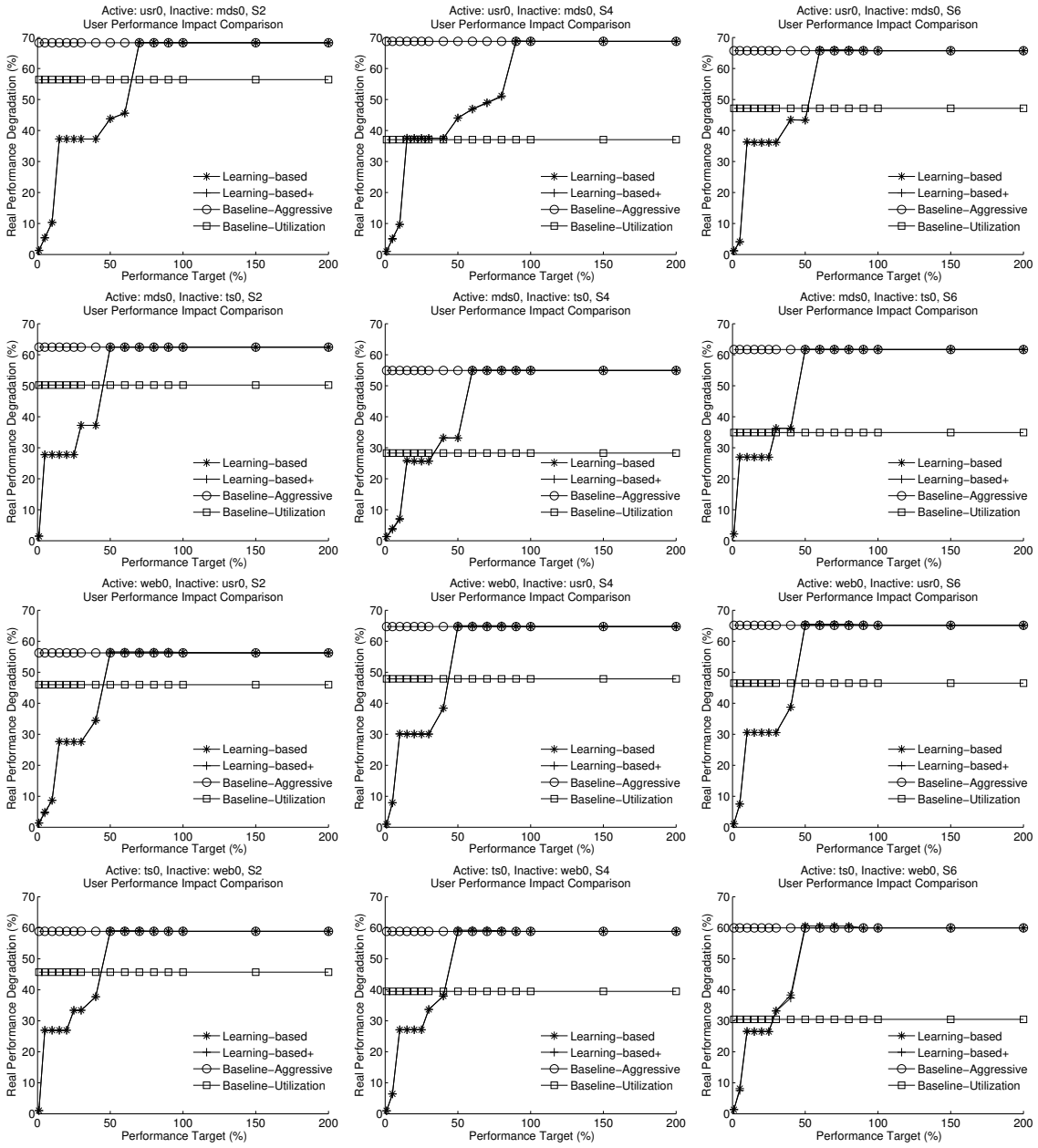



**Figure 6.6**: User performance impact comparison between different scheduling for various active-inactive pairs (first row: usr0 - mds0, second row: mds0 - ts0, third row: web0 - usr0, fourth row: ts0 - web0). Three learning windows are considered: $Start = first\ day$ (left column), $Start = third\ day$ (center column), and $Start = fifth\ day$ (right column).

Although we do not limit buffer availability here, as to be able to assess the maximum buffer requirement for each of the evaluated approaches, in real systems the buffer space is limited. Therefore, buffer size is preferred to be as small as possible. We show the required buffer size for the various policies in Figure 6.7 for the usr0 - mds0 pair. Results for the other three active-inactive pairs are not shown here in the interest of space but we remark that they are qualitatively the same as those reported in Figure 6.7. The x-axis in the graphs of Figure 6.7 is the performance degradation target (%) and y-axis is the required buffer space (in MB).

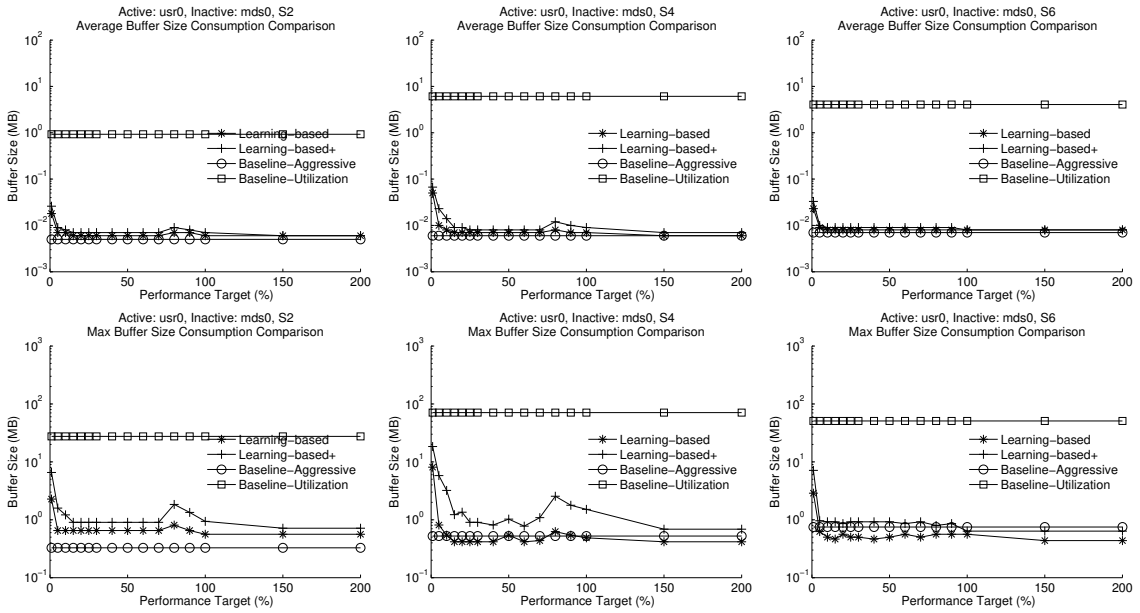



**Figure 6.7**: Buffer consumption comparison between different scheduling for usr0 - mds0 pair, both Standard and Aggressive Version of our framework are provided and also both Mean (first row) and Max (second row) Buffer consumption are provided. Three learning windows are considered: $Start = first\ day$ (left graph), $Start = third\ day$ (center graph), and $Start = fifth\ day$ (right graph).

The Utilization-based policy demands the largest buffer space since under that policy the replica WRITEs accumulate for a long time before being served. The Aggressive policy requires the least buffer space because it serves the incoming asynchronous tasks the fastest.

The buffer space under our scheduling policies depends on the performance target. The smaller the performance target the larger the buffer space. As expected, it converges to the Aggressive policy buffer requirements for higher performance targets. Note, that there are cases when our framework consumes less maximum buffer space than the Aggressive policy. This is because the Aggressive policy causes often the WRITEs to arrive in large batches at the inactive node, while our framework smooths out this bursty behavior for sending out an *almost* equal number of WRITEs every idle interval.

In conclusion, the results presented here support our claim that learning the characteristics of the lengths of idle periods, is crucial to the effectiveness of the two learning-based approaches. In addition, the workload in systems, as seen via captured traces from live systems, does not change drastically. As a result, learning over long periods of time results not only in a more resilient approach, but is also computationally inexpensive. Our framework introduces only a small overhead on the system for monitoring and storing the results. System gains are nevertheless orders of magnitude favorable regarding eventual consistency and user performance impact, which is critical for availability, reliability, and performance in scaled-out systems.

## 6.5 Summary

In this chapter, we presented a scheduling solution that facilitates the efficient synchronization of data distribution in the background for quick eventual data consistency, common in distributed storage systems, with user performance guarantees. The framework learns the idleness characteristics dynamically and determines how fast data can be sent or received without violating performance goals. Once such capabilities are shared among the nodes in the distributed system, each pair can synchronize the speed of sending and receiving. The result is orders of magnitude faster than common

practices without performance loss or large buffer requirements on the receiving end. Extensive experimentation via trace-driven simulation indicates that the learning process is robust and that the near past predicts reasonably the near future, with regard to idleness characteristics. The main findings of this chapter are also reported in [131].

# Chapter 7

# Agile Middleware for Scheduling

Scaling of web services is mostly achieved by deploying distributed systems in large data centers or even across them. Traditional computer systems, particularly those supporting enterprise applications, do not scale well, especially with regard to cost. To mitigate cost at a large scale, the industry is increasingly turning to commodity hardware and open-source software to accomplish large-scale services and computation. In order to scale-out and still operate effectively, the building blocks are off-the-shelf server nodes that operate mostly independently, while exchanging messages with other participating nodes [99]. This goal, exemplified by the *Open Compute* [1] initiative which has been widely adopted by the broader tech community, is to keep down the cost of systems that host big data and provide large scale analytics and other important web services.

One of the salient characteristics of the large distributed systems hosting a wide range of web services is supporting a wide range of features, including eventual consistency of data, data replication, garbage collection, and log data analysis, that run asynchronously in the background, at the level of the individual server node. The goal is to serve user workload as fast as possible and handle most of the management tasks *only when system resources are moderately utilized*. To illustrate the existence of opportunities for effective scheduling

of background tasks in real systems, we illustrate in Figure 7.1 the arrival intensity of requests to store new data or read existing ones in one of the nodes of a large scale web service over a three day period[1]. The strong daily pattern in the arrival intensity allows the system to schedule other important but less time sensitive tasks, e.g., garbage collection, during periods of low user activity, ensuring that these tasks do not affect the user quality of experience.

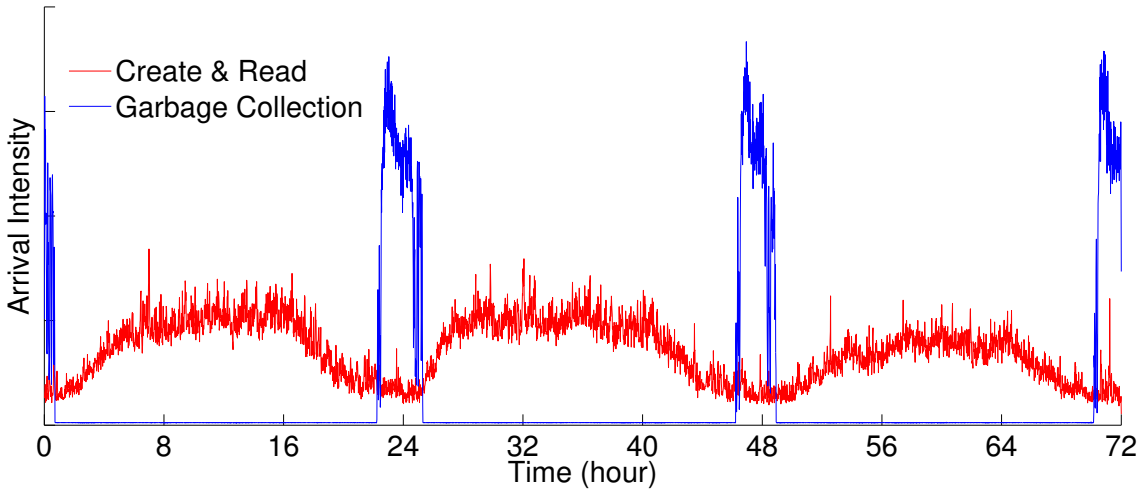



**Figure 7.1**: Overtime plot of arrival intensity for a large scale web service .

In this chapter, we focus on developing scheduling middleware that builds upon standard scheduling prioritization tools that are available in any Linux distribution, which often is the operating system of choice in the individual nodes of scaled-out systems. Standard distributions provide monolithic tools for priority scheduling but these are usually not reactive to changing workload conditions as those depicted in Figure 7.1. The scheduling middleware that we propose is based on effectively launching `nice` and `ionice` [80], the most common prioritization tools, with the appropriate priority levels that best match the existing system conditions. Furthermore, these priority levels are

[1]The data is provided by EMC. Due to confidential reasons, we cannot provide details of the web service.

*continuously adjusted* throughout the lifetime of the system to control relative priorities between the user (or foreground) traffic and the background system features in order to guarantee quality of service targets for foreground work while maximizing completion levels of background features.

While prioritization features have been proposed at the kernel level [134, 79] or at the application level [93, 77, 69], our focus is to provide middleware that is built upon standard tools that are available in any Linux distribution and most importantly operate in user-space. By utilizing `nice` and `ionice` as building blocks, we ensure that at fine time scales (i.e., microseconds) there is correct performance differentiation of the running processes based on their relative priorities. At coarse time scales (i.e., minutes), we control and manage these priorities via `renice` and other utilities to ensure that foreground performance is protected and background work is completed as efficiently as possible. The middleware that we propose is based on several standard monolithic scheduling policies (e.g., `nice` and `ionice`) but also on `smart`, a new (but still monolithic) mechanism that suspends background work briefly when foreground load spikes [123]. During the lifetime of the execution of the background work, the proposed middleware switches among the various basic policies as considered best fit.

## 7.1 Preliminaries

In this section, we first present an overview of the available off-the-shelf scheduling tools for priority scheduling and continue with an overview of resource demands across time for a typical workload to illustrate how background scheduling can become truly opportunistic. Finally, we show evidence that a single background scheduling policy cannot be effective under all circumstances, which further corroborates the need for agile middleware that continuously adjusts priority scheduling parameters in a transparent and autonomic way.

### 7.1.1 Prioritizing Background Work

Proprietary systems often have their own scheduling algorithms that allow them to maintain performance of user workload while other lower priority jobs are running in the background. The available off-the-shelf tools for priority scheduling in any Unix-based system are `nice`, which prioritizes access to the CPU resource, and `ionice`, which prioritizes access to the disk resource. While different distributions of Unix have different implementations of `nice` and `ionice`, they operate similarly: when enabled, they allow users to adjust the execution priority of processes.

A process that is invoked via `nice` can have a scheduling priority between -20 (the highest priority) and 19 (the lowest priority), as determined by a single parameter in the `nice` command. If the priority parameter of `nice` is set to zero or the process is invoked without the `nice` command then the process is run with the default (i.e., normal) priority. `nice` uses the priority parameter to determine the chunk of CPU time for a specific process, i.e., the higher the priority the larger the chunk of CPU time the process gets. The exact relation between the `nice` parameter and the amount of CPU time dedicated to a process are implementation dependent and vary between Unix/Linux distributions. The mechanism is generally simple to use and depends on fine-grained CPU consumption.

Similarly, `ionice` allows ranking the priority of a process from 0 to 3, where 3 is meant to designate a process that should be given IO resources only when the IO system is otherwise idle. A user may select to invoke both `nice` and `ionice`. We combine `nice 19` with `ionice 3` to give the lowest priority setting for both resources, which we label "allnice".

[123] introduces a scheduling policy named `smart` that focuses on adapting background job scheduling to foreground work with demands that are variable across time. The basic premise is to observe and effectively *predict* periods of low and high

utilization of the foreground work and launch or suspend background work based on monitoring system utilization levels. Suspending and resuming utilizes system resources by scheduling background work only when resources are lightly to moderately utilized by high priority processes. Additionally, it better isolates the foreground performance than `nice` or `allnice`, as well as doing so more consistently than either of the off-the-shelf options.

In this chapter, we develop a middleware that utilizes `smart` as well as the Linux priority scheduling tools `nice` and `ionice` and further enhances their capabilities by defining a set of policies which are automatically invoked within the same application run. The policies that are automatically selected by the middleware are the following:

- **nice 0**: the background work and the foreground application are running at the same priority for both CPU and IO resources.
- **allnice**: the background work runs at the lowest priority but it is never suspended, i.e., it is executed using `nice 19` with `ionice 3`.
- **smart+**: the background work is suspended briefly if load spikes using `smart` [123]. Once load returns back to its previous level, **allnice** is used here, unlike to the policy in [123].
- **FGonly**: the background work is suspended completely if high load for an extended period (i.e., at the hour-level granularity) is detected.

Additional policies can be added according to the specific system and application scenarios. In general, more scheduling policies give finer control, but may also result in more overhead. Intermediate policies with different `nice` parameters can be used, as well as more policies between the two extremes of **nice 0** and **FGonly**. For ease of presentation and with no loss of generality, we focus here on the four policies outlined above.

### 7.1.2 Scheduling Background Work: Perils and Opportunities

To illustrate the ample opportunities and perils of background scheduling, we show in Figure 7.2 the CPU utilization and response time as a function of elapsed time for TPC-W [3], a classic multi-tiered benchmark[2] that has significant variability across time in its CPU and memory demands [113]. The figure illustrates three scenarios: one with only 10 emulated browsers (top graph), one with 40 emulated browsers (middle graph), and one with 70 emulated browsers (bottom graph). The figure clearly shows many opportunities to schedule background jobs when there are only 10 emulated browsers (EBs): the CPU utilization is consistently low, with the exception of a few short time periods. Similarly, average response times are low across the entire experiment. The middle graph shows a different situation: with 40 EBs several bursts of short but high CPU activity that are usually clustered together, interspersed with periods of low CPU usage. The average user response time follows closely the CPU usage patterns. The bottom graph, where there are 70 EBs, shows longer periods of high utilization intermixed with periods of low utilization. The figure illustrates that there are plenty of opportunities to schedule background tasks when there are only 10 EBs, but higher load situations require more care, lest background work is scheduled during periods of high utilization and TPC-W performance is compromised.

### 7.1.3 Monolithic Background Scheduling

Figure 7.3 illustrates a first proof-of-concept of the relative advantages and disadvantages of scheduling background jobs using **nice 0**, **allnice**, **smart+**, and **FGonly**. The last policy gives the norm of the ideal response time. The figure illustrates the cumulative distribution histogram (CDH) of response times for TPC-W (first column) and the

[2]For the exact description of the experimental and measurement setting see Section 7.3.

