Dynamic Storage Provisioning with SLO Guarantees

by

Prashant Gaharwar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Static provisioning of storage resources may lead to over-provisioning of resources, which increases costs, or under-provisioning, which runs the risk of violating application-level QoS goals. Toward this end, virtualization technologies have made automated provisioning of storage resources easier allowing more effective management of the resources. In this work, we present an approach that suggests a series of dynamic provisioning decisions to meet the I/O demands of a time-varying workload while avoiding unnecessary costs and Service Level Objective (SLO) violations. We also do a case-study to analyze the practical feasibility of dynamic provisioning and the associated performance effects in a virtualized environment, which forms the basis of our approach. Our approach is able to suggest the optimal provisioning decisions, for a given workload, that minimize cost and meet the SLO. We evaluate the approach using workload data obtained from real systems to demonstrate its cost-effectiveness, sensitivity to various system parameters, and runtime feasibility for use in real systems.

Acknowledgements

I would like to thank my supervisor, Prof. Kenneth Salem for his patient guidance, insightful advices, and constant encouragement. I would also like to thank Prof. Ashraf Aboulnaga and Prof. Johnny W. Wong for being my thesis readers.

I am also thankful to the University of Waterloo staff who have been a great help throughout my graduate studies.

Dedication

Dedicated to Ma, Pa, and Da.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Manual provisioning of storage resources, to meet changing I/O workload demands, is error-prone, not always feasible, and cumbersome [2]. In modern IT data-centers, time-varying resource demands of typical enterprise applications, due to temporal variations of their workloads, lead to under-provisioning or over-provisioning of resources. In case the resources are over-provisioned, the cost incurred increases. In case the resources are under-provisioned, there may be violations of Service Level Objectives (SLOs). A modern approach to optimize resource allocation and improve resource utilization is to consolidate applications in a shared infrastructure using virtualization and employ dynamic resource allocation strategies to balance the seemingly conflicting goals of meeting SLOs and minimizing resource costs.

Storage virtualization forms the backbone of such environments, with storage arrays providing consolidated data access to multiple applications simultaneously. In such arrays, one or several *physical* disks could possibly support a single *virtual* disk. Requests coming to a virtual disk are distributed among the physical disks and processed in parallel, so increasing or decreasing the number of physical disks backing the virtual disk would result in an increase or decrease in I/O performance. This fact could be exploited in data-centers, which experience variable I/O workload demands and call for dynamic allocation strategies for storage.

Changing the configuration of resources to meet resource demands has been studied widely (e.g. [18, 25, 1]). Unlike other resources, e.g. CPU and memory, which can be configured on the fly (in virtualized environments), changing the configuration of storage resources is a time-consuming process because of the significant amount of data-transfer involved during provisioning. Data transfer is required because, if an additional disk is introduced into the storage system, it can not start serving requests right away. It has to have data

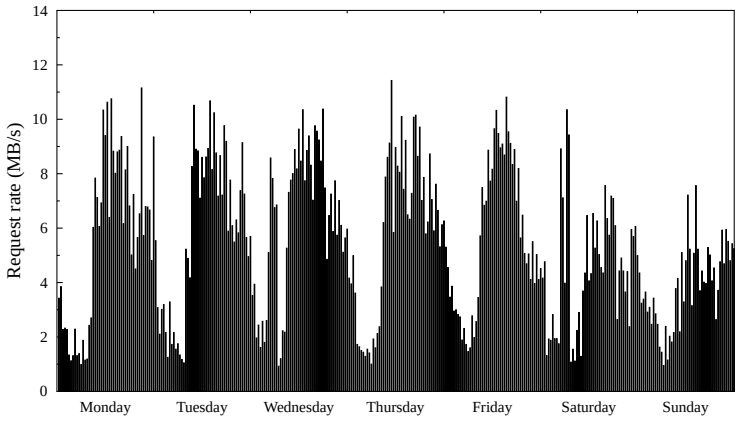


Figure 1.1: I/O workload pattern of a typical week at a UW file server

for serving requests. This data (part of the total data residing on the storage system) is moved from other storage devices already in the system. Similarly, if we remove a disk from the storage system, the data residing on this disk should be moved to the disks that would remain in the system. Depending on the size of the data to be moved, this data migration may last for hours.

In case of dynamically changing I/O workloads, one cannot afford to wait for long-periods for a desired storage configuration to be provisioned, as it may lead to SLO violations in the interim. To offset this provisioning time, one must start the provisioning in anticipation of a future change in workload. Thus, we base our study on the assumption that knowledge of the future workload is provided to us, so that we can determine optimal provisioning decisions that minimize the provisioned resources while meeting the SLOs. Typical I/O workload patterns in data-centers have been found to be cyclic in nature and easily lend themselves to be *predictable* [22, 21]. We found a similar cyclic pattern in I/O request rates of a typical week at a *University of Waterloo (UW) file server*, which are illustrated in Figure 1.1. Therefore, cyclic, predictable workloads in data-centers are possible use cases for our technique, which requires advance knowledge of the workload.

In this work, we build an approach that takes a model of the upcoming workload for a certain time-period, and emits a schedule of provisioning and deprovisioning actions that must be initiated during the given time-period to meet a given SLO, while minimizing the provisioned resources. Specifically, the schedule indicates the number of physical disks that should be backing a virtual disk at each point in time. Changes in the scheduled number of physical disks are implemented by provisioning and deprovisioning operations.

The schedule is such that the performance of the system will be within the requirements of the SLO throughout the time period for which the workload model is given. In addition, the number of physical disks required is minimized. We make the following contributions:

- An exhaustive depth-first search (DFS) technique to find an optimal schedule for dynamically provisioning storage resources.
- Two case-studies of the characteristics of migration mechanisms, so that these characteristics can be accounted for by the optimization technique.
- Search space pruning strategies for increasing the performance of the search algorithm.
- Experimental analysis of the developed approach, using I/O traces from real systems, to demonstrate its efficacy and applicability to real scenarios.

The organization of the rest of the thesis is as follows. Chapter 2 gives the general architecture of a system designed to tackle our problem, followed by Chapter 3 with discussion and presentation of two case-studies to characterize storage provisioning mechanisms. In Chapter 4, we present the specifics of the problem and the approach developed to tackle it. Chapter 5 analyzes the approach for efficiency and effectiveness. We discuss the related work in Chapter 6 and conclude with directions on future work in Chapter 7.

Chapter 2

System Architecture

The problem we have tackled in this work is an instance of a general problem, which is to provision storage resources among n machines such that each machine's SLO is met. We specifically address the problem of minimizing the number of disks provisioned to serve requests, while ensuring that SLO requirements are met. The general architecture of a system designed to tackle the storage provisioning problem is illustrated in Figure 2.1. The rectangular boxes indicate a software or a hardware module of the system that interacts with the input data (represented as a curved gray box) and emits some output (curved gray box), which may be used by another module of the system. The arrows indicate the flow of data or information from one module of the system to another.

As depicted in the figure, the input to the *optimizer* is a workload model, which is a description of upcoming (future) workload. In addition, the SLO specifies the I/O performance that must be guaranteed. The optimizer is responsible for emitting a *schedule* of (de)provisioning actions that will ensure that the SLO is met while minimizing the number of provisioned storage resources. This schedule is fed to the *resource controller*, which effects the actual changes required in the storage system according to the schedule. Brief discussions of various aspects of the system follow.

Workload. The workload is the stimulus applied to a system, application, or component that prompts usage of system resources. An I/O workload is a sequence of block I/O requests. Such a request sequence can be characterized by a set of statistics, describing attributes of the requests, such as I/O request size or I/O request rate. In our work, we assume that a model of the upcoming workload is given. Further details are presented in Chapter 4. Such a workload model could be obtained by, for example, prediction based on past workload measurements [5, 9, 25].

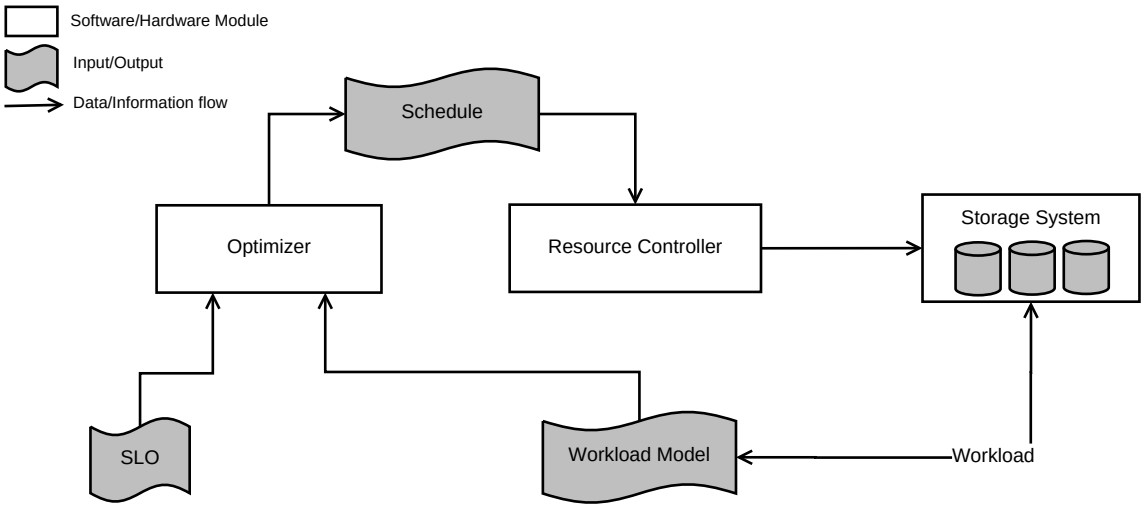


Figure 2.1: System Architecture

SLO. Service level objectives (SLOs) are a means of specifying the quality of a service or application. They are generally a combination of some measurable quantities indicating the performance of the system, e.g. availability, throughput, response time, utilization. To meet SLOs, resources must be provisioned. Thus, to achieve a balance between the conflicting goals of minimizing number of provisioned resources and meeting the SLO, an optimization strategy is needed. Finding such a strategy is the task of the *optimizer* module.

Optimizer. The optimizer is responsible for taking as input the workload model and emitting a **schedule**. A schedule indicates when to provision or deprovision storage resources to meet the current or future workload demands. The schedule emitted by the optimizer ensures that the SLO is met while minimizing the resource provisioning cost. Generally, an *online* optimizer, which only has the past workload data at its disposal, would try to provision resources based on the current demand being experienced by the system. On the other hand, an *offline* optimizer, which is given advance knowledge of all future requests, would try to come-up with a resource provisioning schedule that minimizes the cost over that whole period and meets the SLOs as well. Our main goal is to build an *offline* optimizer in this work.

Resource Controller. This module is responsible for carrying out the actual provisioning of the storage resources based on the schedule fed to it as input. The resource controller could be a series of system calls or OS-dependent scheduling jobs, e.g. *cron* jobs in Linux. Most prior works [2, 21, 1] use their own custom job controllers to control provisioning tasks. Third-party tools, like *svmotion* by **VMware Inc.** and the live migration tool in

Xen [4], could also be used if the host system supports them. We present some empirical data obtained by using *svmotion* in our experimental test-bed in one of the case-studies presented in the next chapter.

Storage System. The storage system is the hardware which stores data and handles requests (I/O workload) to read and write data. It often contains several physical disks as in, for example, a RAID system. The disk arrays are generally mapped to logical disks, which serve as the logical layer for disk access by the applications. Also, a storage system could be redesigned, which may involve reconfiguration of physical disks and their mapping to logical disks, by provisioning commands from the resource controller.

With the above discussion, we aimed to give an overview of various modules and their interactions in our system. Next, we discuss provisioning mechanisms in greater detail, as we wish to develop an optimizer that can take the characteristics of such mechanisms into account when choosing a provisioning schedule.

Chapter 3

Dynamic Storage Provisioning

In the previous chapter, we looked at the general architecture of our system. It includes a *resource controller*, which controls dynamic provisioning of storage resources, based on the *schedule* fed to it. The storage system implements provisioning, under the control of the resource controller. Although the design of a provisioning mechanism is not the focus of this work, we want to characterize (or, model) the provisioning mechanism so that the optimizer can take its characteristics into account when it produces a provisioning schedule. Toward this end, we present brief case studies of two provisioning mechanisms. Based on these examples, we present a simple provisioning model, which is used by the optimizer described in Chapter 4.

We assume that the storage system implements *logical storage volumes*, each of which maps to one or more underlying physical devices. This sort of storage model is widely used in modern data centers [6]. We also assume that the devices are identical. Dynamic storage provisioning is implemented by increasing or decreasing the number of physical devices backing a logical volume, which can serve several purposes. It can be used to adjust the capacity (size) of a volume. It can also be used to adjust the I/O throughput that can be supported by the volume. As our primary purpose is in reacting to a time varying I/O request load, we are concerned with the latter motivation for provisioning. Next, we present two case studies as examples of dynamic storage provisioning mechanisms.

