January 01, 2015

Resource management in enterprise cluster and storage systems

Jianzhe Tai
Northeastern University

Contents

Abstract

In this thesis, we present our works on resource management in large scale systems, especially for enterprise cluster and storage systems. Large-scale cluster systems become quite popular among a community of users by offering a variety of resources. Such systems require complex resource management schemes for multi-objective optimizations and should be specific to different system requirements. In addition, burstiness has often been found in enterprise workloads, being a key factor in performance degradation. Therefore, it is an extremely challenging problem of managing heterogeneous resources (e.g., computing, networking and storage) for such a large scale system under bursty conditions while providing performance guarantee and cost efficiency.

To solve this problem, we first investigate the issues of classic load balancers under bursty workloads and explore the new algorithms for effective resource allocation in cluster systems. We demonstrate that burstiness in user demands diminishes the benefits of some existing load balancing algorithms. Motivated by this observation, we develop a new class of burstiness-aware load balancing algorithms. First, we present a static version of our new load balancer, named ARA, which tunes the schemes for load balancing by adjusting the degree of randomness and greediness in the selection of computing sites. An online version of ARA has been developed as well, which predicts the beginning and the end of workload bursts and automatically adjusts the load balancers to compensate. The experimental results show that this new load balancer can adapt quickly to the changes in user demands and thus improve performance in both simulation and real experiments.

Secondly, we work on data management in enterprise storage systems. Tiered storage architectures provide the shared storage resources to a large variety of applications which might demand for different service level agreements (SLAs). Furthermore, any user query from a data-intensive application could easily trigger a burst of disk I/Os to the back-end storage system, which eventually causes performance degradation. Therefore, we present a new approach for automated data movement in multi-tiered storage systems aiming to support multiple SLAs for applications with dynamic workloads at the minimal cost.

In addition, Flash technology can be leveraged in virtualized environments as a secondary-level host-side cache for I/O acceleration. We present a new Flash Resource Manager, named vFRM, which aims to maximize the utilization of Flash resources with the minimal I/O cost. It identifies the data blocks that benefit most from being put on Flash, and lazily and asynchronously updates Flash. Further, we investigate the benefits of the global versions of vFRM, named G-vFRM, for managing Flash resources among multiple heterogeneous VMs. Experimental evaluation shows that both vFRM and G-vFRM algorithms can achieve better cost-effectiveness than traditional caching solutions, and cost orders of magnitude less memory and I/O bandwidth.

1 Introduction

Large-scale resource management systems are being employed in an increasing number of application areas these days. Examples of these systems include High Performance Computing (HPC), enterprise information systems, data centers, cloud computing and cluster servers. In particular, resource management is an important research topic which is required by any man-made system and affects in system evaluation in two basic criteria, i.e., performance and cost. Efficient resource management has a direct positive effect on system performance and cost. Managing resources at large scale while providing performance guarantee and cost efficiency by using of underlying hardware resources (e.g., computing, networking and storage) is an extremely challenging issue. Such scaled environment provides shared resources as an unified hosting pool, which requires a central mechanism for resource provisioning and resource allocation based on the demands of multiple remote clients [1, 2].

The basic resource management schemes are classified as follows. Admission control prevents system from accepting workloads in violation of high-level policies and avoids the additional loads competing with the works already in progress. Capacity allocation schedules resource for individual instances which might be subject to multiple global optimization constraints. An instance is a service activation. Load balancing usually works as the function to evenly distribute workloads among a shared resource pool or works as the function of server consolidation that concentrates resources and uses the smallest number of servers while switching the others to standby mode. Auto scaling releases or allocations resources on demand and on-the-fly adjusts a pool of system resources for unplanned spike loads. Another classification of resource management policy is directly related to system evaluation of performance criterion aiming for quality of service guarantee and performance isolation. In this dissertation, we works on resource management in large-scale systems, especially, for the load balancing in the cluster systems and the capacity allocation and quality of service guarantee in enterprise storage systems.

A large-scale resource management infrastructure is a complex system with a large number of shared resources. These are subject to different system architectures, heterogeneous underlying hardware resources and unpredictable requests from multiple remote clients. For example, there are different virtualization architectures (e.g., Para-virtualization vs. full virtualization) which require different software designs. Further, such systems can be virtualized in different layers (e.g., software defined networking and software defined data center). Additionally, there are different computing models as platform, e.g., cluster computing, distributed computing, utility computing and grid computing in general. Except the difference in system architectures, the underlying hardware is evolved over time. GPU is in a class by itself which goes beyond CPU and basic graphic

controller functions as a programmable and powerful computational device. NAND-based Flash is being widely deployed as the cache in the storage systems to improve the I/O performance and reduce the power consumption. Flash can be deployed as an entire Flash server or works as a whole storage tier combined with conventional storage devices as a multi-tiers systems. Furthermore, some approaches leverages Flash as a secondary-level host-side cache to accelerate IO operations. Thus, resource management is extremely challenging by requiring complex policies for multi-objective optimization and should be specific to different system requirements. In modern large-scale environments, the variety and the burstiness of workloads are often found as the key factors in performance degradation. Since the cluster system is exposed as a shared resource pool to variously remote applications and many applications are no longer designed as single-program-single-execution, the load balancer in cluster systems may be involved in a large number concurrent and dependent jobs. Furthermore, launching jobs from different applications during a short time can easily introduce an arrival burstiness.

1.1 Computational Resource Management in Cluster Systems

In a traditional multi-server cluster, a front-end dispatcher plays an important roll in dispatching incoming jobs among those back-end servers. Such a front-end dispatcher is featured as a redirect buffer without central waiting queue while each back-end server has its own queue for waiting jobs and a separate processor that operates under the first-come first-serve (FCFS) queuing discipline. A simple yet powerful load balancing in a dispatcher-based cluster system is significantly critical for system performance, such as average waiting time, average response time, system utilization, and job slowdown, etc. Lots of prior researches have investigated the characteristics and algorithm optimization of such featured cluster systems [3–5]. Examples of these policies include Join Shortest Queue (JSQ), the size-based AdaptLoad [6] and the Min-Min/Max-Min algorithms [7], etc. However, these methods only consider the Poisson arrival streams as well as the exponentially distributed service time and the fixed number of choices (i.e., servers). To implement better resource allocation in cluster systems, we present a adaptive load balancing algorithm, named ARA. We first use a queuing model to demonstrate the negative effects of burstiness on system performance. We use a detailed simulation to verify that the conventional load balancing algorithms do not perform well in the presence of burstiness. We then describe our static ARA algorithm which tunes the load balancer by adjusting the best number of servers as targeting candidates based on the trade-off between randomness and greediness. To select an effective server for an incoming job, static ARA periodically queries the load information (e.g., queue length and utilization) from each server as the ranking criteria and then selects the top K servers as the best candidates. The higher

ranking values, the more likely jobs can be served with shorter queuing times. Then, the incoming job will be randomly submitted to a server among these K candidates. K is a fix number in static ARA. While this approach gives very good performance, tuning the number of K can be difficult. We therefore design our online ARA algorithm. In this version, a burst on-off prediction algorithm is designed by using the index of dispersion to accurately forecast the workload changes in user demands and system loads, and then online ARA can re-adjust the degree of randomness on-the-fly according to the workload changes.

1.2 Data Management in Multi-tiered Storage Systems

Except resource management in cluster systems, we also present data management in enterprise storage systems. One of the most important issue in data management is where to store the data sets and how to make them efficiently accessible. In this topic, we mainly focus on two aspects, 1) how to implement an auto-tiering algorithm in tiered storage system which moves data between types or classes of storage so that the most active data is on the fastest type of storage and the least active is on the most cost-effective class of storage, 2) how to improve the conventional caching algorithms in order to best utilize Flash resources in storage systems.

Tiered storage systems have gained prominence in modern enterprise storage arrays which combines SSDs with traditional HDDs. The SSD tier is used to cache the most active data by providing performance guarantee while the HDD tier is used to provide storage capacity for the remaining less active data. However, such hybrid storage architecture introduces the complexity of data management and maintenance. The performance improvement in tiered storage systems is subject to effectively place the right data at the the right tier during the right time. One of the main issues in data management is regarded to the diversity of SLAs. The enterprise storage arrays (e.g., SAN and NAS) often provide the shared resources or services to a large variety of applications which might demand for different performance goals. Hence, these storage systems should have the capability of controlling resources to achieve the performance goals of various applications and then to meet their associated SLAs. Yet, real application workloads are much more diverse and complex, which dynamically change over time. Bursty workloads and traffic surges are often found in enterprise data centers, which can cause different data temperatures (e.g., hot/cold) in the fundamental storage pools. Here, the term of hot/cold means the high/low access rate. Under bursty conditions, the traditional HDDs then yield poor QoS to the application, leading to the high latency of I/O operations and a large number of SLA violations.

Therefore, in the work, we present a new approach for automated data movement in

multi-tiered storage systems, which migrates the data across different tiers, aiming to support multiple SLAs for applications with dynamic workloads at the minimal cost. Using trace-driven simulations, we verify that bursty workloads and traffic surges that are often found in storage systems, dramatically deteriorate the system performance, causing high I/O latency and large numbers of SLA violations in low performance tiers. In order to mitigate such negative effects, hot data that are associated with those bursty workloads should be migrated to high performance tiers. However, extra I/Os due to data migration as well as the newly migrated bursty workloads can incur additional SLA violations to latency-sensitive applications in high performance tiers. Therefore, we designed a live data migration algorithm, called LMsT, in multi-tiered storage systems, which can (1) counteract the impacts of burstiness by efficiently utilizing the high-performance devices to improve the QoS for loose-SLA applications; and (2) minimize the potential delays to latency-sensitive applications by introducing priority-based queuing discipline which classifies I/Os by their corresponding SLA goals and serves them with different logical buffers. Furthermore, we show that LMsT can automatically detect all the possible migration candidates and verify the feasible ones by estimating the risk of SLA violations and quantifying the performance benefits via both the SLA and the performance constraints.

1.3 Data Management in Flash-based Storage Systems

Flash resources in storage systems can be allocated either within a storage array (e.g., NAS and SAN) or it can be located on the server itself. One popular approach of leveraging Flash technology in virtualization environments today is using Flash as a secondary-level host-side cache. Such Flash can be served as both write buffer and read cache to hold metadata and data and be synchronized with back-end magnetic drives. Lots of previous works studied how to utilize Flash within conventional storage solutions to construct an efficient data management system [8, 9]. In these works, however, they leverage some conventional caching policies [10–13] such as LRU and its variants to maintain the most recent accessed data for future reuse while some other works intended to design a better cache replacement algorithm by considering frequency in addition to recency [14,15]. These caching algorithms determine the cache admission and eviction on each data access which is independent of the practical I/O behavior, especially in virtualized storage systems. While straightforward, these conventional caching approaches have disadvantages in the following two aspects: 1) Cost- and performance-effectiveness. Since the cache is statically pre-allocated to each virtual disk, and the caching algorithm computes the cache admission and eviction independent of the I/O activities of other virtual machines, it is difficult for the hypervisor to cost-effectively partition and allocate Flash resources among multiple heterogeneous virtual machines with different workloads. 2) Scalability. Since caching is

usually implemented with a fine-grained cache line size (e.g. $4KB$, $8KB$), it requires a large number of CPU cycles for operations such as cache lookup, eviction, page mapping, etc., a large amount of memory space for maintaining cache metadata such as mapping table, LRU list, hash table, etc., and a fair amount of I/Os to update the content in Flash [16]. As the size of Flash storage grows to hundreds of GB or even several TB, the high cost of CPU, memory and I/O bandwidth reduces the benefit of virtualization, where virtual machines are contending the same pool of resources from host. Even worse, it hinders the deployment of Flash resources in large scale.

To address these problems, we explore the Flash usage model from the hypervisor's point of view, and define a new set of goals: maximize the performance gain, and minimize the incurred cost for CPU, memory and I/O bandwidth. With re-defined goals of using Flash, we design VMware Flash Resource Manager (vFRM) to manage Flash resources in the virtual machine environment. Based on long-term observation of the I/O access patterns, vFRM uses the heating and cooling concepts to model the variation of I/O popularity of individual blocks. With better understanding of the variation of I/O popularity, it predicts the most popular blocks in the future and places them into the Flash tier to maximize the I/O absorption ratio on Flash, which eventually maximizes the performance benefits from Flash resources. In addition, this leads to a great saving in memory space for keeping the metadata, and a significant reduction in I/Os that are needed for updating the contents in Flash, because vFRM updates the placement of data blocks between two tiers in a lazy and asynchronous manner.

In this thesis, our main contributions regarding resource allocation are summarized as follows:

1. Investigation of the impact of burstiness on load balancing in the cluster systems;
2. Presentation of a new class of load balancing algorithms, called ARA, in both static and dynamic versions;
3. Presentation of a new approach for automated data movement in multi-tiered storage systems and verification of the effectiveness and the robustness of LMST algorithm by trace-driven simulations;
4. Presentation of a new Flash resource manager, called vFRM, which aims to maximize the utilization of Flash resources with minimal I/O cost;
5. Investigation the benefits of G-vFRM for managing Flash resources among multiple heterogeneous VMs.

The remainder of this thesis is organized as follows. In Section 2, we discuss the background for resource management in cluster systems and introduce the background for

data management in storage systems. Section 3.1 demonstrates the effects of burstiness and information query delay in load balancer. In Section 3.2, we first present a static version of ARA, (see Section 3.2.1) which tunes the load balancer by adjusting the trade-off between randomness and greediness, as well as an online version, (see Section 3.2) which predicts the beginning and the end of workload bursts and automatically adjusts the load balancer to compensate. Performance evaluation of the load balancer ARA is presented in Section 3.2.2 and Section 3.3. Section 4.1.1 demonstrates the architecture of a multi-tiered storage system. Section 4.1.2 presents the LMST algorithm for automated data migration in multi-tiered storage systems. Section 4.1.3 evaluates the effectiveness and robustness of LMST using trace-driven simulations. In Section 4.2.1, we discuss the goals of leveraging Flash and analyze disk I/O traces of real workloads. Section 4.2.2 describes the details of VFRM design. Section 4.2.3 evaluates VFRM in contrast with existing caching solutions. In Section 4.3.1, we classify workloads into two categories and analyses access patterns of workloads. Section 4.3.2 presents the the global version G-VFRM. Section 4.3.3 evaluates the effectiveness of G-VFRM algorithm in both hit ratio and IO cost. Section 4.4 gives a summary of data management in storage systems. Finally, we draw the conclusion and future works in Section 5.

2 Background

2.1 Multi-tiered Cluster Systems

In today's climate, cluster systems are established as an industry standard for developing client-server applications. The large-scale cluster computing environments are being deployed in an increasing number of application areas these days. Examples of these systems include High Performance Computing (HPC), enterprise information systems, data centers, cloud computing and cluster servers. For example, multi-server cluster systems have been widely deployed in web-based services involved in the well-known flash crowd phenomenon in Web 2.0 workloads [17]. Lots of prior research have investigated the characteristics and algorithm optimization of such featured cluster systems [3–5]. Specifically, in a cluster, shared resources are provided as an unified hosting pool, which requires a central mechanism for resource provisioning and resource allocation based on the demands of multiple remote clients [1, 2].

A lot of research work [18–20] has been carried out on the study of multi-server clusters with a single system image, i.e., a set of homogeneous hosts behave as a single resource pool. Figure 1 illustrates the model of a simple cluster system. In such a multi-server cluster, a front-end dispatcher plays an important roll in dispatching incoming jobs among those back-end servers based on a certain algorithm. Such a front-end dispatcher is featured as a redirect buffer without central waiting queue while each back-end server has its own queue for waiting jobs and a separate processor that operates under the FCFS queuing discipline. In our work, we focus on the load balancing policy design for a front-end dispatcher in a multi-server cluster system. Nowadays, increasingly growing opportunity and commercial interest have arisen for both academic and industrial researchers to design and manage extensive applications and services in cluster systems which have been extended to a variety of platforms and infrastructures by providing pools of fundamental resources.

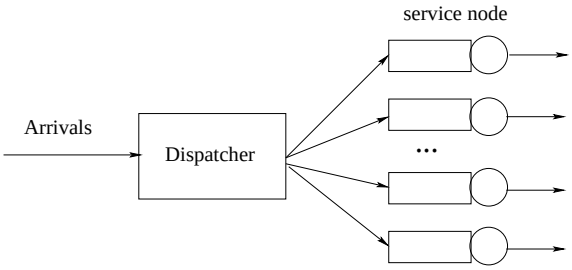

Figure 1: The model of a cluster system with N service nodes.

For example, cloud computing becomes quite popular among a community of cloud

users. Cloud computing platforms, such as those provided by Microsoft, Amazon, Google, and IBM, let developers deploy applications across computers hosted by a central organization. Cloud computing has become an important technology that affects our daily life by providing a shared "cloud" of servers as an unified hosting pool and scalable resources provisioning for multiple remote clients based on their demands [1, 2]. In the cloud, clients could access the services and deploy their own required platforms and infrastructures through application programming interfaces (APIs) over a large network of computing resources. Clients obtain the advantages of a managed computing platform, without having to commit resources to design, build and maintain the network. Yet, an important problem that must be addressed effectively in the cloud is how to manage QoS and maintain SLAs for cloud users that share cloud resources.

In cloud platforms, resource allocation (or load balancing) takes place at two levels. First, when an application is uploaded to the cloud, the load balancer assigns the requested instances to physical computers, attempting to balance the computational load of multiple applications across physical computers. Second, when an application receives multiple incoming requests, these requests should be each assigned to a specific application instance to balance the computational load across a set of instances of the same application. For example, in Amazon EC2, a real cloud platform that provides pools of computing resources to developers, Elastic Load Balancing (ELB) is the default load balancing service that redirects all the incoming application requests for load dispatch across multiple Amazon EC2 instances [21]. Application designers can direct requests to instances in specific availability zones, to specific instances, or to instances demonstrating the shortest response times. In another approach to balancing load in cloud computing systems, the load balancer can itself be an application that accepts incoming requests, monitors the availability of resources within the cloud, and distributes the requests. This approach can be applied the Microsoft and Amazon cloud computing infrastructures, as well as those of Google and IBM.

2.2 Load Balancers in Cluster Systems

A good scheme for load balancing in a dispatcher-based cluster system is significantly critical for system performance, such as average waiting time, average response time, system utilization, and job slowdown, etc. Most importantly, the market for cloud computing services has emerged and been dramatically growing. Providing a simple yet powerful load balancing policy to meet the varying application demands and SLAs in the cloud environment is significant and challenging.

A lot of previous studies have been focusing on developing load balancing policies for a large-scale clustered computing system over the past decades [18–20, 22, 23]. Examples of

these policies include Join Shortest Queue (JSQ) and the size-based AdaptLoad [6]. JSQ has been proven to be optimal [24] for a cluster with homogeneous servers, when there is no prior knowledge of job sizes and the job sizes are exponentially distributed. The extended version of JSQ was later presented in [25] which considers the non-decreasing failure rates in job size distribution. However, [26] evaluated the performance of JSQ under various workloads by measuring the mean response times and found that the performance of JSQ clearly varies with the characteristics of different service time distributions. The optimality of JSQ quickly disappears when job service times are highly variable and heavy-tailed [27–29].

Recently, size-based policies were proposed to balance the load in a cluster system, only using the knowledge of the incoming job sizes. The literature in [6, 22, 28, 30, 31] have shown that such size-based policies can be deployed in cluster systems to achieve the minimum job response times and job slowdowns. The AdaptLoad policy being a representative example of size-based policies, has been developed to improve average job response time and average job slowdown by on-the-fly building the histogram of job sizes and distributing jobs of similar sizes to the same server [6]. However, such size-based solutions are not adequate if the job service times are temporally dependent [19]. Not all size-based policies require a prior knowledge of the job service time distribution as the empirical distribution may be estimated on-the-fly by collecting statistics of the past workload seen by the system [6]. A required condition for size-based policies is that upon job arrival at the dispatcher, an accurate estimate of the job service time is possible. This condition restricts our discussion here to systems where accurate estimation of job service times is possible.

Other load balancing policies have been developed as well for cluster environments. For example, the Min-Min and the Max-Min algorithms focus on the problem of scheduling a bag of tasks and assume that the execution times of all jobs are known in advance [7]. Meta-schedulers, like Condor-G [32], rely on the accurate queuing time prediction in scheduling jobs on computing grids. However, accurate predictions become more challenging in the current virtualized and multi-tiered environments. Recently, techniques of advance-reservation and job preemption were presented to allocate jobs in cluster systems [33]. Yet, extra system overheads and job queue disruptions could decrease the overall system performance.

Each of the above classic policies is widely used because of its simplicity and efficiency. However, there is no universal policy which claims to be the best in all circumstances since the workload distribution is also a key factor on the system performance. Previous studies believed that the requests follow an exponential distribution and thus designed the policies based on this assumption. As research goes deeper, people realized that it is not always the case especially in the cluster systems and cloud environments, since many

applications in these systems are no longer single-program-single-execution applications. These applications involve a large number of concurrent and dependent jobs, which can be executed either in parallel or sequentially. Simultaneously, launching jobs from different applications during a short time period can immediately cause a significant arrival peak, which further aggravates resource competitions and load unbalancing among computing sites. Also, as the number of these applications significantly increases in recent years, the present of traffic surges becomes more frequent. As a result, how to counteract burstiness and maintain high quality of service and system availability becomes imminently important but challenging as well. However, conventional methods unfortunately neglect cases of bursty arrivals and cannot capture the impacts of burstiness on system performance.

2.3 Bursty Workloads in Real Systems

Bursty workloads are often found in multi-tier architectures, large storage systems, and grid services [34–36]. Internet flash-crowds and traffic surges are familiar examples of bursty traffic, where bursts of requests are aggressively clustered together during short periods and thus create spikes with extremely high arrival rate. *Burstiness* or *temporal surges* in the traffic to modern Internet systems generally turns out to be catastrophic for performance, leading to dramatic server overloading, uncontrolled increase of response times. We argue that the presence of burstiness can cause load unbalancing in clouds and consequently degrade the overall system performance.

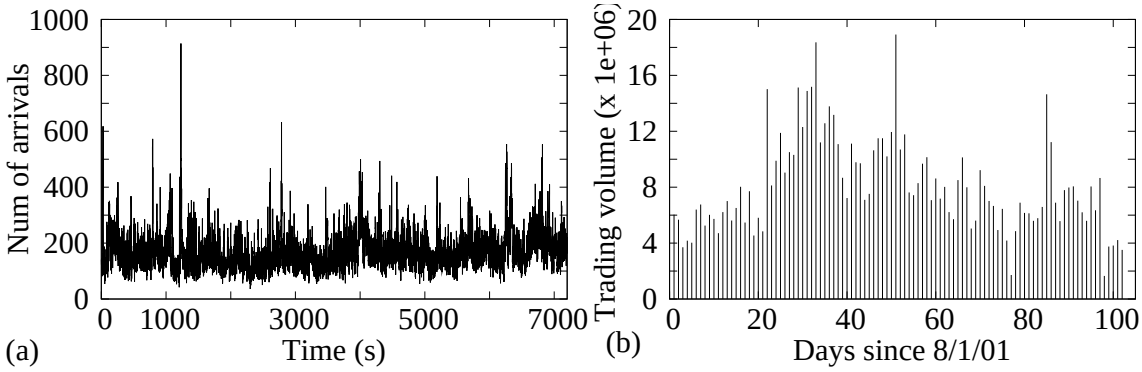

Figure 2: Illustrate burstiness in (a) arrival rates of TCP packages and (b) daily trading volume of IBM stock.