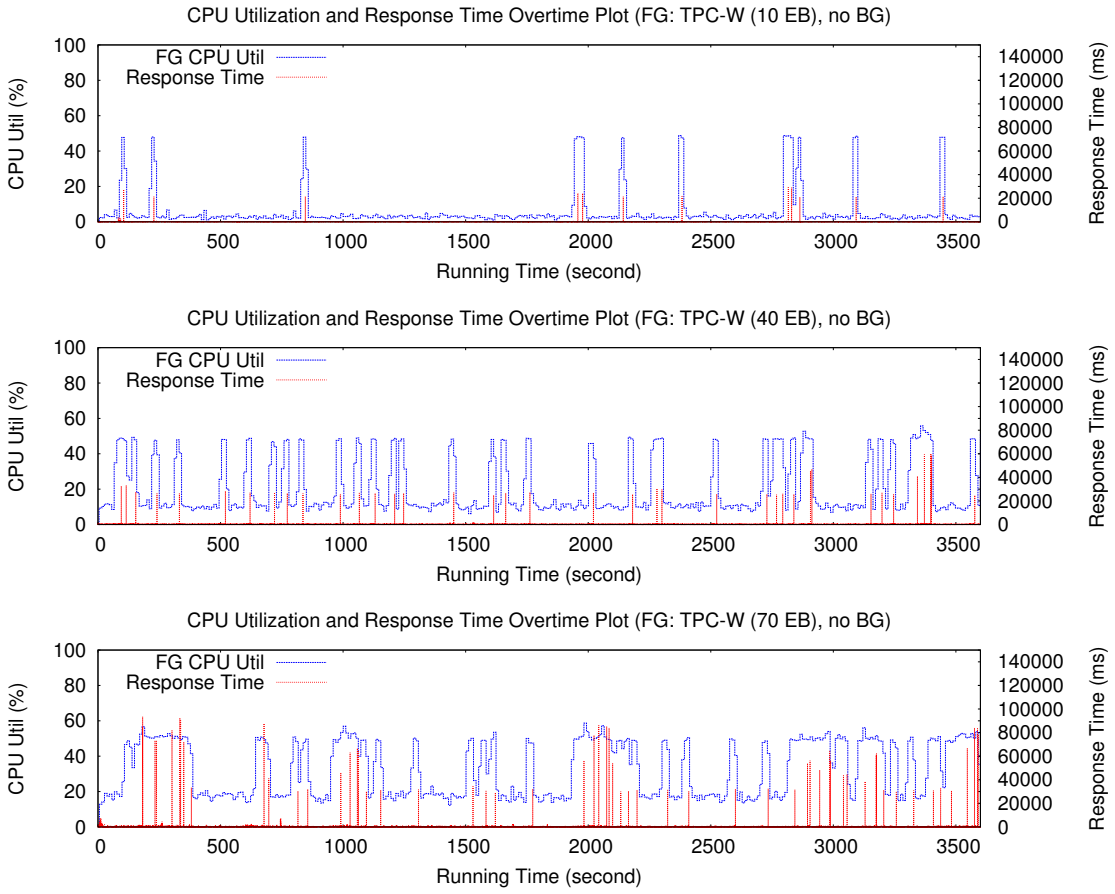



**Figure 7.2**: Overtime comparison of CPU utilization and average response times for 10, 40, and 70 emulated browsers. The duration of of this experiment is one hour.

throughput of background jobs (second column), presented as the number of completed iterations. The CDH figures clearly illustrate that the ranking of the various policies with respect to foreground performance are consistent for 10, 40, and 70 EBs and reflect how conservatively the background work is scheduled, ditto for the respective amount of completed background work. Yet, if there is a certain service level objective, e.g., if the 80th percentile of response time needs to be less than 600 ms, then background scheduling can be tuned to be more or less aggressive, such that it takes into account the

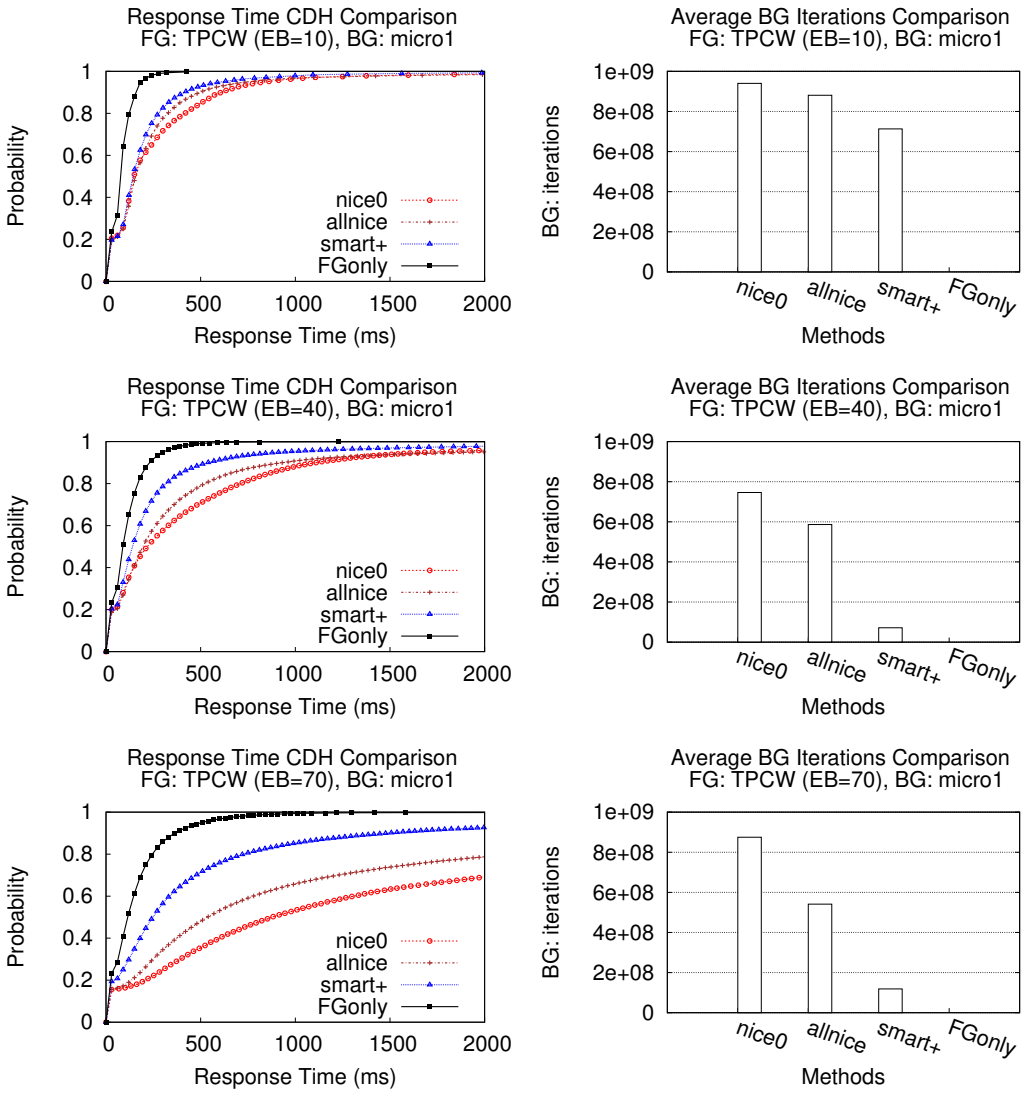



**Figure 7.3**: Performance results for TPC-W (CDH of response times) and background work completed (measured in number of iterations).

load in the system as expressed by the number of EBs is able to guarantee better background throughput. If the system operates with 10 EBs, then **nice 0** is sufficient for performance and maximizes the completed iterations but if the system operates with 40 EBs then **allnice** can offer performance guarantees while keeping iterations at a

maximum. When EBs rise to 70, then **any** background scheduling must be stopped. The figure clearly shows that effective background scheduling needs to be agile and hybrid, i.e., *continuously change its priority parameters* (e.g., switch from **nice 0** to **allnice** to **smart+** to **FGonly**) depending on the system operating conditions. In the following section we define how to develop and launch such middleware.

## 7.2 Methodology

The following summarizes the basic premise of the proposed middleware. If load from the high priority (or foreground) application is light, then running background tasks with the same priority should not violate the foreground performance target. As load from the foreground application increases, the priority of background work should decrease. If the system foreground load is high then the background work should be suspended until the high load period passes. We aim to consistently meet the system's foreground performance target while serving as much background work as possible. To achieve this goal, we learn the corresponding performance for different foreground load levels and monitor the latter to decide at what priority (if at all) to schedule background tasks.

### 7.2.1 Foreground Load Levels Relative to the Target

Load levels are defined relative to the foreground performance target, which, without loss of generality, we define as the percentile of requests whose response time is less than a target value (e.g., 80% of foreground requests are served in less than 600 ms). The system then is said to be under high load if it closely meets the target. If the target is violated, then the system is in overload. If the load results in better performance than the target, then we consider the load to be light to moderate and background work can be scheduled

without violating the target. The lighter the load, the higher the priority of the background work.

In Figure 7.4, we plot the cumulative distribution histogram (CDH) of TPC-W response times when load varies from light to heavy. In TPC-W the load is measured by the number of emulated browsers - EBs - (which corresponds to the number of network connections). In general, the load of a web service (which is the application type of interest given our focus on the individual nodes of a scaled-out system) can be measured similarly, although other metrics of load can be trivially defined and applied to our methodology.

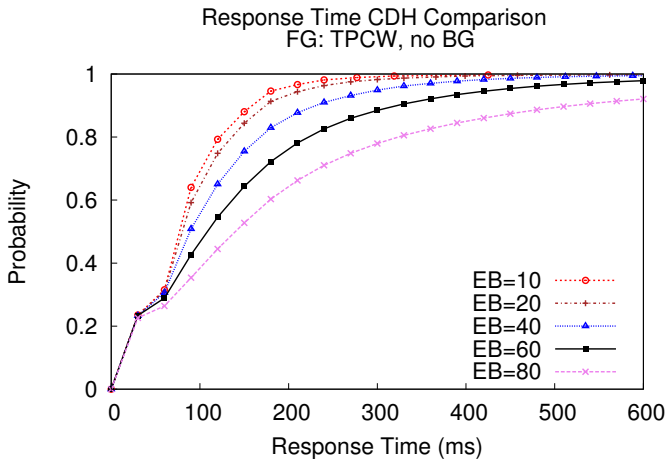



**Figure 7.4**: CDH of response times for different system load (EBs).

For a given target, we define the "high load" level based on the measurements captured in Figure 7.4. For example, the target of 80th percentile being at most 600 ms would result in "high-load" being 80 EBs, because it is the highest load level meeting this target. If the foreground performance target for the 80th percentile is to be at most 300 ms, then 70 EBs would be the "high load" in the system, while a load of 80 EBs would put the system into overload. For a target of 300 ms the system should serve background work alongside foreground only if the foreground load is less than 70 EBs.

The measurement data present in Figure 7.4 can be collected off-line in a test environment or it can be collected in the system as it comes on-line and kept up-to-date over time. Collecting such data should be possible with minimal effort, since the systems we are focusing on are provided by the general Linux distribution with an array of monitoring and logging tools.

### 7.2.2 Priority Policy Decision

The proposed middleware requires identifying the relation between current load and performance target for the foreground application (as described in Subsection 7.2.1), in order to identify the availability of resources to execute background tasks and set correctly the relative priority of the background tasks. As a result, similarly to the learning described in Subsection 7.2.1, we learn the foreground performance with a single representative background task (see more details in Section 7.3) treated with one of the four priority policies defined in Subsection 7.1.1. We again generate the distribution of foreground response times. For example, for each of the evaluated TPC-W loads and **nice 0**, we generate the same set of response time distributions as captured in Figure 7.4.

We learn the foreground performance behavior through a number of representative cases. The background tasks that we use for *training* run concurrently with TPC-W are described in Section 7.3 and can be tuned to demand more or less CPU and memory resources. Specifically, during learning, we measure the system under the foreground application plus heavy background load, i.e., demanding more than 100% CPU utilization and memory, so that the impact on foreground performance would hold for *any* background task that may be served in the system. Because of these choices during the learning period, we consider the measurements conducted as a baseline that can be used reliably to guide our decision on the priority policy for a given foreground load (and its

performance target) and *any* background task. Results shown in Section 7.3 support these choices. Our reliance on fine-grained priority scheduling done by `nice` and `ionice` adds to the robustness of our decisions.

To help visualize the data we collect during the learning process, as well as to clarify our decision-making process with regard to dynamically changing background priorities, we plot the distribution of response times for different load levels and priority policies as stacked bars, see Figure 7.5. The x-axis of Figure 7.5 consists of all possible (*system load, priority policy*) pairs. The y-axis in Figure 7.5 represents the response time percentiles of the foreground requests, measured in milliseconds. The different colors used in each bar mark a specific, i.e., 50, 70, 80, and 90th, percentile of the response time distribution for a specific pair.

The data structure visualized in Figure 7.5 is paired with the foreground performance target which we illustrate with a horizontal line that represents the expected response time percentile. In this figure we have marked performance targets for the 80th percentile of response times to be equal to or less than 600ms. In this case, more than 70 EBs is considered "high load", since the target is met under the **FG-only** policy only. As the foreground load decreases, the 80th percentile of foreground response times is met also by several priority policies that serve background work. For example, for 50 EBs, the 80th percentile of foreground response time is less than 600ms under the **smart+** policy, while for 40 EBs the target is still met if we schedule background work via **allnice**. At 30 EBs or less that background work can be scheduled with the same priority as foreground work via **nice-0** without violating the target. The benefit of increasing the priority of background work (from **FG-only** to **nice-0**) as foreground load reduces, helps to serve more background work while ensuring that the foreground performance target is met. The decision map in Figure 7.5 is used by the scheduling middleware that we propose here as

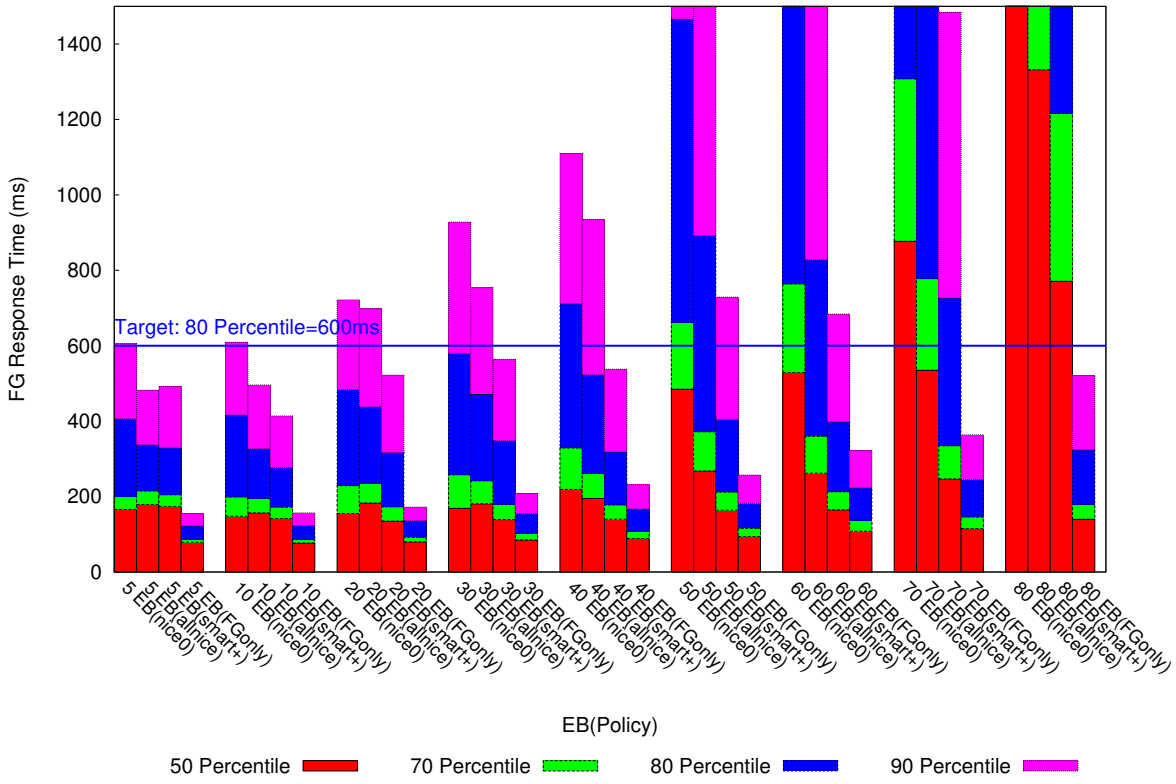



**Figure 7.5**: Decision map.

the decision making engine to automatically adjust priorities as foreground load conditions change over time.

A schematic view of information interchange in our priority scheduling hybrid middleware is provided in Figure 7.6. We reiterate that the learning is done in such a way that it can either be complete off-line or on-line as the system comes up and can be updated overtime with more observations. As we provide more details on our prototype in Section 7.3, we also highlight the standard Linux utilities that we use.

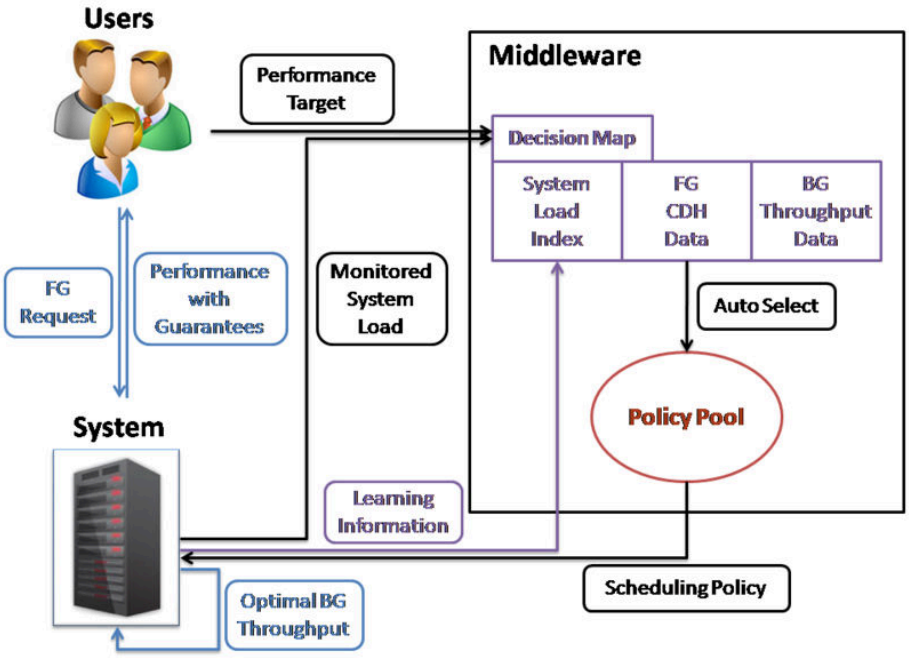



**Figure 7.6**: Schematic view of the middleware scheme.

## 7.3 Experimental Evaluation

All experiments presented here are conducted on a Dell Precision WorkStation with Intel Pentium Dual Core 2.4GHz processor, 1GB memory, Seagate 7.2K SATA hard drives, running openSUSE 11.4 (64 bit). As foreground workload, we use a Java implementation of the TPC-W benchmark. For background work we use our own micro benchmark in order to control the experiments and ensure representative data with regard to learning.

We consider TPC-W to be a challenging workload, because it is characterized by variability in its resource demands across time [15], as also shown in Figure 7.2. TPC-W is a web server and database performance benchmark [3] and in our prototype we use to drive the system the java distribution in [12]. We use tomcat as the application server and mysql as the database server. TPC-W provides a large number of parameters. We

use the browsing mix on a 100000 items in the database.

We develop our own micro benchmark to use as background work, which is built upon the Isolation Benchmark Suite [66]. This micro-benchmark performs multiplications in a tight loop which is embedded in a larger one containing array initializations and file writes. The micro benchmark allows to experiment with a broader range of CPU, memory, and IO background demands. In the results reported in this section, we have used three different variations of the micro benchmark. In each of the three scenarios, four instances of the micro-benchmark are run concurrently, each consuming approximately 20% of memory capacity and some IO traffic. The micro benchmarks parameters are scaled to change the CPU demand as shown below:

- micro1: consumes approximately 100% of the system's total CPU resource.
- micro2: consumes approximately 45% of the system's total CPU resource.
- micro3: consumes approximately 160% of the system's total CPU resource (i.e., uses almost both cores).

In order to provide a simple and easily portable implementation, our monitoring and scheduling algorithms are implemented entirely in user space, making use of readily available Linux commands (e.g., `pidstat` and `kill`). For monitoring, we launch a shell script to call `pidstat` every 10 seconds and extract the CPU utilization for all running processes, classifying the results into three main categories: foreground (TPC-W related) processes, background (micro benchmark related) processes, and other system processes. The coarse granularity of these intervals differs from the fine-grained handling generally used in real-time scheduling algorithms in the literature, but we emphasize that we delegate the fine-grained decisions to `nice` and `ionice`.