3.1 Dynamic Storage Arrays

This case-study was done to study the provisioning characteristics of the storage systems that support dynamic provisioning by restructuring of the storage array. We did this case-study on a specific storage system product by **IBM**, *DS4200*. It has an array of physical

Volume Size	Migration Time	Source disks	Target disks
100 GB	2h50m	1	2
100 GB	2h55m	2	3
100 GB	2h50m	3	4
500 GB	14h45m	1	2
500 GB	14h50m	2	3
500 GB	14h50m	3	4

Table 3.1: Migration times for array expansion

disks. The disk array is formatted as a RAID and then this RAID is partitioned into one or several storage volumes. Each volume is striped across all physical devices in the RAID-system. The storage system supports an *expansion* operation that allows the number of physical devices that implement an array to be increased from, say k to $k+1$. The expansion operation can be done on the fly, without taking the volume off-line. During provisioning, volume data is redistributed so that it resides evenly over $k+1$ disks instead of k disks. Unfortunately, this particular storage system did not support a *compression* operation for reducing the size of the array from, say k physical disks to $k-1$. Since, it did not support a deprovisioning operation, we only did a limited set of experiments for this system.

We wanted to get an idea of the time taken by the expansion operation and the parameters it depends on. For this study, we used the following methodology:

1. Create a volume of size p on a RAID-0 system,
2. Expand the array from k to $k+1$ disks (initially, k = 1),
3. Repeat from step 2, till $k+1$ reaches 4,
4. Repeat from step 1, for various values of p.

The results are shown in Table 3.1. Observe that the migration time does not depend on the number of devices in the array. Also, it scales proportionally with the size of the volume. Thus, the migration takes a long time for large volumes.

We also found that this data redistribution generates extra I/O overhead, and thus, underlying physical devices must support both volume's regular I/O workload as well as this overhead. Although, unfortunately, this particular storage system did not support a *compression* operation, we believe other storage systems should exist that can support

this operation. To address the need for a mechanism with the ability to both increase or decrease the number of physical devices, we present another case-study.

3.2 Virtual Device Migration

We formatted a volume (on the DS4200) with *VMFS*[1]filesystem. As stated earlier, the storage system implements volumes as striped arrays of physical devices. Any file residing on the filesystem is also distributed across all devices. One can host a VM on this filesystem. Each VM has a virtual disk, implemented as a file, which resides on the filesystem. VMware provides a utility, *svmotion* to allow live-migration (which is desirable, to prevent disruption of a running application) of virtual disks from one file-system to another across a network. Thus, by migrating a virtual disk file from a file-system backed by k_1 disks to a file-system backed by k_2 disks, we can effect dynamic provisioning. If $k_1 < k_2$, we increase the I/O capacity of the virtual disk, and vice versa.

An experiment to study and characterize the behavior of this mechanism was designed on our test-bed. We created a VM with 2GB memory, with a virtual disk, initially hosted on a file-system backed by a single physical device. We also created *three* other filesystems, backed by 2, 3, and 4 devices, respectively. To study the effect on actual workload (application performance) during and after migration, we created a workload generator to run in the VM. The workload generator was a simple multi-threaded program with several threads reading random blocks ($4K$ each) of a 2 GB file stored on the virtual disk, thus generating a random I/O workload at the virtual disk. The experiment's methodology was:

1. Launch VM and workload generator,
2. Measure I/O performance when virtual disk is backed by k devices (initially, *one* device),
3. Migrate virtual disk from k to $k+1$ devices using *svmotion*, measuring the time taken to migrate and the workload's performance,
4. Repeat from 2, till $k + 1$ reaches 4.

[1]VMFS (Virtual Machine File System) is **VMware Inc.**'s cluster file system. It is used by **VMware ESX Server** to store virtual machine disk images. Multiple servers can read/write the same filesystem simultaneously. VMware ESX is an enterprise-level virtualization OS product offered by VMware, Inc. that can be used as a host-server for several guest virtual machines.

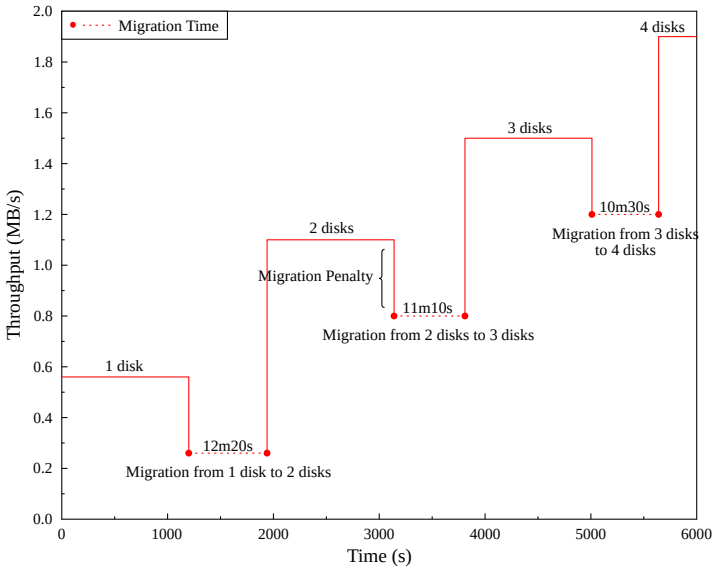


Figure 3.1: *Virtual Device Migration* case-study

The results of our experiment are illustrated in Figure 3.1. As expected, the application performance (throughput) increases with the increase in the number of physical devices backing the file-system. We also see a dip in performance during every migration because of the extra I/O workload generated due to data transfer. Also, the dip in performance, labeled *migration penalty*, largely remains constant at every migration instance. Observe, the migration time taken to go from 1 to 2 disks is slightly more than the time taken to go from 2 to 3 disks. Similarly, the time taken to go from 3 to 4 disks is less than that going from 2 to 3 disks. It indicates that the migration time depends on the number of disks involved in migration and as they increase the migration gets faster. A more important factor in determining migration times was found to be the *size* of the virtual disk. We present a few readings of the migration times with increasing size of virtual disk in Table 3.2. The methodology followed was similar to the one stated earlier and is as follows:

1. Launch VM of size p and the workload generator,
2. Migrate virtual disk from k to $k+1$ devices using *svmotion*, measuring the time taken to migrate,
3. Repeat from 1 for various values of p.

Observe that the migration time increases almost linearly with the increase in the size of the virtual disk. To summarize, we make three main observations from the results obtained

Size	Migration Time	Source disks	Target disks
20 GB	10m30s	3	4
50 GB	24m50s	3	4
100 GB	45m10s	3	4

Table 3.2: Change in migration time with size of virtual disk

from the two case-studies and present them in the next section as characteristics to model dynamic storage provisioning.

3.3 Modeling Dynamic Storage Provisioning

We can produce a simple model of dynamic storage provisioning by abstracting from the two case-studies. We note three key elements of the model to be considered in our optimizer:

1. **Provisioning Time.** From the second case-study we observe that the migration time from one set of disks to another largely depends on the size of the virtual-disk (amount of data) being migrated. We also see a decreasing pattern in the amount of migration time as the number of disks increase. For simplicity, in our optimizer, we assume that the migration time (m) remains constant for any number of devices migrating to any other number. However, our optimizer could easily be extended to consider variable provisioning times. These times could be represented using a matrix of dimensions $k_{max} * k_{max}$, where k_{max} is the practical limit to the number of disks in the system. If the rows represent *from (source)* disks and the columns represent *to (destination)* disks in the matrix, then each element, $\kappa_{i,j}$ represents *the time needed to migrate a virtual disk of size 1GB from i disks to j disks*. From the matrix, we could obtain migration times for virtual disks by simply multiplying $\kappa_{i,j}$ by a scaling factor proportional to the size of the virtual disk.

2. **Provisioning Overhead.** As is clear from the case-studies, the throughput suffers a drop during the process of migration (because of the extra IOs generated by the data-movement). Thus, we must introduce a factor of *migration penalty*, λ_m, as an input to the optimizer and consider it in every migration. Migration penalty is the additional load experienced by a disk during a migration. We treat the penalty as a constant, which is consistent with our observations from the case studies.

3. **Provisioning Cost.** Recall that our optimizer will minimize cost, which is defined in terms of the number of physical disks being used. When the system is operating with k_1 disks, it is clear that the cost is k_1. When migrating from k_1 to k_2, we need to

define cost during the migration. We define it as the number of physical devices that are involved in the provisioning operation. In the first case-study, the data movement was only restricted to $k_1 + 1$ devices (i.e. if we are going from k_1 to $k_1 + 1$ devices). Had there been a mechanism to go from, say k_1 to $k_1 - 1$ devices, the total number of devices involved would have been k_1. So, we see that provisioning mechanisms that rely on restructuring of the array (by expansion or contraction) as in the first case-study, would have a provisioning cost of $max(source, destination)$ (maximum of the number of source and destination disks). On the other hand, in the second case-study every migration involved data movement from a set of source disks (k_1) to a different set of destination disks (k_2). Thus, the provisioning cost in such cases would be $sum(k_1, k_2)$ (sum of the number of source and destination disks). Since, different mechanisms have different notions of cost, the optimization technique we present in the next chapter does not depend on a specific cost model. Instead, any appropriate function of the numbers of source and destination disks can be used.

Chapter 4

Optimizing Dynamic Provisioning

In this chapter, we describe our dynamic provisioning optimization problem and the optimizer, which is meant to solve the problem. Recall from Chapter 2 that the optimizer takes a workload model and an SLO and produces a provisioning schedule.

4.1 Problem

The problem we aim to solve is: *Given a description of the workload model, generate a migration schedule which meets the service level objective (SLO) while minimizing the storage system cost.* In this section, we look at the specifics of the problem, i.e. an abstract model of the storage system, the notion of time in our problem, the kind of workload description that we will use, the schedule that is to be emitted, the specifics of the SLO, and a definition of cost (not necessarily in this order). We restate the problem, towards the end of this section, after these specifics have been defined.

Storage System Model. We present a model of the storage system abstracted from the discussion in Chapter 3, on which our solution is based. The model is a k-server queuing system (Figure 4.1(b)), with servers representing the underlying physical disks. Arrival rate and service times are time varying. The number of servers can also vary over time, as a result of the provisioning and deprovisioning operations which we are trying to schedule. We also assume that the load is distributed evenly across k-devices (illustrated in Figure 4.1).

So, the storage system is characterized by k (varies with time) and the service time. In addition, as discussed in the previous chapter, the storage system is also characterized by a set of provisioning parameters: migration time (m), migration penalty (λ_m), and migration

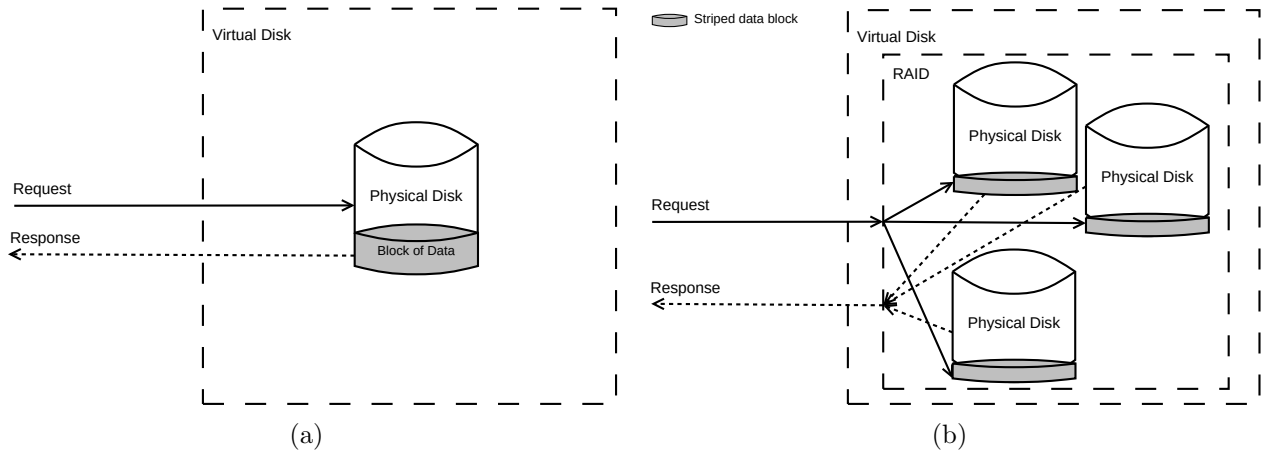


Figure 4.1: Request processing in a single disk and parallel-disks system

cost (C). These parameters describe the behavior of the system during provisioning and deprovisioning operations.