We give two examples of real-world situations where burstiness exists. The first workload, LBL-TCP-3, includes information from all TCP packets sent by Lawrence Berkeley Laboratory over a two-hour period in January, 1994 [37]. Each packet has a time-stamp, and the trace includes 1.8 million packets. Figure 2(a) shows a clear bursty pattern in the number of arrivals per second over the entire trace. The second workload, S&P500,

consists of intra-day trading volumes for all stocks listed in the S&P 500 trading index. The dataset includes the number of trades executed for each stock at one-minute intervals over 11 years from 1998-2009. Trading volumes can be bursty, as described by [38]. Sudden fluctuations in a stock's trading volume can be a result of the distribution of news that changes a stock's value, the effect of nightly and weekly interruptions in trading, as well as the intrinsically chaotic behavior of the stock market. An example of this bursty behavior for a single stock's trading volume is shown in Figure 2(b).

Burstiness and temporal dependent structure can be captured by autocorrelation function (ACF). Autocorrelation is a mathematical measurement of correlation coefficient in statistics and probability theory. Autocorrelation is used as a statistical measure of the relationship between a random variable and itself [39]. Consider a set of random variables X_n, the following equation defines the autocorrelation $\rho_{X(k)}$ of X_n for different time lags k:

$$\rho_{X(k)} = \rho_{X_t, X_{t+k}} = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}, \tag{1}$$

where E is the expected value, μ is the mean and σ^2 is the common variance of X_n. The parameter k is called the lag and represents the distance between the observed value at different time and itself, i.e.,$X_{(t+k)}$ and X_t. The values of $\rho_{X(k)}$ fall in the range $[-1, 1]$. If $\rho_{X(k)}$ =0, then there is no autocorrelation (independence) at lag k . In most cases ACF converges to zero as k increases. A positive value of $\rho_{X(k)}$ indicates there exists autocorrelation of X_i while a negative value implies anti-autocorrelation. A high absolute value of $\rho_{X(k)}$ implies that there is strong temporal locality on X_n, i.e., the value of X_i has a high probability to be followed by another variable with the same value of magnitude. The decay rate of ACF determines if the process has a weak or strong correlation. We consider the process as high autocorrelation case if the value of X_i is greater than 0.4. An independent process is determined only if $\rho_{X(k)} = 0$ at lag k.

In [39], the authors observed the existence of correlated (dependent) flows over a wide range of Internet traffic including WEB servers, E-mail servers, User Accounts servers and Software Development servers in both inter-arrival times (i.e., the arrival process) and service times (i.e., the service process). Their work also examined the different impacts of autocorrelation under both open and close systems. In an open system with infinite buffers, the autocorrelation in the arrival or service process of a queue will only affect the performance of downward queue. While in multi-tiered systems with a close loop structure, if ACF exists in any of the tiers, then it propagates across the entire loop in the closed system and is present in the arrival stream of tiers that precede that tier [39, 40]. Although it balances the load among all queues (decreases mean queue length and mean response time of the bottleneck queue and increases those of the non-bottleneck queue), the overall performance (mean round trip time and mean throughput) still degrades.

Index of dispersion I can also be considered to characterize the burstiness in workloads. From the mathematical and statistical perspective, the index of dispersion is a normalized measure of the dispersion of a probability distribution. From the characterization of burstiness case, it is defined as follow:

$$I = SCV(1 + 2\sum_{k=0}^{\infty} \rho_k), \tag{2}$$

where SCV is the squared coefficient of variation and ρ_k is the autocorrelation on lag-k. Note that $I = 1$ when the distribution is exponential. Thus, the index of dispersion can be considered as the ratio of the observed autocorrelation with respect to a Poisson distribution and the value of I can thus be used as a good indicator of autocorrelation. In Section 3.2.2, we use index of dispersion as a metric to detect the burst on-off status in workloads.

2.4 Multi-tiered Storage System

The volume of data in today's world has been tremendously increased. For example, Facebook revealed that its system each day processes 2.5 billion pieces of content and more than 500 terabytes of data, including 83 million pictures. Being one of the largest databases in the world, Google processes more than 25 petabytes of data per day. As more and more people use different types of devices such as smartphones and laptops, data comes from everywhere, including body sensors for collecting medical data and GPS devices used to gather traffic information. Such massive and diverse data sets will then lead to challenging issues for system designers to address. One of the most important issues is where to store these gigantic data sets and how to make them accessible. By providing a way to combine various storage media types, multi-tiered storage architectures become attractive in enterprise data centers for achieving high performance and large capacity simultaneously.

In 1990s, flash-based SSDs were first introduced to maintain the data in the memory chips. NAND flash memory is the basic building block of SSDs as electronic non-volatile storage medium. A typical SSD is composed of host interface logic (e.g., SATA and PCI Express), an array of NAND flash memories, and a SSD controller. Read and Write operations are performed at the granularity of a page (e.g., 4KB), while erase operation is performed at the granularity of a block (e.g., 64 pages). Nowadays, SSDs have gained prominence in enterprise arrays and been successfully used as a replacement of HDDs because of significant performance improvement (i.e., higher IOPS and lower latency) and low energy consumption. Yet, given the fact that SSDs are more expensive per gigabyte (GB) and have a limited number of writes over the life time, a multi-tiered storage

platform, which combines SSDs with traditional HDDs (e.g., FC/SAS and/or SATA), has become an industrial standard building block in an enterprise data center, where SSDs are used as the top tiers to guarantee fast data access while the traditional HDDs function as the bottom tiers to provide large storage capacity. However, from service provider's perspective, how to efficiently manage big data across these hybrid storage resources in order to provide high quality of service is still a core and difficult problem due to the following two issues.

The first issues in resource management is regarded to the diversity of SLAs. Modern enterprise data centers often provide shared storage resources to a large variety of applications which might demand for different performance goals such that different SLAs have to be met. Hence, these data centers need to be SLA-aware in the management of shared storage resources in multiple storage tiers to achieve different performance goals for applications and meet their associated SLAs. In this work, we refer to SLA as I/O latency (or I/O response time) in millisecond (ms). Intuitively, latency-sensitive applications with strictly high SLAs should occupy SSDs to avoid SLA violations while the traditional HDDs like FC and SATA disks should be assigned to serve the applications with loose SLAs.

Second, an effective resource management has to dynamically adjust its policy according to different application workloads. In practice, workloads may change over time. Bursty workloads and traffic surges are often found in enterprise data centers. For example, a user query from a data-intensive application might easily trigger a scan of a gigantic data set and then bring a burst of disk I/Os into the system, which will eventually cause disastrous SLA violations, performance degradation and even service unavailability on the traditional slow HDDs. Thus, ideally, SSDs should serve applications under bursty workloads in order to mitigate their burdens, even if these applications require loose SLAs.

As enterprises consolidate a variety of applications that require different service levels, it becomes an urgent demand to build a multi-tiered storage platform for providing different levels of service and performance in the storage domain [41]. Therefore, *storage tiering* techniques are introduced to dynamically deliver appropriate resources to the business, targeting at performance improvement, cost reduction and management simplification. Because of its significant importance, the technology of storage tiering has been recognized by ESG's 2011 Spending Intentions Survey [42], as one of the top 10 planned storage investments in the next couple years. The market landscape report [43] from ESG further points out that the present market of storage tiering can be classified into five main segments, including array-based migration, array-based caching, file system-based, software tools and archive emphasis. Especially, many industrial companies have already developed their own automatic tiering technologies and released the relative products, such as IBM Easy Tier for DS8000 [44], EMC Fully Automated Storage Tiering (FAST) for

Celerra [45], and HP Adaptive Optimization for 3PAR [46]. These new tiering techniques indeed differ in many features, including the number and the type of tiers, the direction and the frequency of data migration among tiers, the capability of learning and training, and the granularity in tiering, etc.

A large literatures on storage management have been developed for the years. Recently, [47–54] proposed several new techniques (algorithmic or theoretical) to explore the effective data migration in storage systems. For example, [47,48] have investigated the idea of using edge-coloring theory for data migration and achieved a near-optimal migration plan by using polynomial-time approximation algorithms. Triage, an adaptive controller, has been proposed in [50] to address the problem of performance isolation and differentiation in a consolidated data center. By throttling storage access requests, Triage ensures high system availability even under overload conditions. Later, [49] focused on minimizing the overhead of data migration by automatically detecting hotspots and reconfiguring the system based on the bandwidth-to-space ratio. [54] proposed a dynamic tier manager, named EDT-DTM, which performs dynamic extent placement once the system is running to satisfy performance requirements while minimizing dynamic power consumption. However, we argue that none of the existing studies take account of both the on-the-fly migration penalties and the various application SLAs for data migration in multi-tiered storage systems.

A cost model [51] has been developed to solve the problem of efficient disk replacement and data migration in a polynomial time. [52] implemented the QoS guarantee of performance on foreground work by leveraging a control-theoretical approach to dynamically adapt migration speed. [53] proposed a lookahead data migration algorithm for SSD-enabled multi-tiered storage systems, where the optimal lookahead window size is determined to meet the workload deadlines. However, the work [53] assumes that the I/O profile exhibits a cyclic behavior and does not consider different application SLAs in their algorithm.

2.5 Host-side Flash in Storage System

On the other side, there is an increasing host-based deployment of NAND-based flash as a second-level cache [16,55–62], given its low latency and low power consumption. Host-side flash-based caching offers a promising new direction for optimizing access to data that has been being widely accepted in modern storage systems. Memcached is a distributed memory caching system by adding a scalable object caching layer to speed up dynamic Web applications and alleviate database load [8]. However, Memcache is more like an in-memory data store rather than a caching strategy in a storage system. Flashcache is a kernel module which is built using the Linux Device Mapper (DM) and works primarily

as a write back block cache in general purpose [9]. Recently, Facebook announced a new data management infrastructure, called TAO, in which its caching layer is designed as a globally distributed in-memory cache running on a large collection of geographically distributed server clusters [63].

There are many research literature that studied the problem of how to best utilize the flash resources as a cache-based secondary-level storage system or integrated with HDD as a hybrid storage system. Some conventional caching policies [10–13] such as LRU and its variants maintain the most recent accessed data for future reuse while some other works intended to design a better cache replacement algorithm by considering frequency in addition to recency [14, 15]. These caching algorithms determine the cache admission and eviction on each data access which is independent of the practical I/O behavior. [64] uses the flash resources as a disk cache and adopts an LRU-based wear-level aware replacement policy. SieveStore [59] presented a selective and ensemble-level disk cache by using SSDs to store the popular sets of data. [65] proposed a hybrid storage system, called Hystor, by fitting the SSD into the storage hierarchy. The SSD plays a major role by identifying the performance- and semantically-critical data and timely retaining these data to the SSD. [54] presented a SSD-based multi-tier solutions to perform dynamic extent placement using tiering and consolidation algorithms. In these existing schemes [54,65], Flash resources are managed using caching policies such as LRU or its variants, aiming to maintain the most likely-to-be accessed data for future reuse. While straightforward, these approaches fail to fully exploit Flash resources in terms of cost- and performance-effectiveness. Moreover, they limit the scalability with respect to the ever-increasing size of Flash storage.

To address these problems, a new Flash usage model should be explored with a new set of goals: consolidating the use of flash so as to maximize performance gain, while minimizing management cost and operational cost [66, 67]. In other words, instead of solving the performance issues of individual workload, we focus on improving the utilization of the CAPEX of flash resources, and reducing the OPEX that is needed for managing and operating them.

3 Computational Resource Management in Cluster Systems

Cloud computing nowadays becomes quite popular among a community of cloud users by offering a variety of resources. However, burstiness in user demands often dramatically degrades the performance of applications in the cloud. In order to satisfy peak user demands and meet SLAs, effective resource allocation schemes are highly demanded in the cloud. However, we find that conventional load balancers unfortunately neglect cases of bursty arrivals and thus experience significant performance degradation.

Motivated by this problem, we propose a new class of burstiness-aware algorithms which attempt to balance bursty workloads across all computing sites in the cloud and then improve the overall system performance. The contributions of this work include:

1. investigation of the impact of burstiness on load balancing in the cluster systems;
2. presentation of a new class of load balancing algorithms in both static and dynamic versions;
3. implementation of a new algorithm, named ARA, as a new load balancer in Amazon EC2;
4. evaluation of the proposed algorithms under both bursty and non-bursty workloads by simulations and real experiments in Amazon EC2.

3.1 Motivation

In this section, we first demonstrate the impact of burstiness on load balancing in a distributed simulation environment, which is developed on the CSim library [68]. We refer the interested readers to [69] for the details on system design and remark that such a simulation environment can be used to simulate a cloud computing framework. In our simulation, the system consists of N computing sites, where each site runs the First-In-First-Out (FIFO) policy to schedule the assigned jobs. The specifications of a job, including job inter-arrival time and job execution time, are created based on the specified distributions and methods.

To select an effective site for an incoming job, a load balancer periodically queries the load information (e.g., queue length and site utilization level) about each site as the ranking criteria from the host resource management systems. The load balancer then selects a computing site that has the highest ranking value (such as the shortest queue length) among all sites of the targeting application. The higher ranking values, the more likely we can complete jobs with shorter queuing times and thus obtain better system performance. Such a load balancing scheme can be referred to as "greedy" because it

always selects the top-ranked site for service. We also evaluate another load balancing scheme, dubbed as *Rand*, which randomly selects one among all available sites.

To demonstrate the performance impact of bursty arrivals, we run the simulations under three different arrival processes with burstiness profiles as shown in Figure 3. Each arrival process is drawn from a 2-state Markovian-Modulated Poisson Process (MMPP)[1] that can be parameterized to have the same mean equal to 10s but three different levels of burstiness: strong, weak, and non-bursty, such that the corresponding values of index of dispersion I are equal to 313.5, 32.25, and 1, respectively, see the detailed discussion of I in Section 2.3. Here, we remark that the index of dispersion has been frequently used as a measure of burstiness in the analysis of time series and network traffic [71,72]. The higher I indicates stronger burstiness in workloads. We observe that the number of arrivals are significantly varied under the three different workloads. In all experiments, the system consists of $N = 16$ sites and has an average site utilization equal to 50%.

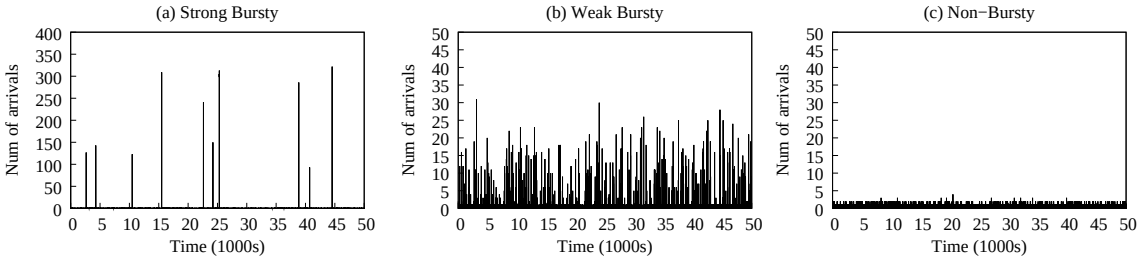

Figure 3: Number of arrives per second under the three workloads with mean inter-arrival times equal to 10s.

Table 1 shows the average response times of two load balancers. We first observe that burstiness in arrivals dramatically degrades the system performance under both two algorithms. As the intensity of burstiness increases, such negative impacts on system performance become more significant. More importantly, the "greedy" load balancer, *Qlen*, outperforms when there is no burstiness in arrivals yet ceases to be effective due to the imbalance of load among computing sites when the workload arrival process is bursty. We interpret this effect by observing that the greedy algorithms cannot detect system load surges on computing sites during bursty arrivals because of the delay in updating load information from sites, and thus make incorrect decisions based on the outdated information. For example, once a job is assigned to a computing site, the associated load information (e.g., the present queue length) of that site cannot be updated immediately at the load balancer. As a result, the load balancer always submits the bursty arrivals to that top-ranked site within the delay period[2]. Consequently, significant load is incurred

[1]Markovian-Modulated Poisson Process (MMPP) is a special case of the Markovian Arrival Process (MAP) [70], which is used here to generate bursty flows because it is analytically tractable.

[2]In our simulation, we set the information query delay D as 1 second. The sensitivity analysis to D

on that particular site, resulting in the performance degradation under bursty workloads.

Table 1: Mean response times of two load balancers under the three workloads. The number of computing sites is $N = 16$ and the information query delay is $D = 1$s.

Response time	Strong-bursty Fig. 3 (a)	Weak-bursty Fig. 3 (b)	Non-bursty Fig. 3 (c)
Rand	1520.9s	168.5s	80.5s
Qlen	6541.5s	466.5s	7.6s

We stress that such an information query delay unfortunately is *unavoidable* in real systems because when a job is submitted to a site, it takes non-negligible time for that particular site to update the information about system load. Similarly, the communication for querying and broadcasting such load information between the distributed load balancers and the sites via network also take a non-negligible amount of time among clouds. Therefore, we argue that *such deleterious effects due to burstiness and information query delay must be considered in the performance evaluation and load balancer design for cloud computing.*

3.2 New Load Balancer: ArA

We present a new load balancing algorithm, called "ARA", for adaptive resource allocation in cloud systems, which attempts to counteract the deleterious effect of burstiness by allowing some randomness in the decision making process and thus improve overall system performance and availability. In the remainder of this section, we first present a static version of ARA, (see Section 3.2.1) which tunes the load balancer by adjusting the trade-off between randomness and greediness in the selection of sites, as well as an online version, (see Section 3.2) which predicts the beginning and the end of workload bursts and automatically adjusts the load balancer to compensate. Performance evaluation of the proposed load balancer ARA is presented in Section 3.2.2 and Section 3.3.

3.2.1 Static Version

To address the load unbalancing problem caused by burstiness, we present a new load balancer which can balance bursty workloads across available resources and thus improve the overall system performance. Later, we show how this new load balancer can be deployed for load balancing across a set of instances of the same application in a real cloud platform.

We observed in Section 3.1 that under non-bursty conditions the "greedy" methods that always select the best site, obtain better performance than the "random" ones. But

will be given in the next subsections.

we also observed the advantage of distributing jobs randomly among all computing sites under bursty conditions. This observation inspires us to design a new ARA algorithm which adjusts the randomness and the greediness in the decision making process.

Algorithm: static version of ARA
1. initialize
  a. number of candidates: $K = k$;
  b. information query delay: $D = d$;
/ load information updating*/*
2. **for** each window of D time
  a. send queries to all computing sites for load information;
  b. update load information received from all computing sites;
  end
/ site selection process */*
3. upon each job arriving
  a. sort all sites S_i, $1 \leq i \leq N$, by current load information;
  b. set $S = \{S_1, S_2, ..., S_K\}$; /* get K sites with least load */
  c. set $s = \texttt{uniform}(1, K)$; /* randomly select one site from the candidate set S */
  d. submit the job to site S_s;
  end

Figure 4: The high level idea of the static ARA.

Given an incoming job and N available computing sites, ARA finds K sites, where $K \leq N$, as the best candidates for serving that job, using queue length as the ranking criterion. Then, that particular job will be randomly submitted or enqueued to one site among the selected K candidates. The value of K in ARA is critical for system performance, which in turn should be set appropriately based on the intensity of burstiness in workloads. For example,

- under the case of no burstiness in arrivals, K is set to small values (i.e., close to 1). It turns out that ARA performs exactly the same as the "greedy" load balancer, *always* selecting the best site with shortest queue length;
- under the case of extremely strong burstiness in arrivals, the number of best candidates is set equal (or close) to the total number of available sites, i.e., $K = N$. Consequently, ARA has behavior similar to the "random" method, which allows the bursty workload to be shared among all sites, therefore alleviating the imbalance of load;
- otherwise, K is set to the value between 1 and N.

As a result, ARA dispatches the load among sites by combining the features of both *Qlen* and *Rand*. Figure 4 presents the high level idea of this static version of ARA.

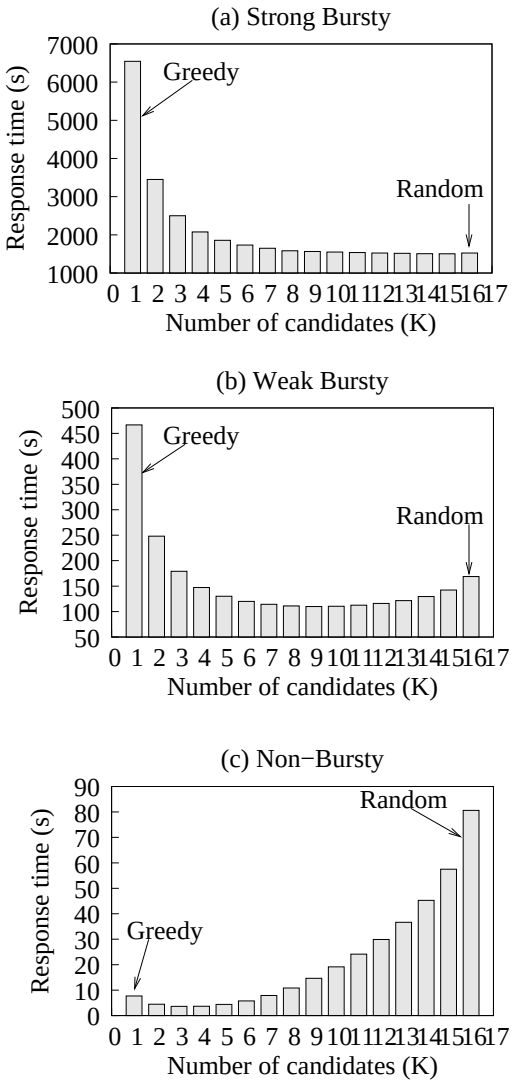

Figure 5: The average response times of the ARA load balancer as a function of the number of candidates K under (a) strong bursty workload, (b) weak bursty workload, and (c) non-bursty workload. The average response times of the *Qlen* and *Rand* load balancers, as well as the best performance of ARA (see the black bars) are also marked in the plots.