**Table 7.1**: Scheduling Computed from Decision Map for different Scenarios.

| **Scenario 1:** Target: $Prt(80\%) <= 600ms$ | | | **Scenario 2:** Target: $Prt(90\%) <= 650ms$ | | |
|---|---|---|---|---|---|
| System Load | EB Range | Policy Selection | System Load | EB Range | Policy Selection |
| low | 0-30 | nice0 | low | 0-10 | nice0 |
| medium | 31-40 | allnice | medium | 11-19 | allnice |
| high | 41-60 | smart+ | high | 20-40 | smart+ |
| extreme | 61-80 | FGonly | extreme | 40-80 | FGonly |

To control the execution of background work, we use the STOP and CONT signals and pass them to process by the `kill` command to "pause" and "resume" the background tasks. The process is suspended by being starved of resources, but because it is not actually killed, it can be immediately resumed from where it is paused. We stress all these native system tools make our method easy to deploy and with low overhead.

### 7.3.1 Results

Initially, we evaluate our hybrid scheduling middleware by running the TPC-W as the foreground task and four micros for a total of 100% additional CPU utilization (i.e., variant *micro1*) as background tasks for different foreground performance targets. We choose two scenarios to present here. Scenario 1's performance target is the 80th percentile to be equal or smaller than 600 ms and Scenario 2's performance target is the 90th percentile to be equal or smaller than 650 ms. Based on the decision map of Figure 7.5 the scheduling strategy is devised and summarized in Table 7.1. The policy transition parameters in Table 7.1 are obtained from our off-line learning. As robustness of this learning approach is key to our evaluation, we run all our tests for 14 hours, during which the foreground load varies from 0 to 80 EBs, including load levels that were not used in learning. For those cases, the decision is done based on the next higher load tested.

We evaluate our hybrid priority scheduling middleware by comparing it with the monolithic scheduling methods for the same scenario. Our experiments are run 14 hours

long to ensure that enough instances of foreground workload changes occur in order to demonstrate the robustness of our hybrid middleware. We plot the results for Scenario 1 and Scenario 2 in Figures 7.7 and 7.8, respectively. In each figure, we plot the system load measured by both CPU utilization and number of EBs (see top plot). As load varies over time, so do the opportunities to schedule background work. For each hour, we report in the top plot of Figures 7.7 and 7.8 along the x-axis the decision of our hybrid middleware based on the parameters devised in Table 7.1 and monitoring of foreground load levels. Figures 7.7 and 7.8 also plot the CDH and CCDF of the response times for the different monolithic policies and our hybrid middleware (see the second row) of both figures.

As expected, the **FG-only** and **nice-0** achieve the best and worst foreground performance, respectively, because **FG-only** suspends background work while **nice-0** treats foreground and background work the same. The other two policies, **allnice** and **smart+** maintain better foreground performance at the cost of background throughput (bottom right plot in Figures 7.7 and 7.8). The bottom left plot in Figures 7.7 and 7.8 shows how hybrid meets the foreground performance target (see vertical line) while achieving highest background throughout among all policies that do meet the performance target (**FG-only**, **smart+**, and **hybrid** for Scenario 1 and **FG-only** and **hybrid** for Scenario 2, see the two right plot in the bottom rows of both figures).

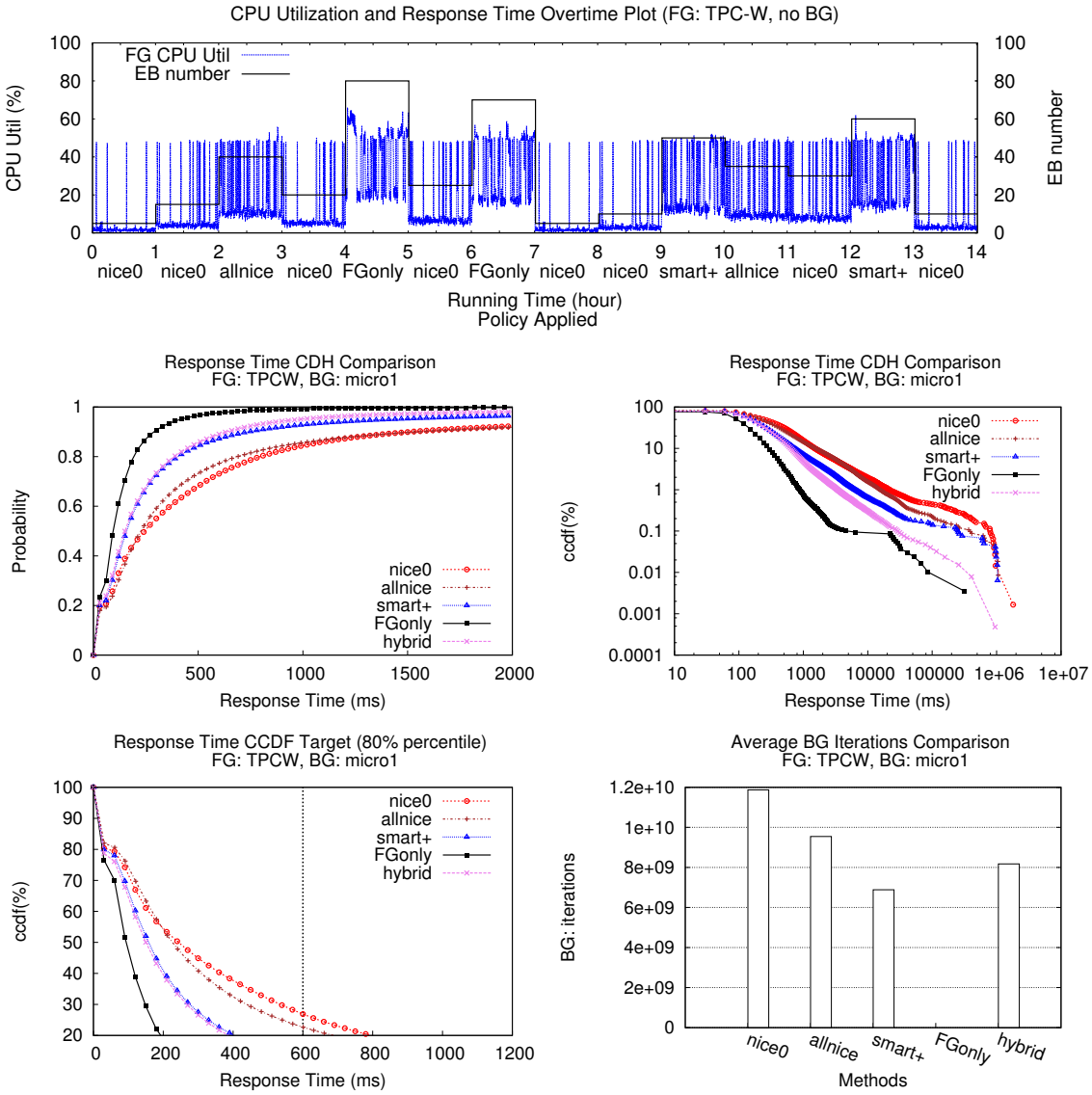



**Figure 7.7**: Scenario 1 - BG: CPU total demand: 100% .

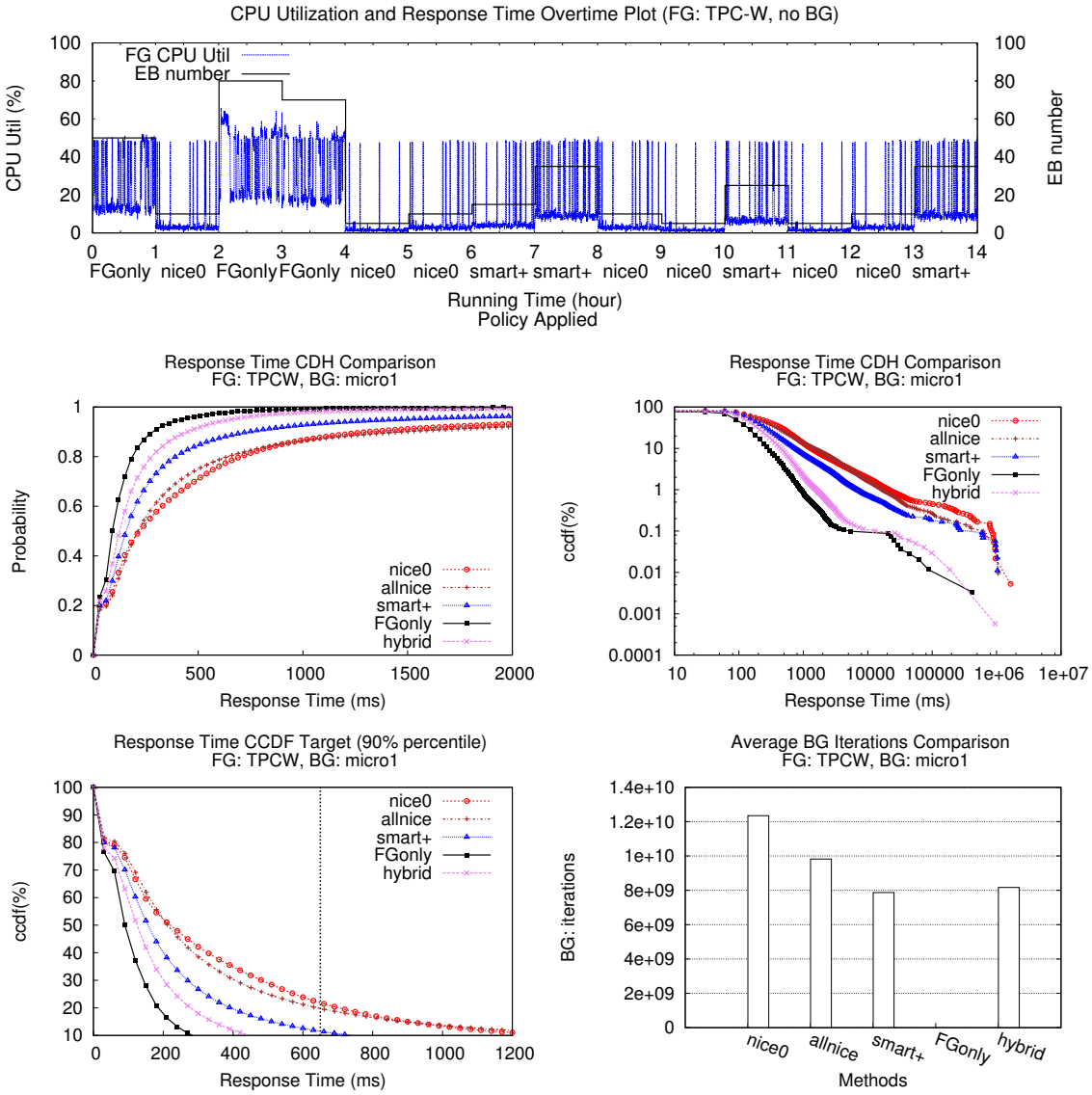



**Figure 7.8**: Scenario 2 - BG: CPU total demand: 100% .

We also evaluate the resilience of our hybrid middleware to the learning methodology. Recall that learning is done with background tasks adding up to 100% CPU utilization. We run the same 14 hours test for Scenario 1 and Scenario 2, but now the background work follows the variant micro2 (45% total CPU demand) and micro3 (total 160% CPU

demand). For micro2 background workload, we show the respective results for Scenario 1 and Scenario 2 in Figures 7.9 and 7.10. For micro3 as background workload, we show the respective results for Scenario 1 and Scenario 2 in Figure 7.11 and Figure 7.12, respectively. The decisions on policy transitions are done according to Table 7.1 for both cases.

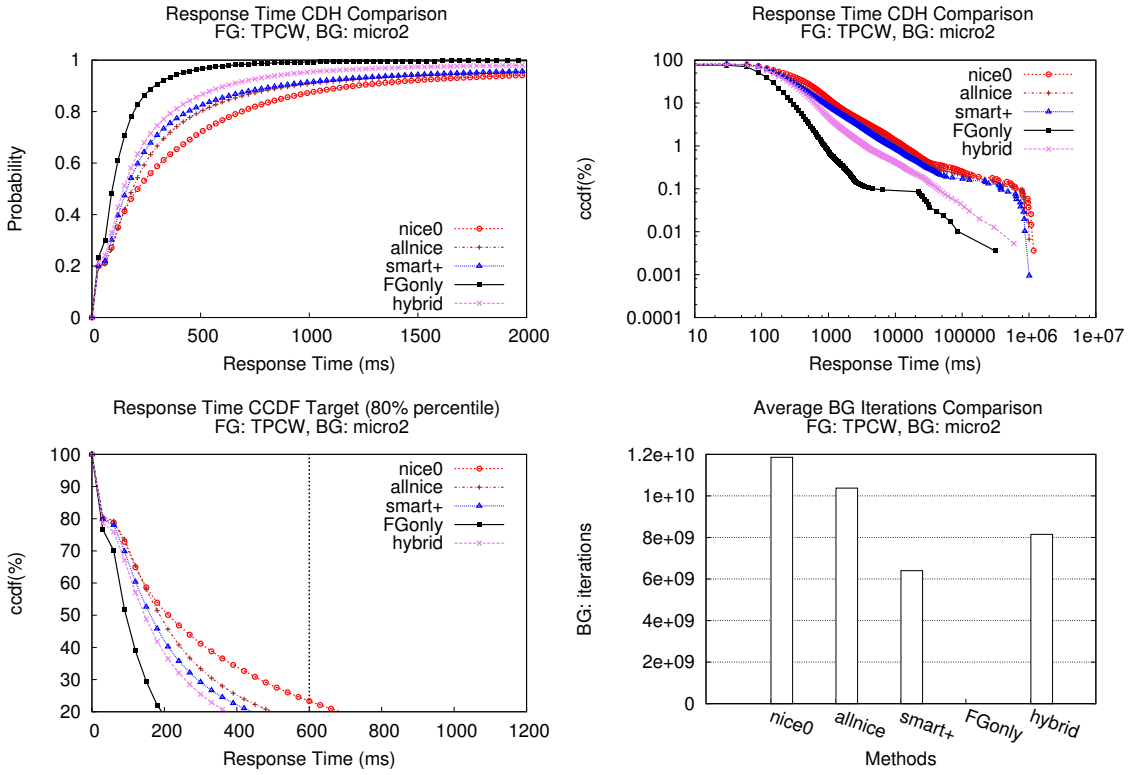



**Figure 7.9**: Scenario 1 - BG: CPU total demand: 45% .

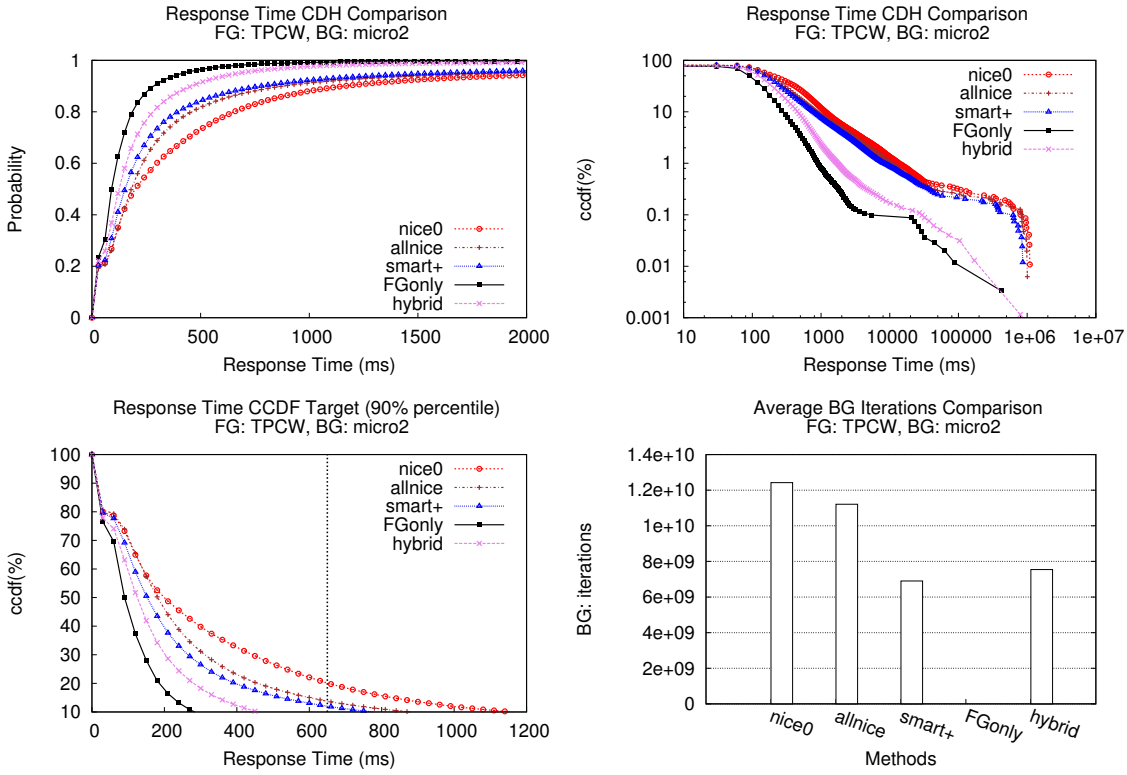



**Figure 7.10**: Scenario 2 - BG: CPU total demand: 45% .

These experiments confirm that the hybrid middleware meets the foreground performance target under all these different combinations of foreground/background work and that the learning approach is effective. The reason behind this is that the low priority workload is only scheduled during low system load periods, where the foreground impact is well controlled. In addition, we are always conservative by approximating the untrained foreground intensity to the higher nearest intensity entry in the decision map. Another critical aspect that contributes to the resiliency of our hybrid middleware is the fact that the scheduling policies used in our scheme are based on `nice` and `ionice`, which control priorities at very fine granularities.