Time. We treat time, t, as a sequence of discrete fixed-length (t_f) intervals. We define

$$\tau = \lceil T/t_f \rceil \tag{4.1}$$

In the above equation, T is the time-period for which we need to find a schedule and thus, τ is the number of discrete time intervals[1]of length t_f in T. It also follows, in our context, $t \, \epsilon \, [1, \tau]$. Arrival rate ($\lambda(t)$) and service rate ($\mu(t)$) are fixed during each interval, but may vary from interval to interval.

Workload. The input to the optimizer consists of two vectors of length τ. One vector defines the service rate of a disk ($\mu(t)$), the other describes the request arrival rate ($\lambda(t)$). Figure 4.2 illustrates an example of the request rate ($\lambda(t)$) vector for a week at a UW file server with $t_f = 30$ minutes and $\tau = 336$. Further details about the workload are presented in Chapter 5.

We treat all of the disks as identical, so their service rates are also the same. Also, as stated earlier, we assume that the load is evenly distributed among the disks. The *offered utilization* ($U'(t)$) of a storage system with k disks at t is given by:

$$U'(t) = \frac{\lambda(t)}{k(t)\mu(t)} \tag{4.2}$$

[1]We use the terms *time interval* and *timeslot* interchangeably.

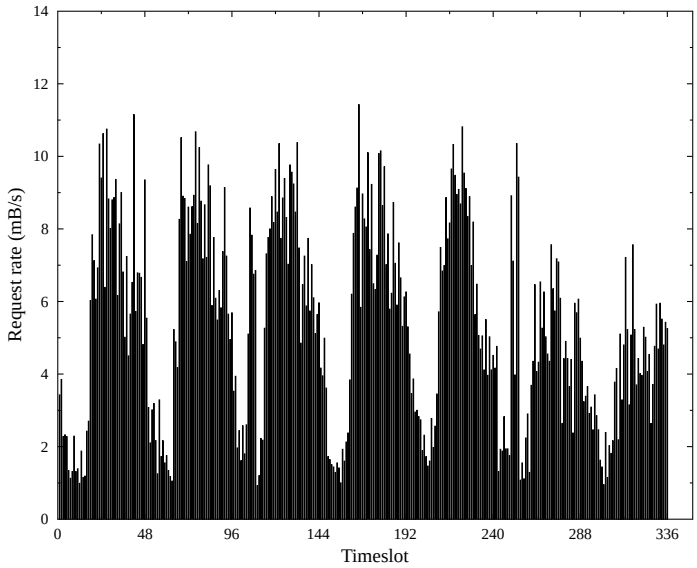


Figure 4.2: Request rate ($\lambda(t)$) at a UW file server

In addition, during migration, an extra load is experienced by the devices due to data transfer. We define λ_m as the extra load experienced by a device during migration, thus, the actual utilization ($U(t)$) of k disks is given by:

$$U(t) := \begin{cases} U'(t) & \text{when not provisioning} \\ U'(t) + \frac{\lambda_m}{k(t)\mu(t)} & \text{during provisioning} \end{cases} \tag{4.3}$$

Service Level Objective (SLO). Utilization is often used by data-center operators to vicariously control application performance because of the monotonic relationship between the two [17]. We define the SLO in terms of keeping the *utilization* ($U(t)$) of storage system devices below a specified threshold (U_{ref}), where U_{ref} is the *reference* utilization. We also introduce a trade-off factor, v, in the SLO to allow a trade-off between performance and cost. We define v as the percentage of time intervals during which the reference utilization may be exceeded. Thus, if v is very small the optimizer will ensure that the U(t) is almost always below U_{ref}, possibly at the expense of provisioning additional devices. As v is increased, the optimizer is permitted more violations of the reference threshold, which may allow it to provision fewer devices. We also define v in terms of a positive integer V as:

$$V = \lceil v\tau \rceil$$

V is simply the number of time intervals during which the U_{ref} may be exceeded. So, we restate the SLO statement as, "Keep $U(t)$ below U_{ref} with at most V violations".

Schedule. The output of the *optimizer* is a *schedule* that indicates the following:

1. The number of devices that are available to serve requests at each time-interval t, which is represented by vector $k = [k(1), k(2), ...k(\tau)]$.
2. The time-interval(s) at which provisioning, if any, should start and end, and the number of devices that would be available after the provisioning ends.

The above can be extracted from a schedule vector S, which we define shortly. We also use the supplied parameter, m (migration time) to determine the amount of time needed to complete a provisioning step. m is given in terms of the number of time-intervals (t) needed to complete a migration. A time-interval t during which provisioning is taking place is called a *migration slot*.

We define a schedule $S = [s(1), s(2), ...s(\tau)]$. Each $s(t)$ represents the number of disks available during the t^{th} time interval, i.e. $s(t) = k(t)$, unless t represents a migration slot. In that case $s(t)$ indicates the number of devices that will be available once the provisioning operation is complete.

Consider the following example (Example 4.1) for clarity. Suppose the optimizer emits a schedule $S = \{1, 2, 2, 2, 1, 1, 1, 1\}$ with $m = 2$. In S, $s(2) \neq s(1)$ and $s(5) \neq s(4)$, indicating the beginnings of a migration at $t = 2$ and $t = 5$, respectively. Each migration would last for $m = 2$ time-intervals, i.e. up to $t = 3$ and $t = 6$ respectively. Thus, $t = 2, 3, 5, 6$ are all *migration slots*. The vector k corresponding to S is $\{1, 1, 1, 2, 2, 2, 1, 1\}$, representing the number of devices available for serving requests.

Next, we define a few kinds of schedule that would be useful while discussing the optimizer:

- *x-Feasible Schedule.* A schedule S is said to be x-feasible if all pairs of provisioning operations in S have starting points separated by at least x timeslots. For example, schedule S in Example 4.1 has 2 provisioning operations, one starting at $t = 2$, the other starting at $t = 5$. Thus, S is 1-feasible, 2-feasible and 3-feasible, but not $4-feasible$. Intuitively, an x-feasible schedule can be implemented by a provisioning mechanism with a migration time (m) of x or less.
- *Valid Schedule.* A schedule that satisfies the SLO. A schedule satisfies the SLO, if for all time-intervals, except for V of them, the number of available disks (k) is sufficient

to keep utilization below U_{ref}. In Example 4.1, we can find $U'(t)$ (in case of t is not a migration slot, i.e. at $t = 1, 4, 7, 8$) and $U(t)$ (in case of t is a migration slot, i.e. at $t = 2, 3, 5, 6$) from Equation 4.2 and Equation 4.3 respectively, for any given workload data and migration penalty, λ_m. If $v = 10\%$, then the utilization obtained should not exceed a given U_{ref}, in more than 1 of the time-intervals ($\tau = 8$).

- *Optimal Schedule.* A schedule which is m-feasible and valid with minimal cost (the *cost* of a schedule is defined a little later in the discussion and can be obtained from Equation 4.6).
- *Suboptimal Schedule.* A schedule which is m-feasible and valid, and is not an optimal schedule.

Cost. We define the cost at each time interval, $C(t)$, as the total number of disks required at that interval. In case there is no migration, the cost would be the number of disks available at that time-slot. Since, migration involves provisioning or deprovisioning to a different number of disks, the cost changes. As discussed in Chapter 3, the *migration cost* varies with the migration mechanism used. Thus, during a migration slot, the cost $C(t)$ using a migration scheme similar to the one in the second case-study (*Section 3.2*) would be:

$$C(t) = s(t) + d \tag{4.4}$$

where d is the number of devices that were available before the provisioning operation began. Similarly, we can define $C(t)$ during a migration slot for a migration scheme like the one in the first case-study (Section 3.1):

$$C(t) = max(s(t), d) \tag{4.5}$$

The total cost of a schedule ($Cost(S)$) is defined as the average number of disks required per time-slot, i.e.

$$Cost(S) = \frac{\Sigma_{t=1}^{\tau} C(t)}{\tau} \tag{4.6}$$

To illustrate further, the cost vector $C(t)$ for the schedule from Example 4.1 is $\{1, 3, 3, 2, 3, 3, 1, 1\}$ using the cost model based on the second case study and is $\{1, 2, 2, 2, 2, 2, 1, 1\}$ using the cost model from the first case study.

Now, equipped with these definitions, we restate our problem statement as: *Given* $\lambda(t)$, $\mu(t)$, U_{ref}, V, λ_m, *and* m, *find an m-feasible, valid schedule* S *such that* $Cost(S)$ is minimized. Table 4.1 summarizes the notation used in our problem.

Notation	Description
$\lambda(t)$	Request rate vector
$\mu(t)$	Service rate vector
t	Time interval or timeslot
t_f	Time interval length or timeslot length
$T(\tau)$	Time period (Schedule length in time intervals)
U_{ref}	Reference utilization
$U'(t)$	Offered utilization
$U(t)$	Total utilization
V	Number of violations allowed
p_{max}	Maximum number of disks allowed in the system
m	Migration time (in time-intervals)
λ_m	Migration penalty (stated as extra load per disk)
S	Schedule vector of $s(t)$
$k(t)$	Available devices in the system to serve requests at t
$C(t)$	Cost at each time interval
$Cost(S)$	Average cost of a schedule

Table 4.1: Summary of notations used

4.2 Optimizer

Here, we explain the exhaustive approach used by the *optimizer* to find an *optimal schedule* and present several pruning techniques for performance improvement.

Exhaustive Depth-first Search (DFS). The idea behind exhaustive approach is to consider every possible *m-feasible schedule* and check whether it is an *optimal schedule*. The approach employs a depth-first search (DFS) into the search graph and progressively finds better *suboptimal schedules* until an optimal schedule is found. We restrict the search-space by a parameter k_{max}, which is the maximum number of physical disks allowed in the system. To give an idea about the structure of the search graph, consider the parameters k_{max} and τ (the length of the schedule). The optimizer tries to fit every possible value from $[1, k_{max}]$ at every time-slot in $[1, \tau]$ that gives an m-feasible schedule. The search graph for *2-feasible schedules*, for $k_{max} = 4$, $\tau = 5$, and $m = 2$ is illustrated in Figure 4.3. In the figure, if the *rows* represent *disks* and the *columns* represent *time-intervals* then, we can see there is a directed edge from every cell (or node or disk configuration) to the cells in the same row in the adjacent column (next time-interval), representing *no migration*. An edge from a node to a node in a different row represents a migration (change in the number of disks) and the edge spans exactly $m = 2$ time-intervals, corresponding to the fact that

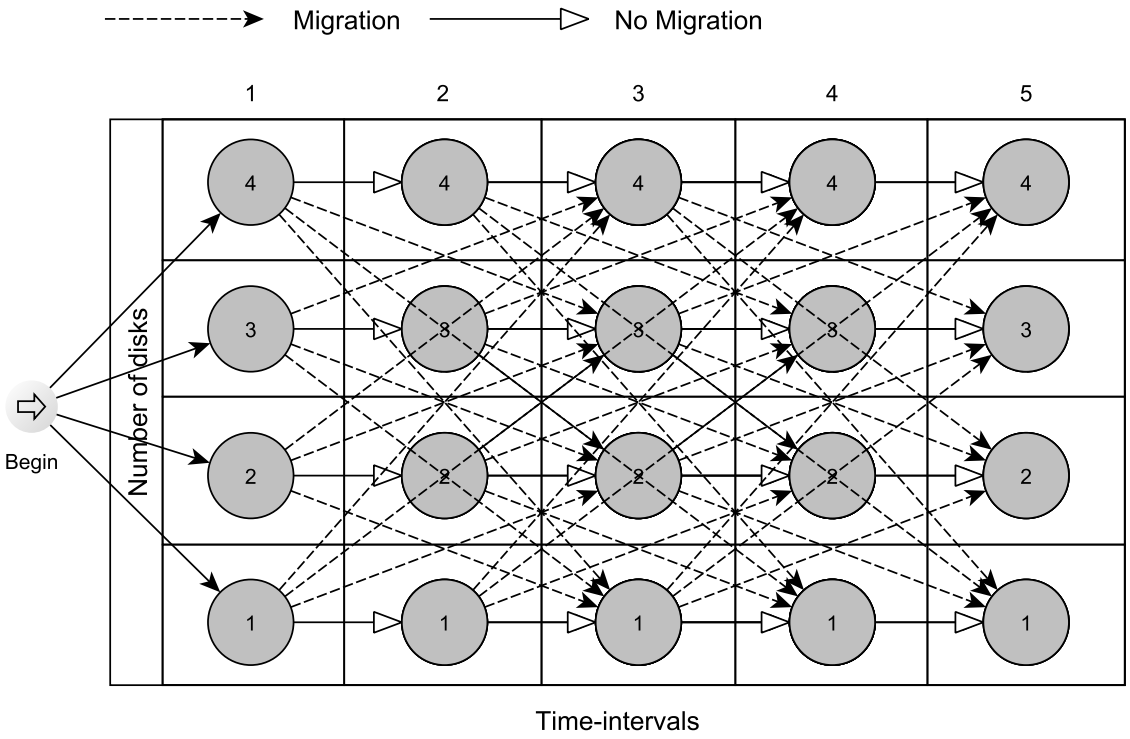


Figure 4.3: Search graph for finding the optimal schedule

a migration takes m time-intervals to complete. Thus, in this graph, which represents 2-feasible schedules, intermediate states that correspond to 1-feasible schedules (in general, $[1, m-1]$-feasible schedules) are not possible.