In order to evaluate the performance of ARA, we here investigate the sensitivity analysis over a range of bursty conditions and statically set the value of K from 1 to N. Figure 5 shows the average response times under ARA as a function of the number of candidates K, as well as ones under both *Qlen* (see the left most bar in the figure) and *Rand* (see the right most bar in the figure) policies. These results give us a first proof of concept that ARA with an appropriate K value can be beneficial for performance of cloud applications

with bursty arrivals. For example, in the case of non-bursty condition, a small K (e.g., $K = 3$ in Figure 5 (c)) allows ARA to achieve performance similar to *Qlen*, which greedily chooses the best candidate for the incoming jobs and thus obtains the best performance. As burstiness becomes stronger, the value of K then keeps increasing which allows ARA to behave almost the same as *Rand* counteracting the load unbalancing problem incurred by burstiness, see Figure 5 (a). We also notice that our static ARA achieves very similar performance as the algorithm in [73], which considers the supermarket model such that customers can randomly choose a constant number of servers and waits for service at the one with the fewest customers.

However, such performance improvements depend on the degree of randomness that is introduced by the number of top candidates K. A good choice of K can result in significant performance improvements, but an unfortunate choice may also result in poor performance. Furthermore, real traffic of dynamic cloud environments indeed changes over times: extremely busy in some periods and quite idle in other periods. We thus remark that with a fixed K both static ARA and the algorithm in [73] cannot always achieve the best performance across different bursty conditions. To quickly adapt to the changes in user demands, an effective way for online adjusting K, instead of using a fixed K, becomes imminently important in cloud systems.

3.2.2 Online Version

Here, we design an online version of ARA which can re-adjust the degree of randomness (i.e., K) on-the-fly according to the workload changes. We first leverage the knowledge of burstiness to develop predictors which can accurately detect the changes in user demands and then present the online ARA which dynamically shifts between the "greedy" and the "random" schemes based on the predicted information.

On-Off Predictor: We incorporate the index of dispersion [71,72] I to detect bursts in the incoming traffic. The advantage of I is that it can qualitatively capture burstiness in a single number and thus provide a simple yet powerful way to promptly identify the start and the end of a bursty period. As discussed in Section 2.3, the joint presence of variance and autocorrelations in I is sufficient to discriminate traces with different burstiness intensities and thus to capture changes in user demands.

To understand how I performs as a single measure, we illustrate the arrival rates (i.e., the number of arrivals per 100 seconds) of a bursty workload across the time in Figure 6 (a). The trace shown in this plot consists of two idle phases and one single peak phase. We divide the whole trace into five parts during the following time windows: $W_1 = [40K, 50K), W_2 = [50K, 55K), W_3 = [55K, 64K), W_4 = [64K, 70K)$, and $W_5 = [70K, 75K)$, where only windows W_2 and W_4 cover both idle and peak phases while the

remaining windows include only one phase. We also measure the corresponding index of dispersion for each window, see the values of I marked in the plot. We notice that the values of I are quite small when the trends of traffic are stable during both idle and peak phases, e.g., windows W_1, W_3, and W_5, however, for the windows with clear changes in traffic, e.g., W_2 with the burst arriving and W_4 with the burst ending, the values of I significantly increase. This observation indicates that dramatic changes in I can be used as a measure criterion to detect the start and the end of bursty arrivals and further predict the changes in user demands.

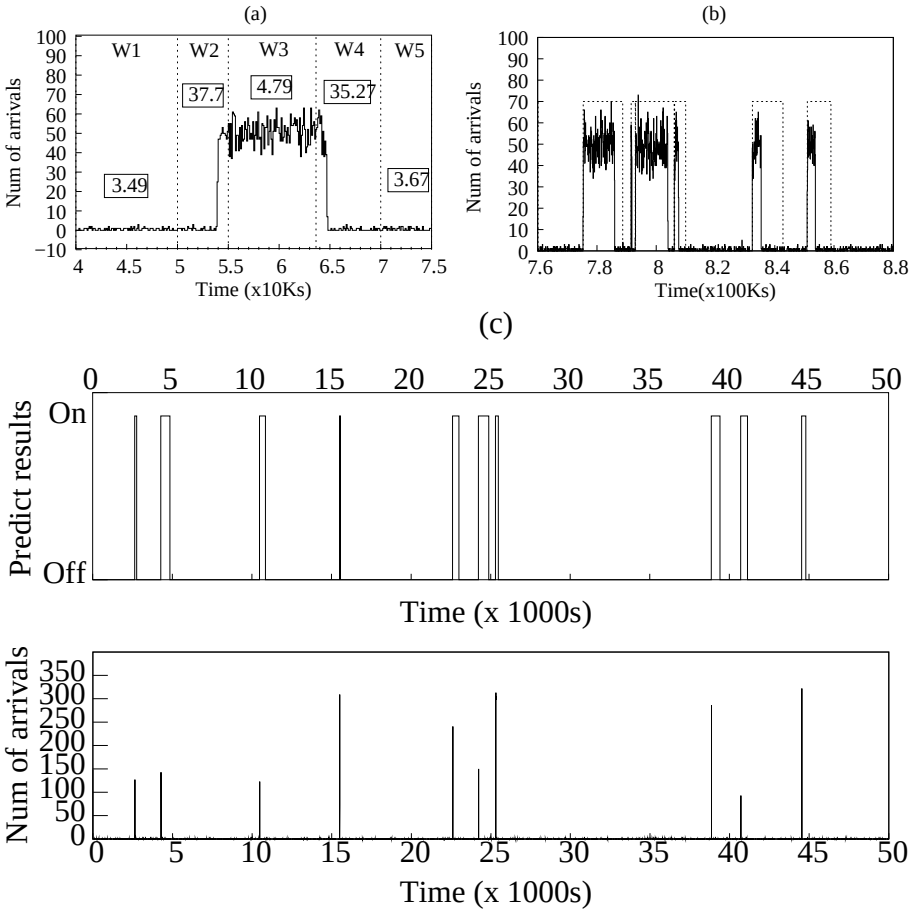

Figure 6: Illustrating (a) the index of dispersion that are measured within five monitoring windows under a bursty workload, (b) prediction results that accurately capture the start and the end of bursts, where the solid lines are actual traffic (i.e., arrival rates across time) and dashed lines show the detection of when burstiness starts (state "on") and when it ends (state "off"), and (c) prediction results for the strong bursty workload, see Fig. 3(a), where the above plot shows the detection and the bottom plot presents the actual traffic.

In [74], an algorithm has been proposed to use I coupled with information about the current and previous arrival rates to detect changes in arrival intensities. In this paper, we

consider to exploit this algorithm for identifying changes in cloud user demands. However, we also find that the algorithm in [74] cannot accurately detect the start and the end of some bursts. Especially, the end of a burst is easily missed because of the deficiency of the algorithm, which results in the unnecessary delay in the detection of changes from peak to idle. In addition, the monitoring window size used in [74] is too large, which although is beneficial to capture the state transition, further extends the delay of detection in the ending of bursts.

In order to improve the prediction accuracy, we refine the algorithm by dynamically adjusting the monitoring window size m instead of a fixed value in [74] to trade off the contradiction of monitoring window size and detection delay. To shorten the detection delay, a small window size is preferred which however may miss the detection of state changes, especially the end of bursts. This is because m now is too short to provide sufficient samples for readjusting I from small values to large ones, see W_3 and W_4 in Figure 6. In our algorithm, we initially choose a small value of m, but dynamically enlarge the monitoring window (e.g., $2m$ requests) to collect enough samples for updating I, given that the original window size (i.e., m) is not large enough.

Figure 6 (b) shows the outputs of the algorithm, where state "on" indicates the start of a burst and state "off" means the end of a burst. We can see that the changes of states "on" and "off" correctly follow the actual bursts plotted in solid lines in the figure. One should notice that the algorithm is slower in the detection of an idle period. This is the outcome of our new dynamic window size, which indeed has negligible impacts on our new load balancer's performance because of few arrivals during idle periods. Figure 6 (c) further validates the effective of this new predictor algorithm, illustrating the accurate prediction results for the arrival traffic with strong burstiness, as shown in Figure 3 (a). We expect that this new refined predictor can accurately forecast the changes in user demands and thus can provide significant valuable information to ARA for effectively load balancing in clouds.

Online Adjusting of K: Motivated by the fact that Internet flash-crowds and traffic surges often present in real systems, we now propose a new load balancing algorithm, named ARA_PRED, that detects the phases of "*burst*" and "*idle*" in user demands and further discriminates these two phases by introducing different degrees of randomness in an online fashion. In particular, when the predictor detects the start of a burst, we increase the degree of randomness by setting K to a large value th_l close to the total number of available sites. On the other hand, when the predictor detects the start of an idle period, the value of K is be decreased to a small value th_s close to 1. The degree of greediness is then increased and ARA performs closely to *Qlen*. As a result, by leveraging the knowledge of burstiness, this new load balancer can quickly adapt to changes in user demands by shifting between the "greedy" and the "random" schemes, and thus optimize

the utilization of available resources and application performance by making a smart site selection for cloud users. The high level idea of the online ARA is described in Figure 7.

Algorithm: online ARA
1. initialize
  a. the large threshold th_l for K; /* e.g., $th_l = \lceil 0.5 * N \rceil$ */
  b. the small threshold th_s for K; /* e.g., $th_s = 1$ */
2. run the prediction algorithm;
3. upon the detection of changes in user demands
  a. if detect the start of "*burst*"
    then increase K to th_l;
  b. if detect the start of "*idle*"
    then decrease K to th_s;
  c. use K for the site selection process as shown in Fig. 4;
  end

Figure 7: The high level idea of the online ARA.

3.2.3 Performance Improvement of ArA_Pred

To investigate the performance of the online ARA, we here consider a case such that user demands arriving during the "burst" and the "idle" phases *both* have non-negligible impacts on the system load, as well as the overall system performance. For example, in the arrival trace used by the following experiments, there are almost half of traffic arriving when the system is relatively idle, although 51% of jobs aggregate in bursts. It becomes sophisticated and time consuming to search a good value of K for the static version. Some value of K may benefit the arrivals during "idle" periods but degrade the performance of those in the "burst" periods, vice versa. Thus, adjusting values of K based on the changes in traffic becomes more important to such a case.

Figure 8 depicts the performance measures (e.g., the average response times) under the online version of ARA. The results under the greedy (e.g., *Qlen*) and the random (e.g., *Rand*) algorithms are plotted in the figure as well. Also, in order to evaluate the prediction algorithm, we present the results for a new version of ARA, dubbed as ARA_OPT, that assumes to have a prior knowledge of each job's arrival time and thus makes an exact detection of when the burst starts and when it ends. This version thus provides an upper bound for ARA_PRED. Note that when both th_s and th_l are equal to 1, ARA_PRED performs exactly as *Qlen*. In all experiments, the number of computing sites is $N = 16$, the average utilization of each computing site is 50%, and the information query delay is $D = 1$s. Additionally, we here fix the small threshold th_s as 1 but change the large threshold th_l from 1 to 16 in Figure 8 (a), while fix the large threshold th_l as 14 but change the small threshold th_s from 1 to 16 in Figure 8 (b).

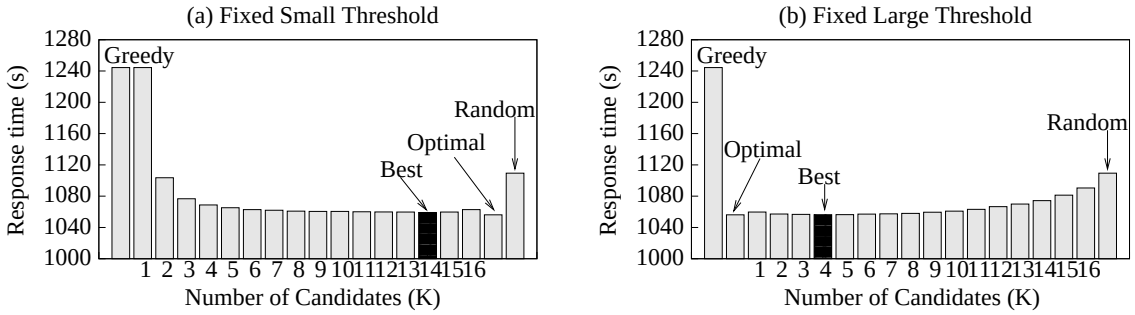

Figure 8: The average response times under the online version of ARA, where (a) the small threshold th_s is kept as 1 while the large threshold th_l is changed from 1 to 16, and (b) the small threshold th_s is changed from 1 to 16 while the large threshold th_l is kept as 14. The performance under *Qlen*, *Rand* and ARA_OPT are also plotted. Here, the number of computing sites is $N = 16$, and the average site utilization is 50%, and the information query delay is $D = 1$s.

The results shown in Figure 8 first confirm that neither *Qlen* nor *Rand* is able to obtain good performance for this workload. *Qlen* even presents the worst behavior because of the moderate burstiness in arrivals. Instead, our new algorithms ARA_PRED and ARA_OPT significantly improve the system performance, using distinguished values of K for the phases with different burstiness intensities. Also, ARA_PRED performs closely to the one with optimal forecasting, validating the accuracy of our prediction algorithm.

More importantly, ARA_PRED can always achieve such performance improvements as long as th_l is larger than some thresholds (e.g., 8 in Figure 8 (a)) and th_s is smaller than some thresholds (e.g., 6 in Figure 8 (b)). This is because jobs in a "burst" phase could be almost equally distributed among all the sites in the following cases: when $K = 8$ and the duration of a "burst" phase is short (e.g., around 2s), the jobs in this phase may be sent randomly to the top 8 sites in the first second and to the remaining 8 sites in the second seconds, leading to the similar results as the case that $K = 16$ and all jobs are sent randomly to 16 sites in two seconds. Therefore, we argue that the results shown in Figure 8 demonstrate that our algorithms ARA_PRED has more robustness, which provides a simple yet flexible knob for deciding the value of K. In contrast, the static version described in Section 3.2.1 and the algorithm proposed in [73] require more efforts to tune the values of K, which is sophisticated when the workload is dynamically changed.

3.2.4 Sensitivity Analysis on Experimental Parameters

Now, we turn to analyze the effects of different experimental parameters on ARA_PRED's performance. We first focus on investigating the sensitivity of ARA to the network size (i.e., the number of computing sites) by evaluating job response time for $N = 8, 16$, and

32. In all experiments, we scale the mean service times in order to fix the site utilization levels equal to 50%. All the other parameters are kept the same as the experiments shown in Figure 8.

The performance results under the four algorithms are shown in Table 2(a). These results first confirm that the conventional algorithms (e.g., *Rand* and *Qlen*) poorly behave under all the three network sizes, and our new ARA ones improve the system performance by discriminating bursty periods from non-bursty ones. We also observe that as the system becomes larger (i.e., N increases), jobs experience worse response times under the "greedy" and the "random" methods. But, such a performance trend disappears under the two ARA ones. We interpret that as the number of sites becomes larger, it is more likely for *Qlen* (resp. *Rand*) to make wrong decisions for bursty (resp. non-bursty) traffic, resulting in more dramatic degradation on system performance. On the other hand, by online adjusting the values of K for bursty and non-bursty traffic, two ARA algorithms select the good sites for incoming jobs, which may have less loads (i.e., the number of queuing jobs) as the number of sites increases and thus reduce the waiting times for those jobs.

(a)

network size	Load Balancer			
	Rand	ARA_OPT	ARA_PRED	*Qlen*
8	1089.25	1063.39	1064.66	1101.02
16	1109.33	1056.07	1059.00	1244.32
32	1148.38	1042.79	1051.21	1751.43

(b)

delay time	Load Balancer			
	Rand	ARA_OPT	ARA_PRED	*Qlen*
1s	1109.33	1056.07	1059.00	1244.32
2s	1111.07	1057.76	1062.97	1692.26
6s	1110.77	1063.23	1070.57	3653.21

(c)

site load	Load Balancer			
	Rand	ARA_OPT	ARA_PRED	*Qlen*
30%	487.83	471.05	473.04	606.62
50%	1109.33	1056.07	1059.00	1244.32
80%	4220.09	3964.39	3968.77	4138.34

Table 2: Sensitive analysis of system parameters (a) network size, (b) delay time, and (c) site load on ARA_PRED performance.

As the existence of delays in computing and communicating the site load information

is critical to the algorithm performance, we investigate the sensitivity of load balancers to information query delay D. In this set of experiments, we fix all the other parameters, e.g., $N = 16$ and site utilization is 50%, but increase D to 2s and 6s. The reason to set $D = 6$s is because the average duration of bursty periods is equal to 6s as well, which then provides an extreme case such that all jobs arriving during bursty periods are either sent to a single site or fully randomly sent to one of all sites in average. Table 2(b) shows the performance results. First, different delay times do not affect the performance of the "random" algorithm because the candidate site is always selected randomly no matter how long the delay is. However, for the "greedy" algorithm, the performance becomes worse as the delay time increases. This is because more jobs in bursty periods are then sent together to the same site due to the outdated load information and thus the load of that particular site significantly increases, causing serious load unbalancing and bad performance. For both of the ARA algorithms, we observe again the performance improvement compared to the other two conventional ones. Also, the delay time has less impact on the ARA performance. This is because after detecting the start of bursty periods, ARA quickly shifts to the "random" scheme.

In order to understand the performance benefit of the algorithm when the system reaches critical congestion, we turn to analyze the impacts of utilization levels on ARA performance. We here conduct experiments with three different site utilization levels: 30%, 50% and 80% by scaling the mean service times, while keeping the other parameters fixed as the experiments shown in Figure 8. The performance measures provided by four load balancing algorithms are illustrated in Table 2(c). We observe that both two ARA algorithms achieve better performance than the conventional ones (e.g., *Rand* and *Qlen*) across all three utilization levels.

In summary, the extensive experimentation produced in this section has validated that ARA using prediction information can effectively improve the system performance, compared to the conventional load balancers which ignore the effects of burstiness in arrivals. The sensitivity results on network size, delay time, and system load have further demonstrated that the gains of ARA are visible in a variety of different conditions.

3.3 Case Study: Amazon EC2

To further verify the effectiveness of our new load balancer, we implement and evaluate the ARA algorithms as well as the conventional ones (i.e., *Rand* and *Qlen*) in Amazon EC2, a real cloud platform that provides pools of computing resources to developers for flexibly configuring and scaling their compute capacity on demand. Figure 9 illustrates the basic framework of our implementation in Amazon EC2.

In particular, we replace the Elastic Load Balancing (ELB) in Amazon EC2 with our

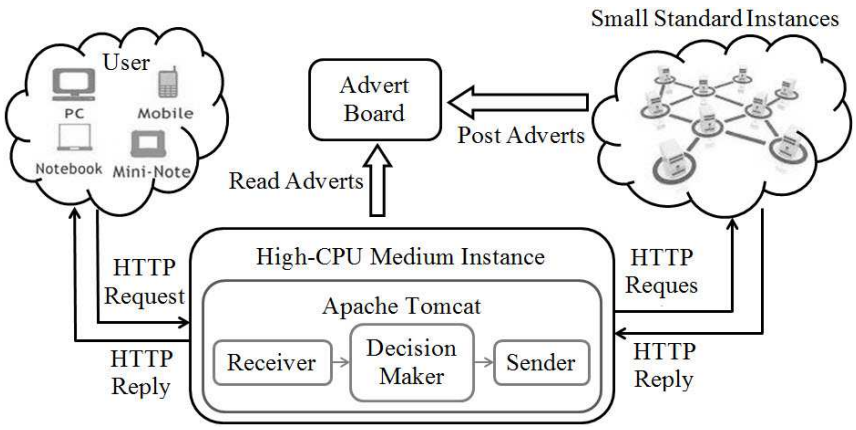

Figure 9: The overview framework of our implementation in Amazon EC2.

load balancing (LB) service and then direct all the incoming application requests to this new LB service for load dispatch across multiple Amazon EC2 instances. This new LB service is then run at a High-CPU Medium Instance which provides five EC2 compute units for compute-intensive applications. We also lease 8 Small Standard Instances as servers, each of which has one EC2 compute unit and 1.7GB memory by default. Such a configuration of instances aims to ensure that the system bottleneck is not our load balancer while the overall performance is dominated by the load balancing algorithms as well as the processing capability of each server instance.

We then conduct real experiments in Amazon EC2 by running microbenchmarks like the execution of `Fibonacci` numbers. As illustrated in Figure 9, multiple users can simultaneously send HTTP requests to our load balancer instance. Each HTTP request contains an URL, which includes a decision maker ID and the corresponding job size parameters. Once the load balancer receives an HTTP request, *Apache Tomcat*, an installed Java Servlet container, parses that request's header and then selects a server instance for serving that request according to the implemented load balancing algorithm. Here, on each of server instances, the `sar` command was run for measuring and reporting the CPU utilization every 1 second to load balancer via advert board. The chosen server instance then calculates a `Fibonacci` number and sends the result back to a client through the load-balancer instance.

In terms of evaluation, we measured end-to-end response times (i.e., the duration between request submission and reply receiving) for the QoS assessment and monitored utilization levels at each site (or application instance) for the load balance assessment. Figure 10 presents the performance of the online ARA in our Amazon EC2 model, where burstiness was injected into the arrivals of HTTP requests. The results under both *Qlen* and *Rand* are also plotted in the figure. We observe that consistently to our simulations,

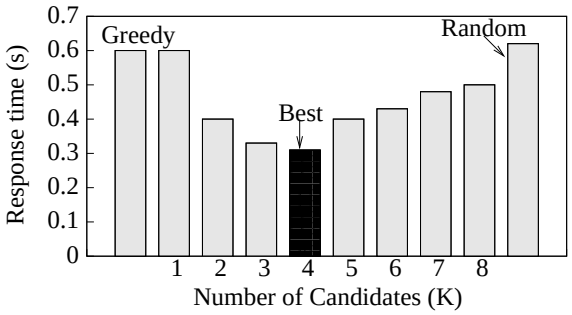

Figure 10: The average end-to-end response times under *Rand*, *Qlen* and our online ARA where the small threshold th_s is kept as 1 while the large threshold th_l is changed from 1 to 8.

none of the conventional load balancers (e.g., *Qlen* and *Rand*) is able to obtain good performance under bursty workload, while our online ARA algorithm achieves significant performance improvements by dynamically shifting between "greedy" and "random" according to the workload changes. The best performance under ARA is obtained when K is equal to 4 such that the relative improvements are 48% over *Qlen* and 50% over *Rand*, respectively. We also observe that the measured utilization levels at all 8 server instances are quiet close to each other, i.e., about 41% in average, which indicates a good load balancing across multiple Amazon EC2 instances.

3.4 Summary

In this work, we have described our new adaptive load balancing algorithms for clouds under bursty workloads. Our new static ARA algorithm tunes the load balancer by adjusting the trade-off between randomness and greediness in the selection of sites. While this approach gives very good performance, tuning the algorithm can be difficult. We therefore proposed our new online ARA algorithm that predicts the beginning and the end of workload bursts and automatically adjusts the load balancer to compensate. We show that the online algorithm gives good results under a variety of system settings. This approach is more robust than the static algorithm, and does not require the algorithm parameters to be carefully tuned. We conclude that an adaptive, burstiness-aware load balancing algorithm can significantly improve the performance of cloud computing systems.