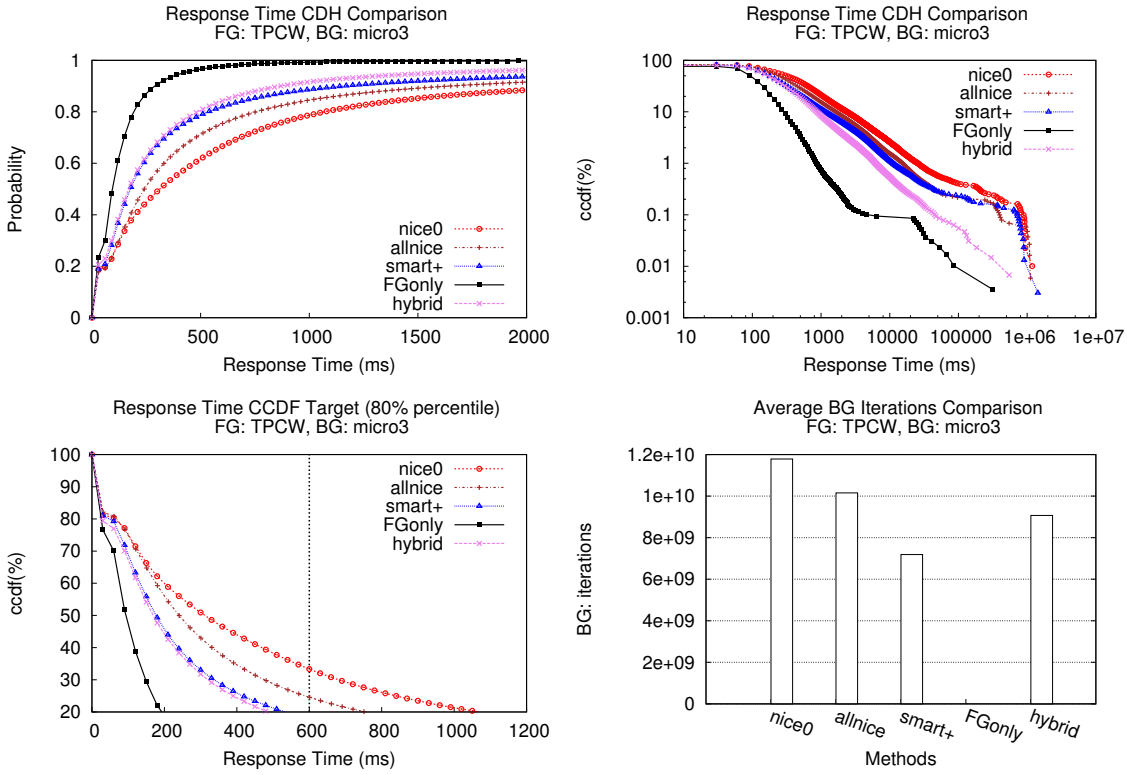



**Figure 7.11**: Scenario 1 - BG: CPU total demand: 160% .

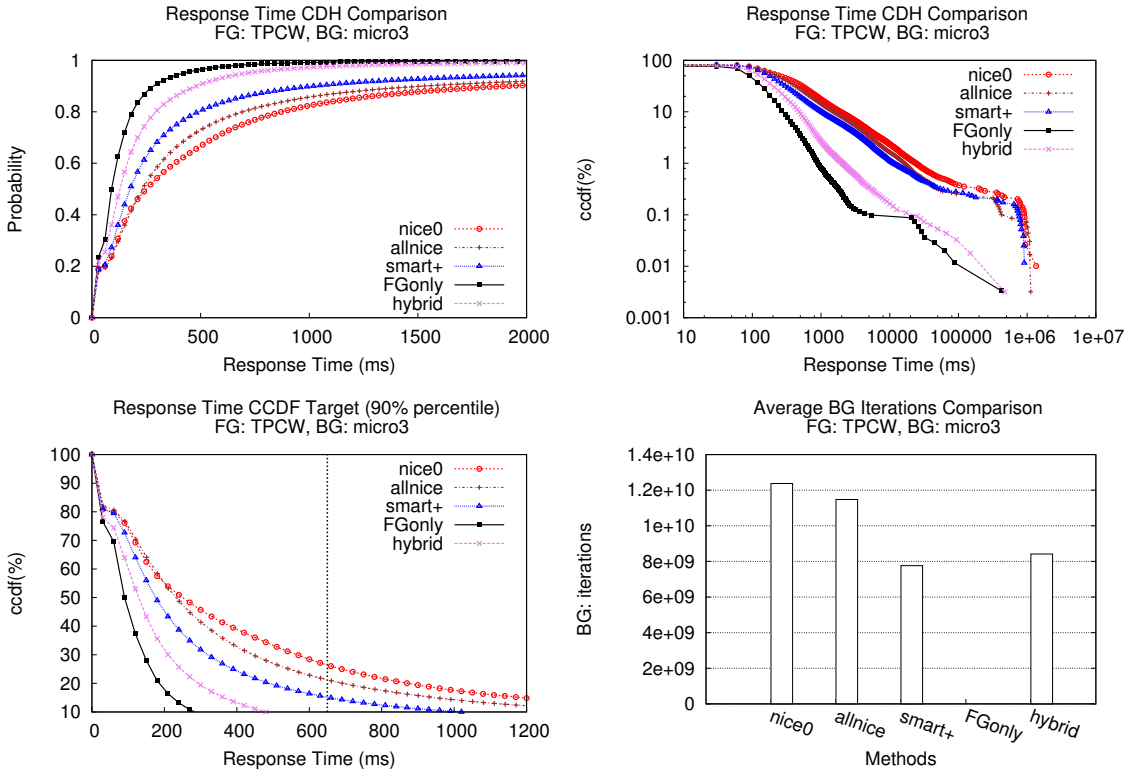



**Figure 7.12**: Scenario 2 - BG: CPU total demand: 160% .

## 7.4 Summary

In this chapter, we proposed a middleware scheme that remedies the shortcomings of monolithic background scheduling and provides strong performance guarantees on foreground work. Our middleware scheme learns the foreground resource requirements and stores such information in a compact way, in the form of a cumulative data histogram. This learning allows the scheme to determine the appropriate scheduling policy based on pre-specified performance targets and current system load levels. The scheduling middleware is built above standard system tools, ensuring that is portable, with low overhead, and that can be deployed easily at the node level within large

scaled-out systems. Detailed experimental results verify its effectiveness and robustness. The main findings of this chapter are also reported in [124].

# Chapter 8

# DyScale Scheduler

To offer diverse computing and performance capabilities, the emergent modern system on a chip (SoC) may include heterogeneous cores that execute the same instruction set while exhibiting different power and performance characteristics. The SoC design is often driven by a power budget that limits the number (and type) of cores that can be put on a chip. The power constraints force designers to exploit a variety of choices within the same power envelope and to analyze decision trade-offs, e.g., to choose between either many slow, low-power cores, or fewer faster, power hungry cores, or to select a combination of them, see Figure 8.1. A number of interesting choices may exist, but once the SoC design is chosen, it defines the configuration of the produced chip, where the number and the type of cores on the chip is fixed and cannot be changed.

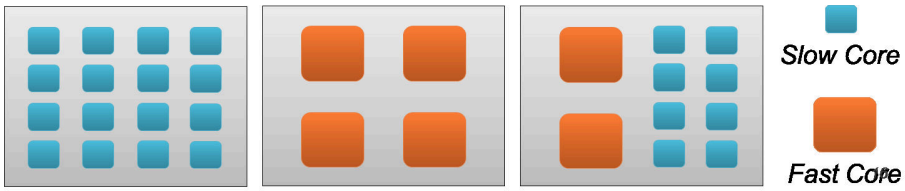



**Figure 8.1**: Different choices in the processor design.

Intuitively, an application that needs to support higher throughput and that is capable of partitioning and distributing its workload across many cores favors a processor with a higher number of slow cores. However, the latency of a time-sensitive application depends on the speed of its sequential components and should benefit from a processor with faster cores to expedite the sequential parts of the computation. This is why a time-sensitive application may favor a SoC processor with faster cores, even if these are few. A SoC design with heterogeneous cores might offer the best of both worlds by allowing to benefit from heterogeneous processing capabilities.

MapReduce and its open source implementation Hadoop offer a scalable and fault-tolerant framework for processing large data sets. MapReduce jobs are automatically parallelized, distributed, and executed on a large cluster of commodity machines. Hadoop was originally designed for batch-oriented processing of large production jobs. These applications belong to a class of so-called scale-out applications, i.e., their completion time can be improved by using a larger amount of resources. For example, Hadoop users apply a simple rule of thumb [115]: processing a large MapReduce job on a double size Hadoop cluster can reduce job completion in half. This rule is applicable to jobs that need to process large datasets and that consist of a large number of tasks. Processing these tasks on a larger number of nodes (slots) reduces job completion time. Efficient processing of such jobs is "throughput-oriented" and can be significantly improved with additional "scale-out" resources.

When multiple users share the same Hadoop cluster, there are many interactive ad-hoc queries and small MapReduce jobs that are completion-time sensitive. In addition, a growing number of MapReduce applications (e.g., personalized advertising, sentiment analysis, spam detection) are deadline-driven, hence they require completion time guarantees. To improve the execution time of small MapReduce jobs, one cannot

use the "scale-out" approach, but could benefit using a "scale-up" approach, where tasks execute on "faster" resources.

A typical perception of a MapReduce processing pipeline is that it is disk-bound (for small and medium Hadoop clusters) and that it can become network-bound on larger Hadoop clusters. Intuitively, it is unclear whether a typical MapReduce application under normal circumstances can benefit from processors with faster cores. To answer this question we perform experiments on a diverse set of MapReduce applications in a Hadoop cluster that employs the latest Intel Xeon quad-core processor (it offers a set of controllable CPU frequencies varying from 1.6 Ghz to 3.3 Ghz, with each core frequency set separately). While the achievable speedup across different jobs varies, many jobs achieved speedup of 1.6-2.1 thanks to the faster processors. Such heterogeneous multi-core processors become an interesting design choice for supporting different performance objectives of MapReduce jobs.

Here, we design and evaluate DyScale, a new Hadoop scheduler that exploits capabilities offered by heterogeneous cores for achieving a variety of performance objectives. These heterogeneous cores are used for creating different virtual resource pools, each based on a distinct core type. These virtual pools consist of resources of distinct virtual Hadoop clusters that operate over the same datasets and that can share their resources if needed. Resource pools can be exploited for multi-class job scheduling. We describe new mechanisms for enabling "slow" slots (running on slow cores) and "fast" slots (running on fast cores) in Hadoop and creating the corresponding virtual clusters. Extensive simulation experiments demonstrate the efficiency and robustness of the proposed framework. Within the same power budget, DyScale operating on heterogeneous multi-core processors provides significant performance improvement for small, interactive jobs comparing to using homogeneous processors with (many) slow

cores. DyScale can reduce the average completion time of time-sensitive interactive jobs by more than 40%. At the same time, DyScale maintains good performance for large batch jobs compared to using a homogeneous fast core design (with fewer cores). The considered heterogeneous configurations can reduce the completion time of batch jobs up to 40%.

There is a list of interesting *opportunities* for improving MapReduce processing offered by heterogeneous processor design. First of all, both *fast* and *slow* Hadoop slots have the same access to the data stored in the underlying Hadoop Distributed File System. This eliminates data locality issues that could make heterogeneous Hadoop clusters comprised of *fast* and *slow* servers[1] being inefficient [7]. However, when each node consists of heterogeneous core processors, then any dataset (or any job) can be processed by either *fast* or *slow* virtual resource pools, or their combination. Second, the possibility of task (job) migration between slow and fast cores enables enhancing performance guarantees and more efficient resource usage compared to static frameworks without the process migration feature. Among the challenges are *i)* the implementation of new mechanisms in support of dynamic resource allocation, including migration and virtual resource pools, *ii)* the support of accurate job profiling, especially, when a job/task is executed on a mix of fast and slow slots, *iii)* the analysis of per job performance trade-offs for making the right optimization decisions, and *iv)* increased management complexity.

The rest of this chapter is organized as follows. Section 9.1.2 provides background of MapReduce processing. Section 8.2 gives a motivating example and discusses the advantages of the scale-out and scale-up approaches. Section 8.3 introduces the DyScale framework. Section 8.4 evaluates DyScale using measurements on actual machines and

[1]Note that the concept of heterogeneous cores within a single processor is very different from heterogeneous servers, where different servers have different capacity and performance. Hadoop clusters that include heterogeneous servers do have a variety of problems with traditional data placement and related unbalanced data processing as has been shown in [7].

via simulation on a diverse variety of settings. We summarize in Section 8.5.

## 8.1 Job Scheduling in MapReduce Processing

Job scheduling in Hadoop is performed by a master node called JobTracker, which manages a number of worker nodes. Each worker node is configured with a fixed number of map and reduce slots, and these slots are managed by the local TaskTracker. The TaskTracker periodically sends heartbeats to the master JobTracker via TCP handshakes. The heartbeats contain information such as current status and the available slots. The JobTracker decides the next job to execute based on the reported information and according to a scheduling policy. Popular job schedulers include FIFO, Hadoop Fair scheduler (HFS) [132], and Capacity scheduler [4]. FIFO is the default and schedules MapReduce jobs according to their submission order. This policy is not efficient for small jobs if large jobs are also present. The Hadoop Fair Scheduler aims to solve this problem. It allocates on average the same amount of resources to every job over time so that small jobs do not suffer from delay penalties when scheduled after large jobs and large jobs do not starve. The Capacity scheduler offers similar features as the HFS but has a different design philosophy. It allows users to define different queues for different types of jobs and to configure a percentage of share of the total resources for each queue in order to avoid FIFO's shortcomings.

The assignment of tasks to slots is done in a greedy way: assign a task from the selected job $J$ immediately whenever a worker reports to have a free slot. At the same time, a data locality consideration is taken into account: if there is a choice of available slots in the system to be allocated to job $J$, then the slots that have data chunks of job $J$ locally available for processing are given priority [132]. If the number of tasks belonging to a MapReduce job is greater than the total number of processing slots, then the task

assignment takes multiple rounds, which are called *waves*.

The Hadoop implementation includes *counters* for recording timing information such as start and finish timestamps of the tasks, or the number of bytes read and written by each task. These counters are sent by the worker nodes to the master node periodically with each heartbeat and are written to logs. Counters help profile the job performance and provide important information for designing new schedulers. We utilize the extended set of counters from [135] in DyScale.

## 8.2 Motivating Example: Scale-out vs. Scale-up Approaches

Workload characterization based on Facebook and Yahoo jobs [132, 86] shows that a MapReduce workload typically can be described as a collection of "elephants" and "mice". Table 8.1 shows the number of map and reduce tasks and the percentage of these jobs in the Facebook workload [132]. In the table, different jobs are grouped into different bins based on their size in terms of number of map and reduce tasks. Most jobs are quite small (mice): 88% of jobs have less than 200 map tasks, but there is a small percentage of the large jobs (elephants) with up to thousands of map tasks and hundreds of reduce tasks.

Pig jobs [38] present a different case of a MapReduce workload with large and small jobs. Pig offers a high level SQL-like abstraction on top of Hadoop. Pig queries are composed of MapReduce workflows. During the earlier stages of the workflows, the datasets to be processed are usually large, and therefore, they correspond to "large" job processing. After some operations such as "select" and "aggregate", the amount of data for processing may be significantly reduced, and the jobs in the second half of the workflow can be considered

| Bin | Map Tasks | Reduce Tasks | # % Jobs |
|---|---|---|---|
| 1 | 1 | NA | 38% |
| 2 | 2 | NA | 16% |
| 3 | 10 | 3 | 14% |
| 4 | 50 | NA | 8% |
| 5 | 100 | NA | 6% |
| 6 | 200 | 50 | 6% |
| 7 | 400 | NA | 4% |
| 8 | 800 | 180 | 4% |
| 9 | 2400 | 360 | 2% |
| 10 | 4800 | NA | 2% |

**Table 8.1**: Job description for each bin in Facebook workload.

"small".

Different types of jobs may favor different design choices. For example, large jobs may benefit from processors with many slow cores to obtain better throughput, i.e., to execute as many tasks in parallel as possible in order to achieve better job completion time. Small jobs may benefit by processors with fewer fast cores for speeding-up their tasks and for getting an improved job completion time. Therefore, heterogeneous multi-core processors may offer an interesting design point because they bring a potential opportunity to achieve a win-win situation for both types of MapReduce jobs.

MapReduce applications are scale-out by design, which means that the completion time is improved when more slots are allocated to a job, see Figure 8.2. The scale-out limit depends on the total number of slots in the system and the job parallelism. MapReduce applications may also benefit from "scale-up", e.g., a job may complete faster on faster cores. The interesting question is how different MapReduce jobs may benefit by both scale-out and scale-up. To understand the possible trade-offs, consider the following example.

**Motivating Example**. Assume that we have a Hadoop cluster with 100 nodes. Under the same power budget each node can have either **two fast** cores or **six slow** cores. We

configure the Hadoop cluster with one map slot and one reduce slot per core. The slots that are executed on the fast or slow cores are called *fast* and *slow slots* respectively. Therefore, the system can have either **200 fast** map (reduce) slots or **600 slow** map (reduce) slots in total. Let us consider the following two jobs:

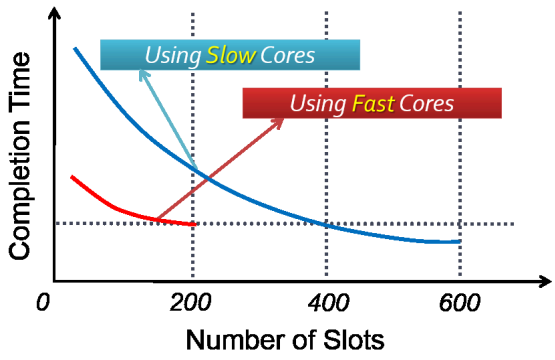
- *Job1* with 4800 map tasks (i.e., similar to jobs in the 10th group of Table 8.1),
- *Job2* with 50 map tasks (i.e., similar to jobs in the 4th group of Table 8.1).

Assume a map task of *Job1* and *Job2* requires $T$ amout of time to finish with a *fast* slot, and $2 \cdot T$ to execute with a *slow* slot. Let us look at the job completion time of *Job1* and *Job2* as a function of an increased number of fast or slow slots that are allocated to the job.

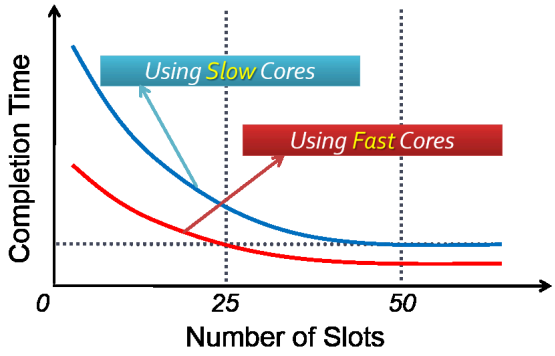
The scenarios that reflect possible executions of *Job1* and *Job2* are shown in Figure 8.2(a) and 8.2(b), respectively. The graphs in Figure 8.2 are drawn based on calculations using the analytic model for estimating the completion time of a single job [105]. Completion times are graphed as a function of the number of slots allocated to a job. The figure illustrates that both jobs achieve lower completion times with a higher number of allocated slots.

When a large *Job1* is executed with fast slots, and *all 200 fast* slots are allocated to the job, then its completion time is: $4800 \cdot T/200 = 24 \cdot T$, i.e., it takes 24 rounds of $T$ time units. The best job completion time is achieved when using *all 600 slow* slots. In this case, *Job1* finishes in $4800 \cdot 2 \cdot T/600 = 16 \cdot T$, i.e., it takes 8 rounds of $2 \cdot T$ time units. The job completion time with slow slots is 30% better than with the fast slots. Thus, using a larger number of slow slots leads to a faster completion time.

The a small *Job2* shown in Figure 8.2(b) cannot take advantage of more than 50 slots, either slow or fast, because it only has 50 tasks. In this case, when *Job2* is executed with fast slots, it takes $50 \cdot T/50 = T$ time units to complete. If executed with slow slots, the




(a) Large *Job1* Processing.




(b) Small *Job2* Processing.

**Figure 8.2**: Processing MapReduce jobs *Job1* and *Job2* with *slow* or *fast* slots available in the cluster.

completion time is $50 \cdot 2 \cdot T/50 = 2 \cdot T$ units, which is twice longer than using fast slots. The small jobs are usually interactive and thus are time sensitive. For such jobs, 50% of a completion time improvement represents a significant performance opportunity.

From the example, it is clear that large batch jobs (similar to *Job1*) can achieve better performance when processed by a larger number of slow slots, while the smaller jobs (like *Job2*) can execute faster on a smaller number of fast slots. Such diverse demands in MapReduce jobs indicate that traditional homogeneous multi-core processors may not provide the best performance and power trade-offs. This motivates a new Hadoop

scheduler design with a tailored slot assignment to different jobs based on their scale-out and scale-up features.

## 8.3 DyScale Framework

We propose a new Hadoop scheduling framework, called DyScale, for efficient job scheduling on the heterogeneous multi-core processors. First, we describe the DyScale scheduler that enables creating statically configured, dedicated virtual resource pools based on different types of available cores. Then, we present the enhanced version of DyScale that allows the shared use of spare resources among existing virtual resource pools.

### 8.3.1 Problem Definition

The number of *fast* and *slow* cores is SoC design specific and workload dependent. Here, we focus on a given heterogeneous multi-core processor in each server node and on the problem of taking advantage of these heterogeneous capabilities, especially compared to using homogenous multi-core processors with the same power budget. Our goal is twofold: 1) design a framework for creating virtual Hadoop clusters with different processing capabilities (i.e., clusters with *fast* and *slow* slots); and 2) offer a new scheduler to support jobs with different performance objectives for utilizing the created virtual clusters and sharing their spare resources. The problem definition is as follows:

**Input:**

- $C$: cluster size (number of machines)
- $N_f$: number of fast cores on each machine
- $N_s$: number of slow cores on each machine
- $S$: job size distribution
- $A$: job arrival process

**Output:**

*Sched:* schedule of Map/Reduce task placement

**Objective:**

minimize$_{\{Sched\}}$ Job Completion Time ( *Sched* )

A natural first question is why a new Hadoop scheduler is a necessity and why the default Hadoop scheduler can not work well. To answer this question, we show the performance comparison under the same power budget of using the default Hadoop scheduler on heterogenous and homogenous multi-core processors respectively, and also our DyScale scheduler with the same heterogenous multi-core processors, see Figure 8.3. The details of the experiment configurations are given in Section 8.4.3. The important message from Figure 8.3 is that the default Hadoop scheduler cannot use well the heterogenous multi-core processors and may even perform worse than when using it on a cluster with homogenous multi-core processors with the same power budget due to the random use of fast and slow cores.