Given such a graph, we would like to find out the minimum cost path (schedule) [Equation 4.6] that begins at a node at timeslot $t = 1$ and finishes at timeslot $t = \tau$ ($\tau = 5$ in the figure) and does not violate the SLO. The DFS exploration begins at the *begin* node in the figure, and exhaustively searches all of the paths in the search graph. After a complete m-feasible schedule is obtained, it is checked for being *valid* and saved for later comparison if its cost is lower than the lowest cost schedule found so far. The search finishes when all the m-feasible schedules have been explored, and the last saved schedule is an *optimal schedule*. We present the essential pseudo-code of the technique in Algorithm 1. It also includes some strategies to better the performance of the basic exhaustive algorithm, which are discussed a bit later.

We can estimate an upper-bound on the size of search space of all complete m-feasible

Input: U_{ref}, V, τ, $\lambda(t)$, $\mu(t)$, λ_m, m, k_{max}
Output: $S_{optimal}$ (Optimal schedule)

```
1  OPTIMIZER(schedule S_tillNow, integer V_tillNow, integer Cost(S_tillNow))
2  if length(S_tillNow) = τ then
3  |  if Cost(S_tillNow) < Cost(S_optimalTillNow) then
4  |  |  S_optimal = S_tillNow;
5  |  end
6  |  else
7  |  |  return;
8  |  end
9  end
10 /* Insert minAbove pruning here, refer text for details                    */
11 foreach disk config k in [1, k_max] at t = (length(S_tillNow) + 1) do
12 |  bool migration = false;
13 |  if t > 1 AND, in S_tillNow, k ≠ s(t − 1) then Migration possible
14 |  |  if (τ − t) ≥ m then
15 |  |  |  Add(k) to the schedule S_tillNow, m times;
16 |  |  |  update(V_tillNow, Cost(S_tillNow));
17 |  |  |  if V_tillNow ≤ V AND Cost(S_tillNow) * t ≤ Cost(S_tillNow) * τ then
18 |  |  |  |  migration = true;
19 |  |  |  end
20 |  |  end
21 |  |  if migration = false then
22 |  |  |  rollback(S_tillNow, V_tillNow, Cost(S_tillNow));
23 |  |  |  continue to the next disk configuration (k + 1);
24 |  |  end
25 |  end
26 |  else
27 |  |  Add(k) to the schedule S_tillNow;
28 |  |  update(V_tillNow, Cost(S_tillNow));
29 |  |  if V_tillNow > V OR Cost(S_tillNow) * t > Cost(S_tillNow) * τ then
30 |  |  |  rollback(S_tillNow, V_tillNow, Cost(S_tillNow));
31 |  |  |  continue to the next disk configuration (k + 1)_t;
32 |  |  end
33 |  end
34 |  OPTIMIZER(S_tillNow, V_tillNow, Cost(S_tillNow));
35 |  rollback(S_tillNow, V_tillNow, Cost(S_tillNow));
36 end
```

Algorithm 1: OPTIMIZER: DFS-algorithm with *pruning*

schedules by considering the space of all *1-feasible schedules*, which is $(k_{max})^{\tau}$. Evidently, the search space gets huge for even moderately large values of τ. In our experiments, we use $k_{max} = 20$ and $\tau = 336$, which is a number with 438 decimal digits! Thus, we employ several strategies that dramatically shrink our search space, make the exploration of paths faster and help in improving the performance of our technique. We list them below:

- *Violations based pruning.* By definition, an *optimal schedule* must be a *valid schedule*. Recall, from the SLO, $U(t)$ should not exceed U_{ref} more than V times. It follows from above that an optimal schedule should have at most V violations. Therefore, we can prune all those schedules that have more than V violations. Moreover, if during the course of finding a schedule, if we encounter a partial schedule ($length(S) < \tau$) with violations greater than V, we can avoid exploring any complete schedules with that prefix, because all schedules with that partial schedule as a prefix would have at least $V + 1$ violations. We employ this strategy to prune in both migration and non-migration cases in Algorithm 1, at *line* 17 and 29 respectively.

- *Cost based pruning.* We use this strategy to *prune if the current partial schedule cost is greater than the cost of the best complete schedule found so far.* From Equation 4.6, $Cost(S_{partial})$ is monotonically increasing in the schedule length. Thus, any partial schedule with cost greater than current best schedule would yield a suboptimal (complete) schedule. Note that the addition of this cost-based pruning turns our algorithm into a traditional *branch-and-bound* technique. As the algorithm progresses, the bound (minimum cost schedule found so far) gets tighter, resulting in more pruning. We again use this strategy to prune in both migration and non-migration cases in Algorithm 1, at *line* 17 and 29 respectively.

- *Exploration strategy.* The order in which we explore the nodes of the graph at each level is also pertinent to the performance of the algorithm. We explore the path with the smallest number of devices (lowest cost) first, and rely on violations-based pruning to prune quickly, in case the devices are too low in number to meet the SLO. This strategy provides good cost-based pruning when the number of devices are large, as a relatively low-cost plan is found first that acts as a good bound for pruning the branches leading to higher numbers of disks.

- *Dominant path based pruning.* In Figure 4.3, consider two paths, P_1 and P_2 from a node at the starting timeslot, $t = 1$, converging to the same node at time t, and let, (C_1, V_1) and (C_2, V_2) be the costs and the violations of the two path. We define,

  $$P_1 \textit{ dominates } P_2 \textit{ iff } C_1 < C_2 \textit{ and } V_1 \leq V_2.$$

  It follows from above that any schedule beginning with P_2 need not be considered, as P_2 can be replaced by P_1, and if the schedule with P_2 is valid, the modified schedule

with P_1 will be both valid and of lower cost. It is so because the schedule with the dominant path always has at least as many violations allowed as the schedule with the dominated path in the lower parts of the search graph below the common node at t.

This pruning technique can be used in our algorithm. At each node n in the search graph, and for each $v \,\epsilon\, [0, V]$, we record the lowest cost of any path found to n having exactly v violations. Any subsequent path that reaches n with v violations can be pruned if its cost is higher than the saved cost, as it is dominated by the path whose cost we recorded. We call this strategy, *minAbove*. It works because from Equation 4.6:

$$Cost(S_{dominated}) = \frac{\Sigma_{t=1}^{t} C_{dominated}(t) + \Sigma_{t=t}^{\tau} C_{common}(t)}{\tau}$$
$$Cost(S_{dominant}) = \frac{\Sigma_{t=1}^{t} C_{dominant}(t) + \Sigma_{t=t}^{\tau} C_{common}(t)}{\tau}$$

And,

$$\Sigma_{t=1}^{t} C_{dominated}(t) > \Sigma_{t=1}^{t} C_{dominant}(t).$$

Therefore, cost of the schedule with the dominant path is always lower than the dominated path one. We use this pruning strategy in our implementation. It can be inserted at *line* 10 in Algorithm 1.

Similar to minAbove, we can define *minBelow* for diverging paths, which works by bounding the sub-graph below the common node at t with the dominant path cost (and corresponding violations allowed). We implemented it as well, but found it to be not as effective as the previous strategy, because of the fewer opportunities to prune branches at the top of the tree as compared to the minAbove strategy. For our experiments, we used minAbove along with the cost-based and violations-based pruning strategies presented earlier. We also did a comparison of optimization times of Algorithm 1 with various pruning strategies and is presented in Chapter 5.

In the next section, we present several experimental results obtained by employing our approach and try to evaluate the same in light of those results.

Chapter 5

Experiments and Results

In this chapter, we present several experiments designed to study the effectiveness and efficiency of our approach. We want to know whether our dynamic provisioning approach helps in lowering the cost as opposed to a static system that over-provisions the resources to meet the SLO. If so, we would also like to study how significant are the savings, to what parameters is the technique sensitive, and whether the required optimization time is reasonable.

We used four sets of workload data from two sources, namely *University of Waterloo Home Directory Server* and *Microsoft Research (MSR) Cambridge I/O Traces*. We collected the *UW* traces over a period of time by monitoring the I/O traffic generated by the primary network file server in the School of CS at the University of Waterloo. It supports mail and general computing needs of the faculty, staff, and graduate students. The *MSR* traces are available from Microsoft [16]. They include 36 I/O traces from 36 different volumes on 13 servers. We chose two sets of data from each source exhibiting some periodicity (closer to real I/O workloads in data-center environments [22, 21]) and enough variability (significant difference between the average and the maximum demand) so that dynamic provisioning would be potentially useful.

The methodology was to run an instance of our algorithm on these workload traces and study the properties of the resulting schedules (suboptimal and optimal) under default parameters vis-à-vis an optimal static schedule. We also measure the optimization times. In addition, we varied the values of the algorithm's parameters to study how these parameters affect the resulting schedules.

We present results from two workload traces, one each from UW and MSR, in this section. Additionally, we present results from the other two traces in Appendix A. Each of the two

Parameter	Default Value	Description
t_f	0.5 hr.	Time interval (timeslot length)
$T(\tau)$	1 week (336)	Time period (Schedule length in timeslots)
U_{ref}	0.5	Reference utilization
v	5%	Violations allowed (5% of τ)
k_{max}	20	Maximum number of disks in the system
m	8	Number of timeslots taken for a migration
λ_m	1 mb/s	Migration penalty

Table 5.1: Parameters with default values

UW traces represent a week's I/O workload, one from the month of *January 2010* and one from the month of *March 2010*. Each trace records the server's I/O throughput, in $bits/s$, measured over 30 minutes intervals. Our optimizer requires both request rates ($\lambda(t)$) and service rates ($\mu(t)$) as input. Our traces describe time-varying request rates, but unfortunately they do not include information about service rates. Thus, for the purpose of our experiments we assumed a constant service rate of 10 *megabits/s*. As a result, we are not able to directly evaluate the impact of time-varying service rates. However, our traces do capture significant variation in request rates.

From the MSR traces, we chose I/O data from two servers, **src1** and **web** servers (See [16] for details). We assumed a service rate 10 *megabits/s*, as in case of UW data, for MSR traces. Also, we scaled the request rates of **web** by a factor of 10 as its request rate was very low. The I/O workload of *UW January* server was presented earlier in Figure 4.2. The I/O workload of *MSR src1* server is illustrated graphically in Figure 5.1. In both the figures, $t_f = 30$ *minutes* and $T = 1$ *week*, i.e. $\tau = 336$ time-intervals of 30 minutes each.

We mainly used the *minAbove* pruning strategy with *Algorithm 1* for the experiments, but also implemented *minBelow* and *Algorithm 1 with basic pruning*. Also, we use the additive provisioning cost-model, as represented in Equation 4.4, as the default cost model. The default values of various other parameters are listed in Table 5.1. Unless otherwise stated, we use the default values in our experiments.