4 Flash Resource Management in Storage Systems

Storage is another critical resource in contemporary computer systems. Especially, data intensive applications demands on high I/O throughput, small response time and large capacity. Today, the volume of data in the world has been tremendously increased. Large-scaled and diverse data sets are raising new big challenges of storage, process, and query. Particularly, real-time data analysis becomes more and more frequently. With low latency and low power consumption, NAND-based flash memory is being widely deployed as the cache in the storage systems to improve the I/O performance and reduce the power consumption. Flash is either combined with magnetic mediums as a tiered storage or is directly used as a second-level cache for read cache and write buffer. Such two architectures introduce more layers in both the hardware implementation and software design and management. By this way, it indeed increases the complexity and challenges in the resource management of Flash-based storage systems. Therefore, the traditional designs of I/O path and the conventional caching management methods are no longer fit for the new storage architectures.

The contributions of our works include:

1. Investigation of the architecture of tiered storage systems;
2. Exploration of efficient methods to support multiple SLAs for applications with dynamic workloads;
3. Presentation of a new approach for automated data movement in multi-tiered storage systems and verification of the effectiveness and the robustness of the algorithm by trace-driven simulations;
4. Investigation of I/O access patterns in enterprise storage systems;
5. Presentation of a new Flash resource manager and evaluation of the performance improvement in Flash hit ratio and I/O cost.

4.1 Live Data Migration in Multi-tiered Storage Systems

Tiered storage architectures, which provide a way to combine SSDs with HDDs, therefore become attractive in enterprise data centers for achieving high performance and large capacity simultaneously. However, from service provider's perspective, how to efficiently manage all the data hosted in data center in order to provide high QoS is still a core and difficult problem. The modern enterprise data centers often provide the shared storage resources to a large variety of applications which might demand for different SLAs. Furthermore, any user query from a data-intensive application could easily trigger a scan

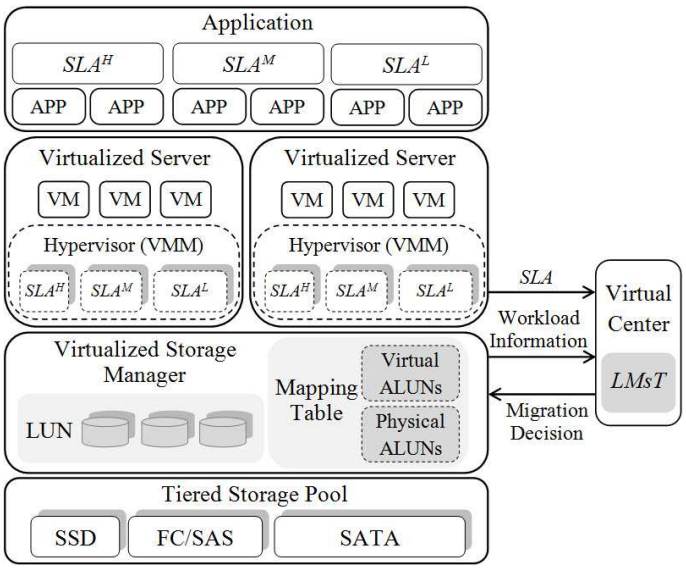

Figure 11: The structure of a multi-tiered storage system.

of a gigantic data set and inject a burst of disk I/Os to the back-end storage system, which will eventually cause disastrous performance degradation. Therefore, in the paper, we present a new approach for automated data movement in multi-tiered storage systems, which migrates the data on-the-fly across different tiers, aiming to support multiple SLAs for applications with dynamic workloads at the minimal cost. Detailed trace-driven simulations show that this new approach significantly improves the overall performance, providing higher QoS for applications and reducing the occurrence of SLA violations. Sensitivity analysis under different system environments further validates the effectiveness and robustness of the approach.

4.1.1 System Architecture

In a traditional data center, the storage system straightforwardly provides resources to meet application demands. There is no coordinated orchestration in such an environment. However, with the introduction of storage tiering and virtualization technologies, the layers of abstraction between applications and back-end fundamental resources become obscured, which raises a big issue for allocating storage resources among the applications in order to meet various performance goals. To well understand the interaction of these individual layers, we here present an architectural overview of a multi-tiered storage system which is considered in this paper. As shown in Figure 11, the system consists of four main components: application, server, logical unit (LUN), and back-end storage pool.

The *application* component in the top layer is used to represent the applications who can access the shared storage resources in data centers. We classify these applications into

several categories according to their SLA requirements, e.g., SLA^H, SLA^M, and SLA^L. For example, the latency-sensitive applications should be assigned to SLA^H class and then be allocated to high-performance tiers in order to meet their strictly high SLAs. Furthermore, each application with its own I/O workload specifications is assigned to a virtual machine (VM) which provides a virtual disk to support the associated SLA requirement. The hypervisor, as a virtual machine monitor (VMM) in the *server* component, supports multiple VMs to access the shared back-end storage pool and allocates virtualized disk resources among VMs to achieve their different performance goals.

The *LUN* component abstracts the fundamental storage pool and supports the storage virtualization by building a mapping table to connect virtual disk resources with physical disk resources. Therefore, the LUN component hides the information of the underlying hardware devices to applications while enables multiple applications to share virtualized storage resources without noticing the accesses and the contentions from the others.

The *back-end storage pool* is modeled as a multi-tiered system, which consists of different disk devices such as SSD, FC/SAS, and SATA. For example, Figure 11 shows three tiers in the storage pool, each of which groups the same type of disk devices and is specified with different performance features, e.g., service rate, power consumption, and physical capacity.

Through storage virtualization in the LUN component, the storage pool can provide the fundamental disk resources as the module of allocation unit (ALUN) which is set to 1GB as the minimal capacity/migration unit for thin-provisioning in sub-LUN level. Via the mapping table, each virtual ALUN in the hypervisor is then dedicated to a physical ALUN in the storage pool. The virtual center (e.g., VMware vCenter [75]) is responsible to analyze the resource usage in virtualization layers and to deploy tools for resource management. The virtualized storage manager monitors workload changes of physical ALUNs in sub-LUN level and transfers such information to virtual center. Meanwhile, the hypervisors provide the corresponding SLA requirements to the virtual center. We remark that our new migration method can be implemented as a new module in the virtual center, which is able to use all these information to make the decisions for data migration, and then send the decisions back to the virtualized storage manager which will execute the corresponding migration procedure.

4.1.2 Migration Algorithm LMsT

In this section, we present our new data migration algorithm LMsT. Our objective is to improve the system performance in terms of I/O response time while the application SLAs are still satisfied after the migration processes. In the rest of this section, we first present a formulation for data migration and give an overview of our new algorithm. Then, we

show how LMsT addresses the formulated problem in detail.

4.1.2.1 Overview and Problem Formulation

We use *data temperature* as an indicator to classify data into two categories according to the access frequency: *hot data* has a frequent access pattern and *cold data* is occasionally queried. Also, we consider a multi-tier storage structure consisting of two tiers, high performance tier equipped with SSDs and low performance tier using FCs. Because of the high hardware cost, high performance tier has a much smaller capacity than low performance tier. We note that our solution can be easily extended for data categories with more temperature levels and for storage systems with more than two tiers.

In practice, high performance tier is often reserved for applications which have strictly high SLA requirements. However, from the perspective of improving the overall system performance, high performance tier is also expected to host hot data regardless of the data owner's SLA. To best coordinate between the SLA-based and the performance-based resource allocations, LMsT automatically reallocates the data across multiple tiers of drives based on data temperature and SLA requirements. In designing the new algorithm, we define the following goals that allow LMsT to efficiently utilize the high performance SSD-tier.

- **Goal 1:** Latency-sensitive applications with strict SLAs should always be served in SSD-tier while the applications with loose SLAs should be initially served in HDD-tier.
- **Goal 2:** Once the applications with loose SLAs suffer bursty workloads, the corresponding hot ALUNs should be migrated to SSD-tier in order to mitigate their burdens in HDD-tier and avoid SLA violations.
- **Goal 3:** Extra I/Os caused by the migration process should not violate SLAs of any applications at both the source and the destination devices.
- **Goal 4:** The newly migrated hot data in SSD-tier should not bring additional SLA violations to latency-sensitive applications with strict SLAs.

In particular, assume there are n ALUNs $\{A_1, A_2, \ldots, A_n\}$ across m disks $\{D_1, D_2, \ldots, D_m\}$. Let $x_{i,j} \in \{0, 1\}$ indicate the association between A_i and D_j, i.e., $x_{i,j} = 1$ if ALUN A_i is hosted on disk D_j. Apparently, we have $\forall i, \sum_j x_{i,j} = 1$. In our solution, an ALUN is the minimum storage unit to be migrated. LMsT monitors the workload and the performance for each ALUN and each disk in a predefined time window t_{win} (e.g., 20 minutes in our

experiments[3]), to assist our migration decision.

Let λ_{A_i} and λ_{D_j} represent the arrival rates (KB/ms) of ALUN A_i and disk D_j, respectively. Then, we have

$$\lambda_{D_j} = \sum_i x_{i,j} \cdot \lambda_{A_i}, \tag{3}$$

$$\lambda_{A_i} = m(\lambda_{A_i'}) + \alpha \cdot \Delta(\lambda_{A_i'}), \tag{4}$$

where $m(\lambda_{A_i'})$ and $\Delta(\lambda_{A_i'})$ represent the mean and the standard deviation of previous recorded arrival rates $\lambda_{A_i'}$, and α is a tuning parameter for conservation. We further classify I/Os into four categories, i.e., sequential read (SR), random read (RR), sequential write (SW), and random write (RW) and let $\mu_{D_j}^{SR}$, $\mu_{D_j}^{RR}$, $\mu_{D_j}^{SW}$, and $\mu_{D_j}^{RW}$ denote the corresponding average service rates for these patterns, respectively. Then, the overall average service rate for disk D_j can be estimated as,

$$\begin{aligned} \mu_{D_j} = {} & P_{SR} \cdot \mu_{D_j}^{SR} + P_{RR} \cdot \mu_{D_j}^{RR} + \\ & P_{SW} \cdot \mu_{D_j}^{SW} + P_{RW} \cdot \mu_{D_j}^{RW}, \end{aligned} \tag{5}$$

where P_{SR}, P_{RR}, P_{SW}, and P_{RW} represent the fraction of each category. We also let s_{D_j} denote the average I/O size (KB) for each disk D_j.

In addition, assume each disk D_j has a single I/O queue consisting of l consecutive "logical" buffers $\{Q_{j,1}, Q_{j,2}, \ldots, Q_{j,l}\}$ and each $Q_{j,k}$ serves I/Os with a different SLA requirement, SLA_k (ms), see Figure 12. Without loss of generality, we assume $\forall i < k,\ SLA_i < SLA_k$. Let $y_{i,k} \in \{0, 1\}$ indicate if A_i is associated with SLA_k. Thus, A_i belongs to the buffer $Q_{j,k}$ if $x_{i,j} \cdot y_{i,k} = 1$. Table 3 gives the notations that are used in this paper. The high level idea of our new migration algorithm is also shown in Figure 13.

4.1.2.2 Migration Candidate Selection and Validation

In this subsection, we present how to select candidate ALUNs for migration. Our scheme consists of two phases: In the first Selection phase, we choose a set of potential migration candidates based on the workloads of each ALUN and the performance of each disk. Each potential candidate is represented by a pair value (A_i, D_j) indicating a migration of ALUN A_i to D_j ($x_{i,j} = 0$). These migration candidates, if accomplished, will help either improve the system performance or release SSD resources. In the second phase of *validation*, we carefully examine each migration candidate, quantify the benefits, and estimate the risk of SLA violations. A subset of validated candidates will be selected for actual migration.

[3]We remark that the setting of t_{win} depends on how frequently the workload changes. If the workload changes fast, then a small t_{win} is preferred, vice versa.

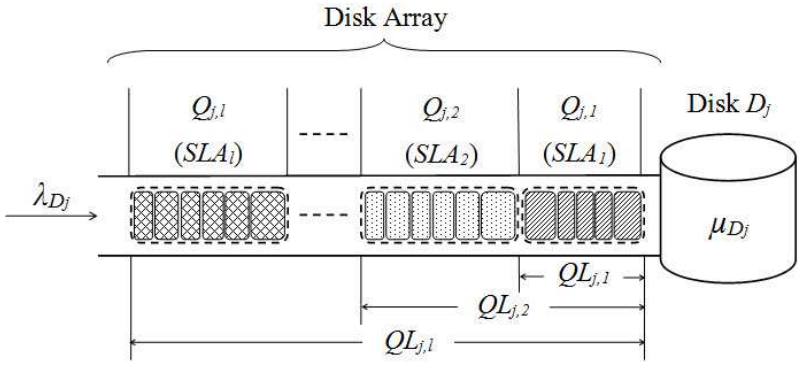

Figure 12: The profile of logical buffers and disk array.

ALGORITHM: The high level description of LMsT
for each time window t_{win}
a. Determine the migration candidates (i.e., ALUNs), see Sec. 4.1.2.2
I. *Selection Phase:* find the potential candidates (A_i, D_j) for forward and backward migration;
II. *Validation Phase:* select a subset of potential candidates using two sets of constraints, see Eq.s(8) and (16);
b. Determine the trigger time for a migration process, see Sec. 4.1.2.3
I. Estimate the migration duration t_{mgt} for each migration candidate using Eq.(17);
II. Schedule backward migrations;
III. Schedule forward migrations till the end of window or no more migration candidates;
end

Figure 13: The high level description of LMsT.

Table 3: Notations in this work.

$A_i,\ i \in [1,n]$	n ALUNs.
$D_j,\ j \in [1,m]$	m disks.
$x_{i,j} \in \{0,1\}$	indicator of association between A_i and D_j.
λ_{A_i} or λ_{D_j}	I/O arrival rates of A_i or D_j (KB/ms).
μ_{D_j}	average service rate of disk D_j (KB/ms).
s_{D_j}	average I/O size on disk D_j (KB).
SLA_k	the kth SLA requirement (ms).
$Q_{j,k}$	the kth logical buffer on disk D_j with SLA_k.
$y_{i,k} \in \{0,1\}$	indicator of association between A_i and SLA_k.
t_{win}	duration of a time window (ms).
t_{mgt}	time duration of the migration process (ms).

Selection Phase: There are two types of effective migrations that the system can benefit from. First, if an ALUN hosts hot data in low performance tier, it should be migrated to high performance tier for improving the performance. We call this migration as *forward migration*. Second, if the workload of an ALUN from loose-SLA application becomes cold in high performance tier, we may migrate that ALUN back to low performance tier in order to release the space in high performance tier. Such a migration is called *backward migration*. However, it might happen that all ALUNs currently in high performance tier are hosting hot data. Under this case, LMsT will cancel all forward migrations if there is no available space in high performance tier.

Two thresholds of I/O workloads τ_h and τ_l ($\tau_l < \tau_h$) are defined to eligible ALUNs for migration as follows. For an ALUN A_i, if its average workload $\lambda_{A_i} > \tau_h$, we consider the data hosted on A_i is hot. If A_i resides in low performance tier, it would be beneficial for the system to migrate it to high performance tier. Similarly, if an ALUN's workload is less than the lower threshold, i.e., $\lambda_{A_i} < \tau_l$, the data stored on A_i is regarded as cold. We may move this ALUN A_i to low performance tier, if A_i is allocated in high performance tier but belongs to an application with loose SLA. By this way, we find a set of ALUNs that are eligible for either forward or backward migrations.

Furthermore, the destination disk D_j for each eligible ALUN to migrate to is found such that D_j has the lowest load among those disks that can provide at least one available ALUN space. Finally, the selection phase yields a set of migration candidates (A_i, D_j) for the next validation phase.

Validation Phase: In validation phase, we quantify each migration candidate (A_i, D_j) through the following two conditions: (1.) SLAs have to be met; (2.) average I/O response time is expected to be decreased (for forward migration). A candidate is validated for migration only if both of these two conditions are satisfied. In the next, we quantify and analyze these performance metrics.

(1.) SLA Constraint: Recall that in our model, each disk array keeps multiple logical buffers and each buffer servers I/Os with a different SLA as shown in Figure 12. Upon the arrival of an I/O request, the I/O scheduler inserts it into a particular logical buffer which contains the requests having the same SLA requirement as the arriving one. While, within each buffer, all requests are scheduled based on First-In-First-Out (FIFO) discipline. Specifically, each buffer $Q_{j,k}$ can just hold a limited number of I/O requests in order to avoid introducing heavy loads to disk D_j and causing additional SLA violations.

Thus, for logical buffer $Q_{j,k}$, we define $ML_{j,k}$ as the maximal queue length in which the disk j can handle without causing any SLA violation,

$$ML_{j,k} = SLA_k \cdot \mu_{D_j}.$$

Additionally, we use $QL_{j,k}$ to denote the accumulated average queue length of logical buffers from $Q_{j,1}$ to $Q_{j,k}$. Let $\overline{\lambda}_{j,k}$ represent the overall arrival rates of the ALUNs whose SLAs are equal to or smaller than SLA_k in disk j,

$$\overline{\lambda}_{j,k} = \sum_{t=1}^{k} \sum_{i=1}^{n} x_{i,j} \cdot y_{i,t} \cdot \lambda_{A_i}.$$

Thus, using Little's Law, $QL_{j,k}$ can be expressed as

$$QL_{j,k} = f(\overline{\lambda}_{j,k}) = \frac{\overline{\lambda}_{j,k}}{\mu_{D_j} - \overline{\lambda}_{j,k}} \cdot s_{D_j}. \tag{6}$$

According to the definitions, $QL_{j,k} \leq ML_{j,k}$.

With the above analysis, we check the following two rules for each migration candidate (A_i, D_j),

$$\lambda_{D_j} + \lambda_{A_i} < \mu_{D_j}, \tag{7}$$

$$ML_{j,k} \geq QL'_{j,k} = f(\overline{\lambda}_{j,k} + \lambda_{A_i}), \text{ for } y_{i,k} = 1. \tag{8}$$

The first rule requires the total arrival rate on the destination disk D_j to be less than the processing rate μ_{D_j}. Similarly, in order to process migration, the arrival rate of the source disk to which A_i belongs should also be less than its processing rate. The second rule is for the particular logical buffer with the corresponding SLA that A_i belongs to. After migration, the new queue length of $Q_{j,k}$ should not exceed the maximal limit $ML_{j,k}$.

(2.) Response Time Constraint: Now, we turn to the performance constraint in terms of I/O response time for validating migration candidates. Basically, we estimate the I/O response time of both the source and the destination disks under the policies with and without migration and then evaluate the benefit (or the penalty) of each migration candidate.

For a migration candidate (A_i, D_j), assume A_i is currently hosted on disk D_k, i.e., $x_{i,k} = 1$. Let $\lambda'_{D_k}, \lambda'_{D_j}$ and λ'_{A_i} represent the workloads of D_k, D_j, and A_i in the next time window, respectively. Additionally, let t_{mgt} be the time duration to process a live migration ($t_{mgt} < t_{win}$) and $\Delta\lambda$ be the extra transfer rate for serving migration I/Os during the migration process. We will discuss how to derive t_{mgt} and $\Delta\lambda$ in Section 4.1.2.3. Assume if validated, the migration (A_i, D_j) will be launched at the current window. With this particular migration, for both the source disk D_k and the destination disk D_j, the workloads during t_{mgt} of the current window become $\lambda_{D_k} + \Delta\lambda$ and $\lambda_{D_j} + \Delta\lambda$, respectively. Additionally, in the next time window, their new workloads will be $\lambda'_{D_k} - \lambda'_{A_i}$ and $\lambda'_{D_j} + \lambda'_{A_i}$, respectively.

Based on the Little's Law, we can calculate the average response time RT_j of disk D_j as follows,

$$RT_j = g(j, \lambda_{D_j}) = \frac{s_{D_j}}{\mu_{D_j} - \lambda_{D_j}}. \tag{9}$$

With Eq.(9), we can evaluate the average I/O response time of both the source and the destination disks in three periods, i.e., before, during and after the migration process. Let $RT_{k/j}(A_i, D_j)$ and $RT'_{k/j}(A_i, D_j)$ be the average response times of the source disk D_k or the destination disk D_j under the policies with and without a particular migration (A_i, D_j), respectively, and $\overline{RT}_{k/j}(A_i, D_j)$ be the relative benefit (or penalty) in terms of response time. We then have the following equations:

$$\begin{aligned} RT_k(A_i, D_j) &= (g(k, \lambda_{D_k}) + g(k, \lambda_{D'_k})) \cdot t_{win}, & (10)\\ RT'_k(A_i, D_j) &= g(k, \lambda_{D_k}) \cdot (t_{win} - t_{mgt}) + \\ &\quad g(k, \lambda_{D_k} + \Delta\lambda) \cdot t_{mgt} + & (11)\\ &\quad g(k, \lambda'_{D_k} - \lambda'_{A_i}) \cdot t_{win}, \\ \overline{RT}_k(A_i, D_j) &= \frac{RT'_k(A_i, D_j) - RT_k(A_i, D_j)}{RT_k(A_i, D_j)}, & (12) \end{aligned}$$

$$\begin{aligned} RT_j(A_i, D_j) &= (g(j, \lambda_{D_j}) + g(j, \lambda_{D'_j})) \cdot t_{win}, & (13)\\ RT'_j(A_i, D_j) &= g(j, \lambda_{D_j}) \cdot (t_{win} - t_{mgt}) + \\ &\quad g(j, \lambda_{D_j} + \Delta\lambda) \cdot t_{mgt} + & (14)\\ &\quad g(j, \lambda'_{D_j} + \lambda'_{A_i}) \cdot t_{win}, \\ \overline{RT}_j(A_i, D_j) &= \frac{RT'_j(A_i, D_j) - RT_j(A_i, D_j)}{RT_j(A_i, D_j)}. & (15) \end{aligned}$$

The response time constraint is designed to compare the overall improvement in average response time to a threshold $e\%$. The migration candidate (A_i, D_j) is validated only if the following condition is satisfied.

$$\frac{\overline{RT}_k(A_i, D_j) + \overline{RT}_j(A_i, D_j)}{2} > e\% \tag{16}$$

In summary, we defined two sets of migration constraints, related to *SLA* and *performance* in our migration policy, LMsT, for evaluating each migration candidate. Once a candidate is validated, the corresponding forward or backward migration process can be actually performed by LMsT.

4.1.2.3 Migration Trigger Time

Given all the validated migration candidates, we now turn to schedule them for actual migration. To fulfill this schedule, the first key issue is to find out when to trigger the migration processes. So, in this section, we first introduce the estimation of migration duration for each candidate, and then present our migration trigger policy for both forward and backward migration candidates.

Estimation of Migration Duration: If a migration candidate (i.e., ALUN) meets the SLA constraint in Section 4.1.2.2, then one can expect that both the source and the destination disks might have extra capabilities to process migration I/Os which acquire additional transfer bandwidth. Since our migration policy can execute migration I/Os at any disk idle period, we assume that the disk utilization during the migration process is 100%, then the maximal arrival rate can be equal to the service rate.