### 8.3.2 Dedicated Virtual Resource Pools for Different Job Queues

DyScale offers the ability to schedule jobs based on performance objectives and resource preferences. For example, a user can submit small, time-sensitive jobs to the *Interactive Job Queue* to be executed by fast cores and large, throughput-oriented jobs to the *Batch Job Queue* for processing by (many) slow cores. This scenario is shown in Figure 8.4. It is also possible for the scheduler to automatically recognize the job type and schedule the job on the proper queue. For example, small and large jobs can be categorized based on the number of tasks. A job can be also classified based on the application information or by adding a job type feature in job profile.

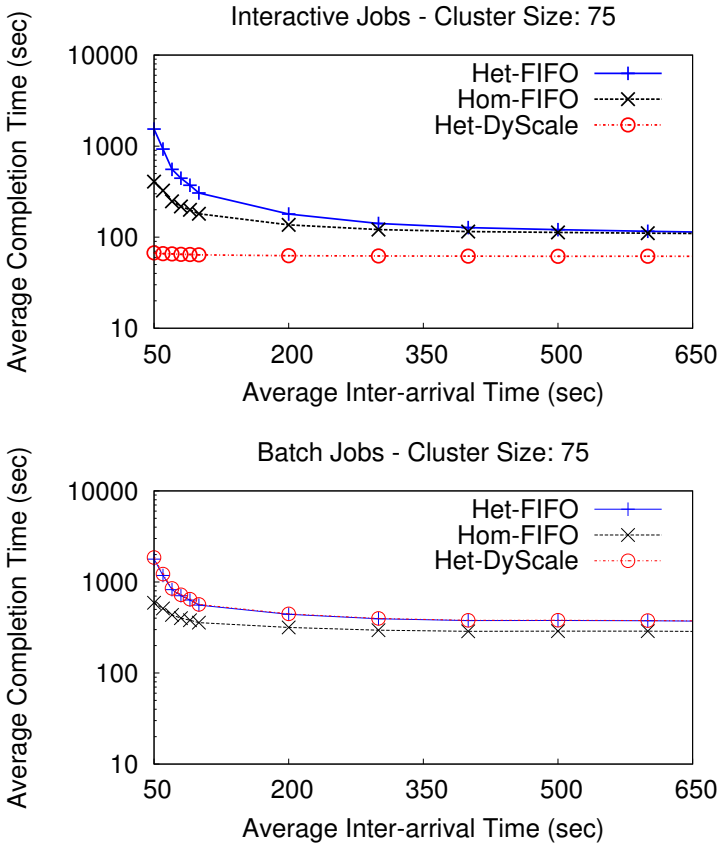



**Figure 8.3**: The completion time of interactive jobs and batch jobs under different configurations: heterogenous cluster using FIFO, homogenous cluster using FIFO and heterogenous cluster using DyScale.

To allocate resources according to the above scenario, a dedicated virtual resource pool has to be created for each job queue. For example, as shown in Figure 8.4, fast slots can be grouped as a Virtual Fast (vFast) resource pool that is dedicated to the Interactive Job Queue. Slow slots can be grouped as a Virtual Slow (vSlow) resource pool that is dedicated to the Batch Job Queue.

The attractive part of such virtual resource pool arrangement is that it *preserves data locality* because both fast and slow slots have the same data access to the datasets stored

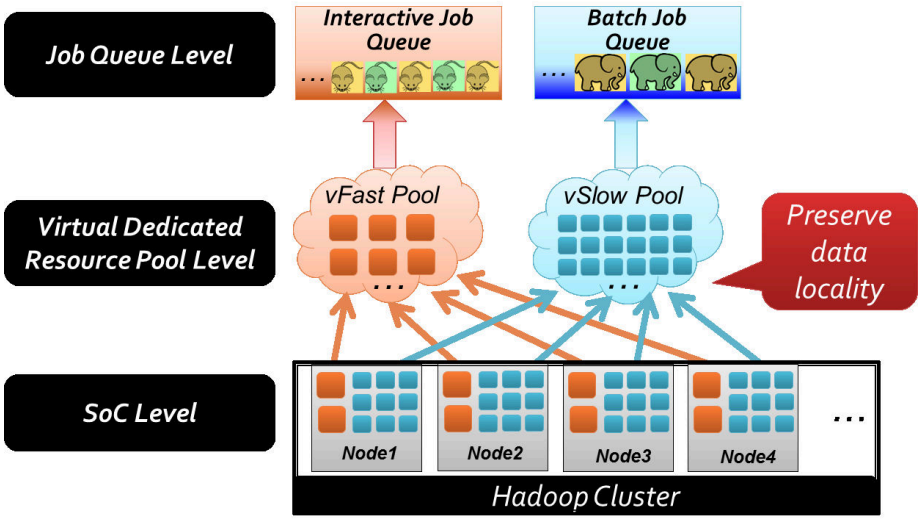



**Figure 8.4**: Virtual Resource Pools.

in the underlying Hadoop Distributed File System. Therefore, any dataset can be processed by either *fast* or *slow* virtual resource pools, or their combination[2]. To support a virtual resource pool design, the TaskTracker needs additional mechanisms for the following functionalities:

- the ability to start a task on a specific core, i.e., to run a slot on a specific core and assign a task to it;
- to maintain the mapping information between a task and the assigned slot type.

The TaskTracker always starts a new JVM for each task instance (if the JVM reuse feature in Hadoop is disabled). It is done such that a JVM failure does not impact other tasks or does not take down the TaskTracker. Running a task on a specific core can be achieved by binding the JVM to that core. We use the *CPU affinity* to implement this feature. By setting the CPU affinity, a process can be bound to one or a set of cores. The TaskTracker

[2]Note the difference of this approach compared to node level heterogeneity, where data may reside on different node types, and therefore, it leads to data locality issues as data is not always available on the desired node types.

calls *spawnNewJVM* class to spawn a JVM in a new thread. The CPU affinity can be specified during spawn to force the JVM to run on the desired fast or slow core.

An additional advantage of using the CPU affinity is that it *can be changed during runtime*. If the *JVM reuse* feature is enabled in the Hadoop configuration (note, that the JVM reuse can be enabled only for the tasks of the same job), the task can be placed on a desired core by changing the CPU affinity of the JVM.

The mapping information between tasks and cores is maintained by recording *(task_ID, JVM_pid, core_id)* in the TaskTracker table. When a task finishes, the TaskTracker knows whether the released slot is fast or slow.

The JobTracker needs to know whether the available slot is a slow or fast slot to make resource allocation decisions. DyScale communicates this information through the *heartbeat*, which is essentially a RPC (Remote Procedure Call) between the TaskTracker at a worker node and the JobTracker at the master node.

The TaskTracker asks the JobTracker for a new task when the current running map/reduce tasks are below the configured maximum allowed number of map/reduce tasks through a boolean parameter *askForNewTask*. If the TaskTracker can accept a new task, then the JobTracker calls the Hadoop Scheduler for a decision to assign a task to this TaskTracker.

The Scheduler checks *TaskTrackerStatus* to know whether the available slots are Map or Reduce slots. DyScale's Scheduler also needs to distinguish the slot type. There are four types of slots: *i)* fast map, *ii)* slow map, *iii)* fast reduce, and *iv)* slow reduce.

In the DyScale framework, the Scheduler interacts with the *JobQueue* by considering the slot type, e.g., if the available slot is a fast slot, then this slot belongs to vFast pool, and the *InteractiveJobQueue* is selected for a job/task allocation. After selecting the *JobQueue*, it allocates the available slot to the first job in the queue.

Different policies exist for ordering the jobs inside the *JobQueue* as well as different slot allocation policies. The default policy is FIFO. The job ordering/resource allocation depends on the performance objectives and can be defined by the Hadoop Fair Scheduler (HFS) [132] or the ARIA SLO-driven scheduler [105]. DyScale can be easily augmented with additional policies for improving fairness, meeting completion time objectives, or other metrics. The JobTracker puts a list of current actions, such as LAUNCH_TASK, in the TaskTrackerAction list to tell the TaskTracker what to do next through the *heartbeatResponse*.

### 8.3.3 Managing Spare Cluster Resources

Static resource partitioning and allocation may be inefficient if a resource pool has spare resources (slots) but the corresponding *JobQueue* is empty, while other *JobQueue*(s) have jobs that are waiting for resources. For example, if there are jobs in the *InteractiveJobQueue* and they do not have enough fast slots, then these jobs should be able to use the available (spare) slow slots.

We use the Virtual Shared (vShare) Resource pool to utilize spare resources. As shown in Figure 8.5, the spare slots are put into the vShare pool. Slots in the vShare resource pool can be used by any job queue.

The efficiency of the described resource sharing could be further improved by introducing the *TaskMigration* mechanism. For example, the jobs from the *InteractiveJobQueue* can use spare slow slots until the future fast slots become available. These tasks are migrated to the newly released fast slots so that the jobs from the *InteractiveJobQueue* always use optimal resources. Similarly, the migration mechanism allows the batch job to use temporarily spare fast slots if the *InteractiveJobQueue* is empty. These resources are returned by migrating the batch job from the fast slots to the

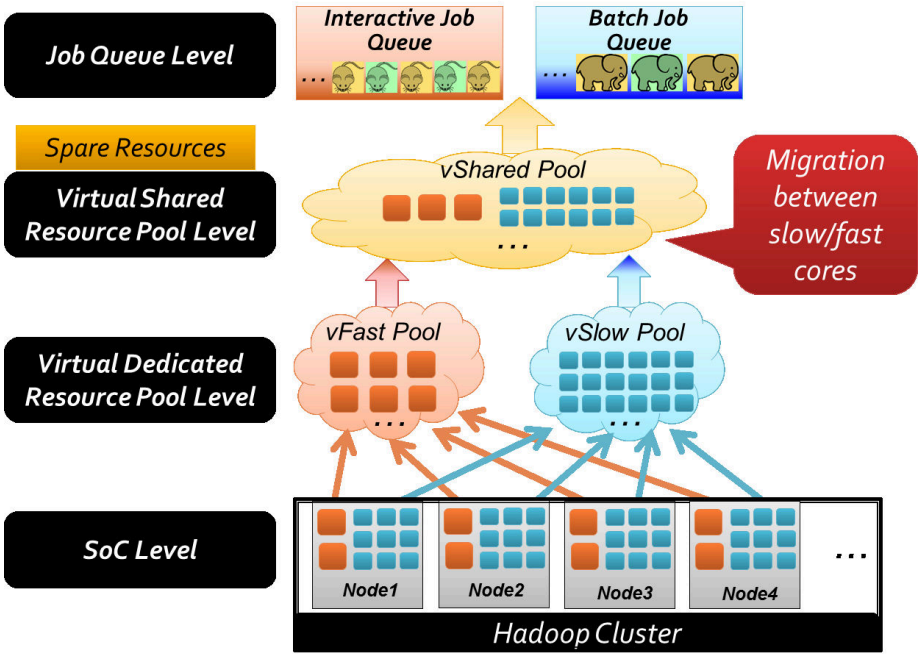



**Figure 8.5**: Virtual Shared Resource Pool.

released slow slots when a new interactive job arrives.

DyScale allows to specify different policies for handling spare resources. The migration mechanism is implemented by changing the JVM's CPU affinity within the same SoC. By adding the MIGRATE_TASK action in the TaskTrackerAction list in heartbeatResponse, the JobTracker can inform the TaskTacker to migrate the designated task between slow and fast slots.

DyScale can support SLOs by adding priorities to the queues and by allowing different policies for ordering the jobs inside each queue. For example, let the interactive jobs have deadlines to meet. The batch jobs are the best-effort jobs. When there are not enough fast slots for interactive jobs, these jobs can be given priority for using the available slow slots. This can be supported by the vShared resource pool and task migration.

## 8.4 Case Study

In this section, we first present our measurement results with a variety of MapReduce applications executed on a Hadoop cluster configured with different CPU frequencies. Then, we analyze and compare simulation results based on synthetic Facebook traces that emulate the execution of the Facebook workload on a Hadoop cluster to quantify the effects of homogeneous versus heterogeneous processors. We also analyze the DyScale scheduler performance under different job arrival rates and evaluate its performance advantages in comparison to the FIFO and Capacity[4] job schedulers that are broadly used by the Hadoop community.

### 8.4.1 Experimental Testbed and Workloads

We use an 8-node Hadoop cluster as our experimental testbed. Each node is a HP Proliant DL 120 G7 server that employs the latest Intel Xeon quad-core processor E31240 @ 3.30Ghz. The processor offers a set of controllable CPU frequencies varying from 1.6 Ghz to 3.3 Ghz, and each core frequency can be set separately. The memory size of the server is 8 GB. There is one 128 GB disk dedicated for system usage and 6 additional 300 GB disks dedicated to Hadoop and data. The servers use 1 Gigabit Ethernet and are connected by a 10 Gigabit Ethernet Switch. We use Hadoop 1.0.0 with 1 dedicated server as JobTracker and NameNode, and the remaining 7 servers as workers. We configure 1 map and 1 reduce slot per core, i.e., 4 map slots and 4 reduce slots per each worker node. The file system blocksize is set to 64MB and the replication level is set to 3. We use the default Hadoop task failure mechanism to handle task failures.

We select 13 diverse MapReduce applications [7] to run experiments in our Hadoop cluster. The high level description of these applications is given in Table 8.2.

Applications 1, 8, and 9 use synthetically generated data as input. Applications 2 to 7

| *Application* | *Input data (type)* | *Input data (GB)* | *Interm data (GB)* | *Output data (GB)* | *#map,red tasks* |
|---|---|---|---|---|---|
| *1.TeraSort* | Synth | 31 | 31 | 31 | 450, 28 |
| *2.WordCount* | Wiki | 50 | 9.8 | 5.6 | 788, 28 |
| *3.Grep* | Wiki | 50 | $3x10^{-8}$ | $1x10^{-8}$ | 788, 1 |
| *4.InvIndex* | Wiki | 50 | 10.5 | 8.6 | 788, 28 |
| *5.RankInvIndex* | Wiki | 46 | 48 | 45 | 768, 28 |
| *6.TermVector* | Wiki | 50 | 4.1 | 0.002 | 788, 28 |
| *7.SeqCount* | Wiki | 50 | 45 | 39 | 788, 28 |
| *8.SelfJoin* | Synth | 28 | 25 | 0.15 | 448, 28 |
| *9.AdjList* | Synth | 28 | 11 | 11 | 507, 28 |
| *10.HistMovies* | Netflix | 27 | $3x10^{-5}$ | $7x10^{-8}$ | 428, 1 |
| *11.HistRatings* | Netflix | 27 | $2x10^{-5}$ | $6x10^{-8}$ | 428, 1 |
| *12.Classification* | Netflix | 27 | 0.008 | 0.006 | 428, 50 |
| *13.KMeans* | Netflix | 27 | 27 | 27 | 428, 50 |

**Table 8.2**: Application characteristics.

process Wikipedia articles. Applications 10 to 13 process Netflix ratings. The intermediate data is the output of map task processing. This data serves as the input data for reduce task processing. If the intermediate data size is large, then more data needs to be shuffled from map tasks to reduce tasks. We call such jobs *shuffle-heavy*. Output data needs to be written to the distributed storage system (e.g., Hadoop Distributed File System). When the output data size is large, we call such jobs *write-heavy*. *Shuffle-heavy* and *write-heavy* applications tend to use more networking and IO resources.

Selected applications for our experiments represent a variety of MapReduce processing patterns. For example, TeraSort, RankInvIndex, SeqCount, and KMeans are both *shuffle-heavy* and *write-heavy*. Grep, HistMovies, HistRatings, and Classification have a significantly reduced data size after the map stage and therefore belong to the *shuffle-light* and *write-light* category. In addition, some applications including Classification and KMeans are computation-intensive because their *map* phase processing time is orders of magnitude higher than other phases. The selected applications exhibit different processing patterns and allow for a detailed analysis on a diverse set of

MapReduce workloads.

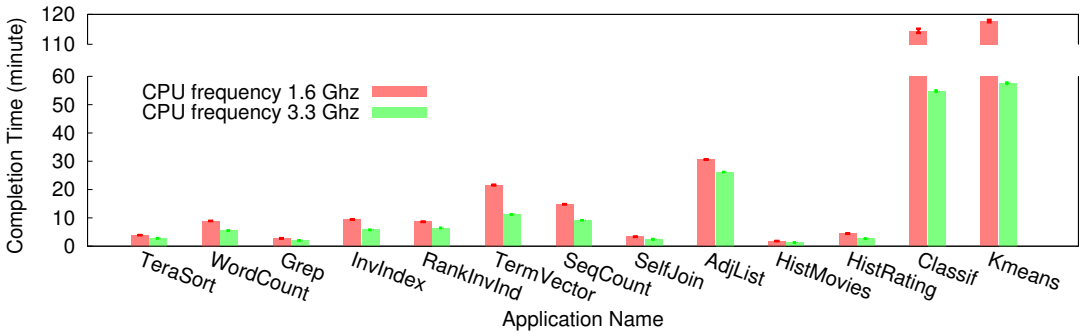
### 8.4.2 Experimental Results with Different CPU Frequencies

Since the heterogeneous multi-core processors are not yet available for provisioning a real testbed and performing experiments directly, we need to understand how execution on "fast" or "slow" cores may impact performance of MapReduce applications. Here we aim to empirically evaluate the impact of "fast" and "slow" cores on the completion time of representative MapReduce applications. We mimic the existence of fast and slow cores by using the CPU frequency control available in the current hardware. These experiments are important, because Hadoop and MapReduce applications are considered to be disk-bound, and intuitively, it is unclear what is the performance effect of different CPU frequencies.

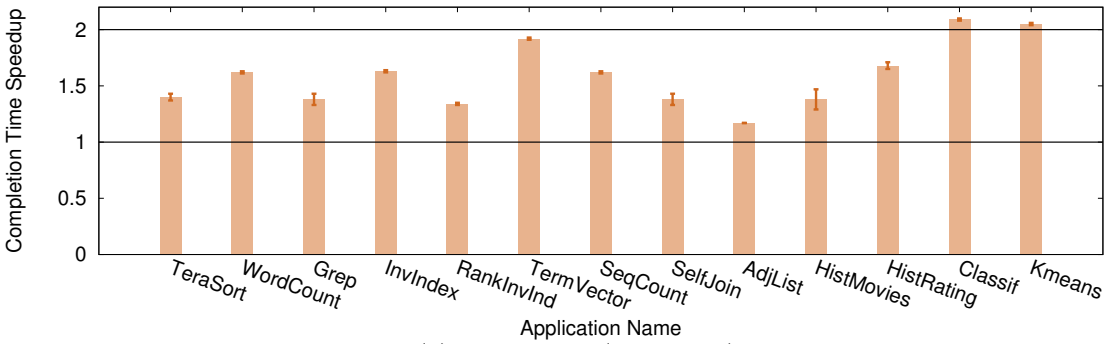
We run all applications from Table 8.2 on our experimental cluster using two scenarios: *i)* CPU frequency of all processors is set to 1.6 Ghz for emulating "slow" cores, and *ii)* CPU frequency of all processors is set 3.3 Ghz, e.g., two times faster, for emulating "fast" cores. We flush the memory after each experiment and disable the write cache to avoid caching interference.

All measurement experiments are performed five times. We show the mean and the variance, i.e., the minimal and maximal measurement values across the 5 runs. This comment applies to the results in Figures 8.6, 8.8, and 8.9.