Schedule Quality. First, we analyze the *quality of the optimal schedule* produced by the optimizer. In Figure 5.2, we compare the cost of the optimal dynamic plan found by the optimizer to the cost of the optimal static plan. The optimal static plan is the least costly static plan that satisfies the SLO. In both, Figure 5.2(a) and Figure 5.2(b), we clearly see the *cost* to be lower in our dynamic approach. To elaborate the idea, we study the benefit obtained empirically. Here, we use the results for the default values (i.e. corresponding

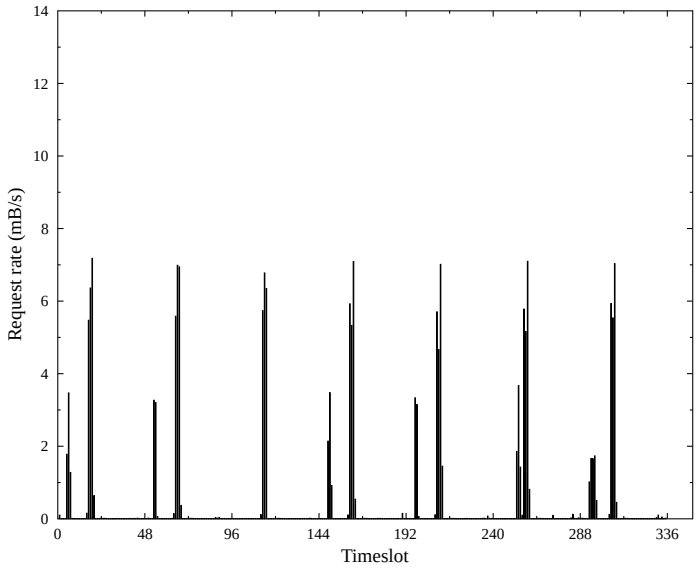
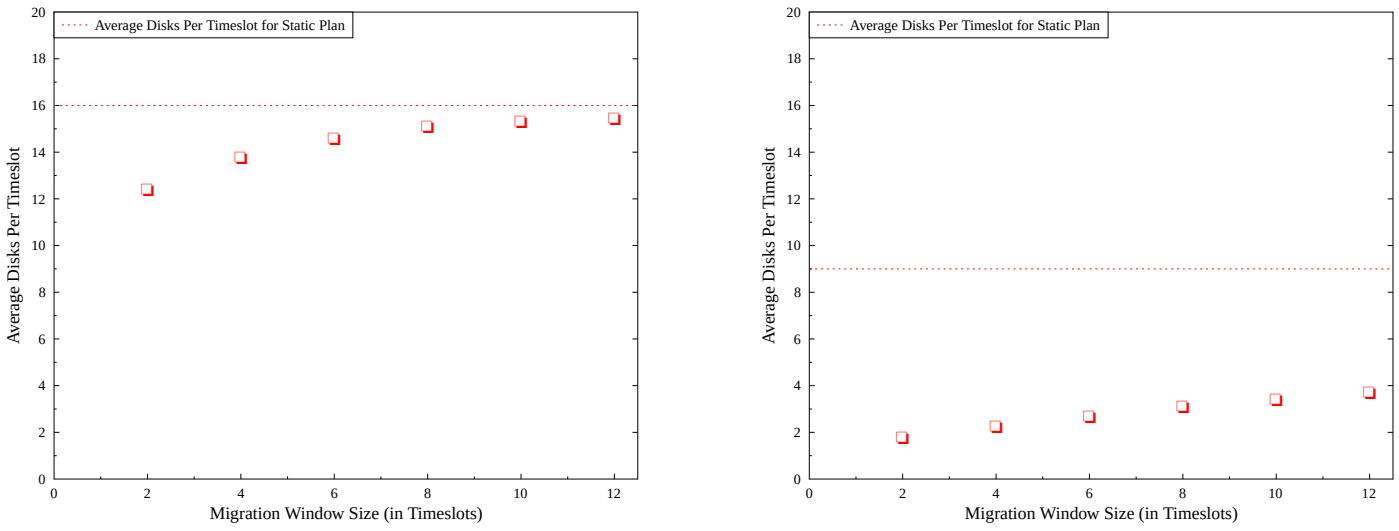


Figure 5.1: Request rate for MSR Source1 server



(a) UW Home Directory (January 11-17, 2010)

(b) MSR Source1 Server

Figure 5.2: Average disk cost per time-interval with varying migration window size (m)

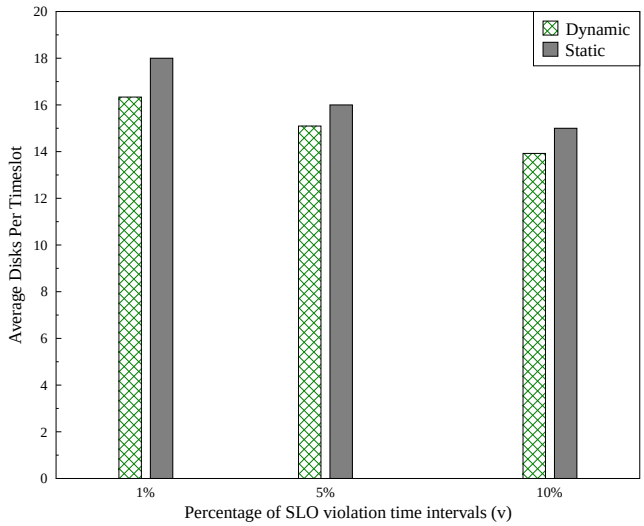
to $m = 8$ in Figure 5.2). In Figure 5.2(a), the cost of the static schedule is 16, while the cost of the dynamic schedule is 15, a cost reduction of around 6% or cost savings of 1 disk per timeslot. Similarly, in Figure 5.2(b), the cost of the static schedule is 9, while the cost of the dynamic schedule is 3.1, a cost reduction of around 65% or cost saving of around 6 disks per timeslot. The latter has greater cost savings due to the nature of the *MSR* workload which sees long periods of low activity with periodic bursts of high activity. Clearly, our approach does much better than the best static approach and, depending on the dynamics of the workload, cost-savings could be very substantial.

Sensitivity Analysis. We did some experiments to observe the effect of changing parameter values on the properties of the schedule. Referring to Figure 5.2 again, we see that the margin of benefit increases as we decrease the length of the migration slots. This indicates that if we use a faster migration scheme, we could profitably use a more dynamic plan (as migrations take fewer time-intervals to complete), which results in even lower costs.

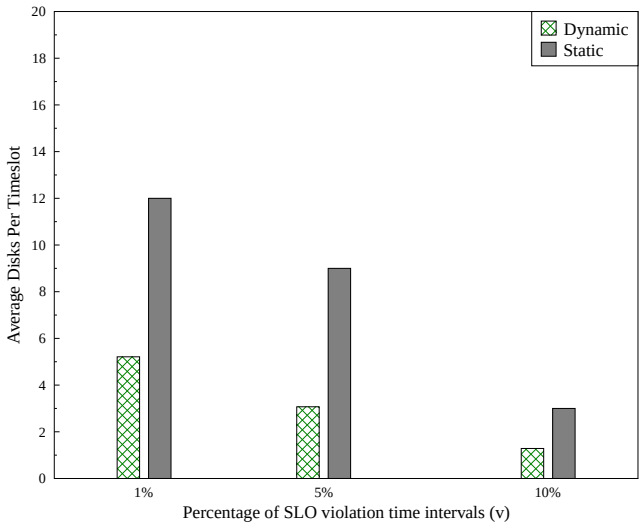
Next, we study the effect on schedule quality (cost) as we change number of SLO violations allowed. The results are shown in Figure 5.3. For both UW (Figure 5.3(a)) and MSR (Figure 5.3(b)) data, we see that the average cost decreases as we get more liberal with the amount of violations we are willing to allow. Clearly, it is so because as we increase the number of violations allowed, one is able to use a lower number of disks, instead of a higher number (which meets the SLO), in increasingly more time-intervals, driving the average disk cost down.

As already discussed, our technique is independent of the cost-model used. Figure 5.4 illustrates this and compares two cost-models, viz. *costSum*, represented by Equation 4.4 and *costMax*, represented by Equation 4.5. Since, the latter imposes lesser penalty on the use of disks during migration, its average disk cost is significantly lower. The benefit of using our dynamic approach over a static approach is greater under costMax, since the cost of migrations is lower.

Optimization Time. We can see significant cost benefits in our approach, but it is also necessary that the approach provides results in a reasonable amount of time. Figure 5.5 illustrates a run of our algorithm successively finding lower-cost plans as it proceeds over time. The increasing quality is because of the way our algorithm works. It maintains the minimum cost plan and replaces it with some other plan if it is lower in cost. A clear benefit of the approach is that it finds a good *suboptimal schedule* quite quickly (because of the liberty to grossly overprovision initially for meeting the SLO), which could be used in case there are some optimization time constraints. Also observe that the quality of plans found initially has a steep slope. It is so because initially, the bound conditions in the search graph are very loose. They get tighter as the algorithm proceeds and successively better



(a) UW Home Directory (January 11-17, 2010)



(b) MSR Source1 Server

Figure 5.3: Average disk cost per timeslot comparison of static and dynamic plans for various constraint values

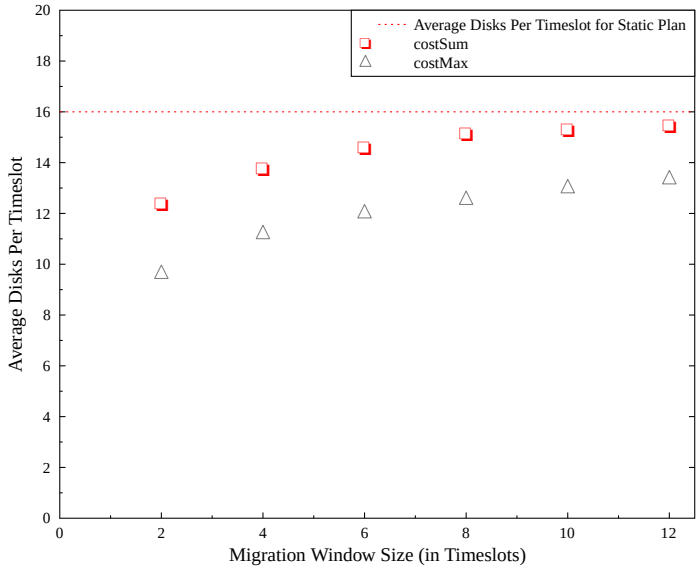
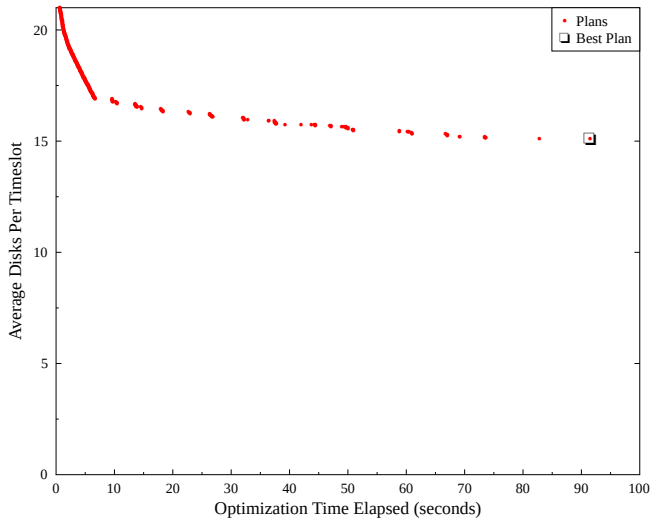
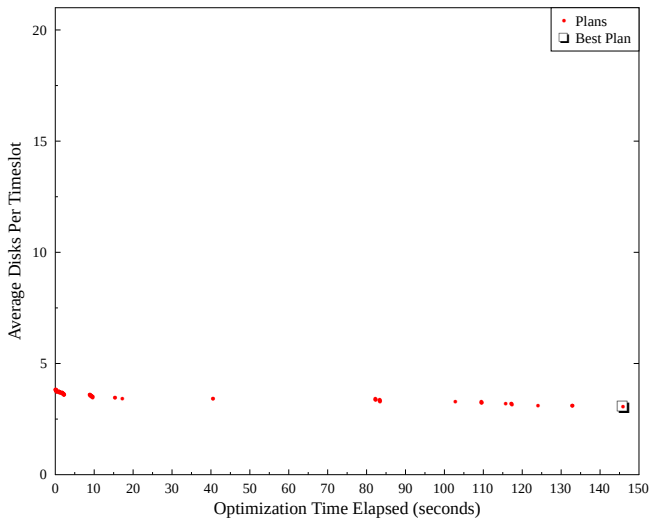


Figure 5.4: Average disk cost per timeslot with different cost functions for UW data



(a) UW Home Directory (January 11-17, 2010)



(b) MSR Source1 Server

Figure 5.5: Increasing quality (lower cost) of plans as the algorithm proceeds over time

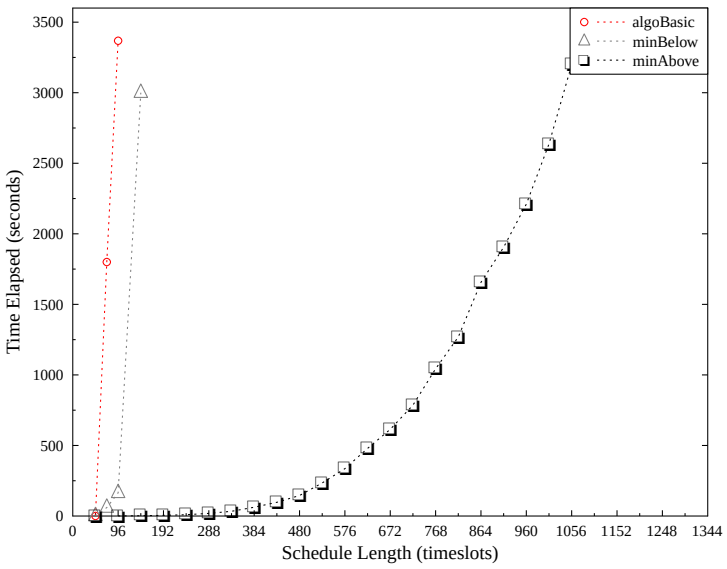


Figure 5.6: Comparison of run time of various optimization techniques

plans are found. Thus, grossly overprovisioned (suboptimal) plans are slowly replaced by near-optimal plans. Hence, later, the plans' costs are quite comparable to one another. This fact could be used in scenarios, where one would like to use a plan which may not necessarily be optimum but is better than a static plan.