Thus, the extra transfer rate $\Delta\lambda_{D_j}$ for serving migration I/Os can be obtained as the gap between the service rate μ_{D_j} and the actual arrival rate λ_{D_j} of disk D_j. Both the source disk D_k and the destination disk D_j have their own estimated transfer rates. We thus conservatively select the smaller one, i.e., $\min\{\Delta\lambda_{D_k}, \Delta\lambda_{D_j}\}$, as the mutual transfer rate for serving extra migration I/Os between the source and the destination disks. In addition, the total migration capacity might be larger than the capacity of a migration ALUN (i.e., 1GB) because of additional application write I/Os during the migration process. We here use a tuning parameter β to multiply the capacity of migration ALUN to assess the actual migration capacity. Therefore, the migration duration t_{mgt} for each candidate can be estimated by using the ratio of the estimated migration capacity to the extra transfer rate $\Delta\lambda$, see Eq.(17).

$$t_{mgt} = \frac{\beta \cdot \text{Capacity of ALUN}}{\mu_i - \lambda_i}. \tag{17}$$

Migration Trigger Policy: Once we obtain the migration duration for each candidate, the migration trigger policy needs to decide when to start the migration process. Since the forward and the backward migrations have different goals, they need to be triggered separately in each time window. For example, the candidates from backward migrations need to be triggered at the beginning of each time window in order to release the spaces in high performance tier. Once all backward migrations are done, the policy starts to schedule forward migrations. But, it is possible that the whole migration processes (i.e., backward plus forward migrations) exceeds a time window. As a result, some candidates cannot be successfully migrated within that particular time window. Under this case, the policy abandons all the migration candidates which have not been responded yet. For simplicity, our migration trigger policy assumes that each disk, either as the source or as the destination, only serves one migration candidate at any time.

Table 4: Device parameters of two tiers

Disk Type	Disk Number	Total Capacity	Service Rate
SSD	2	40GB	500MB/s
FC	5	100GB	160MB/s

4.1.3 Performance Evaluation of LMsT

In this section, we use representative case studies to evaluate the effectiveness of our new LMsT algorithm. A trace-driven simulation model to emulate a multi-tier storage system as shown in Figure 11. Without loss of generality, we assume that in our model the application components have two priority levels with different SLA requirements such that the SLAs of high and low priority applications are equal to SLA^H = 1ms and SLA^L = 20ms, respectively. We also assume two tiers of disk drives in the storage pool, i.e., SSD and FC. We remark that the number of disk drives in each tier is fixed in all the following experiments. The device parameters of these two tiers are shown in Table 4. Initially, the virtual ALUNs of applications with strict SLAs (i.e., SLA^H) are all mapped to SSDs. Whereas, FCs are initially assigned to low priority applications with loose SLAs.

In the following subsections, we first investigate the benefits of LMsT through a representative case study and later use synthetic traces to show the performance of LMsT under different system workloads. We also investigate the performance impact by tuning two constraints (see Section 4.1.2.2) and further validate the effectiveness of LMsT by comparing its performance with an existing migration policy.

4.1.3.1 Performance Improvement

To validate the performance improvement of LMsT, we consider a representative case, where the total active storage capacity is 70GB and 50% of each virtual ALUN's arrival flows are bursty across the overall simulation period. Here, active storage capacity refers to the amount of space that is used to store the data for application. That is, given the total 100GB storage capacity and the 70GB active storage capacity, we have the remaining 30GB capacity to be free. Recall that each applications requests 10 ALUNs, each of which has the capacity of 1GB. Therefore, we have totally 7 applications in this experiment, i.e., 2 applications with high priority and 5 applications with low priority. In particular, we generate a synthetic trace of I/O jobs for each virtual ALUN such that an I/O trace consists of 30 time slots, each of which lasts around 20 minutes. The specifications of an I/O job include I/O arrival time, I/O locality, and I/O size, in which I/O locality is uniformly distributed within an ALUN and I/O size is drawn from an exponential distribution. In addition, according to different arrival rates, a time slot can be further characterized as "idle" or "bursty", such that the inter-arrival times of I/O jobs that arrive during a bursty

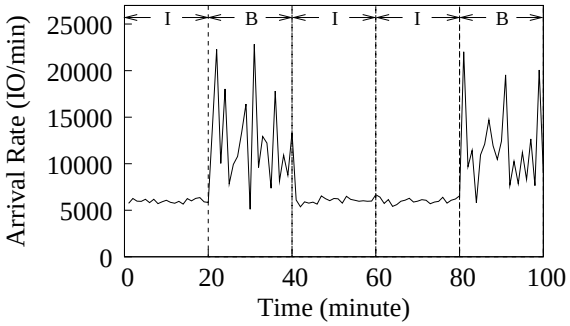

Figure 14: Example of arrival flows (i.e., number of I/Os per minute) to a virtual ALUN. In this window, we have 3 "idle" (I) and 2 "bursty" (B) time slots.

period are drawn from a 2-state Markov-Modulated Poisson Process (MMPP) while the I/O arrival process of an idle period is exponentially distributed. Figure 14 shows an example of the arrival flows to a virtual ALUN.

In all experiments, we set the mean arrival rates in "idle" time slots of high and low priority applications to be equal to 10 KB/ms and 5 KB/ms, respectively, while we double the mean arrival rates in "bursty" time slots for both two types of applications, i.e., 20 KB/ms for high priority ones and 10 KB/ms for low priority ones. The mean I/O size is 100KB and the FIFO policy is used to schedule I/Os from all the virtual ALUNs.

In addition, the performance metrics that are considered here include:

- M_Resp: the mean of I/O response times that are measured from the moment when an application submits an I/O request to the moment when that particular I/O request is completed.

- V_Ratio: the fraction of I/Os whose response times exceed the predefined SLAs, e.g., $SLA^H = 1$ms and $SLA^L = 20$ms, for those from high and low priority applications, respectively.

- V_Time: the mean violation times that are the difference between the actual I/O response times and the predefined SLAs.

Figure 15 shows the performance comparisons between NMsT and LMsT, where we use the NMsT policy as the base case to normalize our LMsT's performance. The NMsT policy is defined as no migration processes among multiple storage tiers in which the hot data is served in SSDs and the cold data is served in FCs. We observe that LMsT significantly improves the overall system performance. For example, I/Os experience faster response times under LMsT than under NMsT. This is because our new policy better utilizes SSDs by migrating all validated bursty traffic from FCs to SSDs. Consequently, LMsT enables fewer I/Os from low priority applications to be violated and meanwhile

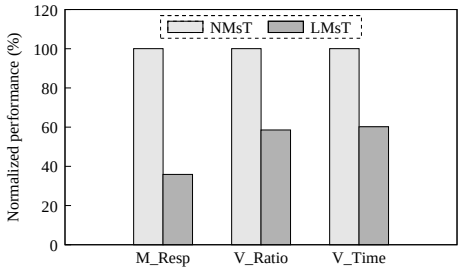

Figure 15: Comparing the system performance between NMsT and LMsT in the case of 70GB capacity and 50% burst.

reduces the violation times. More importantly, LMsT executes its migration processes using the priority-based queuing discipline at SSDs, i.e., migration I/Os always have the lowest priority and new application I/Os migrated from FCs still have lower priority than the original application I/Os with high SLAs. Therefore, the I/Os from high priority applications in SSDs are still guaranteed to meet the corresponding SLAs.

To further investigate the migration impacts, we illustrate each disk's utilization as well as each application's performance under both NMsT and LMsT in Table 5. Here, disk utilization is defined as the ratio of the disk busy time over the duration of the whole simulation. Compared to NMsT, the SSDs (i.e., D_1 and D_2) under LMsT are now better utilized with higher utilization while the I/O loads on FCs (i.e., $D_3, ..., D_7$) are significantly reduced, resulting in lower utilization, see Table 5.

Table 5 provides the performance data for each application by showing its average I/O response time (i.e., M_Resp) and the fraction of I/Os whose response times are beyond the predefined SLAs (i.e., V_Ratio). Given the results shown in the table, it is clear that all the low priority applications (i.e., $App3, ..., App7$) obtain tremendous performance improvement, experiencing lower response times and less violation ratios, and thereby receiving the high QoS. Moreover, the performance of high priority applications (i.e., $App1$ and $App2$) keeps almost the same despite a very slight degradation due to the extra migrated I/Os.

Table 5: Each application's performance under NMsT and LMsT in the case of 70GB data size and 50% burst.

Data Size 70GB 50% Burst		High Priority		Low Priority				
		App1	App2	App3	App4	App5	App6	App7
NMsT	M_Resp (ms)	1.30	1.27	382.29	366.60	368.52	350.12	380.55
	V_Ratio (%)	7.06	6.90	41.75	40.37	41.65	40.28	40.59
LMsT	M_Resp (ms)	1.34	1.32	131.01	137.75	139.53	125.49	124.53
	V_Ratio (%)	7.73	7.53	21.55	22.36	22.28	19.94	19.34

4.1.3.2 Sensitivity Analysis on System Workloads

Now, we turn to analyze the effectiveness and robustness of LMsT under various experimental conditions. We first focus on exploring the sensitivity of LMsT to different system workloads. Later, we investigate the sensitivity analysis of LMsT to our migration constraints.

In order to study the impacts of system loads and burstiness profiles on LMsT's performance, we vary the total active storage capacities (e.g., 40GB, 70GB and 100GB) and the percentage of arrival flows which are bursty, e.g., 30%, 50% and 70%. Recall that the active capacity refers to the amount of space in the storage pool that has been used to store the data for applications. Table 6 shows the configuration of applications and disks in the case of different active capacities. As we assume that each application requests ALUNs of 10GB, varying the overall active capacities change the number of applications as well, see the columns of Num_H and Num_L in Table 6. For example, in the case of 40GB, we have totally 4 applications such that 1 application has high priority and the other 3 ones have low priority. Since we keep the fixed number of disks (i.e., 2 SSDs and 5 FCs), the initial active capacity (see the columns of C_SSD and C_FC in Table 6) and the load of each disk is increased as well when the total active capacity increases. Finally, we remark that the combination of increasing active capacity and burst ratio further exacerbates the bursty load to the system. For example, when the active capacity increases to 100GB and the burst ratio is 70%, the total number of time slots (see N_TS in Table 6) at FCs reaches to 2100, i.e., 70 ALUNs at FCs times 30 slots/ALUN. Consequently, the total number of bursty slots at FCs becomes larger then 1400, see the row of $Burst\_TS$ in Table 8.

Table 6: Configuration of applications and disks under different storage active capacities, where Num_H (Num_L) is the number of high (low) priority applications and C_SSD (C_FC) gives the initial active capacity of SSD (FC), and N_TS is the total number of time slots at the FC-tiers.

Data Size	Num_H	Num_L	C_SSD ($\times 2$)	C_FC ($\times 5$)	N_TS
40GB	1	3	5GB	6GB	900
70GB	2	5	10GB	10GB	1500
100GB	3	7	15GB	14GB	2100

The experimental results of LMsT under 9 workload combinations are shown in Table 7. We also present the results of NMsT as well as the relative improvement with respect to NMsT in the table. First of all, we observe that under all the 9 workloads, LMsT achieves non-negligible performance improvement in terms of the mean I/O re-

sponse time (M_Resp), the fraction of I/Os that are SLA violated (V_Ratio), and the mean SLA violation times (V_Time). For example, under the case of 40GB and 30% burst, LMsT dramatically accelerates the average I/O response times by up to 83% relative improvement over NMsT and decreases the number of SLA-violated I/Os with the relative improvement over NMsT up to 78%. This indicates that LMsT provides the high QoS to low priority applications and meanwhile maintains the SLAs for high priority ones.

Also, we find that burstiness in arrival flows does deteriorate the overall system performance under both LMsT and NMsT policies. Such performance degradation becomes more significant when the arrivals become more bursty, i.e., the bursty ratio increases. Similarly, the increasing active storage capacity degrades the system performance as well because the overall disk loads are increased. This further results in strict migration constraints, allowing fewer bursty ALUNs to be migrated. In addition, the combination of large active storage capacity and high bursty ratio makes the relative improvement over NMsT less visible, e.g., in the case of 100GB and 70% burst, the relative improvements with respect to all the three performance metrics diminish.

Table 7: Sensitive analysis of system workloads with active storage capacity of (a) 40GB, (b) 70GB, and (c) 100GB. The burst ratio is set to 30%, 50% and 70%.

(a)

Data Size 40GB	NMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	29.45	66.93	127.43
V_Ratio (%)	7.16	13.72	21.80
V_Time (ms)	381.17	463.15	562.29
Data Size 40GB	LMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	5.04 (82.89%)	15.17 (77.33%)	35.71 (71.98%)
V_Ratio (%)	1.58 (77.94%)	4.67 (65.98%)	9.87 (54.70%)
V_Time (ms)	271.16 (28.86%)	295.45 (36.21%)	337.15 (40.04%)

(b)

Data Size 70GB	NMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	67.11	205.20	433.90
V_Ratio (%)	12.00	25.78	39.68
V_Time (ms)	533.41	775.47	1075.62
Data Size 70GB	LMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	24.65 (63.27%)	73.51 (64.18%)	140.44 (67.63%)
V_Ratio (%)	6.21 (48.23%)	15.09 (41.46%)	28.35 (37.96%)
V_Time (ms)	370.86 (30.47%)	466.64 (39.82%)	553.46 (48.55%)

(c)

Data Size 100GB	NMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	195.02	716.30	1675.73
V_Ratio (%)	30.00	47.18	61.74
V_Time (ms)	631.72	1502.26	2699.83
Data Size 100GB	LMsT		
	30% Burst	50% Burst	70% Burst
M_Resp (ms)	72.59 (62.78%)	290.50 (59.44%)	1324.17 (20.98%)
V_Ratio (%)	18.85 (37.15%)	36.49 (22.66%)	56.67 (8.21%)
V_Time (ms)	368.39 (41.68%)	781.28 (47.99%)	2322.53 (13.95%)

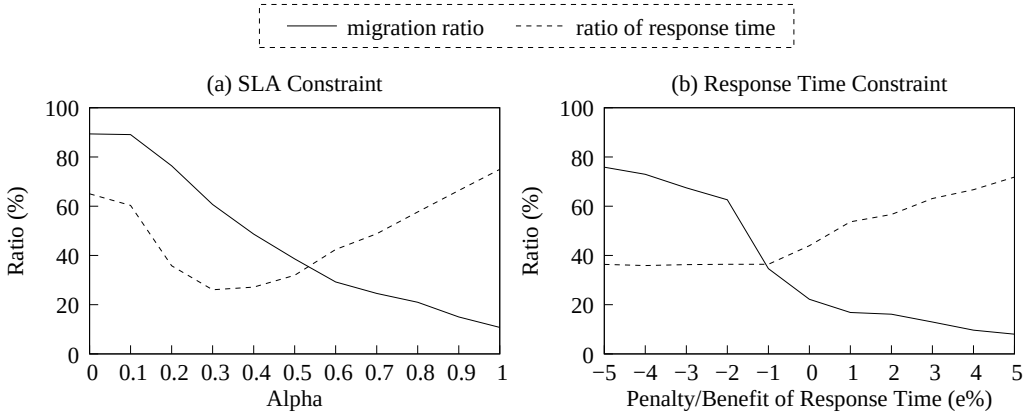

Figure 16: Ratios of system performance between LMsT and NMsT using different parameters in (a) SLA constraint and (b) response time constraint.

4.1.3.3 Sensitivity Analysis on Parameters in Migration Constraints

Recall that the key idea of LMsT is to improve the QoS for low priority applications via on-the-fly moving their hot data to SSDs, and meanwhile without causing additional SLA violations to high priority applications. Therefore, it is critical to set the right parameters for the migration constraints in order to achieve the best performance of LMsT. To address this issue, we here conduct a set of experiments to investigate the sensitivity of LMsT to the key parameters in two sets of migration constraints, i.e., SLA constraint shown in Eq.(8) and response time constraint shown in Eq.(16).

We show the performance results measured from the experiments, where we vary the parameter α of SLA constraint in Figure 16 (a) and the parameter $e\%$ of response time constraint in Figure 16 (b). All other parameters in these experiments are kept the same. Additionally, the total active storage capacity is set of 70GB and 50% of arrival flows to FCs are bursty. Then, we have 2 applications with high priority and 5 applications with low priority, see Table 6. In these experiments, we measure the migration ratio of bursty ALUNs in FC-tier and the response time ratio between LMsT and NMsT.

We first tune the parameter α in Eq.(4) to control the arrival rates to an ALUN and the associated disk. As the value of α increases, LMsT then conservatively migrates the ALUNs of low priority applications from FCs to SSDs due to the current heavy load (i.e., larger arrival rate) at SSDs. That is, the number of ALUNs which can be validated for migration by the SLA constraint in Eq.(8) becomes less, so that the migration ratio decreases, see Figure 16 (a). However, the trend of the response time is not straightforward. As we discussed, LMsT with a large α conservatively migrates ALUNs and thus obtains less improvement in response times. On the other hand, when α is set to a small value, a large number of ALUNs may be aggressively moved to SSDs, which thus dramatically in-

creases the load at SSDs and degrades the performance of the corresponding applications. We observe that the most benefits of LMsT are actually achieved when α is close to 0.3.

As shown in Eq.(16), the parameter $e\%$ is used as a threshold to determine the overall improvement in average response time, where the negative or positive value of $e\%$ (e.g., -5% or 5%) indicates the penalty or benefit in terms of response time. That is, when the relative benefit or penalty over NMsT is more or less than $e\%$, an ALUN will then be validated for migration. Consequently, a large value of $e\%$ indicates a conservative migration process, so that the migration ratio decreases and the ratio of response time between LMsT and NMsT increases, see Figure 16 (b).

4.1.3.4 Performance Comparisons with SMsT

In this section, we further validate the effectiveness of LMsT by comparing its performance with a migration algorithm, named SMsT, which is commonly used to suspend all the I/Os from an associated application when an ALUN is being in the migration process. That is, with a synchronization consideration, the migration I/Os are given the higher priority than the regular I/Os ones which request to access those to-be-migrated ALUNs. Table 8 shows the resultant comparison between LMsT and SMsT. We remark that under SMsT, all the other application I/Os which do not access to-be-migrated ALUNs still have higher priority than the migration I/Os during the migration process.

To identify the effectiveness of LMsT, we here use Mgr_Resp to represent the mean response time of the particular application I/Os which would access the to-be-migrated ALUNs during migration processes. In Table 8, the first important observation is that our new migration policy achieves much better performance (i.e., Mgr_Resp) under different workloads compared to SMsT. This is because by using the synchronization mechanism, LMsT can effectively eliminate the negative impacts of migration on regular application I/Os, avoiding unnecessary delays to their execution.

Besides the results of mean response times, we also show the total number of time slots ($Burst\_TS$) that have bursty arrivals in all the ALUNs of 5 FCs, as well as the fraction of bursty time slots (Mgr_Ratio) which are validated to be migrated, see Table 8. As the heavy bursty loads reduce the capability of both SSDs and FCs to migrate data, few migration candidates can be validated by our migration constraints. Therefore, we observe that as active storage capacity and bursty load increase, the total number of bursty time slots increases, whereas the migration ratio decreases. This further verifies the experimental results in Table 7. Both large $Burst\_TS$ and low Mgr_Ratio incur non-negligible performance degradation. For example, in the case of 100GB and 70% burst, LMsT achieves the smallest relative improvements over NMsT, compared to all the other cases.

Table 8: Sensitive analysis of migration policies under system workloads with active storage capacity of (a) 40GB, (b) 70GB, and (c) 100GB. The burst ratio is set to 30%, 50% and 70%. Here, Mgr_Resp is the mean response time of the application I/Os which access the to-be-migrated ALUNs and $Burst\_TS$ is the number of bursty time slots in all the ALUNs of 5 FCs, and Mgr_Ratio is the migration ratio over $Burst\_TS$.

(a)

Data Size 40GB	30% Burst		50% Burst		70% Burst	
	LMsT	SMsT	LMsT	SMsT	LMsT	SMsT
Mgr_Resp (ms)	13.31	140.22	66.89	165.81	194.79	296.38
$Burst\_TS$	267		428		591	
Mgr_Ratio (%)	87.64		89.72		89.85	

(b)

Data Size 70GB	30% Burst		50% Burst		70% Burst	
	LMsT	SMsT	LMsT	SMsT	LMsT	SMsT
Mgr_Resp (ms)	73.39	162.22	235.75	329.73	336.99	495.15
$Burst\_TS$	418		725		1038	
Mgr_Ratio (%)	89.71		76.41		49.62	

(c)

Data Size 100GB	30% Burst		50% Burst		70% Burst	
	LMsT	SMsT	LMsT	SMsT	LMsT	SMsT
Mgr_Resp (ms)	79.60	188.55	266.88	371.58	323.67	418.77
$Burst\_TS$	644		1073		1464	
Mgr_Ratio (%)	53.73		25.82		12.98	

As a final remark, it is interesting to observe that in the case of 40GB active capacity, the migration ratios are similar across different bursty loads (i.e., 30%, 50% and 70% burst). This is because that the overall system load is very low under this case, therefore, most of the bursty ALUNs can be migrated to SSDs no matter how heavy the bursty load is.

4.2 vFRM: Flash Resource Manager in VMware ESX Server

One popular approach of leveraging Flash technology in virtualization environments today is using Flash as a secondary-level host-side cache. Although this approach delivers I/O acceleration for a single VM workload, it might not be able to fully exploit the outstanding performance of Flash and justify the high cost-per-GB of Flash resources. In this section, we propose a new VMware Flash Resource Manager, named vFRM, which aims to maximize the utilization of Flash resources with minimal CPU, memory and I/O cost for managing and operating Flash. vFRM adopts the ideas of heating and cooling to identify data blocks that can benefit the most from being put in Flash, and lazily and asynchronously migrates data blocks between Flash and spinning disks. Experimental evaluation of the prototype shows that vFRM achieves better cost-effectiveness than traditional caching solutions, and costs orders of magnitude less I/O cost.

4.2.1 Motivation

Flash resources are usually deployed as host-side cache for the data center. The most significant benefit by deploying Flash in a system is mainly in the consideration of performance improvement, i.e., increasing I/O throughput and reducing I/O latency. However, this deployment inevitably introduces extra operational cost to system.

4.2.1.1 Goals and Metrics

Instead of improving I/O performance of an individual VM, we aim to maximize the utilization of Flash resources and minimize the cost incurred in managing Flash resources.
Maximizing Flash Utilization: When people buy an SSD, they are actually paying for performance rather than storage space. Therefore, we consider Input/Output Operations Per Second (IOPS), a common performance measurement, as the metric of Flash utilization and redefine one of our primary goals as maximizing IOPS utilization. As IOPS capabilities of Flash devices vary across different models, we alternatively use *I/O hit ratio* as the metric of Flash utilization. I/O hit ratio is defined as the fraction of I/O requests that are served by Flash. The higher the I/O hit ratio, the better the utilization of Flash resources. In order to achieve high I/O hit ratio, the most frequently accessed data should be put on Flash media. As I/O hit ratio increases, the processing efforts required for

these I/O requests are offloaded from the back-end storage array to the Flash tier and the storage array can thus allocate more processing power to serve other I/O requests, which actually improves the I/Os that are not served from Flash. This further improves the total cost of ownership (TCO) in terms of financial (IOPS/$) and power (IOPS/KWH) efficiency of storage systems.