Figure 8.6 summarizes the results of our experiments. Figure 8.6(a) shows the completion times for each job. Note the gap in the Y-axis that is introduced for better visualizing of all 13 applications in the same figure: the map task durations of Classification and Kmeans are much higher compared to the other 11 applications. Figure 8.6(b) shows the normalized results of the relative speedup obtained by executing the applications on the servers with 3.3.Ghz compared to the application completion time




(a) Measured job completion time




(b) Job speedup (normalized)

**Figure 8.6**: Measured job completion time and speedup (normalized) when the CPU frequency is scaled-up from 1.6 GHz to 3.3 GHz.

on the servers with 1.6 Ghz. Speedup of 1 means no speedup, i.e., the same completion time. Few jobs have a completion time speedup of 1.2 to 1.3, while the majority of jobs enjoy speedups of 1.6 to 2.1.

To better understand the above, we perform further analysis at the phase level duration. Each map task processes a logical split of the input data (e.g., 64 MB) and performs the following steps: *read, map, collect, spill*, and *merge* phases, see Figure 8.7. The map task *read*s the data, applies the *map* function on each record, and *collect*s the resulting output in memory. If this intermediate data is larger than the in-memory buffer, it is *spill*ed on the local disk of the machine executing the map task and *merge*d into a single file for each

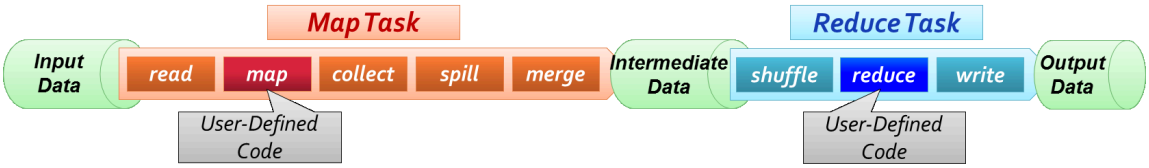
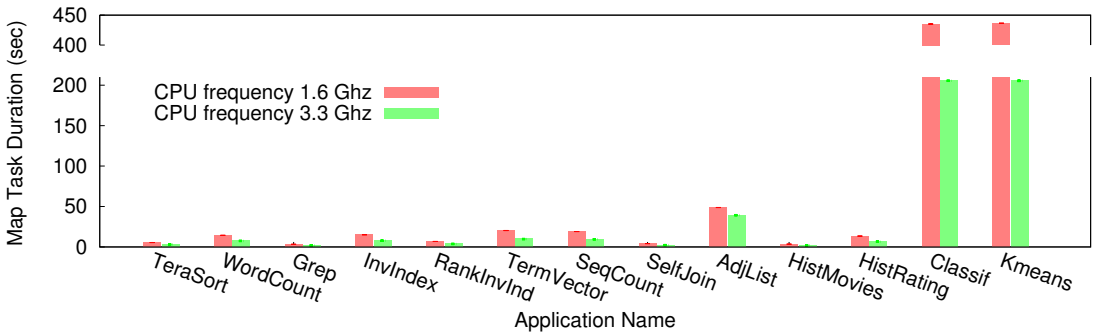
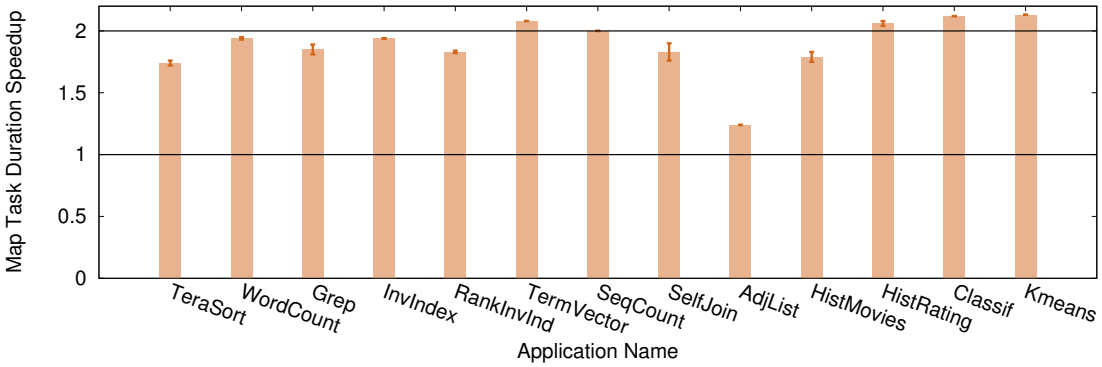



**Figure 8.7**: Map and Reduce Tasks Processing Pipeline.




(a) Average measured map task duration

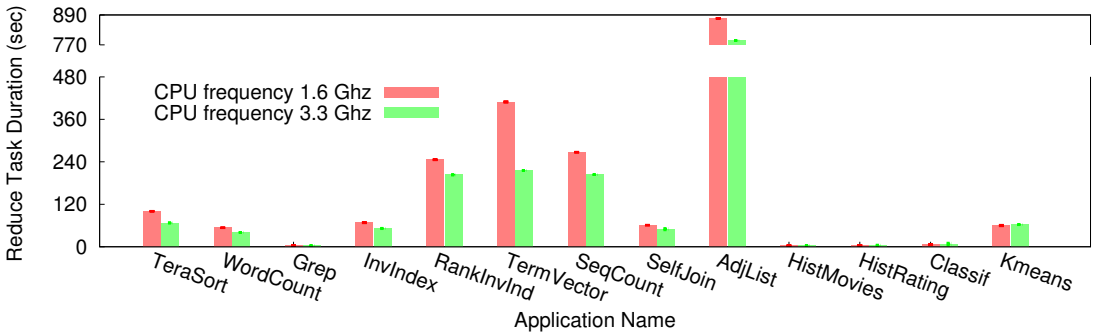



(b) Average map task speedup (normalized)

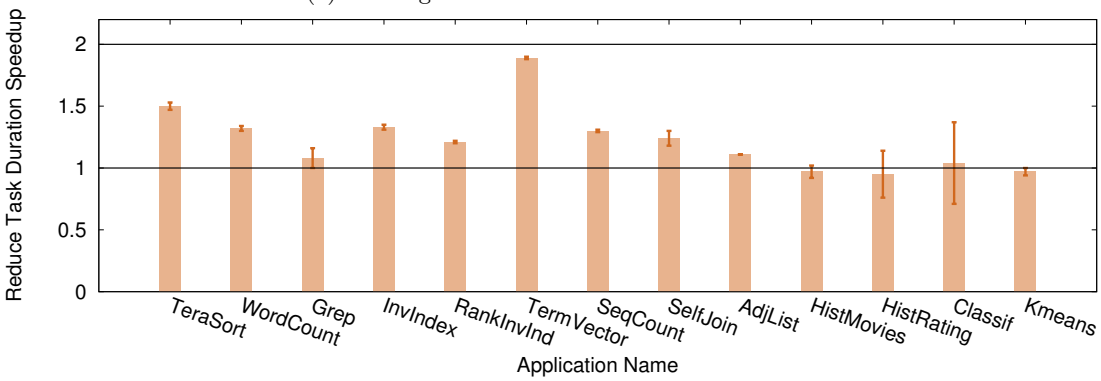
**Figure 8.8**: Average measured map task duration and normalized speedup of map tasks in the experiments when the CPU frequency is scaled-up from 1.6 Ghz to 3.3 Ghz.

reduce task.

The reduce task processing is comprised by the *shuffle, reduce*, and *write* phases. In the *shuffle* phase, the reduce tasks fetch the intermediate data files from the already completed map tasks and sort them. After all intermediate data is shuffled, a final pass is made to




(a) Average measured reduce task duration




(b) Average reduce task speedup (normalized)

**Figure 8.9**: Average measured reduce task duration and normalized speedup of reduce tasks in the experiments when the CPU frequency is scaled-up from 1.6 Ghz to 3.3 Ghz.

merge sorted files. In the *reduce* phase, data is passed to the user-defined reduce function. The output from the reduce function is written back to the distributed file system in the *write* phase. By default, three copies are written to different worker nodes.

We report the average measured map task durations with CPU frequencies of 1.6 Ghz and 3.3 Ghz in Figure 8.8(a) and the reduce task durations in Figure 8.9(a). For different applications, the time spent in the *shuffle* and *write* phases is different and depends on the amount of intermediate data and output data written back to Hadoop Distributed File System (i.e., whether the application is *shuffle-heavy* and/or whether it writes a large

amount of output data such as TeraSort, RankInvIndex, AdjList). These *shuffle* and *write* portions of the processing time influence the outcome of the overall application speedup.

Our analysis reveals that the map task processing for different applications have a similar speedup profile when executed on a 3.3 Ghz CPU. In our experiments, this speedup is close to 2 across all 13 applications, see Figure 8.8(b). However, the *shuffle* and *write* phases in the reduce stage often show very limited speedup across applications (on average 20%, see Figure 8.9 (b)) due to different amount of data processed at this stage.

By looking at the results in Figures 8.8(b)- 8.9(b), one may suggest the following *simple* scheduling policy for improving MapReduce job performance and taking advantage of heterogeneous multi-processors. Run map tasks on faster cores and reduce tasks on slower cores. However, performance of many large jobs is critically impacted not only by the type of slots allocated to the job tasks, but by the number of allocated slots. For example, if each processor has 2 *fast* cores and 6 *slow* cores, then the proposed *simple* scheduling does not work as expected: using only *fast* cores for processing map tasks result in degraded performance for large jobs compared to their processing by using the available *slow* cores as has been shown in the related motivating example in Section 8.2. Therefore, to efficiently utilize the heterogeneous multi-core processors, one needs to consider a number of factors: 1) the type and the size of the job, workload job mix, jobs arrival rate, user performance objectives, and 2) the composition of the heterogeneous multi-processor, i.e., the number of *fast* or *slow* cores per processor, as well as the Hadoop cluster size.

### 8.4.3 Simulation Framework and Results

As the heterogeneous multi-core processors are not yet readily available, we perform a simulation study using the extended MapReduce simulator SimMR [106] and a synthetic Facebook workload [132]. In addition, simulation allows more comprehensive sensitivity

analysis. Our goal is to compare the job completion times and to perform a sensitivity analysis when a workload is executed by different Hadoop clusters deployed on either homogeneous or heterogeneous multi-core processors.

The event-driven simulator SimMR consists of the following three components, see Figure 8.10:

- A *Trace Generator* creates a replayable MapReduce workload. In addition, the *Trace Generator* can create traces defined by a synthetic workload description that compactly characterizes the duration of map and reduce tasks as well as the shuffle stage characteristics via corresponding distribution functions. This feature is useful to conduct sensitivity analysis of new schedulers and resource allocation policies applied to different workload types.

- The *Simulator Engine* is a discrete event simulator that accurately emulates the job master functionality in the Hadoop cluster.

- *The pluggable scheduling policy* dictates the scheduler decisions on job ordering and the amount of resources allocated to different jobs over time.

We extend SimMR[3] to emulate the DyScale framework. We also extend SimMR to emulate the Capacity scheduler [4] for homogeneous environments. We summarize the three schedulers used in this chapter below:

- FIFO: the default Hadoop scheduler that schedules the jobs based on their arrival order.

- Capacity: users can define different queues for different types of jobs. Each queue can be configured with a percentage of the total number of slots in the cluster, this

[3]SimMR accurately reproduces the original job processing: the completion times of the simulated jobs are within 5% of the original ones, see [106].

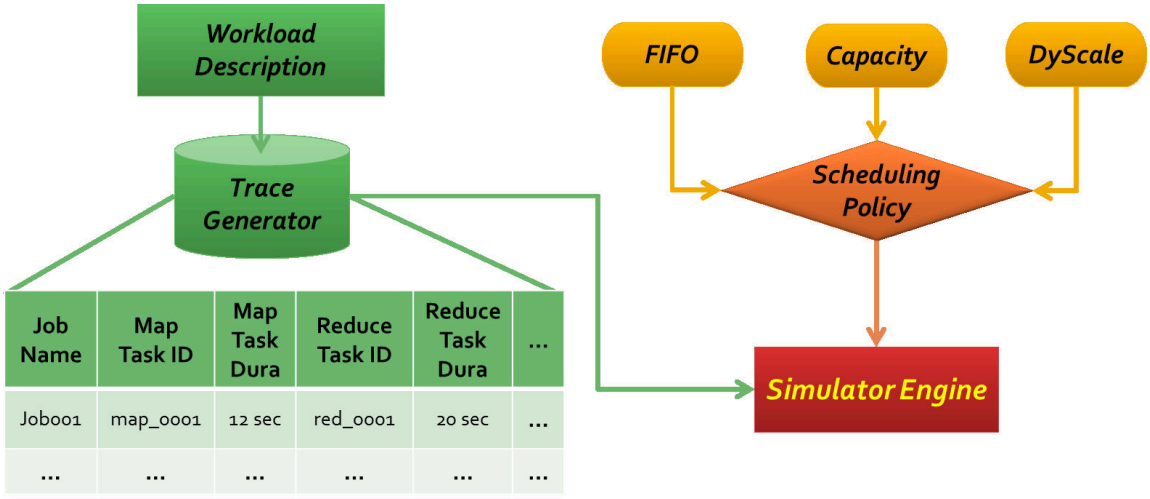



**Figure 8.10**: Simulator Design.

parameter is called *queue capacity*. This scheduler has an *Elasticity* feature that allows free resources to be allocated to a queue above its capacity to prevent artificial silos of resources and achieve better resource utilization.

- DyScale: we use two different versions: *i)* the *basic version* without task migration and *ii)* the *advanced version* with the migration feature enabled.

We approximate the performance and power consumption of different cores from the available measurements of the existing Intel processors [87, 31] executing the PARSEC benchmark [13]. We observe that the Intel processors i7-2600 and E31240 (used in the HP Proliant DL 120 G7 server) are from the same Sandy Bridge micro-architecture family and have almost identical performance [98]. We additionally differentiate the performance of map and reduce tasks on the simulated processors by using our experimental results reported in Section 8.4.2. We summarize this data in Table 8.3.

With a power budget of 84W, we choose three multi-core processor configurations, see Table 8.4. In our experiments, we simulate the execution of the Facebook workload on three

different Hadoop clusters with multi-core processors. For sensitivity analysis, we present results for different cluster sizes of 75, 120, and 210 nodes as they represent interesting performance situations.

| Type | Processor Name | Tech. | Freq. | Power per Core | Norm. Power | Norm. PARSEC Perform. | Norm. Map Task Perform. | Norm. Reduce Task Perform. |
|---|---|---|---|---|---|---|---|---|
| Type1 | i7-2600 Sandy Bridge | 32nm | 3.4Ghz | 21W | 1.0 | 1.0 | 1.0 | 1.0 |
| Type2 | i5-670 Nehalem | 32nm | 3.4Ghz | 16W | 0.81 | 0.92 | 0.92 | 0.98 |
| Type3 | AtomD Bonnell | 45nm | 1.7Ghz | 4W | 0.19 | 0.45 | 0.45 | 0.83 |

**Table 8.3**: Processor Specifications (Freq.: Frequency, Norm.: Normalized, Perform.: Performance).

| Configuration | Type 1 | Type 2 | Type 3 | Power |
|---|---|---|---|---|
| *Hom-fast* | 4 | 0 | 0 | 84W |
| *Hom-slow* | 0 | 0 | 21 | 84W |
| *Heterogeneous* | 0 | 3 | 9 | 84W |

**Table 8.4**: Processor configurations with the same power budget of 84 W.

We configure each Hadoop cluster with 1 map and 1 reduce slot per core[4], e.g., for a Hadoop cluster size with 120 nodes, the three considered configurations have the following number of map and reduce slots:

- the *Homogeneous-fast* configuration has 480 fast map (reduce) slots,
- the *Homogeneous-slow* configuration has 2640 slow map (reduce) slots, and
- the *Heterogeneous* configuration has 360 fast map (reduce) slots and 1080 slow map (reduce) slots.

We generate 1000 MapReduce jobs according to the distribution shown in Table8.1, with a 3-fold increase in the input datasets[5]. Jobs from the 1st to the 5th group are small

[4]We assume that each node has enough memory to configure map and reduce slots with the same amount of RAM for different SOC configurations.

[5]In our earlier conference paper [121], we have evaluated DyScale on a smaller workload defined by Table8.1 and

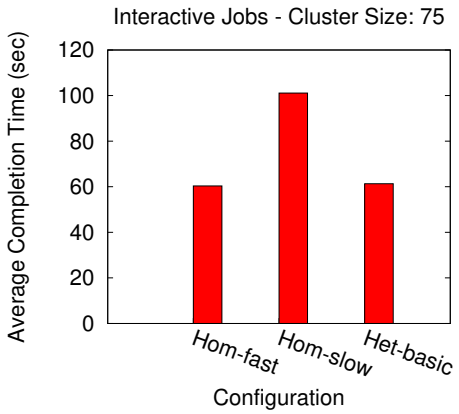
interactive jobs (e.g., with less than 300 tasks) and the remaining jobs are large batch jobs. The interactive jobs are 82% of the total mix and the batch jobs are 18%. The task duration of the Facebook workload can be best fit with a LogNormal distribution [108] and the following parameters: LN(9.9511, 1.6764) for map task duration and LN(12.375, 1.6262) for reduce task duration.

First, we perform a comparison of these three configurations when jobs are processed by each cluster in isolation: each job is submitted in the FIFO order, there is no bias due to the specific ordering policy nor queuing waiting time for each job, e.g., each job can use all cluster resources. For the *heterogeneous* configuration, the SimMR implementation supports the vShared resource pool so that a job can use both fast and slow resources. Results are plotted in Figure 8.11. Each row shows two graphs that correspond to the clusters with 75, 120, and 210 nodes respectively. The graphs show the average completion time of interactive jobs (left column) and batch jobs (right column).

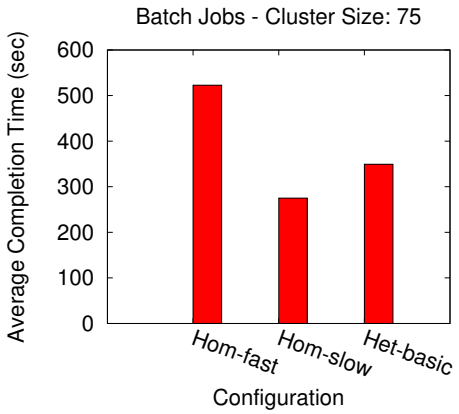
For interactive jobs, *Homogeneous-fast* and *Heterogeneous* configurations achieve very close completion times and significantly outperform the *Homogeneous-slow* configuration by being almost twice faster. The small, interactive jobs have a limited parallelism and once their tasks are allocated the necessary resources, these jobs cannot take advantage of the extra slots available in the system. For such jobs, fast slots are the effective way to achieve better performance (i.e., to scale-up).

For batch jobs, as expected, the scale-out approach shows its advantage since batch jobs have a large number of map tasks. The *Homogeneous-slow* configuration consistently outperforms *Homogeneous-fast*, and can be almost twice faster when the cluster size is small (e.g., 75 nodes). The interesting result is that the *Heterogeneous* configuration is almost neck-to-neck with the *Homogeneous-slow* configuration for batch jobs.

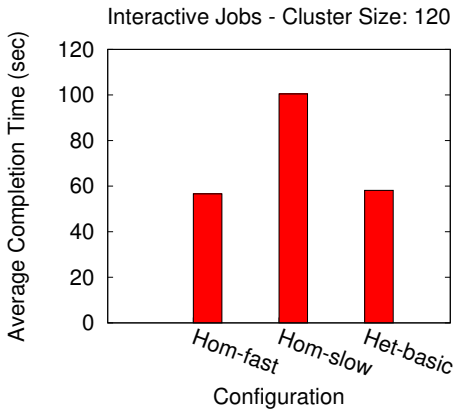
smaller size Hadoop clusters with 25, 40, and 70 nodes. To test the scalability of the solution, we increased the application datasets and the Hadoop clusters for processing.