Finally, Figure 5.6 shows a comparison of the optimization time of the algorithm with various pruning techniques as discussed in the previous chapter. The migration time in this case is 12 time-intervals. We compare three versions of Algorithm 1.

- *algoBasic.* Algorithm 1 in its basic implementation, with only *violations* and *cost* based pruning.
- *minBelow. algoBasic* with *minBelow* pruning.
- *minAbove. algoBasic* with *minAbove* pruning.

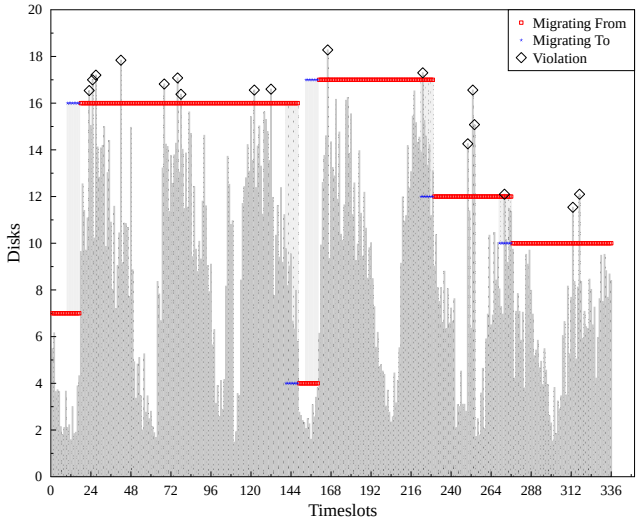
Evidently, minBasic takes too much time to provide results beyond a day's data, which indicates that the violations and cost-based pruning alone are not sufficient for pruning the search space for the problem size we target. Similarly, *minBelow* also suffers as the problem size gets larger. *minAbove* is quite fast and gives results for a week's data within a few minutes. However, its exponential nature is visible as we increase the problem size beyond a couple of weeks. Nevertheless, for the problem size of building weekly schedules for data-centers (which is a prominent problem), it is quite effective.

Now, we present a few migration schedules, plotted graphically in Figure 5.7, for illustrative purposes. Figure 5.7(a) and Figure 5.7(c) show a graphical plot of the optimal migration plans obtained in Figure 5.5. Observe, the workload requirement is superimposed on the number of disks required to meet the same. Specifically, we superimpose the minimum number of devices that would be required to meet the SLO at that time interval. Thus, the superimposed values can be viewed as representing an idealized resource allocation schedule that we would construct if provisioning were instantaneous and produced no overhead. To calculate the superimposed values, we make use of Equation 4.2. We need to plot $k(t)$ for $U(t) = U_{ref}$, where:

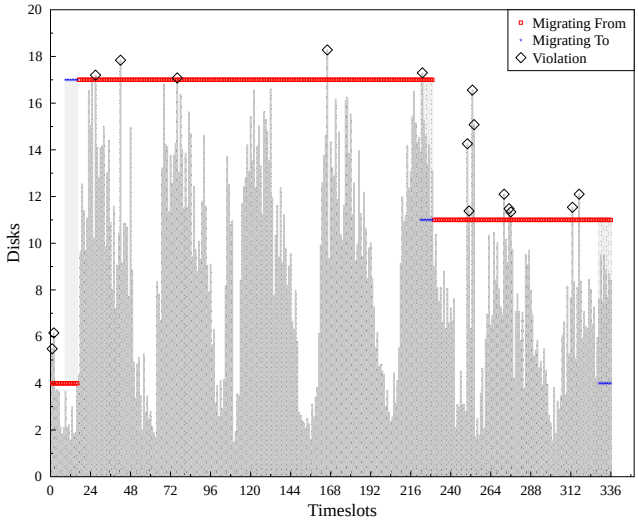
$$k(t) = \frac{\lambda(t)}{U_{ref}\mu(t)}$$

Since, we take $\mu(t)$ constant for our experiments, we can just scale the request rate $\lambda(t)$ by a factor of μU_{ref}. Also observe that the number of violations do not exceed the violations limit ($V = 5\%\, or\, 17$, in this case) in any of the schedules. The migrations try to capture the dynamic nature of the workload, but only if the migration time allows it, which brings

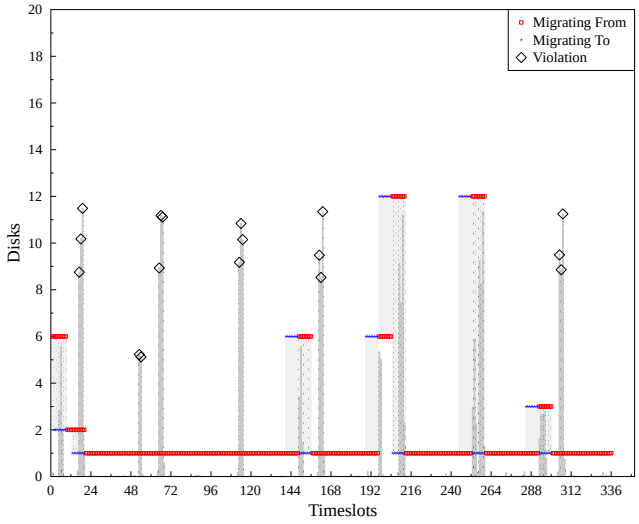
up our earlier observation that if we reduce the migration window size we could see more dynamic plans that closely emulate the workload requirement. Moving on to Figure 5.7(b) and Figure 5.7(d), notice that these plans are also for the same data but with an additional requirement of the *initial* and the *final* disk configuration to be *equal*. In scenarios, where one expects a similar workload in the following time-period as in the previous one, this condition may well be an inherent one.



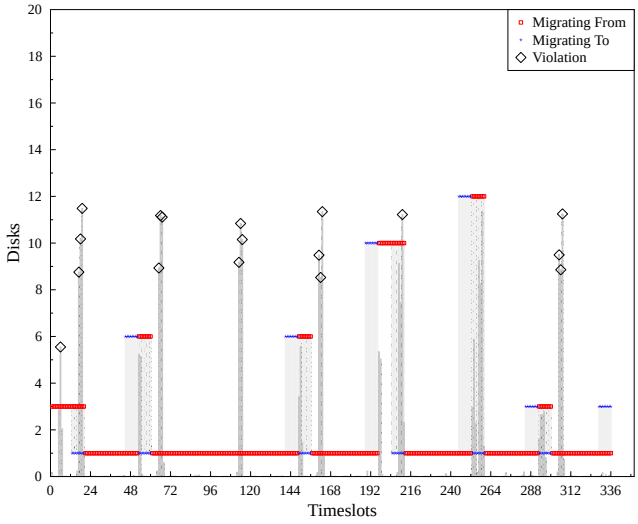
(a) UW Home Directory (January 11-17, 2010)



(b) Initial Config same as Final Config. UW Home Directory (January 11-17, 2010)



(c) MSR Source1 Server



(d) Initial Config same as Final Config. MSR Source1 Server

Figure 5.7: Graphical plots of migration plans

Chapter 6

Related Work

Dynamic resource allocation has been studied over the years with varying objectives of achieving desired QoS levels, load balancing, minimizing resource contention and higher resource utilization. Padala et al. [18, 17] formulate this as a feedback control problem and use tools from adaptive control theory to design online control algorithms. Such techniques use a closed-loop control model where the objective is to converge to a target performance level by taking control actions that try to minimize the error between the measured output and the reference input. Aron et al. [3] formulate it as an online optimization problem using periodic utilization measurements, and resource allocation is implemented via request distribution. Much prior work on resource allocation is based on prediction-based provisioning [25, 20, 27], in which statistical models for the workloads are first developed using historical traces offline or via online learning. Resource allocation decisions are then made to satisfy such predicted demand. This approach is often considered to be proactive, in that it uses predicted demand to periodically allocate resources before they are needed. On the other hand, in reactive resource control, the system reacts to immediate demand fluctuations before periodic demand prediction is available or in case the predicted demand data is erroneous. Thus, both approaches are important and necessary for effective resource control in dynamic operating environments [23]. Zhang et al. [28] outline the provisioning approaches as typically involving the following steps:

1. Constructing an application performance model that predicts the number of application instances required to handle demand at each particular level, in order to satisfy QoS requirements,

2. Periodically predicting future demand and determining resource requirements using the performance model, and

3. Automatically allocating resources using the predicted resource requirements.

In our work, we specifically study dynamic provisioning of storage resources. Our approach fits the outlined steps. Our performance model is based on a k-server queuing model for the storage system, relating utilization of the system with the request rate, the service rate and the number of servers (devices) in the system. We do not predict future demand, but our approach requires a knowledge of the future workload. We use the demand to determine resource requirements using the performance model, and using our technique *proactively* find a plan that allocates minimal resources and meets the SLO.

Automatic system design tools [2, 1] build systems that satisfy declarative, user-specified QoS requirements. They effectively minimize overprovisioning by taking into account workload correlations and detailed device capabilities to design device configuration and data layouts. The whole storage system may be redesigned in every refinement iteration, and there typically is a substantial delay to migrate the data online. Both the systems cater to only specific workload and do not reprovision resources based on dynamically changing workload. Façade [15] is a virtual store controller, which sits between hosts and storage devices in the network, and throttles individual I/O requests from multiple clients so that devices do not saturate. It may be viewed as complementary to the former works as it can handle short-term workload variations through adaptive scheduling without migrating data, and possibly postpone the need for a heavyweight system redesign. It does not reallocate resources to manage demand. In this work, our system provisions resources based on a schedule of *'when to provision or deprovision?'* and is always mindful of the SLO and the upcoming workload.

Researchers have studied impact of backend data-movement tasks on frontend application. Aqueduct [14] uses a control-theoretical approach to statistically guarantee a bound on the amount of impact on foreground work during a data migration, while still accomplishing the data migration in as short a time as possible. QoSMig [7] is an adaptive rate-control based data migration system that achieves the optimal application performance in a differentiated QoS setting, while ensuring that the specified migration constraints are met. QoSMig uses both long term averages and short term forecasts of client traffic to compute a migration schedule. Both the systems are targeted at maximizing performance of the foreground application while a migration is taking place, but do not consider a sequence of data-movements for dynamic storage provisioning to meet a specified SLO.

Pulstore [21], similar to our work, relies on migrations to provision storage resources for a given workload, while providing QoS guarantees. However, there are several differences. In Pulstore, the authors choose a specified I/O *latency* for a logical disk as the QoS goal, and predict performance outcome for a given workload and storage system configuration using models that represent I/O latency as a function of the workload and the logical disk configuration. Latency is difficult to predict at high utilization levels. We use a k-server

queuing model for the storage system to predict its performance in terms of utilization, and use a reference utilization as the QoS goal. We also allow a cost/performance trade-off parameter to be introduced in the SLO making our schedules more flexible to cost and performance requirements. Also, in Pulstore, the migration model is such that a migration action changes the performance of the disks at every data movement. Thus, during a migration, the performance of the system depends on the amount of data at the source disks and the destination disks. This makes Pulstore consider the effect of even a single data-movement action on QoS. Our case-studies suggest that the performance of the storage system changes only after a provisioning action completes. Thus, we base our solution on that principle.

Zhang et al. [26] build a 2-level scheduler to achieve storage performance isolation between several applications while meeting the individual SLOs described in terms of throughput and latency. As discussed earlier, we describe the SLO in terms of keeping the utilization of devices below a reference utilization value. We also introduce a trade-off factor (violations allowed) to allow a trade-off between performance and cost.

In Hippodrome [2], to meet the workload's performance requirements while using a system design that uses the least resources (disks) among the set of candidate *valid designs*, a solver component is used that efficiently searches the exponentially large space of storage system designs. The authors map the problem of efficiently packing a number of stores, with both capacity and performance requirements, onto disk arrays to the problem of multi-dimensional bin packing and use best-fit approaches [12, 13] for a quicker solution to this NP-complete problem. Similarly, BRAHMA [24] uses a constraint-based optimizer that takes the candidate resource list and generates an optimal allocation of resources for a given customer SLO by formulating it as the 0/1 multi-knapsack problem [19].

The problem presented in Chapter 4 can be seen as the *Restricted Shortest Path (RSP)* problem. It is defined as follows. Let G be a graph with n vertices and m edges. Each edge ij has an associated positive integral cost c_{ij} and positive integral delay d_{ij}. The cost (and delay, respectively) of a path is defined as the summation of the costs (and delay, respectively) along all of its edges. The *source* and *target* nodes are also given. The aim is to find the minimum cost $source - target$ path in G such that the total delay along this path does not exceed a given bound D. We can define our problem as RSP using Figure 4.3. Imagine all nodes at $t = 5$, converging to a *target* node. Then, in Figure 4.3, our aim is to find a path (schedule) that starts at the *begin* node and ends at the *target* node. We also observe that the *cost* of a path is the schedule cost (Cost(S) from Equation 4.6) and D in the RSP problem can be seen as V (violations allowed). As observed, our problem is the RSP problem, which was shown by Handler and Zang [10] to be NP-hard. To obtain a solution in polynomial time, researchers have employed *fully polynomial time approximation*

schemes (FPTAS) [11, 8]. These techniques take an instance of the optimization problem and a parameter $\epsilon > 0$ and, in polynomial time, produce a solution that is within a factor ϵ of being optimal. In the current work, we employ an *exhaustive DFS with pruning* in the search space to come up with an *optimal* solution in a reasonable amount of time for our targeted problem size.