Minimizing CPU, Memory and I/O Cost in Managing Flash: CPU, Memory and I/O bandwidth are needed in Flash resource management. Today, a single Flash-based SSD can easily reach up to 1TB and Flash resources are usually managed at a fine granularity (e.g., $4KB$ or $8KB$). Hence, it is fairly likely to incur a high fraction of in-memory footprint for the Flash related metadata. For example, if the memory footprint equals to 1% of Flash space, then $10GB$ metadata is required for a SSD with $1TB$ size. Such a large memory footprint limits the scalability of deploying Flash resources with large capacity. Therefore, our second primary goal is to minimize the cost incurred in managing and operating Flash resources.

4.2.1.2 I/O Access Patterns

The benefits of fully utilizing Flash are mainly motivated by the observations of I/O access patterns from workload studies. The effective workload studies can imply the accurate modeling, simulation, development and implementation of storage systems. [76] introduced twelve sets of long-term storage traces from various Microsoft production servers and analyzed workload characterizations in terms of block-level statistics, multi-parameter distributions, file access frequencies, and other more complex analyses. [77] presented an energy proportional storage system by effectively characterizing the nature of I/O access on servers using dynamic I/O workloads by consolidating the cumulative workload on a subset of physical volumes proportional to the I/O workload intensity. [78] developed a mechanism for accelerating cache warm-up based on detailed analysis of block-level data-center traces. They examined traces to understand the behavior of I/O reaccesses in two dimensions, e.g., temporal and spatial behaviors. [79] is another good example of technique design motivated by workload analysis in which they proposed a write offloading design to save energy in enterprise storage by a better understand of I/O patterns.

We analyze disk I/O traces of various workloads to understand I/O access patterns. First, we use a set of block I/O traces collected by MSR Cambridge in 2007 to understand volume access patterns in production systems [79]. Each trace contains block I/Os within one week and each data entry in the trace describes an I/O request, including time-stamp, disk number, logical block number (LBN), number of blocks and the type of I/O (i.e., read or write). Totally, there are 36 MSR-Cambridge traces representing a variety of workloads. In this paper, we select six of them as representative and summarize these

traces in Table 9.

Table 9: Selected MSR-Cambridge Traces. VS denotes the volume size and WSS denotes the working set size.

Name	Server	VS(GB)	WSS(GB)
mds0	Media Serv.	33.9	3.23
src12	Source Control Serv.	8.0	2.80
stg0	Web Staging Serv.	10.8	6.63
usr0	User Home Dir.	15.9	4.28
web0	Web/SQL Serv.	33.9	7.48
prn0	Printer Serv.	66.3	16.90

For each trace, we partition the entire LBNs address space into bins (with an equal width of $1MB$) and count the number of I/O accesses for each bin in every hour over a period of seven days. Figure 17 plots the distributions of I/O popularity across different bins and its variation over time, where the x-axis represents the LBN range, the y-axis represents the time and the z-axis represents the I/O popularity of bins. In addition, the grey-scale is used to represent I/O popularity. A darker scale represents a greater popularity.

To further validate our observations in I/O access patterns, we select the other three real workload variants. The first trace is collected from Microsoft Exchange Server 2007 SP1 using the event tracing for a duration of 24 hours. The second one is measured from Microsoft RADIUS Back-end SQL Server for a duration about 17 hours [76]. The third trace is collected by Florida International University (FIU) from the first of four different end-user and developer home directories during 24 hours [77]. All these traces are block level disk I/Os with the same I/O properties as MSR traces. Figure 18 shows the distributions of I/O popularity across different LBN bins and different time periods. Among a number of interesting findings, we summarize three key observations that inspire the design of vFRM:

[Obv. 1] The block access frequency exhibits a bimodal distribution. Most of the bins are accessed rarely (i.e., less than 10 times a day), while a small fraction of the bins are accessed extremely frequently (i.e., more than thousands of times a day). This implies that only a small number of bins are popular enough to be placed on Flash tier, while most of the remaining bins are not deserved for the high performance yet expensive Flash resource. This observation also motivates that vFRM is suitable to be managed in a coarse granularity (i.e., $1MB$ bin).

[Obv. 2] The distribution of I/O popularity does not vary significantly over time. This implies that vFRM does not need to actively and frequently update contents of Flash. A lazy and asynchronous approach should be sufficient for minimizing operational cost.

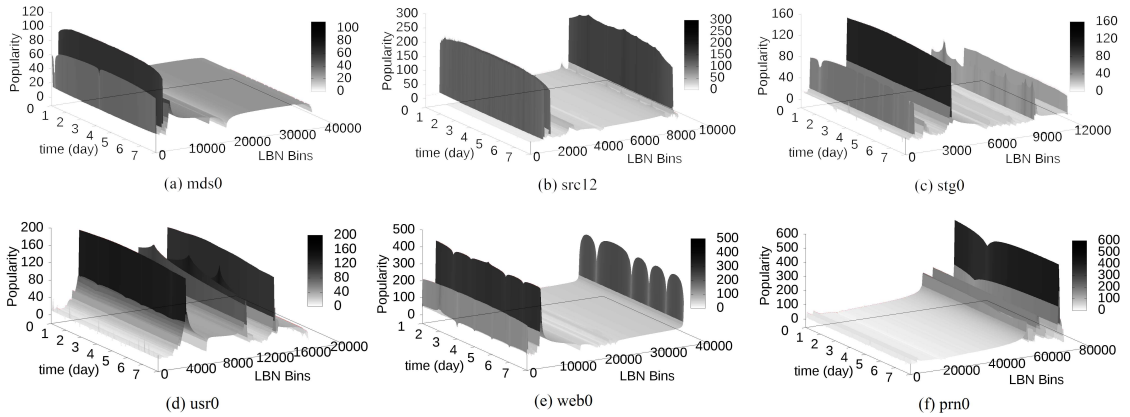

Figure 17: I/O popularity analysis of selected Cambridge traces

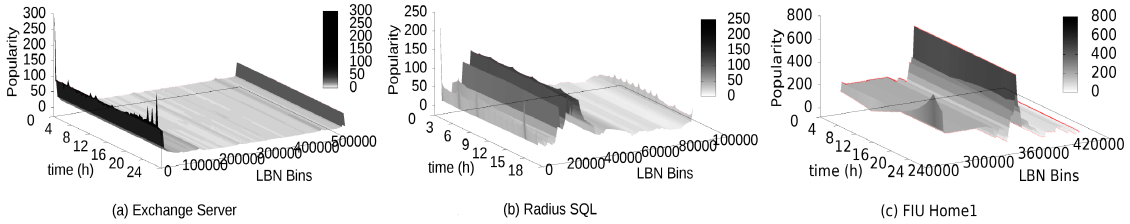

Figure 18: I/O popularity analysis of three traces

[Obv. 3] The distribution of I/O popularity varies across workloads and volumes. This implies that different applications lead to diverse distributions of popular bins and thus need different amount of Flash resources.

4.2.2 vFRM Design and Algorithms

Inspired by the above observations, we design vFRM, a Flash resource manager to manage data blocks at the granularity of hypervisor file system block. vFRM dynamically relocates the data blocks between the Flash tier and the spinning disk tier to gain the most performance benefits from Flash. Additionally, it does the data block relocation lazily and asynchronously, which significantly reduces the cost for CPU, memory and I/O incurred in managing Flash resources. By having the Flash tier absorbing more I/O requests from VMs, vFRM lessens the contention for the I/O bandwidth of the underlying storage, which in turn accelerates the I/O access for data on the spinning disk tier. Note that we intentionally skip the availability problem of locally attached Flash device, which is beyond of the goals of this paper. In this paper, we assume that the Flash device already has a high availability.

4.2.2.1 Main Architecture

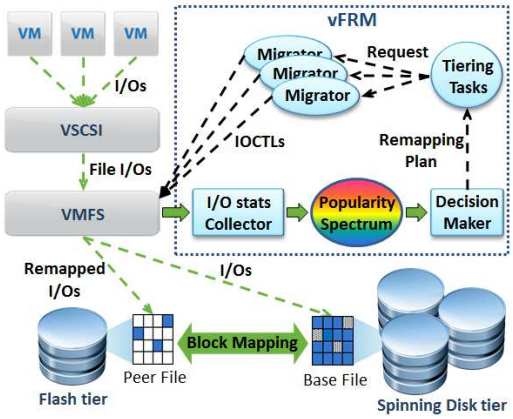

Figure 19: vFRM 's architecture overview.

Figure 19 shows the architecture overview of vFRM, which consists of three major components: (1) a modified VMware Virtual Machine File System (VMFS) that allows composing a hybrid file with mixed blocks from both the spinning disk tier and the Flash tier via block mapping; (2) a tiering manager that monitors I/O activities, makes migration decisions, and then generates tiering tasks for migrating hot blocks into the Flash tier and cold blocks out to the spinning disk tier; and 3) a pool of migrator threads that execute the migration tasks.

4.2.2.2 Hybrid File

A Virtual Machine Disk (VMDK) is essentially a file on a VMFS volume with all of its blocks allocated from the same VMFS volume [80]. In this work, we propose a new type of file, called hybrid file, to extend the VMDK from spinning media to Flash media. A hybrid file comprises two files: a base file and a peer file. As such, the hybrid file can span across both tiers with the hot blocks in its peer file on the Flash tier and the cold ones in its base file on the spinning disk tier.

The peer file is a sparse file and its internal blocks keep the same logical offset in VMDK as their corresponding blocks in the base file. When overlapping these two files, we get a hybrid file with the mixed blocks from both the spinning media and the Flash media. The VMFS file block address resolution mechanism is designed to identify the location of a requested block (i.e., in the peer file or in the base file) and to seamlessly re-direct the I/O to the right tier. Although the peer file has the same size of address space as its base file, it does not necessarily occupy the same size of Flash resources. In fact, it is mostly sparse, because only a small portion of the blocks are allocated as hot

blocks on the Flash tier. As each hot block keeps the same logical offset in both files, there is no need to add an extra mapping table to store the location mapping information of hot blocks between the Flash tier and the spinning disk tier. Moreover, we can use the inode pointer cache of the peer file as the block look up table, which further eliminates the need for an extra lookup table. If a block has been migrated to the Flash tier, the corresponding block will have been allocated and the inode pointer cache of the peer file can indicate the existence of this block. As a result, we have another saving of the memory space for the lookup table.

During the migration of Flash resources, the dirty blocks on the Flash tier of the source host need to be migrated to the Flash tier of the destination host if the Flash tier cannot be accessed by both source and destination hosts. If the Flash tier is not shared and there is not Flash on the destination host, vFRM will collapse this hybrid file via writing the dirty blocks back to the spinning disk tier. In the virtualized environment, a virtual disk is a file on VMFS, the design of hybrid file automatically enables hybrid storage for VMs.

4.2.2.3 Basic Data Structure

Heat map: Heat map is used to represent the I/O popularity statistics. Each $1MB$ block of the files on VMFS has on-Flash metadata associated with it as heat map. The per-block metadata contains 16 bytes to record the number of I/O accesses in which each 2 Bytes denotes the I/O access count that happened in one epoch (e.g., $5min$). In our implementation, we store the I/O statistics for the previous 8 epochs. The details of the usage of I/O statistics to predict the I/O popularity can be found in Section 4.2.2.4. In addition, we have 8 bytes of metadata to represent the logical address of the file descriptor and 4 bytes for the logical offset of the block. Thus, each $1MB$ block requires 28 bytes to hold the popularity statistics, which is only 0.0027% of the size of VMDK. More importantly, heat map does not necessarily to be pinned in memory. It only needs to be retrieved in memory for every 5 minutes when we want to use it to figure what blocks need to be migrated into Flash tier and what blocks need to be migrated out. This essentially can be translated to zero memory consumption. We will discuss more of the details in the following sections.

Tiering map: Tiering map is used to represent placement of the blocks between two tiers. A tiering map is specifically associated with a file and saved alongside the VMDK descriptor. It can be used to quickly warm-up the hot blocks after migration of Flash resources. In the tiering map, one bit represents in which tier a block is located. Therefore the metadata footprint overhead is only about 0.00001% of the size of VMDK. The same as the heat map, tiering map does not need to be pinned in memory permanently.

4.2.2.4 Temperature-based Tiering Manager

The main task of a tiering manager is to migrate data blocks between spinning disk tier and Flash tier to gain the most performance benefit from Flash. There are four steps to place a block on the right tier.

- The I/O stats collector collects the I/O activities at runtime and periodically flushes the I/O popularity statistics to disk.
- The tiering manager identifies the most popular blocks in the scope of all VMDK files based on a temperature-based model. We will discuss the temperature-based model in the following section.
- The tiering manager further generates a set of migrate-in (i.e., hot data into Flash) and migrate-out (i.e., cold data out of Flash) tasks.
- The migrators finally execute migration tasks. As a block migration involves modifying the file inode, all migration tasks are performed in the context of transactions to ensure the consistency of VMFS in case of host crash.

I/O Popularity Prediction Model: We now define a temperature-based model for predicting the I/O popularity of each block. In this model, we apply the concepts of heating and cooling to represent the variation of I/O popularity with time passing. When I/O requests flow to a block, that particular block gets heated. With time passing, the heated block cools down. In general, we consider m minutes (e.g., $m = 5$ in our experiments) as an epoch and let $T(i)$ denote the estimated (or predicted) temperature of a block during the i^{th} epoch. Assume that for each epoch, we always have N previous epochs available. We then use the following equation to calculate a block's temperature:

$$T_i = \sum_{j=1}^{N} H(M_{i-j}) \cdot C(j), \tag{18}$$

where M_{i-j} is the number of I/O requests to that block in the past $(i-j)^{th}$ epoch. $H(M_{i-j})$ and $C(j)$ denote the heating contribution and cooling factor respectively that are from the I/O requests in the past $(i-j)^{th}$ epoch.

Specifically, we define $H(M_{i-j})$ as a linear function, such that the heating temperature in the $(i-j)^{th}$ epoch is proportional to the number of I/O requests during that epoch.

$$H(M_{i-j}) = \lambda \cdot M_{i-j}. \tag{19}$$

Here, λ is a tunable constant that determines how important one workload is relative to other workloads. The greater the λ is, the faster the block gets warmed up with the

same number of I/O requests. We define the cooling factor $C(j)$ as a function of the time distance (i.e., j epochs) from the current epoch.

$$C_j = \begin{cases} \frac{N+1-j}{N}, & 1 \leq j < \frac{N}{2}+1 \\ \frac{1}{2^{j-3}}, & \frac{N}{2}+1 \leq j \leq N \end{cases} . \quad (20)$$

Such a cooling factor represents the declining heating effects with time passing. Currently we adopt a cooling scheme that linearly cools down in the first half of epochs and exponentially cools down in the second half of epochs. The heuristic behind this cooling scheme is that recent I/O activities have more influence than the ones in the past.

Using the above equations, we update instant and cumulative temperatures for each block every m minutes and then re-order all the blocks according to their cumulative temperatures. The hottest blocks should be placed in the Flash tier based on the available capacity of Flash resources while the remaining blocks will be kept on the spinning disk tier.

4.2.3 Evaluation

In this section, we present our experimental results to demonstrate the effectiveness of vFRM for a single enterprise workload with respect to our primary goals: maximizing Flash utilization and minimizing IO cost incurred in managing Flash. We first introduce the performance metrics and how they are measured to evaluate the effectiveness of our Flash managing algorithms. We then present the evaluation by implementing vFRM as a trace-replay simulation program. For comparison, we also treat Flash as a second-level cache and implement the LRU, ARC [14] and CAR [81] caching solutions in our simulation.

Table 10: The necessary SSD and MD operations for all caching conditions.

(a) Operations for IO Access Cost

	Read Hit	Read Miss	Write Hit	Write Miss
LRU/ARC/CAR ($4KB$)	SSD Read	MD Read + SSD Write	SSD Write	SSD Write
vFRM/g1-vFRM ($128KB$)	SSD Read	MD Read	SSD Write	MD Write

(b) Operations for Flash Update Cost

LRU/ARC/CAR ($4KB$)	Evict Dirty Page	
	SSD Read + MD Write	
vFRM/g1-vFRM ($128KB$)	Admin Hot Bin	Evict Cold & Dirty Bin
	MD Read + SSD Write	SSD Read + MD Write

Table 11: Measured average IO response times of various types of IO operations at Flash and spinning disk.

Latency	$T_{SsdRead}$ (μs)	T_{SsdWrt} (μs)	T_{MdRead} (μs)	T_{MdWrt} (μs)
$4K$ Sequential	53	59	63	92
$128K$ Sequential	558	1242	1070	1104
$4K$ Random	135	58	7671	3922
$128K$ Random	790	1241	8665	4942

4.2.3.1 Performance Metrics

We first introduce two performance metrics: IO hit ratio and IO cost. We consider a combination of these two metrics as a criterion to evaluate the effectiveness of our Flash managing algorithms. We also discuss the approaches which we used to calculate the overall IO cost under both the proposed and the conventional Flash managing algorithms.

IO Hit Ratio: IO hit ratio is defined as the fraction of IO requests that are served by Flash. An IO request might contain more than one page. We say an IO request to be Flash hit only when all of its associated pages are cached in Flash. Higher IO hit ratio indicates that more IOs can be accessed from Flash directly which accelerates the overall IO performance. Thus, one of our primary targets is to increase IO hit ratio for improving Flash utilization.

IO Cost: IO cost consists of two parts: IO access cost and Flash contents updating cost. Specifically, IO access cost can be represented as IO response time or IO throughput (e.g., IOPS). For example, in the case of read miss, LRU reads missed pages from MD and caches them in Flash. Thus, the corresponding IO access cost is the time spent during this procedure. Moreover, extra time is needed to flush (or evict) dirty pages when newly accessed pages are administrated but Flash is full. We here consider such data movements between Flash and MD as Flash contents updating cost and include this cost in the overall IO cost.

We use Eq.(21) to calculate the overall IO cost C_{IO}, where C_{IOResp} and $C_{FlashUpdate}$ represent the IO access cost and the Flash contents updating cost, respectively. All N terms indicate the access numbers of SSD Read ($N_{SsdRead}$), SSD Write (N_{SsdWrt}), MD Read (N_{MdRead}), and MD Write (N_{MdWrt}), while all T terms (e.g., $T_{SsdRead}$ and T_{MdRead}) show the corresponding average IO latency for each operation. Specially, the basic IO sizes for the conventional caching algorithms and vFRM/G1-vFRM are specified as $4KB$ and $128KB$, respectively. Since our Flash resource manager uses bins of large spacial granularity (i.e., $1MB$) as migration unit, large IOs (e.g., $128KB$) can be employed in operation to improve disk IO performance. Therefore, all T terms for the conventional caching algorithms and vFRM/G1-vFRM are the corresponding disk performance of

$4KB$ and $128KB$ IOs, respectively.

$$\begin{aligned} C_{IO} &= C_{IOAccess} + C_{FlashUpdate} \\ &= N_{SsdRead} \cdot T_{SsdRead} + N_{SsdWrt} \cdot T_{SsdWrt} \\ &\quad + N_{MdRead} \cdot T_{MdRead} + N_{MdWrt} \cdot T_{MdWrt}, \end{aligned} \tag{21}$$

Table 10 further presents the related IO operations for IO access (see (a) in the table) and Flash contents updating (see (b) in the table) under both the conventional caching algorithms and our Flash resource managers (vFRM and G1-vFRM) when we are in four different scenarios, i.e., read hit, read miss, write hit, and write miss. As shown in Table 10(a), when we have a read or write miss, our Flash managers always redirect IOs to the spinning disk without updating the contents in Flash, and thus only trigger the operation of MD read/write, which is different from the conventional caching algorithms. As shown in Table 10(b), the conventional caching algorithms need a SSD read and a MD write to evict a dirty page from Flash to spinning disks. While our Flash managers only trigger move-in (for hot bins) and move-out (for cold bins) operations every epoch (e.g., 5 minutes). Thus, we count the number of hot and cold bins and consider 8 IOs of MD Read and SSD Write (resp. SSD Read and MD Write) for administrating (resp. evicting) a hot (resp. cold) bin in Flash as each IO operation is $128KB$ and the bin size is $1MB$.

Table 11 illustrates the actual average IO response times (in microseconds) of various types of IO operations at both Flash and spinning disk devices. These results were measured from an `Intel DC S3500 Series` SSD with the capacity of $80GB$ and a `Western Digital WD20EURS-63S48Y0` hard drive with $2TB$ and 5400 RPM. As the conventional caching algorithms use $4KB$ as the cache line size while vFRM and G1-vFRM set the bin size of $1MB$ and update Flash contents using the IO size of $128KB$, we present in Table 11 the measured response times for two levels of granularity (i.e., $4KB$ and $128KB$) in both sequential and random modes. These results will be used to calculate the overall IO cost as shown in Eq.(21).

4.2.3.2 Performance Evaluation For A Single Enterprise VM

IO Hit Ratio: In this section, we conduct experiments to verify the effectiveness of vFRM for a single enterprise VM. We first evaluate the IO hit ratio (i.e., the fraction of IO requests that are served by Flash) under vFRM using the representative MSR-Cambridge traces introduced in Section 4.2.1.2. Each trace represents the workload from a dedicated VM in the virtualized storage systems. For simplicity, we treat every workload as equally important (i.e., setting λ equal to one). We will evaluate the impact of λ in

the clustering environment in our future work. The IO hit ratios with the conventional caching schemes (e.g., LRU, ARC and CAR) are also measured. We conduct experiments with various Flash sizes ranging from $100MB$ to $4GB$ and replay each trace separately. Figure 20 clearly shows that as the size of Flash increases, the IO hit ratio of vFRM catches up or even outperforms those of LRU, ARC and CAR for most of the workloads. As the capacities of Flash devices are usually large, vFRM is pratically better in improving Flash utilization (e.g., IOPS) than classical caching solutions.