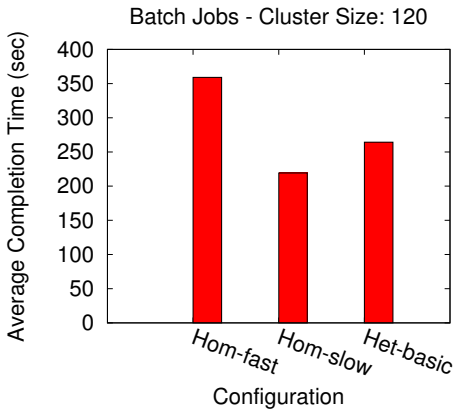

(a) Interactive Jobs, Cluster Size: 75



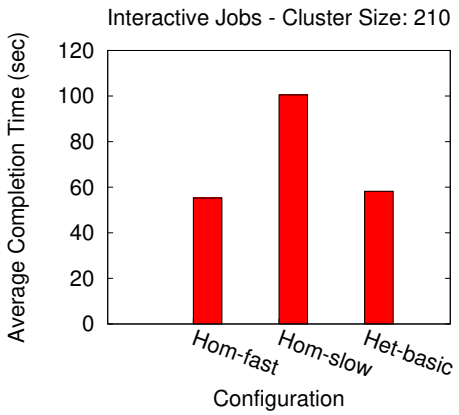

(b) Batch Jobs, Cluster Size: 75



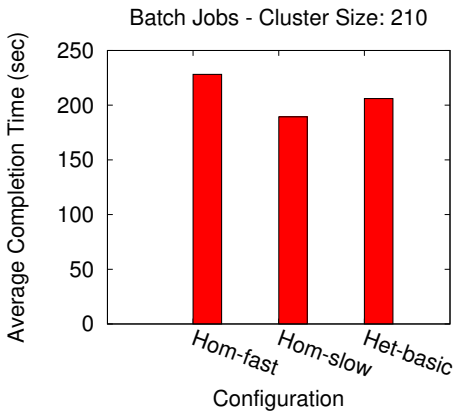

(c) Interactive Jobs, Cluster Size: 120




(d) Batch Jobs, Cluster Size: 120




(e) Interactive Jobs, Cluster Size: 210




(f) Batch Jobs, Cluster Size: 210



**Figure 8.11**: Completion time of interactive and batch jobs under different configurations.

By comparing these results, it is apparent that the heterogeneous multi-core processors with fast and slow cores present an interesting design point. It can significantly improve the completion time of interactive jobs with the same power budget. The large batch jobs are benefiting from the larger number of the slower cores that improve throughput of these jobs. Moreover, the batch jobs are capable of taking advantage and effectively utilizing the additional fast slots in the vShared resource pool supported by DyScale.

### 8.4.4 Simulation Results with Arrival Process

In this section, we conduct further experiments for comparing the performance of different configurations under varying job arrival rates. We use the same experimental setup as in Section 8.4.3. We use exponential inter-arrival times to drive the job arrival process and vary the average of the inter-arrival time between 50 sec and 1000 sec (between 50 sec and 100 sec with a step of 10 sec, and between 100 sec and 1000 sec with a step of 100 sec). We analyze three scenarios:

- **Scenario 1**: We compare the job completion times of DyScale (used in the *Heterogeneous* cluster configuration) with **FIFO** (used in both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations).
- **Scenario 2**: We compare the job completion times of DyScale (used in the *Heterogeneous* cluster configuration) with **Capacity** (used in both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations).
- **Scenario 3**: We compare the performance of DyScale with migration enabled and disabled to illustrate how a task migration feature can provide additional performance opportunities.

Figure 8.12 illustrates the performance comparison of *DyScale vs FIFO* (**Scenario 1**). The completion times of interactive jobs (left column) for both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations with FIFO are much higher than for the *Heterogeneous* configuration with DyScale. The *Homogeneous-fast* configuration is very sensitive to the cluster size and is least resilient to high arrival rates. The *Heterogeneous* configuration with DyScale consistently provides best performance for interactive jobs.

For batch jobs (right column in Figure 8.12), the *Heterogeneous* configuration with DyScale is slightly worse than the *Homogeneous-slow* configuration because batch jobs have more slots to use in *Homogeneous-slow* configuration. However, it outperforms the *Homogeneous-fast* configuration by up to 30%.

Overall, the *Heterogeneous* configuration with the DyScale scheduler shows very good and stable job completion times compared to both *Homogeneous-slow* and *Homogeneous-fast* cluster configurations with the FIFO scheduler. It is especially evident under higher loads, i.e., when the inter-arrival times are small and traffic is bursty. Overall, the system performance for the *Heterogeneous* configuration with the DyScale scheduler is very robust. When the inter-arrival time becomes larger (i.e., under light load), the observed performance gradually converges to the case when each job is executed in isolation, and the completion times are similar to the results shown in Figure 8.11.

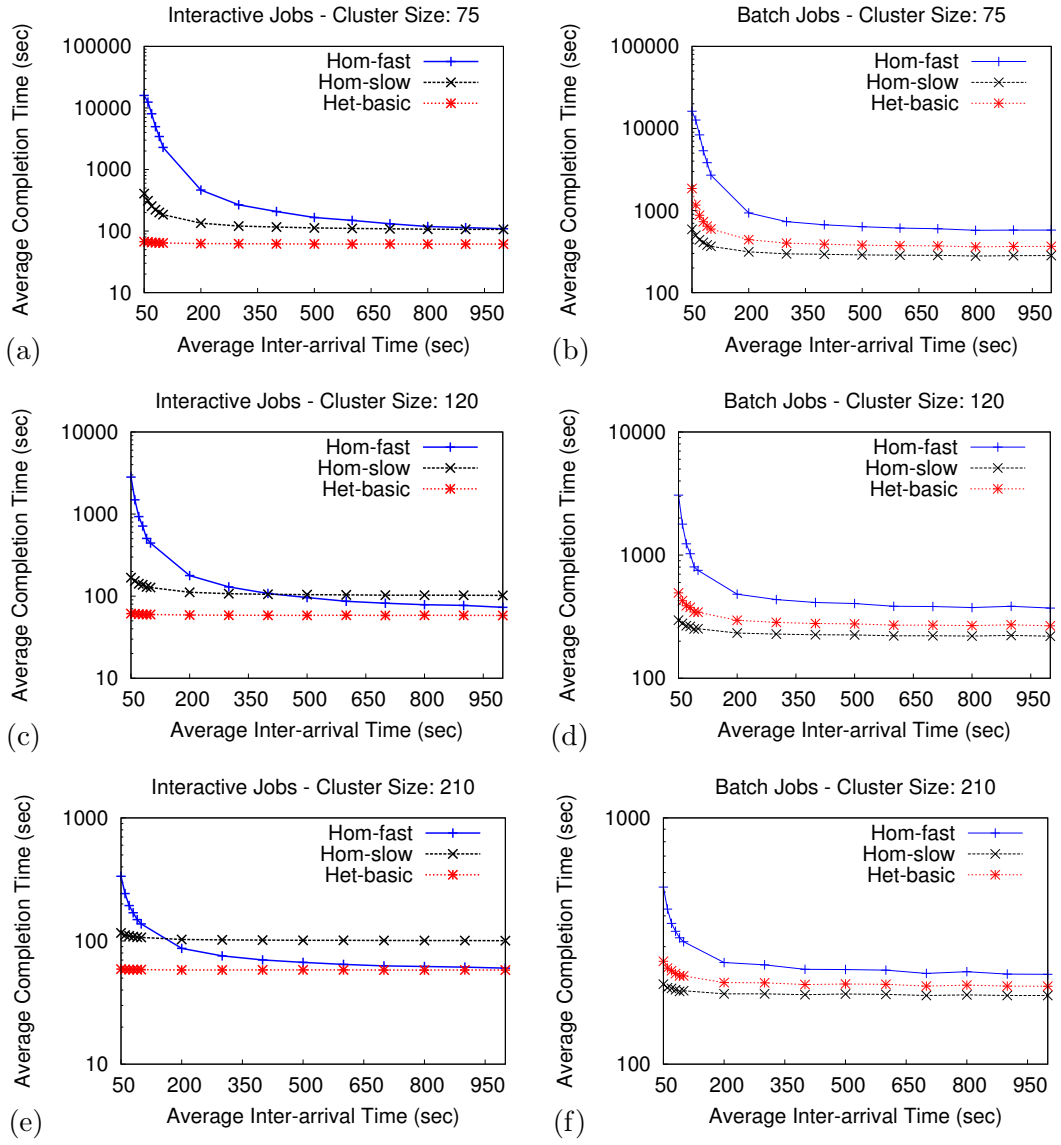



**Figure 8.12**: DyScale vs FIFO scheduler: the completion time of interactive jobs and batch jobs under different configurations, (a)-(b)the Hadoop cluster with 75 nodes, (c)-(d) the Hadoop cluster with 120 nodes, (e)-(f) the Hadoop cluster with 210 nodes.

Figure 8.13 illustrates the performance comparison of *DyScale vs Capacity* (**Scenario 2**). The Capacity Scheduler is configured with two queues for interactive jobs and batch jobs, respectively. Each queue capacity is determined based on the ratio of the interactive

and batch jobs in the Facebook workload, see Table 8.5. We can see that the performance of interactive jobs (shown in top row) of Figure 8.13 is supported better with the Capacity Scheduler compared to FIFO (the previous experiments shown in left column of Figure 8.12). The completion times of interactive jobs for *Heterogeneous* configuration is slightly worse than for the *Homogeneous-fast* configuration, but much better than for *Homogeneous-slow*, by up to 40%. For batch jobs, the *Heterogeneous* configuration is slightly worse than *Homogeneous-slow*, but it outperforms the *Homogeneous-fast* configuration by up to 30%. Again the *Heterogeneous* configuration with DyScale provides an interesting solution and exhibits a flexible support for job classes with different performance objectives compared to homogeneous-core configurations with either FIFO or Capacity schedulers.

| **Config.** | **Interactive-Queue capacity (total slots for cluster size 120)** | **Batch-Queue capacity (total slots for cluster size 120)** |
|---|---|---|
| *Hom-fast* | 18% (87) | 82% (393) |
| *Hom-slow* | 18% (453) | 82% (2067) |

**Table 8.5**: Capacity scheduler: *queue capacity* configurations (in the brackets, we provide the number of slots in each queue for the cluster with 120 nodes as an example).

Finally, we compare the *basic* DyScale (no task migration) and the *advanced* DyScale (with the task migration feature) and present the results for **Scenario 3** in Figure 8.14. We can see that the migration feature always brings additional performance improvement for both interactive and batch jobs because it allows more efficient use of the cluster resources. When the cluster size is small, such feature provides a higher performance boost for interactive jobs, see Figure 8.14(a). In this case, there is only a limited number of fast slots, and the chance is higher that some interactive job is allocated to a slow slot. Task Migration allows migrating tasks when fast slots become available, and utilizes fast slots more efficiently.

When the cluster size increases, the task migration is more beneficial for batch jobs, see Figure 8.14(f). In this case, there are more fast slots in the system and the batch jobs can utilize them. However, when an interactive job arrives, the fast slots occupied by batch jobs can be released by migrating batch tasks to slow slots.

In summary, the *Heterogeneous* configuration with the DyScale scheduler allows for significant performance improvements for interactive jobs while maintaining and improving the performance of batch jobs compared to both *Homogeneous-slow* and *Homogeneous-fast* configurations with different job schedulers.

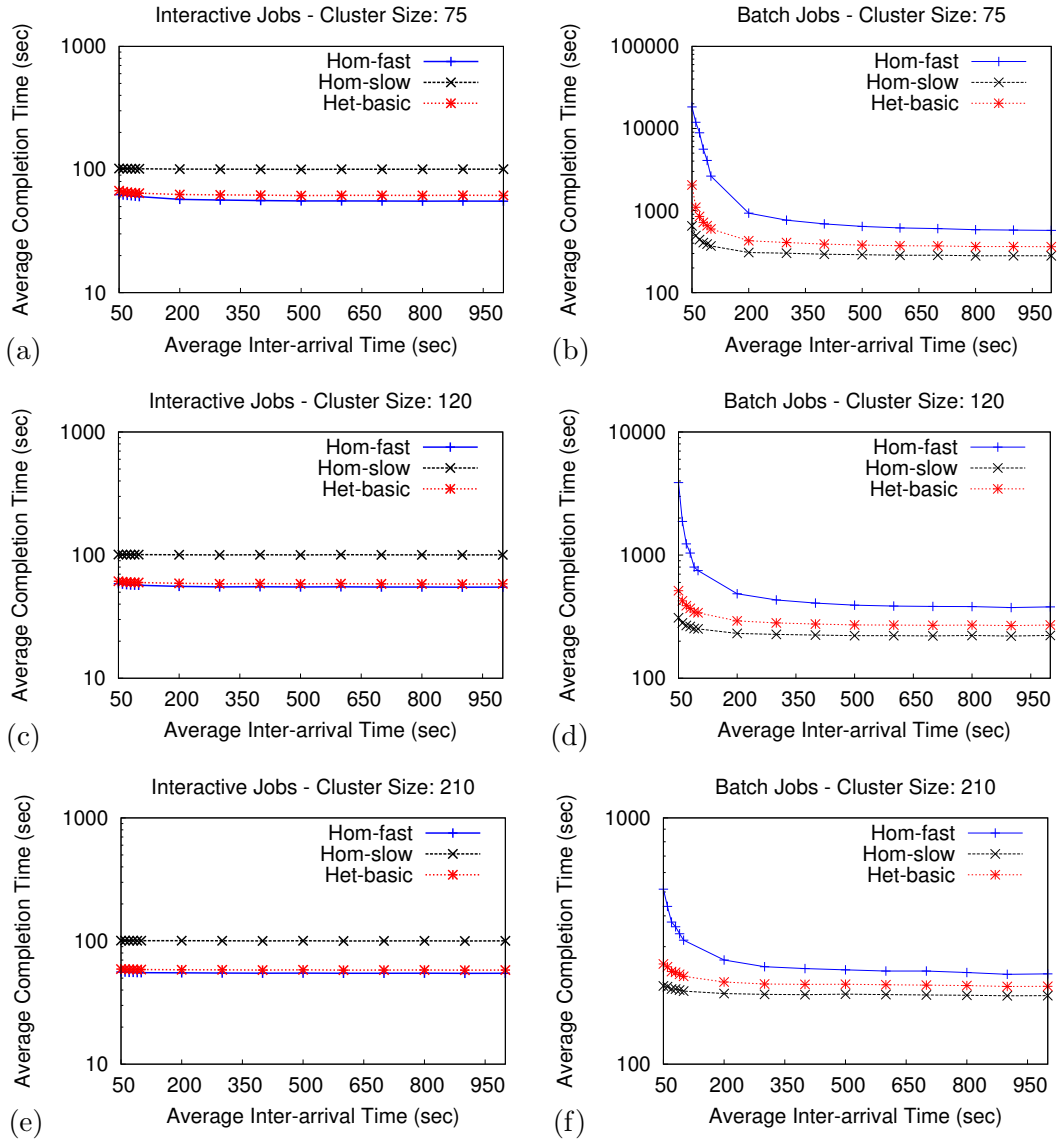



**Figure 8.13**: DyScale vs Capacity Scheduler: the completion time of interactive jobs and batch jobs under different configurations, (a)-(b)the Hadoop cluster with 75 nodes, (c)-(d) the Hadoop cluster with 120 nodes, (e)-(f) the Hadoop cluster with 210 nodes.

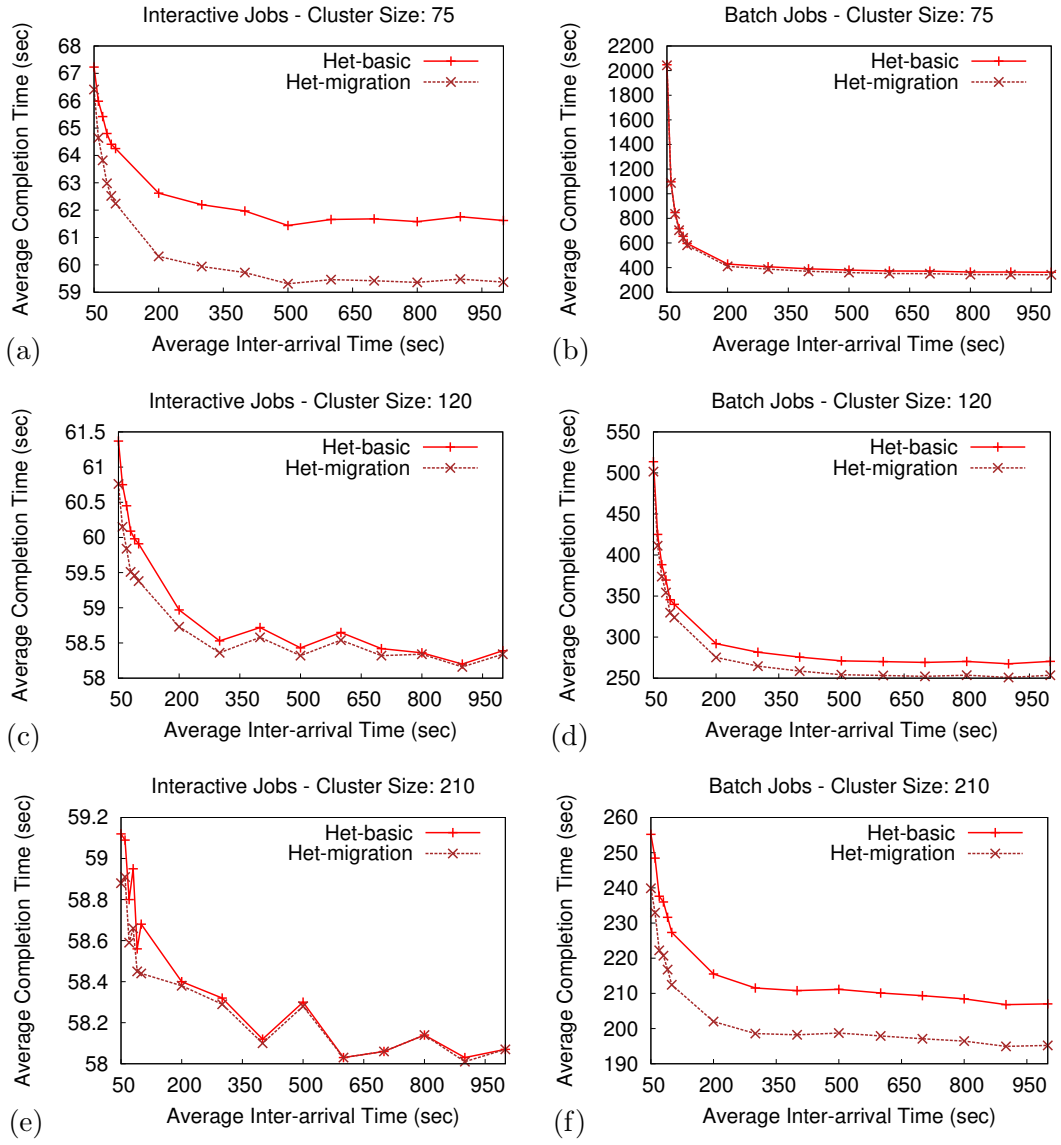



**Figure 8.14**: Comparison of *basic DyScale* and *DyScale with migration* for *Heterogeneous* configuration: the completion time of interactive jobs and batch jobs under different cluster sizes, (a)-(b) cluster of 75 nodes, (c)-(d) cluster of 120 nodes, (e)-(f) cluster of 210 nodes.

## 8.5 Summary

In this charpter, we exploit the new opportunities and performance benefits of using servers with heterogeneous multi-core processors for MapReduce processing. We present a new

scheduling framework, called DyScale, that is implemented on top of Hadoop. DyScale enables creating different virtual resource pools based on the core-types for multi-class job scheduling. This new framework aims at taking advantage of capabilities of heterogeneous cores for achieving a variety of performance objectives. DyScale is easy to use because the created virtual clusters have access to the same data stored in the underlying distributed file system, and therefore, any job and any dataset can be processed by either *fast* or *slow* virtual resource pools, or their combination. MapReduce jobs can be submitted into different queues, where they operate over different virtual resource pools for achieving better completion time (e.g., small jobs) or better throughput (e.g., large jobs). It is easy to incorporate the DyScale scheduler into the latest Hadoop implementation with YARN [5], as YARN has a pluggable job scheduler as one of its components. The main findings of this chapter are also reported in [122].