Chapter 7

Conclusions and Future Work

We developed a dynamic storage provisioning technique that balances the conflicting goals of meeting SLOs and minimizing the provisioned resources. Our approach takes a model of the upcoming workload and builds an optimal schedule of dynamically provisioning and deprovisioning storage resources. We perform two case-studies to help in characterizing the behavior of the provisioning mechanisms, which is taken into account while developing the solution to our problem. It renders our approach to be easily applicable in real systems. We model the storage system as a k-server queuing model. The SLO is defined as keeping the utilization of such a system below a threshold, with a performance/cost trade-off factor also introduced. The provisioning works by varying the number of disks in the system (using migrations that take non-zero time) at each timeslot, and our technique searches through this space to emit an optimal schedule that has minimal average number of disks and is valid for the given SLO. An experimental analysis of our technique using I/O traces from real systems demonstrated the efficacy of the approach.

There is a lot of scope for future work in this problem area. There may be errors in the predicted model of the upcoming workload. How to account for search errors while coming up with an optimal schedule? We can introduce self-tuning mechanisms into the system to handle small variations, but any large variation in observed workload would require re-provisioning of storage resources (which takes time). The current work provides guarantees on the utilization of the storage system. Work on guaranteeing performance in terms of other metrics (e.g. response time, throughput) using a similar mechanism would be interesting and challenging as those metrics may depend on other system parameters as well. Lastly, using our approach for building schedules for large time-periods may not be as effective as for shorter periods. In Chapter 6, we discussed this problem as an RSP and work employing polynomial-time approaches (FPTAS) for obtaining near-optimal schedules could be explored for such large time-periods.

APPENDIX

Appendix A

Graphs For Other Collected Data

As mentioned in Chapter 5, we present results from the remaining two workload traces, one each from UW (March 2010) and MSR (**web** server), in this appendix. The request rates of the two I/O workloads are presented in Figure A.1.

As earlier, we used the *minAbove* pruning strategy with Algorithm 1 for these experiments, with the additive provisioning cost-model, as represented in Equation 4.4, as the default cost model. The default values of various other parameters are the same as were listed in Table 5.1.

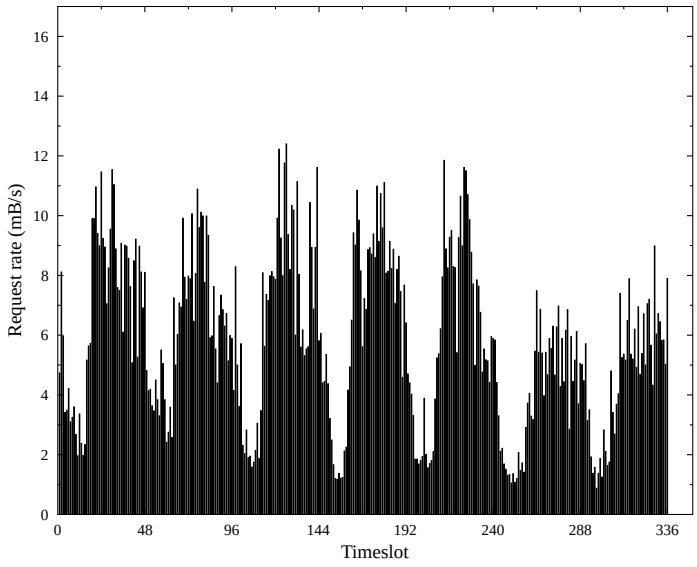
In Figure A.2, as observed in Figure 5.2, we find the cost to be lower in our dynamic approach as compared to the cost of the best static plan. Observe, both the dynamic plan costs and the static plan cost of MSR data are quite low. This is so because the MSR web data (even after scaling) had very few high load values in the workload data, which allowed the plans to ignore them as violations (set at 5%). Also, although not quite apparent in MSR data, we see that the margin of benefit increases as we decrease the length of the migration slots, which is consistent with the previous results.

Figure A.3 illustrates the average cost decreasing as we get more liberal with the amount of violations we are willing to allow. Note, in MSR data, the costs of static and dynamic plan for violations set at 10% are almost equal. The reason is same as stated earlier that the number of violations allowed are so many that even a static plan gets away by not over-provisioning for the very few high values of I/O workload.

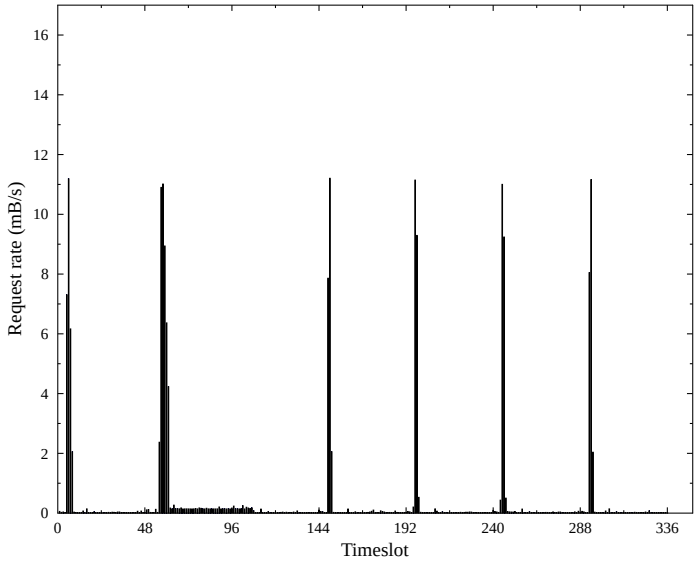
The effect of changing the cost-model is shown in Figure A.4 and Figure A.5, and is consistent with our earlier observation in Chapter 5. MSR data again shows little variation in cost with changing cost-model because of the reason discussed earlier.

Figure A.6 illustrates a run of the algorithm successively finding lower-cost plans as it proceeds over time. Due to the nature of the workload, the algorithm finds the optimal plan quite quickly in the case of MSR data. Note, that the scale of the time axis is in milliseconds for the MSR data.

Finally, Figure A.7 shows graphical plots of the optimal migration schedule obtained in Figure A.6. Figure A.7 (b) and (d) show plots when the initial and the final config must be equal. These figure are obtained in a way similar to the figures in Figure 5.7, which was discussed in Chapter 5.

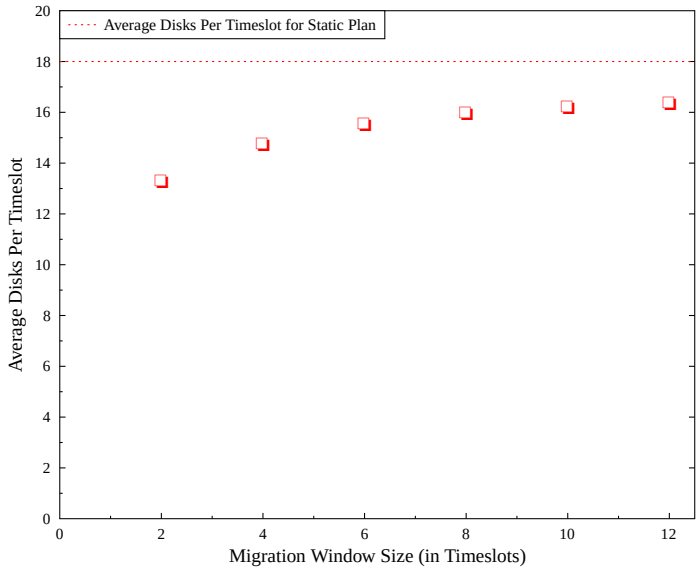


(a) UWaterloo Home Directory (March 01-07, 2010)

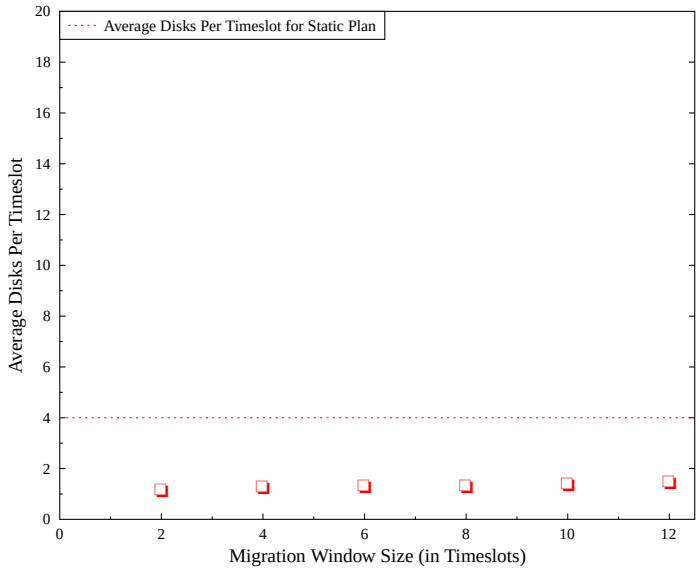


(b) MSR Web Server (Scaled by 10 units)

Figure A.1: Request rates for UW and MSR server

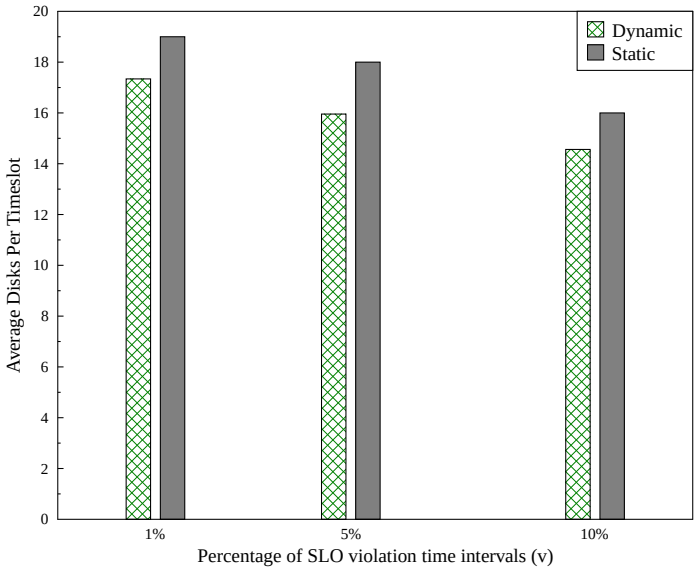


(a) UWaterloo Home Directory (March 01-07, 2010)

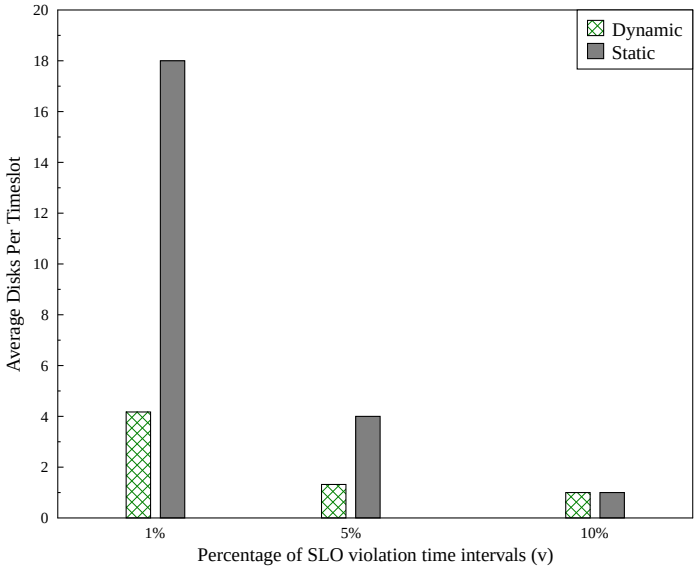


(b) MSR Web Server

Figure A.2: Average disk cost per timeslot with varying migration window size (m)



(a) UWaterloo Home Directory (March 01-07, 2010)



(b) MSR Web Server

Figure A.3: Average disk cost per timeslot comparison of static and dynamic plans for various constraint values