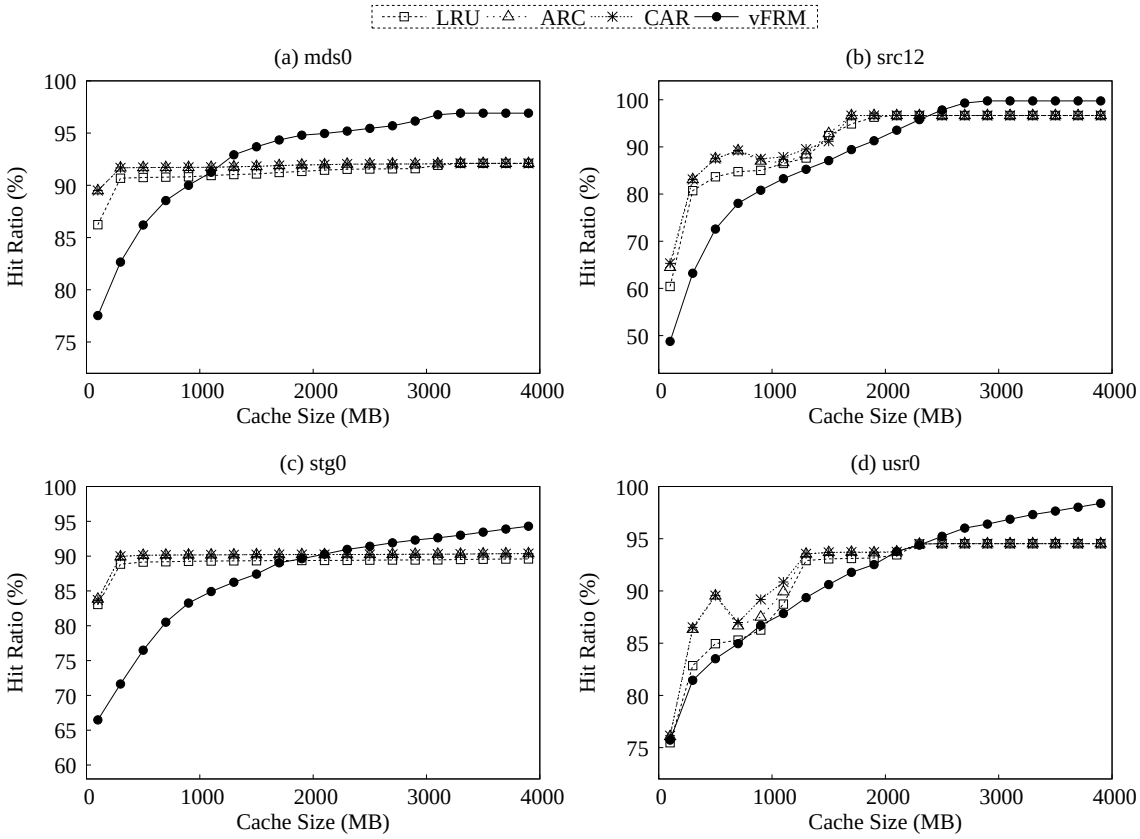

Figure 20: IO hit ratios of vFRM, LRU and ARC.

IO Cost: For both vFRM and existing caching solutions, internal IO costs are needed for both IO response and Flash contents updating, which is another type of performance criterion incurred in managing and operating Flash resources. vFRM only updates the contents every migration epoch (e.g., 5 minutes). In contrast, conventional caching updates the contents on every cache miss. Figure 21 shows the overall IO costs under both vFRM and LRU/ARC/CAR caching schemes. Here, Flash size is set to $4GB$. The numbers on top of each vFRM bar denote the relative improvement of the number of IOs in relative of LRU. Lower percentage implies more reduction. We observe that in all cases, the IO costs of vFRM is far less than those of the other three classic caching solutions. In

fact, most of them are order of magnitude better than the costs with LRU, ARC or CAR. For example, IO costs for *mds0* workload is only 31.87% of that of LRU solution. With such a great saving, vFRM can have more Flash IO bandwidth serving the IO requests, which further improves the VM's IO performance.

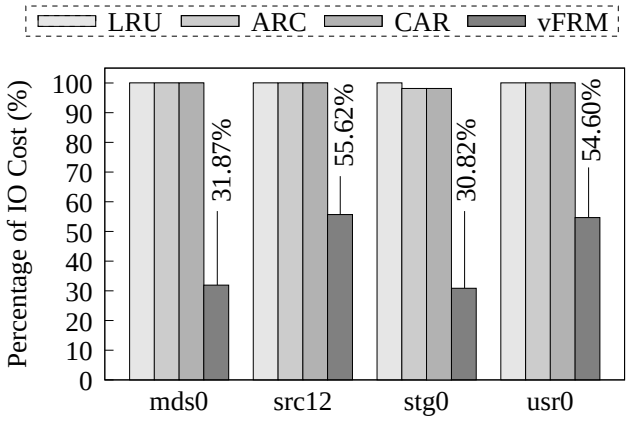

Figure 21: IO costs by using MSR-Cambridge traces. The relative IO costs with respect to LRU are also shown on the bars of vFRM.

4.3 G-vFRM: Improving Flash Resource Utilization Among Multiple Heterogeneous VMs

In a virtualization environment, multiple VMs often share storage services and each VM has their own workload pattern and caching requirement. In most of such shared virtualization platforms, Flash is statically pre-allocated to each virtual disk (VMDK) for simplicity and the caching algorithm decides the cache admission and eviction for each VM only based on IO requests to that particular VM regardless of IOs to the others. Therefore, it is difficult for the hypervisor to cost-effectively partition and allocate Flash resources among multiple heterogeneous VMs, particularly under diverse IO demands. In this section, we further investigate the benefits of g-vFRM for managing Flash resources among multiple heterogeneous VMs. Our goal is to fully leverage the outstanding performance of shared Flash resources under the global view of caching management.

4.3.1 Motivation

To understand various access patterns among multiple heterogeneous VMs, we extend our trace study by selecting 8 representative IO traces from MSR Cambridge trace set. For each workload, we calculate IO hit ratios using the LRU caching algorithm with fully associative cache, $4KB$ cache line and $1GB$ cache size. The results in Table 12 show that the conventional caching algorithms (e.g., LRU) cannot always perform well. For example,

Table 12: Statistics for Selected MSR-Cambridge Traces. Volume size denotes the maximum LBN accessed in disk volume. Working set size denotes the amount of data accessed. Re-accessed ratio denotes the percentage of IOs whose re-access time is within 5 minutes.

Name	Server	Volume Size (GB)	Working Set Size (GB)	Hit Ratio by LRU	Re-access Ratio
mds0	Media Serv.	33.9	3.23	90.84%	95.35%
src12	Source Control Serv.	8.0	2.80	85.64%	94.81%
stg0	Web Staging Serv.	10.8	6.63	89.28%	92.71%
usr0	User Home Dir.	15.9	4.28	88.25%	96.03%
stg1	Web Staging Serv.	101.7	81.5	34.60%	90.94%
usr2	User Home Dir.	530.4	382.7	19.49%	95.50%
web2	Web SQL Serv.	169.6	76.4	6.20%	95.45%
src21	Source Control Serv.	169.6	22.0	2.82%	96.04%

the IO hit ratio is less than 3% under the "src21" workload. We thus coarsely classify the workloads into two categories: "cache-friendly" workloads (e.g., mds0, src12, stg0 and usr0) and "cache-unfriendly" workloads (e.g., stg1, usr2, web2 and src21). As shown in Table 12, cache-friendly workloads always obtain high IO hit ratios under conventional caching algorithms, while cache-unfriendly workloads have relatively low hit ratios. We interpret these results by observing that cache-unfriendly workloads often have larger volume sizes and working set sizes (see the third and the forth columns in Table 12) than cache-friendly workloads, where volume size indicates the maximum LBN accessed in disk volume and working set size indicates the amount of data accessed. This means that the effectiveness of a cache is decided by its size to some extent. A small cache can only hold a small amount of data such that most of the cached data might be evicted or flushed out from the cache before it is reused if the actual working set size is large. Consequently, it is highly likely that the most recent or frequent data are not buffered in the cache which thus incurs low IO hit ratio.

To further investigate the differences between cache-friendly and cache-unfriendly workloads, we partition the entire LBNs address space of each workload into bins (with an equal width of $1MB$) and count the number of accessed bins per 5 minutes over a period of seven days. Figure 22 shows the results of two representative workloads from each category. We observe that the cache-unfriendly workloads, (see Figure 22 (c) and (d)), have more IO spikes than the cache-friendly workloads, (see Figure 22 (a) and (b)). We also observe that these spikes in cache-unfriendly workloads are much stronger and longer, which can dramatically degrade IO hit ratios due to the first-time cache miss and even worse pollute the critical data in Flash. This motivates us to design a new Flash resource manager which can perform well for both cache-friendly and cache-unfriendly workloads.

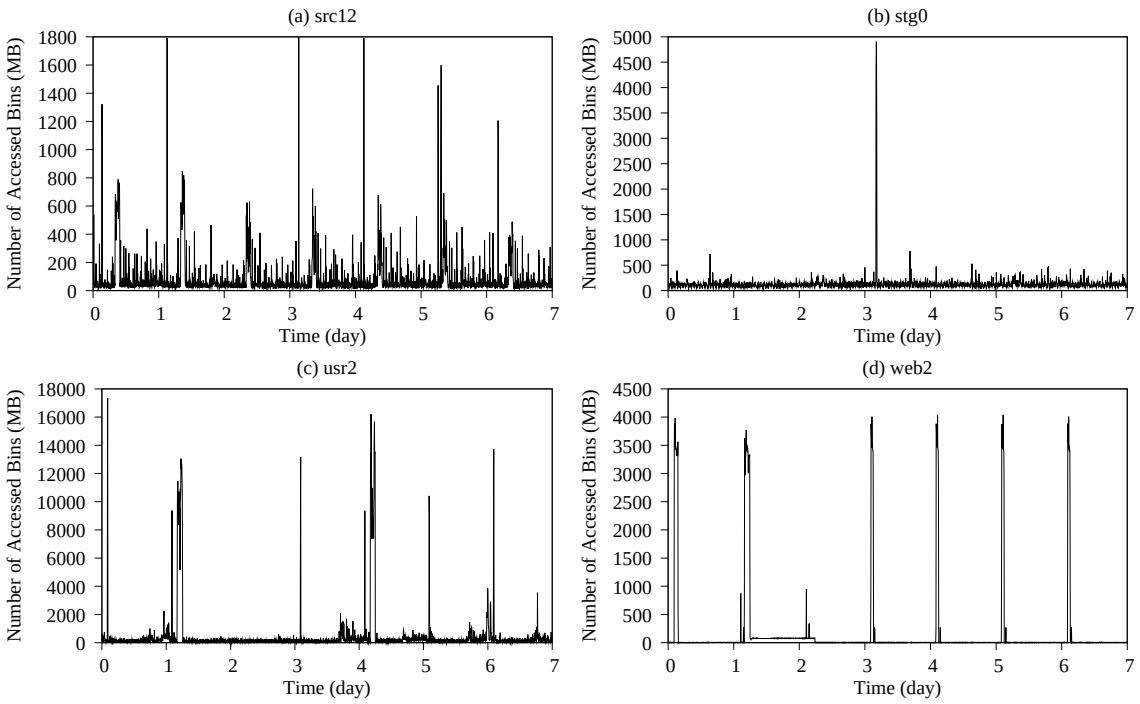

Figure 22: Number of accessed bins per 5 minutes of selected Cambridge traces.

4.3.2 New Global Version of vFRM Among Multiple Heterogeneous VMs

The basic idea of the global version of vFRM is to divide Flash resources among multiple VMs with the goals of fully utilizing Flash and minimizing the operational coat. Intuitively, there are two straightforward approaches which simply allocate Flash resources among VMs by either equally assigning Flash to each VM or managing Flash resources in a fair competition mode. In the former approach, all VMs are purely isolated in using their own Flash resource and the caching management is fully affected by their own workload changes. While, the second approach allows all VMs to freely use or share the entire Flash, such that the caching management is centrally interfered by the intensity of all workload changes.

Unfortunately, these two straightforward approaches cannot fully utilize the benefits of Flash, particularly when the workloads frequently change and bursts or spikes of IOs occur from time to time. If Flash is equally reserved and assigned to all VMs, then VMs with bursty IOs (or strict SLAs) cannot obtain more Flash resources. On the other hand, the second approach solves this issue by allowing all VMs to preempt or compete the Flash based on their present IO demands. Thus, VMs with higher IO demands can occupy more Flash resources by evicting less-accessed data from other VMs. However, under this approach, VMs with bursty IOs might occupy almost all the Flash resources and thus

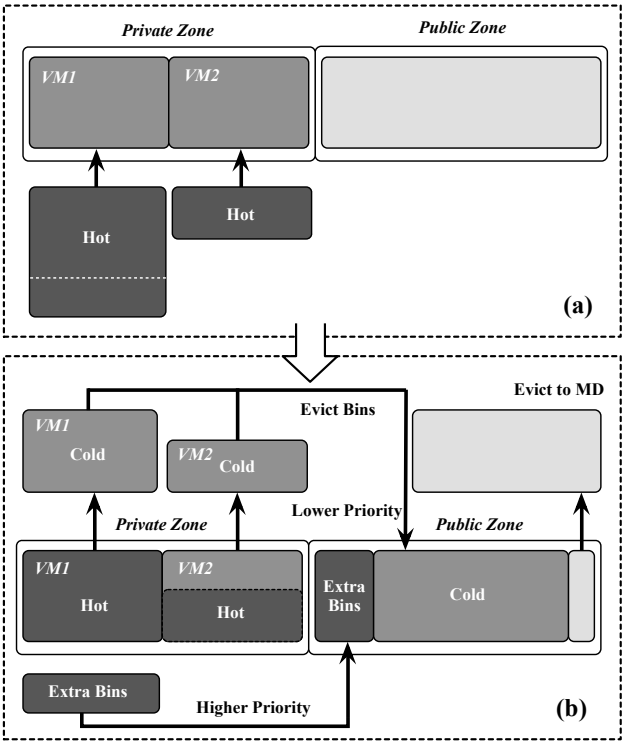

Figure 23: Flash contents updating procedure of G1-vFRM

pollute the critical caching of other VMs. It is even worse that bursty workloads usually have less re-accesses in the long term.

To wisely allocate Flash resources among all VMs, we develop the global versions of vFRM and use the term of G-vFRM to represent all the following global versions of algorithms. G-vFRM takes the dynamic IO demands of all VMs into consideration and divides Flash into a private zone and a public zone. Specially, the private zone is designed for reserving Flash for each VM in order to cache their recently accessed working sets, while the public zone is used to absorb and handle bursty IOs by being fairly competed among VMs according to their data popularities. We first implement a global vFRM algorithm, named "G1-vFRM", such that all VMs are assigned the equal portion of Flash that is pre-allocated in the private zone. Algorithm 1 shows the pseudo code of G1-vFRM. Figure 23 illustrates the Flash contents updating procedure. To manage each VM's private Flash, we sort its recently accessed bins (i.e., $1MB$) in the non-increasing order of their IO popularities. The top bins (i.e., with highest IO popularities) are then assigned to private Flash, see Figure 23(a). This procedure is denoted as *UpdatePrivateZone* in Algorithm 1. Meantime, both the residual of the recently accessed bins that cannot be cached in the private zone due to the limited space (i.e., *extraBin* in Algorithm 1) and the bins that are evicted from the private zone with less recency (i.e., *evictBin* in Algorithm 1) are

then flushed into the public zone, see Figure 23(b). The public zone collects these data sets from all VMs and stores the critical data as much as possible according to their IO popularities, see the procedure of *UpdatePublicZone* in Algorithm 1. By this design, if some VMs receive higher IO demands than others, they can then occupy more Flash resources in the public zone (e.g., the extra bins of $VM1$ in Figure 23(b)), especially to handle their bursty demands. More importantly, bursty VMs cannot arbitrarily pollute the critical data of other VMs because each VM now owns their isolated Flash in the private zone which cannot be preempted by other VMs and thus guarantees the performance to some extent.

Algorithm 1: Initial Task Assignment

Input: n: the number of VMs, $popBin[i]$: accessed bins of the i^{th} VM in last epoch (e.g., 5 min), $prvBin[i]$: cached bins of the i^{th} VM in private zone, $pubBin$: cached bins of all VMs in public zone

Output: $flashBin$: bins need to be cached in Flash

```
1  Procedure G1-vFRM()
2      UpdatePrivateZone();
3      UpdatePublicZone();
4      for i ← 1 to n do
5          flashBin += prvBin[i];
6      flashBin += pubBin;
7      return flashBin;
8  Procedure UpdatePrivateZone()
9      for i ← 1 to n do
10         popDiff = bins of popBin[i] which are not in prvBin[i];
11         prvDiff = bins of prvBin[i] which are not in popBin[i];
12         if len(popBin[i]) < len(prvBin[i]) then
13             j = len(popDiff);
14             itemL = number of j bins in prvBin[i] with lowest IO popularity;
15             evictBin += itemL;
16             prvBin[i] −= itmeL;
17             prvBin[i] += popDiff;
18         else
19             evictBin += prvDiff;
20             j = len(prvBin[i]);
21             prvBin[i] = number of j bins in popBins[i] with highest IO popularity;
22             extraBin += the remaining bins of popBins[i] which are not in prvBin[i];
23     return;
24 Procedure UpdatePublicZone()
25     if len(extraBin) ≥ len(pubBin) then
26         j = len(pubBin);
27         pubBin = number of j bins in extraBin with highest IO popularity;
28     else if len(extraBin) + len(evictBin) ≥ len(pubBin) then
29         j = len(pubBin) − len(extraBin);
30         itemH = number of j bins in evictBin with highest IO popularity;
31         pubBin = extraBin + itemH;
32     else
33         j = len(extraBin) + len(EvictBin);
34         itemL = number of j bins in pubBin with lowest IO popularity;
35         pubBin −= items;
36         pubBin += extraBin + evictBin;
37     return;
```

Nonetheless, evenly partitioning the private zone cannot achieve the best Flash utilization when heterogeneous VMs share Flash resources and their workloads are diverse and changing across time. There exist two primary drawbacks of G1-vFRM. First, not all of the VMs fully utilize their private Flash all the time, because a VM's working data set might be less than its private Flash partition. In such a case, the under-utilized private space should be used by other VMs more valuably. The second drawback lies in the fact that evenly partitioning the private zone does not consider the diversity in workload intensity (e.g., IOPS/GB) as a crucial criterion. For example, some VMs have bins with relatively high IOPS compared to the other VMs but cannot cache all these popular bins in the private zone, due to the limited and fixed space by the evenly partition. Therefore, from a global view, those VMs with more popular bins should reserve more Flash resources in the private zone in order to improve the overall utilization of Flash.

To overcome these two drawbacks, we propose an improved global vFRM algorithm, named "G2-vFRM", which dynamically divides the private zone for each VM based on their bins' frequency (i.e., the accumulated access number for each bin) and allows all VMs to fairly compete the public zone according to bins' rencencies (i.e., the most popular bins accessed in recent 5 minutes). Specially, we maintain a counter for each bin in the working sets of all VMs to record that bin's accumulated access number, which is used to represent the bin's frequency. Such a counter is increased by 1 when the associated bin is accessed. Thus, a larger counter indicates that a bin is accessed more frequently. As time elapses, a counter may overflow. Thus, we periodically aging all the counters by right shifting the values by one bit, so that we can still preserve the relative frequency presented by the values of counters. The private zone is then used to cache the most frequent bins with the highest counter values, see the procedure of *UpdatePrivateZone* in Algorithm 2. By this way, G2-vFRM selects the cached data in the private zone fully based on their global frequency, and thus reserves more private Flash for those VMs which have more popular bins. Moreover, when the distribution of bin popularities varies, G2-vFRM can dynamically adjust the reservation of private Flash for each VM to cache the most popular bins and thus fully utilizes the private zone all the time. Meantime, the public zone is responsible to cache recent working sets of all VMs as well as the data just evicted from the private zone, see the procedure of *UpdatePublicZone* in Algorithm 2.

Algorithm 2: Initial Task Assignment

Input: $binFreq$: a dictionary in which key is bin IDs of all VMs and value is the relative access count for each bin, $popBin$: accessed bins of all VMs in last epoch (e.g., 5 min), $prvBin$: cached bins of all VMs in private zone, $pubBin$: cached bins of all VMs in public zone

Output: $flashBin$: bins need to be cached in Flash

```
1  Procedure G2-vFRM()
2      UpdatePrivateZone();
3      UpdatePublicZone();
4      flashBin = prvBin + pubBin;
5      return flashBin;
6  Procedure UpdatePrivateZone()
7      if len(binFreq) ≤ len(prvBin) then
8          prvBin = binFreq.keys;
9      else
10         j = len(prvBin);
11         itemH = number of j bins in binFreq.keys with highest binFreq.values;
12         evictBin = bins of prvBin which are also in itemH;
13         prvBin = itemH;
14     return;
15 Procedure UpdatePublicZone()
16     if len(prvBin) < len(binFreq) ≤ len(flashBin) then
17         pubBin = the remaining bins of binFreq.keys which are not in prvBin;
18     else if len(binFreq) > len(flashBin) then
19         pubBin −= bins of pubBin which are also in prvBin;
20         popBin −= bins of popBin which are also in prvBin;
21         if len(popBin) ≥ len(pubBin) then
22             j = len(pubBin);
23             pubBin = number of j bins in popBin with highest IO popularity;
24         else
25             pubBin += evictBin;
26             pubBin −= bins of pubBin which are also in popBin;
27             j = len(pubBin) − len(popBin);
28             pubBin = number of j bins in pubBin with highest IO popularity;
29             pubBin += popBin;
30     return;
```

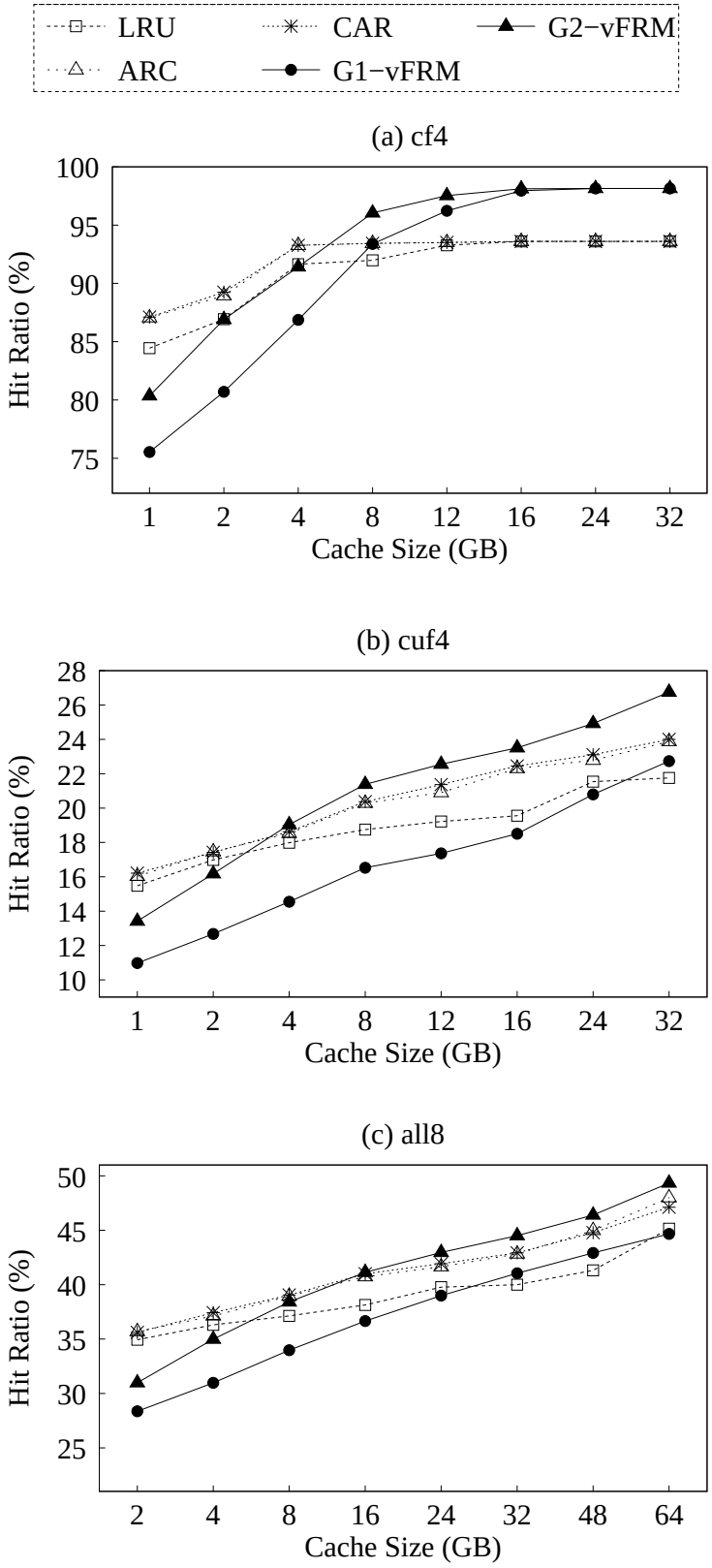

Figure 24: IO hit ratios under three workloads (a)"$cf4$", (b)"$cuf4$", and (c)"$all8$".