# Chapter 9

# Conclusions and Future Work

The main contribution of this dissertation is on the investigation of workload interleaving challenges in data centers and the design of efficient workload interleaving approaches with performance guarantees. A set of metrics, models and algorithms are developed in this dissertation to achieve performance isolation of different workloads in today's data centers and are summarized as follows.

- At the storage node level:
  - There are various tasks maintained at the background of systems for achieving performance, reliability, and availability targets. In this dissertation, BusyBee, an expedient scheduling framework for background tasks is devised by investigating the statistical characterization of busy periods (user traffic).
  - Spinning down disk drives to save energy while *transparently* to the end users and *reliably* to the disk drive's lifetime is a challenging open problem. In this dissertation, PREFiguRE - a robust framework for performance, power, and reliability is developed based on an analytic methodology that uses the histogram of idle times to quantify the performance, power, and reliability impacts and determine optimal schedules for power saving modes.

- At the storage cluster level:
  - To improve load balancing and reduce power assumption, work consolidation is a technique widely used in today's under-utilized data centers. In this dissertation, an automated consolidation framework with performance guarantees is developed, which can estimate beforehand the benefits and overheads of each consolidation options to help make intelligent and automatic consolidation decisions.
  - Today's systems and data are distributed across multiple geographic locations for the purpose of reducing the chance of data unavailability or its services unavailability in case of network, power, or other outages. The eventual consistency has become a common mechanism to manage the consistency across data centers. In order to achieve faster eventual consistency without impacting the performance of user traffic, an eventual consistency framework with performance guarantees is developed. The framework investigates the statistical characterization of idle periods in distributed storage systems and determines how fast the data can be sent or received without violating performance goals.
- At the computing node level: off-the-shelf server nodes are the building blocks for scale-out systems and are expected to utilize off-the-shelf management tools to handle their work independently. However, the standard Linux tools such as nice and ionice fail to achieve the best trade-off between protecting the performance of high priority workload and completing as much low priority work as possible. In this dissertation, an agile priority scheduling tool is developed for off-the-shelf components in data centers. The proposed priority scheduling middleware is a hybrid approach that is capable of selecting the optimal policy intelligently based on off-line and on-line

learning of the high priority workload characteristics and the imposed performance impact due to low priority work.

- At the computing cluster level: MapReduce and its open source implementation Hadoop offer a scalable and fault tolerant framework for processing large data sets. Today's MapReduce workload does not only have the traditional large batch jobs, but there are also many interactive adhoc queries and small MapReduce jobs that are completion-time sensitive. In this dissertation, DyScale - a new Hadoop scheduler is developed that can efficiently use heterogeneous resources to optimize the performance trade-offs of MapReduce jobs. DyScale supports multi-class MapReduce job scheduling by using different virtual resource pools that are created based on the core-types.

## 9.1 Future Work

There are several potential extensions to the methodologies and results in this dissertation.

### 9.1.1 Scheduling of Sensitive Background Tasks

The background scheduling framework proposed in Chapter 3 can be extended to learn and detect the length of both busy and idle periods, aiming for the best outcome on scheduling time sensitive background work. The proposed framework can also be used to schedule work with different but close priorities, where foreground work can be delayed more than background work, at least for some periods of time.

### 9.1.2 Accurate Performance Prediction in Work Consolidation and Beyond

The automated and intelligent work consolidation framework proposed in Chapter 5 can be enhanced by using different learning methods to detect any regular cycles, such as the daily and weekly business cycles, in order to reduce the impact of abrupt changes in workload characteristics. For further accuracy, an off-line component can be incorporated into the process of populating the look-up tables to ensure uniformity over the entire state space, particularly to capture extreme cases that are only rarely encountered in average workloads that are expected to run on the system. The framework can also be extended to cater to other storage features (in addition to consolidation) such as data replication for availability, reliability, back up, and virtualization.

### 9.1.3 Agile Middleware Scheduling with Different Scheduling Objectives

The agile middleware scheduling proposed in Chapter 7 can be extended to support other scheduling objectives. For example, explore the case of meeting background work targets (e.g., catch a deadline) while with minimum foreground performance impact. In addition, other statistical information can also be used in addition to the cumulative data histogram to better understand the resource requirement thus improve the scheduling performance.

### 9.1.4 Better Workload Interleaving using Heterogenous Resources

There is an interesting application of the DyScale proposed in Chapter 8 for improving performance of Pig queries [38]. In many cases, in the beginning, the Pig queries process large datasets (that corresponds to processing large jobs) and then have data transformation over small datasets in the second part of the workflow (that corresponds to processing small jobs).

Using models from earlier work [107] could quantify the impact of node and slot failures on the job completion time as the impact of failed fast or slow slots may be different. Similarly, the allocation of additional slots for re-running failed tasks may impact job completion times and can be supported by special policies in DyScale. There are also a variety of job ordering scheduling policies for achieving fairness guarantees or job completion objectives can be combined with DyScale to achieve better performance.

# Bibliography


[1] The Open Compute Project. http://www.opencompute.org/.

[2] Snia Iotta Repository. http://iotta.snia.org/traces/.

[3] TPC-W. http://www.tpc.org/tpcw/.

[4] Capacity Scheduler Guide. http://hadoop.apache.org/common/docs/r0.20.1/capacity-scheduler.html, 2010.

[5] Apache Hadoop Yarn. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2013.

[6] Yoshihisa Abe, Hiroshi Yamada, and Kenji Kono. Enforcing appropriate process execution for exploiting idle resources from outside operating systems. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 27–40, 2008.

[7] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 61–74, 2012.

[8] Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, and Jay J. Wylie. Efficient eventual consistency in pahoehoe, an erasure-coded key-blob archive. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 181–190, 2010.

[9] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 55–65. ACM Press, 2002.

[10] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*, pages 289–300, 2007.

[11] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the Third Conference on Computing Frontiers, 2006, Ischia, Italy, May 3-5, 2006*, pages 29–40, 2006.

[12] Todd Bezenek, Trey Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles, and Mikko Lipasti. Java TPC-W Implementation Distribution. http://pharm/ece.wisc.edu/tpcw.shtml, 2011.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In

*17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008*, pages 72–81, 2008.

[14] John L. Bruno, José Carlos Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems, ICMCS 1999, Florence, Italy, June 7-11, 1999. Volume II*, pages 400–405, 1999.

[15] Emmanuel Cecchet, Anupam Ch, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. A comparison of software architectures for e-business applications. Technical report, In Proceedings of 4th Middleware Conference, Rio de, 2002.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

[17] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[18] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. SAMR: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, pages 2736–2743, 2010.

[19] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*, pages 56:1–56:11, 2002.

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[21] Ayse Kivilcim Coskun, Richard D. Strong, Dean M. Tullsen, and Tajana Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15-19, 2009*, pages 169–180, 2009.

[22] Kenzo Van Craeynest and Lieven Eeckhout. Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *TACO*, 9(4):32, 2013.

[23] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 55–68, 2010.

[24] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *VLDB*, pages 715–726, 2006.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[27] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*, pages 247–260, 1999.

[28] Fred Douglis, P. Krishnan, and Brian N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing (MLICS'95), Ann Arbor, MI, USA, 10-11 April 1995*, pages 121–137, 1995.

[29] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *USENIX Winter 1994 Technical Conference, San Francisco, California, January 17-21, 1994, Conference Proceedings*, pages 292–306, 1994.

[30] L. Eggert and J.D. Touch. Idletime scheduling with preemption intervals. In *In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 249–262, 2005.

[31] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Commun. ACM*, 55(7):105–114, 2012.

[32] Alexandra Fedorova, David Vengerov, and Daniel Doucette. Operating system scheduling on heterogeneous core systems. In *In Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[33] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In *Replication*, pages 1–17, 2010.

[34] Andrew D. Ferguson, Peter Bodík, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 99–112, 2012.

[35] Rohan Gandhi, Di Xie, and Y Charlie Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proceedings of 2013 USENIX Annual Technical Conference*.

[36] Lei Gao, Michael Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003*, pages 449–460, 2003.

[37] Rajat Garg, Seung Woo Son, Mahmut T. Kandemir, Padma Raghavan, and Ramya Prabhakar. Markov model based disk power management for data intensive workloads. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009*, pages 76–83, 2009.

[38] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[39] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[40] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *In Proceedings of the Usenix Technical Conference UNIX Advanced Computer Systems*, pages 201–212, 1995.

[41] Paul M. Greenawalt. Modeling power management for hard disks. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS '94, pages 62–66, 1994.

[42] Ed Grochowski, Ronny Ronen, John Paul Shen, and Hong Wang. Best of both latency and throughput. In *22nd IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD 2004), 11-13 October 2004, San Jose, CA, USA, Proceedings*, pages 236–243, 2004.

[43] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 2, 2012.

[44] Jorge Guerra, Wendy Belluomini, Joseph S. Glider, Karan Gupta, and Himabindu Pucha. Energy proportionality for storage: impact and feasibility. *Operating Systems Review*, 44(1):35–39, 2010.

[45] Jorge Guerra, Himabindu Pucha, Joseph S. Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, pages 273–286, 2011.

[46] A. Gulati, A. Merchant, and P.J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. *SIGMETRICS Performance Evaluation Review*, 35(1):13–24, June 2007.

[47] Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan de Kleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pages 159–165, 2013.

[48] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin-down for mobile computers. *MONET*, 5(4):285–297, 2000.

[49] Hitachi Global Storage Technologies. Power and acoustics management. White paper at: http://www.hitachigst.com, 2007.

[50] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 263–276, Brighton, United Kingdom, 2005. ACM Press.

[51] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on*

*Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, pages 241–252, 2008.

[52] International Data Corporation (IDC). Digital universe report, 2010. `http://www.emc.com/digital\_universe`.

[53] Sandy Irani, Sandeep K. Shukla, and Rajesh K. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Trans. Embedded Comput. Syst.*, 2(3):325–346, 2003.

[54] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew V. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 261–276, 2009.

[55] Patricia Kim and Mike Suk. Ramp load/unload technology in hard disk drives. White paper at: http://www.hitachigst.com/, 2007.

[56] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. Fair and timely scheduling via cooperative polling. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 103–116, 2009.

[57] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[58] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao.

Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, pages 190–201, 2000.

[59] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, pages 81–92, 2003.

[60] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 64–75, 2004.

[61] Gunho Lee and Randy H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'11, Portland, OR, USA, June 14-15, 2011*, 2011.

[62] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter 1994 Technical Conference, San Francisco, California, January 17-21, 1994, Conference Proceedings*, pages 279–291, 1994.

[63] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M. Shipman, and Chita R. Das. D-factor: a quantitative model of application slow-down in multi-resource shared systems. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems,*

*SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 271–282, 2012.

[64] Yung-Hsiang Lu and Giovanni De Micheli. Adaptive hard disk power management on personal computers. In *9th Great Lakes Symposium on VLSI (GLS-VLSI '99), 4-6 March 1999, Ann Arbor, MI, USA*, page 50, 1999.

[65] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003.

[66] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Experimental Computer Science*, page 6, 2007.

[67] Marshall K. McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*, pages 1–17, 1999.

[68] Joe Meehean, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. CPU Futures: Scheduler support for application management of cpu contention. Technical Report at: http://research.cs.wisc.edu/techreports/2010/TR1684.pdf, 2011.

[69] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In

*Proceedings of the International Conference on Multimedia Computing and Systems, ICMCS 1994, Boston, Massachusetts, USA, May 14-19, 1994*, pages 90–99, 1994.

[70] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*, pages 37–48, 2007.

[71] Ningfang Mi, Alma Riska, Xin Li, Evgenia Smirni, and Erik Riedel. Restrained utilization of idleness for transparent scheduling of background tasks. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance*, pages 205–216, 2009.

[72] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. Efficient management of idleness in storage systems. *TOS*, 5(2), 2009.

[73] X. Mountrouidou, A. Riska, and E. Smirni. Adaptive workload shaping for power savings on disk drives. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, pages 109–120. ACM, 2011.

[74] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, pages 253–267, 2008.

[75] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. Write off-loading: Practical power management for enterprise storage. In *FAST*, pages 253–267, 2008.

[76] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony I. T. Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 145–158, 2009.

[77] Jason Nieh and Monica S. Lam. A SMART scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21(2):117–163, 2003.

[78] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB*, pages 126–137, 1999.

[79] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *EuroSys*, pages 13–26, 2009.

[80] Jerry Peek, Tim O'Reilly, and Mike Loukides. Unix power tools. 1998.

[81] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[82] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, pages 253–265, 2009.

[83] Jorda Polo, David Carrera, Yolanda Becerra, Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Performance management of accelerated mapreduce workloads in heterogeneous clusters. In *39th International Conference*

*on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 653–662, 2010.

[84] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 105–120, 2005.

[85] Dinesh Ramanathan, Sandy Irani, and Rajesh K. Gupta. Latency effects of system level power management algorithms. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*, pages 350–356, 2000.

[86] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 4, 2012.

[87] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S. McKinley. Exploiting processor heterogeneity in interactive services. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pages 45–58, 2013.

[88] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 97–102, 2006.

[89] Alma Riska and Evgenia Smirni. Autonomic exploration of trade-offs between power and performance in disk drives. In *Proceedings of the 7th International*

*Conference on Autonomic Computing, ICAC 2010, Washington, DC, USA, June 7-11, 2010*, pages 131–140, 2010.

[90] Seagate Technology. Constellation ES: High capacity storage designed for seamless enterprise integration. Product overview at: http://www.seagate.com, 2009.

[91] Seagate enterprise capacity 3.5 hdd v4 serial ata, 2014. Product manual at: http://www.seagate.com.

[92] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2000.

[93] Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[94] Daniel Shelepov and Alexandra Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[95] Aameek Singh, Madhukar R. Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, page 53, 2008.

[96] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, pages 169–184, 2009.

[97] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving storage system availability with d-graid. *TOS*, 1(2):133–170, 2005.

[98] PassMark Software. CPU Benchmarks. http://www.cpubenchmark.net/, 2013.

[99] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[100] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *CCGRID*, pages 102–110, 2002.

[101] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: practical power-proportionality for data center storage. In *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, pages 169–182, 2011.

[102] Eno Thereska, Jiri Schindler, John S. Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A framework for building unobtrusive disk maintenance applications (awarded best student paper!). In *Proceedings of the FAST '04 Conference on File and Storage Technologies, March 31 - April 2, 2004, Grand Hyatt Hotel, San Francisco, California, USA*, pages 213–226, 2004.

[103] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 213–224. IEEE Press, 2012.

[104] Anil Vasudeva. Are SSDs ready for enterprise storage systems? White paper at: http://www.snia.org/, 2011.

[105] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pages 235–244, 2011.

[106] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Play it again, simmr! In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*, pages 253–261, 2011.

[107] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. In *Middleware 2011 - ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, pages 165–186, 2011.

[108] Abhishek Verma, Ludmila Cherkasova, Vijay S. Kumar, and Roy H. Campbell. Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle. In *NOMS*, 2012.

[109] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *8th USENIX*

*Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 267–280, 2010.

[110] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

[111] Matthew Wachs and Gregory R. Ganger. Co-scheduling of disk head time in cluster-based storage. In *SRDS*, pages 278–287, 2009.

[112] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, pages 134–143, 2011.

[113] Qingyang Wang, Yasuhiko Kanemasa, Motoyuki Kawaba, and Calton Pu. When average is not average: large response time fluctuations in n-tier systems. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 33–42, New York, NY, USA, 2012. ACM.

[114] C. Weddle, M. Oldham, J. Qian, and A. Wang. PARAID:a gear-shifting power-aware raid. In *Proceedings of 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 245–269, 2007.

[115] T. White. *Hadoop:The Definitive Guide*. Page 6,Yahoo Press.

[116] Jiong Xie et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proc. of the IPDPS Workshops: Heterogeneity in Computing*, 2010.

[117] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Storage workload isolation via tier warming: How models can help. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.*, pages 1–11, 2014.

[118] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Proactive management of systems via hybrid analytic techniques. In *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 137–148. IEEE, 2015.

[119] N.J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX conference on File and storage technologies*, pages 14–14. USENIX Association, 2010.

[120] Feng Yan, Ludmila Cherkasova, Zhuoyao Zhang, and Evgenia Smirni. Heterogeneous cores for mapreduce processing: Opportunity or challenge? In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–4, 2014.

[121] Feng Yan, Ludmila Cherkasova, Zhuoyao Zhang, and Evgenia Smirni. Optimizing power and performance trade-offs of mapreduce job processing with heterogeneous multi-core processors. In *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 240–247, 2014.

[122] Feng Yan, Ludmila Cherkasova, Zhuoyao Zhang, and Evgenia Smirni. Dyscale: a mapreduce job scheduler for heterogeneous multicore processors. *IEEE Transactions on Cloud Computing*, to appear 2016.

[123] Feng Yan, Shannon Hughes, Alma Riska, and Evgenia Smirni. Overcoming limitations of off-the-shelf priority schedulers in dynamic environments. In *MASCOTS 2013, 21st IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, CA, USA, August 14-16, 2013*, pages 505–514, 2013.

[124] Feng Yan, Shannon Hughes, Alma Riska, and Evgenia Smirni. Agile middleware for scheduling: meeting competing performance requirements of diverse tasks. In *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, pages 185–196, 2014.

[125] Feng Yan, Xenia Mountrouidou, Alma Riska, and Evgenia Smirni. Copy rate synchronization with performance guarantees for work consolidation in storage clusters. *SIGMETRICS Performance Evaluation Review*, 39(3):82–86, 2011.

[126] Feng Yan, Xenia Mountrouidou, Alma Riska, and Evgenia Smirni. Toward automating work consolidation with performance guarantees in storage clusters. In *MASCOTS 2011, 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Singapore, 25-27 July, 2011*, pages 326–335, 2011.

[127] Feng Yan, Xenia Mountrouidou, Alma Riska, and Evgenia Smirni. Quantitative estimation of the performance delay with propagation effects in disk power savings. In *HotPower'12 (in conjunction with OSDI'12), Hollywood, CA, USA, October 7, 2012*, 2012.

[128] Feng Yan, Xenia Mountrouidou, Alma Riska, and Evgenia Smirni. Prefigure: an analytic framework for hdd management. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)*, to appear 2016.

[129] Feng Yan, Alma Riska, and Evgenia Smirni. Busy bee: how to use traffic information for better scheduling of background tasks. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*, pages 145–156, 2012.

[130] Feng Yan, Alma Riska, and Evgenia Smirni. Fast eventual consistency with performance guarantees for distributed storage. In *32nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2012), Macau, China, June 18-21, 2012*, pages 23–28, 2012.

[131] Feng Yan, Alma Riska, and Evgenia Smirni. Toward fast eventual consistency with performance guarantees. In *9th International Conference on Autonomic Computing, ICAC'12, San Jose, CA, USA, September 16 - 20, 2012*, pages 167–172, 2012.

[132] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 265–278, 2010.

[133] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 29–42, 2008.

[134] Ronghua Zhang, Tarek F. Abdelzaher, and John A. Stankovic. Kernel support for open qos-aware computing. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), May 27-30, 2003, Toronto, Canada*, pages 96–105, 2003.

[135] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Benchmarking approach for designing a mapreduce performance model. In *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*, pages 253–258, 2013.

[136] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Exploiting cloud heterogeneity for optimized cost/performance mapreduce processing. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms, CloudDP@EuroSys 2014, Amsterdam, The Netherlands, April 13, 2014*, pages 1:1–1:6, 2014.

[137] M. Zhao and R.J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, VTDC '07, pages 5:1–5:8, 2007.

[138] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, and Y. Zhou. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 118–129, Febuary 2004.

[139] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.