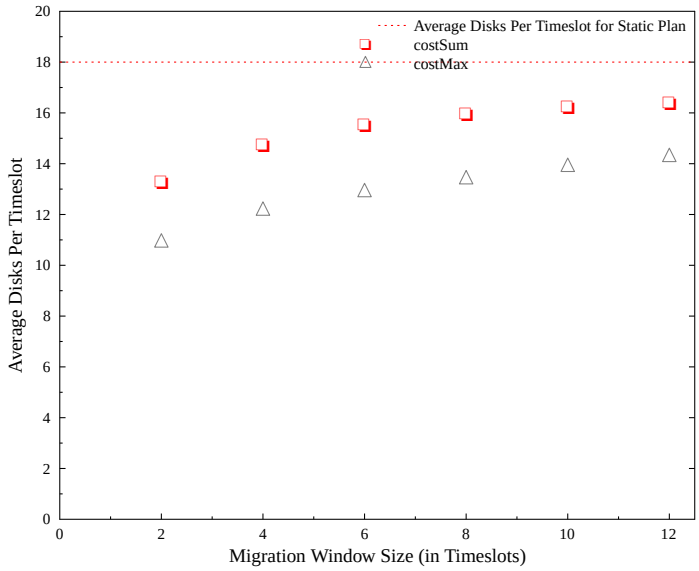


Figure A.4: Average disk cost per timeslot with different cost functions for UW data

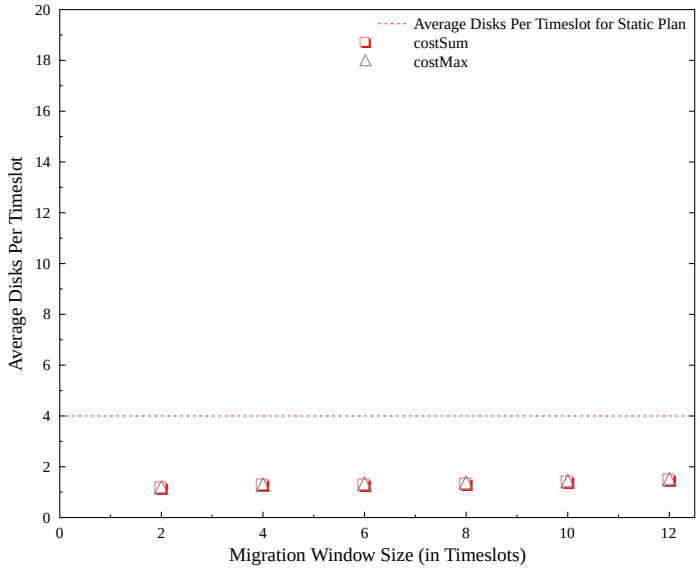
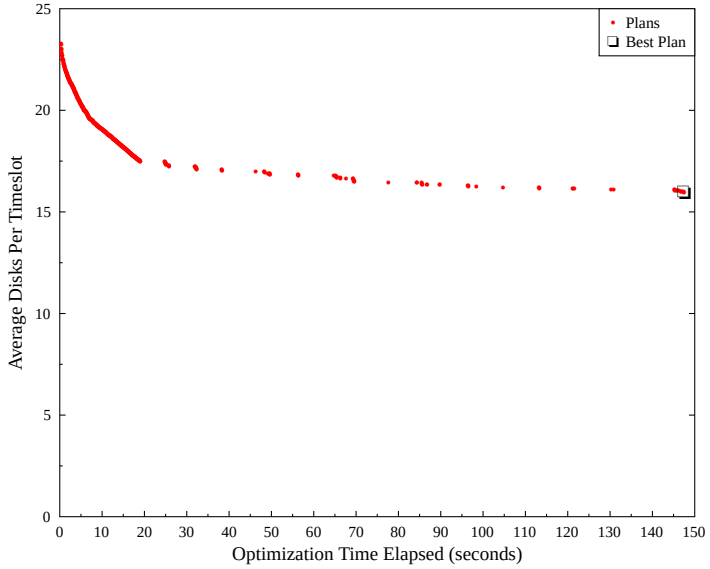
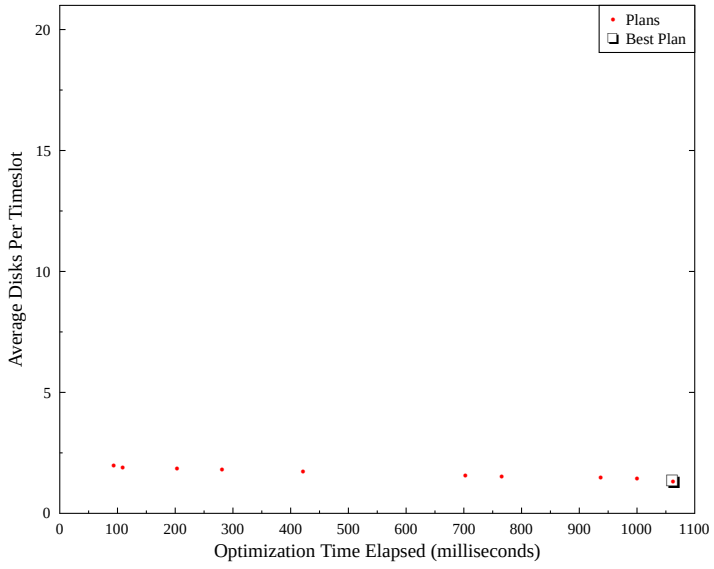


Figure A.5: Average disk cost per timeslot with different cost functions for MSR data

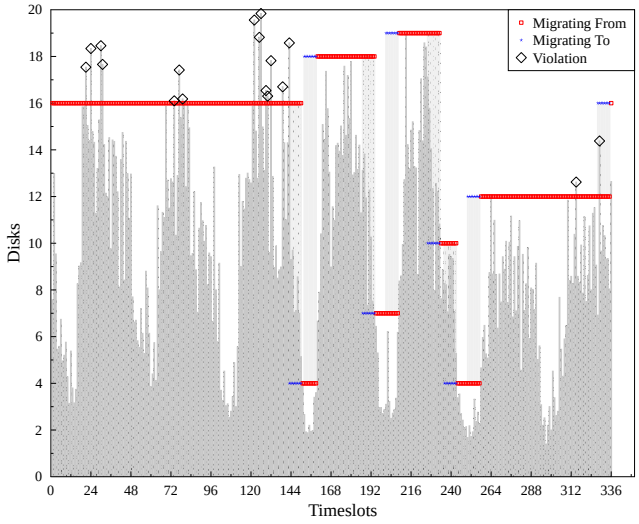


(a) UWaterloo Home Directory (March 01-07, 2010)

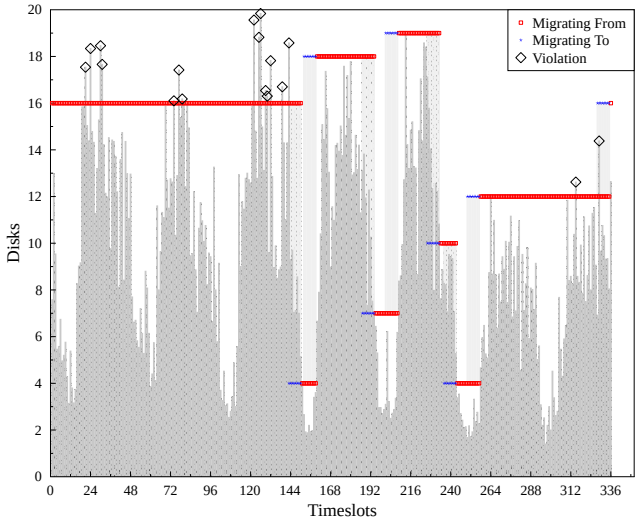


(b) MSR Web Server

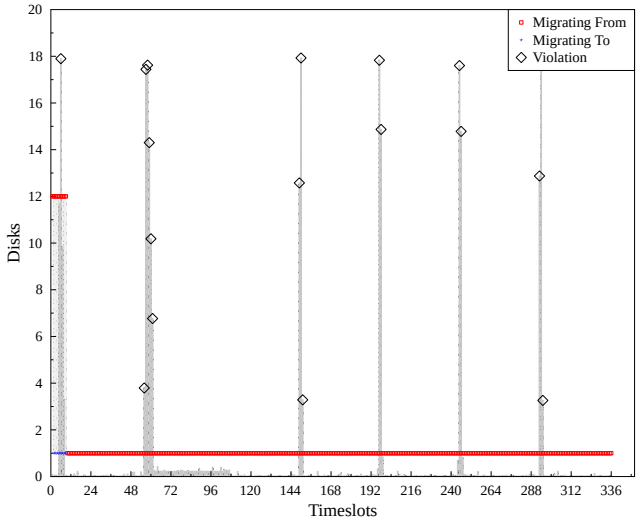
Figure A.6: Increasing quality (lower cost) of plans as the algorithm proceeds over time



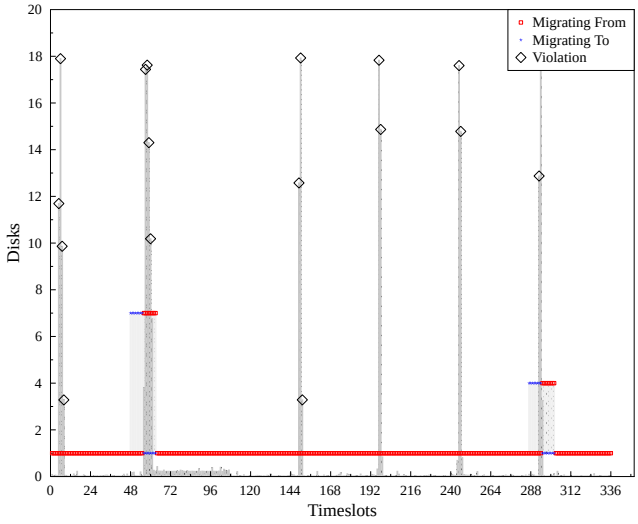
(a) UWaterloo Home Directory (March 01-07, 2010)



(b) Initial Config same as Final Config. UWaterloo Home Directory (March 01-07, 2010)



(c) MSR Web Server, Constraint: 5p



(d) Initial Config same as Final Config. MSR Web Server

Figure A.7: Graphical plots of migration plans

References

[1] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. In *ACM Transactions on Computer Systems (TOCS) v.19 n.4, p.483-518*, November 2001. 1, 5, 33

[2] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Usenix Conference on File and Storage Technologies (FAST)*, January 2002. 1, 5, 33, 34

[3] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *In Measurement and Modeling of Computer Systems*, pages 90–101, 2000. 32

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003. 6

[5] G. E. P. Box, G. Jenkins, and G. Reinsel. In *Time Series Analysis: Forecasting and Control*, USA, 1994. Prentice Hall. 4

[6] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. In *IBM System Journal*, page 28(1):6276, 1989. 7

[7] Koustuv Dasgupta, Sugata Ghosal, Rohit Jain, Upendra Sharma, and Akshat Verma. Qosmig: Adaptive rate-controlled migration of bulk data in storage systems. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, 2005. 33

[8] Funda Ergun, Rakesh Sinha, and Lisa Zhang. An improved fptas for restricted shortest path. *Information Processing Letters*, 2002. 35

[9] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *IEEE International Symposium on Workload Characterization*, September 2007. 4

[10] Gabriel Y. Handler and Israel Zang. A dual algorithm for the constrained shortest path problem. In *Networks*, pages pg. 293–309, 1980. 34

[11] Refael Hassin. Approximation schemes for the restricted shortest path problem. In *Mathematics of Operations Research*, pages pg. 36–42, 1992. 35

[12] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. In *SIAM Journal on Computing*, 1974. 34

[13] C. Kenyon. Best-fit bin-packing with random order. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1996. 34

[14] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002. 33

[15] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Faade: Virtual storage devices with performance guarantees. In *2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003. 33

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008. 23, 24

[17] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, March 2007. 15, 32

[18] 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 *4th ACM European conference on Computer systems*, Nuremberg, Germany, April 2009. 1, 32

[19] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. In *Journal of Operations Research*, page 758767, 1997. 34

[20] Radu Prodan and Vlad Nae. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems*, 2009. 32

[21] Lin Qiao, Balakrishna R. Iyer, Divyakant Agrawal, Amr El Abbadi, and Sandeep Uttamchandani. Pulstore: Automated storage management with qos guarantee. In *International Conference on Autonomic Computing (ICAC)*, 2005. 2, 5, 23, 33

[22] Chris Ruemmler and John Wilkes. A trace-driven analysis of disk working set sizes. In *Technical Report HPLOSR9323*, 1993. 2, 23

[23] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Second International Conference on Automatic Computing*, pages p.217–228, June 2005. 32

[24] S. Uttamchandani, K. Voruganti, R. Routray, L. Yin, A. Singh, and B. Yolken. Brahma: Planning tool for providing storage management as a service. In *the IEEE International Conference on Services Computing*, 2007. 34

[25] X. Wang, D. Lan, X. Fang, M. Ye, and Y. Chen. A resource management framework for multi-tier service delivery in autonomic virtualized environments. In *NOMS*, April 2008. 1, 4, 32

[26] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages p.135–142, 2005. 34

[27] Q Zhang, L Cherkasova, and E Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *In Proc. of the 4th IEEE Int. ICAC*, 2007. 32

[28] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 2010. 32