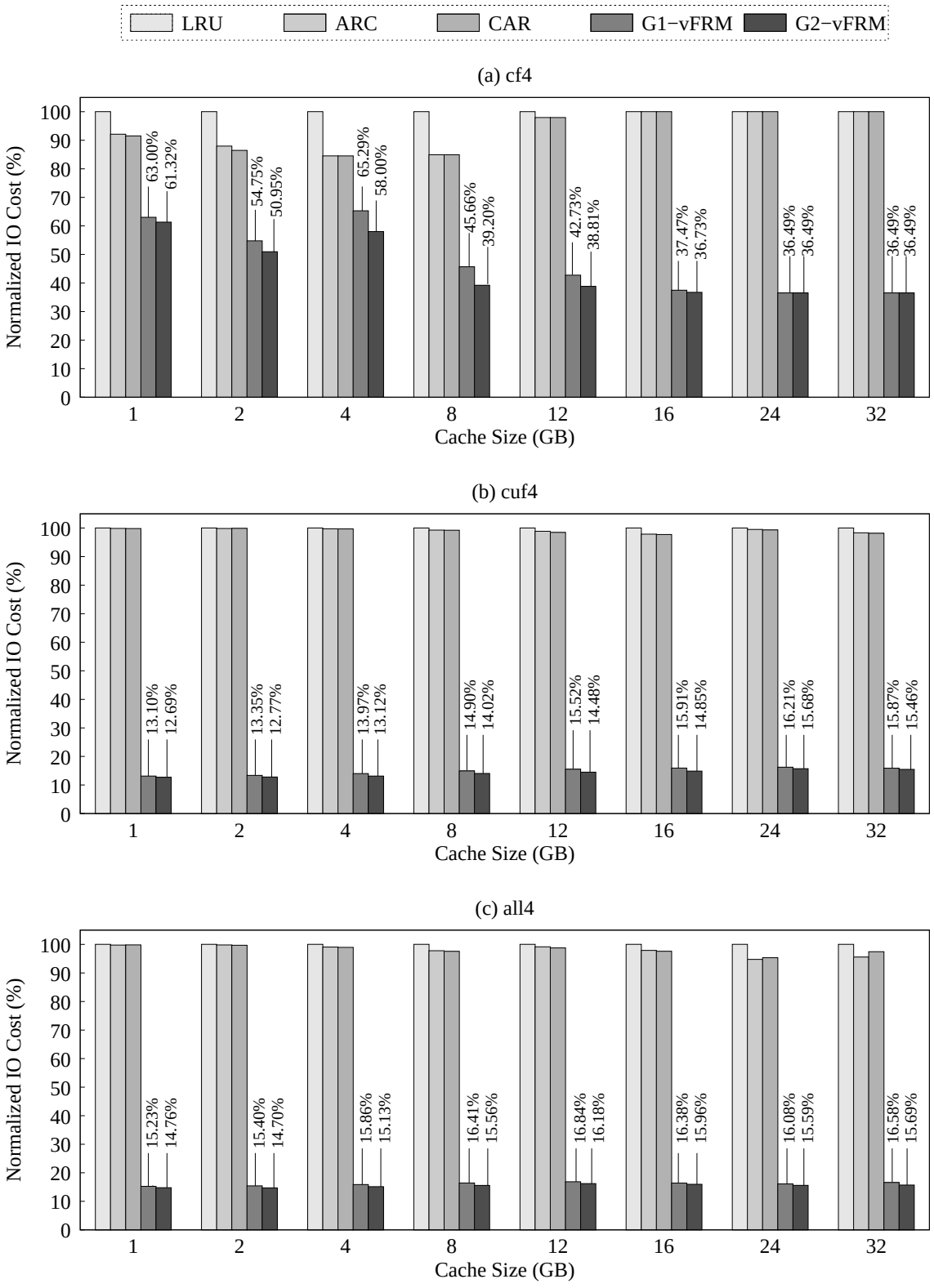

Figure 25: Normalized I/O costs (with respect to LRU) under three workloads (a)"$cf4$", (b)"$cuf4$", and (c)"$all8$'.'

4.3.3 Performance Evaluation of G-vFRM

In this section, we evaluate the effectiveness of G-vFRM algorithms on allocating Flash resources among multiple enterprise applications (or VMs). The evaluation is conducted by using trace-replay simulations with 8 selected MSR-Cambridge IO traces (see Table 12). As shown in Section 4.3.1, these MSR-Cambridge IO traces can be classified into two categories, cache-friendly and cache-unfriendly. Thus, we generate three workloads ("$cf4$", "$cuf4$", and "$all8$") by mixing 4 cache-friendly traces, 4 cache-unfriendly traces, and all 8 traces, respectively. The time-stamps of IO requests in each trace are normalized by a unified simulation start time and then used to determine the arrival times for each IO request in the workload. The metrics considered in our evaluation include Flash utilization (in terms of IO hit ratio) and Flash managing overhead (with respect with IO cost). For comparison, we also present the results under three conventional caching algorithms, e.g., LRU, ARC and CAR and conduct experiments with various Flash sizes ranging from $1G$ to $32G$.

4.3.4 Hit Ratio

Figure 24 illustrates IO hit ratios as a function of Flash size under three workloads (i.e., "$cf4$", "$cuf4$", and "$all8$"). We first observe that all these algorithms (including our G-vFRM) achieve high IO hit ratios when we have 4 cache-friendly traces (or VMs), see plot (a) in Figure 24. More importantly, under this cache-friendly workload, G-vFRM gains better Flash utilization than the conventional caching algorithms. For example, IO hit ratios under G1-vFRM keep rising to 99% as the capacity of Flash increases, while IO hit ratios under the conventional ones stop at around 93% when Flash size is larger than $4GB$. We also observe that under the cache-friendly ("$cuf4$") and mixed ("$all8$") workloads, the IO hit ratios of G1-vFRM catch up and even slightly overcome some of the conventional algorithms as Flash size increases. Furthermore, we observe that the hit ratios of G2-vFRM can overcome the conventional algorithms under all kinds of workloads as long as the Flash size is larger than a threshold. Such results verify that G2-vFRM can wisely allocate the Flash resources for VMs based on their dynamic IO characteristics in both frequency and recency.

We further look closely at IO accesses in these three workloads. As illustrated in Figure 22, IO spikes frequently appear in most traces such that a large number of bins are accessed during a short period which thus degrades IO hit ratios due to the first-time cache miss. Moreover, as the conventional caching algorithms cache data once there is a cache miss, it is highly likely that those IO spikes pollute the critical data of other applications (VMs) in Flash, especially bins in these spikes are rarely reaccessed in near future. Our G1-vFRM algorithm attempts to avoid such cache pollution by reserving private Flash

for each VM and further improve IO hit ratio by caching data blocks in both private and public zones based on their IO popularities. By this way, the spike IOs in a VM cannot pollute the critical data in other VMs. Consequently, as long as Flash has enough capacity to hold active working sets of all VMs, G1-vFRM is able to improve IO hit ratio (or Flash utilization) although G1-vFRM does not update Flash contents upon every IO miss as the conventional caching algorithms do. On the other hand, when Flash size is relatively small, especially for those cache-unfriendly traces which have relatively large working sets (see Table 9), the conventional caching algorithms obtain higher hit ratios than G1-vFRM by using small cache line size (e.g., $4KB$) and on-the-fly updating Flash contents for each cache miss. However, the cost of such caching algorithms is higher as well, which will be discussed in the following subsection. Although, G1-vFRM can overcome the conventional caching algorithms in some cases, but the results are not satisfied enough under the cache-unfriendly and the mixed workloads. This is because that G1-vFRM still cannot avoid the cache pollution of spikes for any specific VM in its own private zone. Thus, G2-vFRM is designed to wisely divide the private zone based on the bins' global frequency of all VMs. In this circumstance, the spikes which have barely re-access feature cannot be cached into the private zone. Thus, G2-vFRM obtains better IO hit ratios by further eliminating the pollution.

4.3.5 IO Cost

Figure 25 illustrates the normalized overall IO costs with respect to LRU under both G-vFRM and the conventional caching algorithms when we have 4 cache-friendly traces in "$cf4$", 4 cache-unfriendly traces in "$cuf4$", and 8 mixed traces in "$all8$". Consistently with the results for a single VM shown in Section 4.2.3.2, both G1-vFRM and G2-vFRM significantly reduce the overall IO costs for allocating Flash among multiple VMs compared to the conventional caching solutions. For example, under the "$cuf4$" workload (see Figure 25(a)), the overall IO costs under G1-vFRM is decreased up to 65.29% and the relative reduction is increasing as Flash size increases. Furthermore, G2-vFRM can beat G1-vFRM by saving more IO cost, since it can more accurately allocate resources resulting in better performance in both IO hit ratio and cost. There are two main reasons for both G1-vFRM and G2-vFRM to have such low IO costs. First, instead of updating Flash contents upon each cache miss, G-vFRM, like vFRM, generates move-in/move-out tasks for both private and public zones in every epoch (e.g., 5 minutes). Such a lazy and asynchronized way allows G-vFRM to reduce the number of extra IOs for Flash contents updating. Secondly, G-vFRM adopts $1MB$ as the minimal size of each bin and uses 8 IOs, each of which has the size of $128KB$, to move a bin into (or from) Flash, which reduces the number of IOs and shorten the latency for migrating a bin as well. More

importantly, G1-vFRM consumes much less IO cost for managing Flash resources when we have the "$cuf4$" and "$all8$" workloads (see Figure 25(b) and (c)) although the IO hit ratios of G1-vFRM are slightly lower. We thus conclude that under the consideration of both Flash utilization (i.e., IO hit ratio) and Flash managing overhead (i.e., IO cost), both G1-vFRM and G2-vFRM are more effective than the conventional caching algorithms.

4.4 Summary

In this section, we propose LMsT, a live data migration algorithm for efficiently utilizing the shared storage resources and meeting various application SLAs in a multi-tiered storage system. Using trace-driven simulations, we have shown that bursty workloads and traffic surges that are often found in storage systems, dramatically deteriorate the system performance, causing high I/O latency and large numbers of SLA violations in low performance tiers. Therefore, we designed LMsT to counteract the impacts of burstiness and minimize the potential delays to latency-sensitive applications. Furthermore, we showed that LMsT can automatically detect all the possible migration candidates and verify the feasible ones by estimating the risk of SLA violations and quantifying the performance benefits via both the SLA and the performance constraints. We conducted trace-driven simulations to evaluate the performance of LMsT policy. A series of sensitivity analysis with respect to storage capacities, burstiness profiles, and parameter settings in the SLA and the performance constrains further validated the effectiveness and robustness of LMsT.

The second main contribution in this section is that we explored the way to leverage Flash as a secondary-level host-side cache in virtualization environments. We analyzed disk I/O traces of various workloads to understand I/O access patterns. Based on this study, we designed vFRM to make a cost-effective use of Flash resources in virtualization environments while reducing the cost for CPU, Memory, and Flash device I/O bandwidth. Simulation results showed that vFRM not only outperforms traditional caching solutions in terms of performance utilization, but also incurs orders of magnitude lower cost for memory and Flash device I/O bandwidth. In addition, vFRM effectively avoids cache pollution and eventually yields more improvement in I/O performance.

5 Conclusion and Future Works

This dissertation presents our works on resource management in large scale systems, especially for enterprise cluster and storage systems. Resource management is an important research topic which is required by any man-made system and affects in system evaluation in two basic criteria, i.e., performance and cost. Efficient resource management has a direct positive effect on system performance and cost. Large-scale systems provide shared resources as a pool, which requires a central mechanism for resource provisioning and resource allocation based on the remote demands. The basic resource management schemes include admission control, capacity allocation, load balancing, auto scaling and the kind of policies directly related to the performance criterion, such as quality of service guarantee and performance isolation. In this dissertation, we works on the load balancing in the cluster systems and the capacity allocation and quality of service guarantee in enterprise storage systems.

We first investigated the impact of burstiness on load balancing in the cluster systems and then described our adaptive algorithms for load balancing of computing resources under bursty workloads. Our new static ARA algorithm tunes the load balancer by adjusting the trade-off between randomness and greediness in the selection of sites. While this approach gives very good performance, tuning the algorithm can be difficult. We therefore proposed our new online ARA algorithm that predicts the beginning and the end of workload bursts and automatically adjusts the load balancer to compensate. We show that the online algorithm gives good results under a variety of system settings. This approach is more robust than the static algorithm, and does not require the algorithm parameters to be carefully tuned. We conclude that an adaptive, burstiness-aware load balancing algorithm can significantly improve the performance of computing systems.

Apart from resource management in cluster systems, we also investigated data management in enterprise storage systems. Storage is another critical resource in contemporary computer systems. Especially, with low latency and low power consumption, Flash-based drive is being widely deployed as storage or cache in the storage systems to improve I/O performance and reduce power consumption. Thus, we further investigate the complexity and challenges in the resource management of Flash-based storage systems.

We proposed LMST, a live data migration algorithm for efficiently utilizing the shared storage resources and meeting various application SLAs in a multi-tiered storage system. We have shown that bursty workloads in storage systems can deteriorate system performance, causing high I/O latency and large numbers of SLA violations in low performance tiers. In order to mitigate such negative effects, hot data that are associated with those bursty workloads should be migrated to high performance tiers. However, extra I/Os due to data migration as well as the newly migrated bursty workloads can incur addi-

tional SLA violations to high priority applications in high performance tiers. Therefore, we designed LMsT to counteract the impacts of burstiness by efficiently utilizing the high-performance devices, and to minimize the potential delays to latency-sensitive applications. Trace-driven simulations have been conducted to evaluate the performance of our new LMsT policy. Compared to the null migration policy, LMsT significantly improves average I/O response times, I/O violation ratios and I/O violation times.

In the other hand, Flash can be leveraged as a secondary-level host-side cache in virtualization environments. We proposed a new Flash Resource Manager, named vFRM, which aims to maximize the utilization of Flash resources with minimal I/O cost for managing and operating Flash. vFRM adopts the ideas of heating and cooling to identify data blocks that can benefit the most from being put in Flash, and lazily and asynchronously migrates data blocks between Flash and spinning disks. We further investigated the benefits of g-vFRM for managing Flash resources among multiple heterogeneous VMs and presented three global versions of Flash resource managing algorithms by extending to deliver IO acceleration in fully leverage the outstanding performance of shared Flash resources under the global view of caching management. Experimental evaluation shows that both vFRM and g-vFRM can achieve better cost-effectiveness than traditional caching solutions, and costs orders of magnitude less I/O cost.

In the future, we propose to optimize both vFRM and g-vFRM algorithms to further explore the benefits by using Flash as a secondary-level host-side cache. We plan to optimize vFRM and g-vFRM in space granularity by leveraging the knowledge of workload statistics. For example, if most of the global bins are sparse, i.e., only a few pieces of contiguous LBAs are actually accessed in a $1MB$ size of logical bin, then it is not necessary to keep the whole bin in Flash. So, we will introduce a mapping table which only records the logically accessed LBAs in the cached global bins and maps them to the physical LBAs of Flash. Based on such optimization, more Flash resources can be saved. In other words, more logical bins can be cached in Flash which will increase the Flash utilization and then improve the performance.

References

[1] P. Chaganti, "Cloud computing with amazon web services," *IBM Technical Library*, pp. 1–10, 2008.

[2] J. Varia, "Building greptheweb in the cloud," 2008.

[3] V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, pp. 28–39, May-June 1999.

[4] T. Kwan, R. McGrath, and D. Reed, "Ncsa's world wide web server: Design and performance," *IEEE Computer*, pp. 68–74, Nov. 1995.

[5] V. P. et al, "Locality-aware request distribution in cluster-based network servers," in *Proceedings of ACM 8th Int'l Conf. Architectual Support for Prog. Langs. and Op. Sys.*, Oct. 1998.

[6] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, Mar. 2005.

[7] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," vol. 24(2), 1977.

[8] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 124, no. 5, 2004.

[9] "Facebook Flashcache," https://github.com/facebook/flashcache.

[10] E. O'Neil, P. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, DC, 1993, pp. 297–306.

[11] M. Kampe, P. Stenstrom, and M. Dubois, "Self-correcting lru replacement policies," in *Proceedings of the 1st conference on Computing frontiers*, Ischia, Italy, 2004, pp. 181–191.

[12] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, 1994, pp. 439–450.

[13] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, 2001, pp. 91–104.

[14] N. Megiddo and D. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[15] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[16] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *IEEE 28th Symposium on Mass Storage Systems and Technologies*, Pacific Grove, CA, 2012, pp. 1–12.

[17] T. Schroeder, S. Goddard, and B. Ramamurthy, "Scalable web server clustering technologies," *IEEE Network*, pp. 38–45, May/June 2000.

[18] Y. M. Teo and R. Ayani, "Comparison of load balancing strategies on cluster-based web servers," *Trans. Soc. for Modeling and Simulation*, vol. 77, no. 5-6, pp. 185–195, Nov. 2001.

[19] Q. Zhang, N. Mi, A. Riska, and E. Smirni, "Performance-guided load (un)balancing under autocorrelated flows," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 19, no. 2, pp. 652–665, 2008.

[20] N. Mi, Q. Zhang, A. Riska, and E. Smirni, "Load balancing for performance differentiation in dual-priority clustered servers," pp. 385–395, 2006.

[21] J. Tai, J. Zhang, J. Li, W. Meleis, and N. Mi, "Ara: Adaptive resource allocation for cloud computing environments under bursty workloads," in *Proc. of IEEE International Performance Computing and Communications Conference (IPCCC'11)*, 2011, pp. 1–8.

[22] H. Feng, M. Visra, and D. Rubenstein, "Optimal state-free, size-aware dispatching for heterogeneous m/g/-type systems," *Performance Evaluation J.*, vol. 62, no. 1-4, pp. 475–492, Nov. 2005.

[23] M. Harchol-Balter and A. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Trans. Computer Systems*, vol. 15, no. 3, pp. 253–285, Aug. 1997.

[24] W. Winston, "Optimality of the shortest line discipline," *Journal of Applied Probability*, vol. 14, pp. 181–189, 1977.

[25] R. Weber, "On the optimal assignment of customers of parallel servers," *Journal of Applied Probability*, vol. 15, pp. 406–413, 1978.

[26] R. Nelson and T. Philips, "An approximation for the mean response time for shortest queue routing with general interarrival and service times," *Performance Evaluation*, vol. 17, pp. 123–139, 1998.

[27] W. Whitt, "Deciding which queue to join: Some counterexamples," *Operations Research*, vol. 34, no. 1, pp. 226–244, Jan. 1986.

[28] M. Harchol-Balter, M. Crovella, and C. Murta, "On choosing a task assignment policy for a distributed server system," *J. Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, Nov. 1999.

[29] F. Bonomi, "On job assignment for a parallel system of processor sharing queues," *IEEE Trans. on Computers*, vol. 39, no. 7, pp. 858–869, 1990.

[30] H. Mor, S. Alan, and Y. R. Andrew, "Surprising results on task assignment in server farms with high-variability workloads," in *Proceedings of ACM SIGMETRICS 2009 Conference on Measurement and Modeling of Computer Systems.*, Seattle, WA, June 2009.

[31] E. Bachmat and H. Sarfati, "Analysis of sita policies," *Performance Evaluation*, vol. 67, no. 2, pp. 102–120, 2010.

[32] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, pp. 237–246, 2002.

[33] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," pp. 127 – 132, 2000.

[34] H. Li and M. Muskulus, "Analysis and modeling of job arrivals in a production grid," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 4, pp. 59–70, 2007.

[35] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel, "Performance impacts of autocorrelated flows in multi-tiered systems," *Perform. Eval.*, vol. 64, no. 9-12, pp. 1082–1101, 2007.

[36] A. Riska and E. Riedel, "Disk drive level workload characterization," in *USENIX Annual Technical Conference, General Track 2006*, 2006, pp. 97–102.

[37] V. Paxson and S. Floyd, "Wide-area traffic: The failure of poisson modeling," vol. 3, pp. 226–244, 1995.

[38] M. Vlachos, K.-L. Wu, S.-K. Chen, and P. S. Yu, "Correlating burst events on streaming stock market data," *Journal of Data Mining and Knowledge Discovery*, vol. 16, no. 1, pp. 109–133, 2008.

[39] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel., "Performance impacts of autocorrelated flows in multi-tiered systems," *Performance Evaluation*, 2007.

[40] R. Onvural and H. Perros, "equivalencies between open and closed queueing networks with finite buffers," *Performance Evaluation*, vol. 9, 1989.

[41] B. Laliberte, "Automate and Optimize a Tiered Storage Environment-FAST!" White Paper, 2009, http://www.emc.com/collateral/analyst-reports/esg-20091208-fast.pdf.

[42] B. Lundell, J. Gahm, and J. McKnight, "2011 IT Spending Intentions Survey," Research Report, 2011, http://www.enterprisestrategygroup.com/2011/01/2011-it-spending-intentions-survey/.

[43] M. Peters, "Storage Tiering," Market Landscape Reports, 2011, http://www.enterprisestrategygroup.com/2011/07/storage-tiering/.

[44] "IBM DS8000," http://www-03.ibm.com/systems/storage/disk/ds8000/.

[45] "EMC FAST," http://www.emc.com/products/launch/fast/.

[46] "HP 3PAR Adaptive Optimization Software," http://h18006.www1.hp.com/storage/software/3par/aos/index.html.

[47] S. Khuller, Y. Kim, and Y. Wan, "Algorithms for data migration with cloning," in *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. San Diego, California: ACM, 2003, pp. 27–36.

[48] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes, "An experimental study of data migration algorithms," in *Workshop on Algorithm Engineering*. London, UK: Springer, 2001, pp. 145–158.

[49] V. Sundaram and P. Shenoy, "Efficient data migration in self-managing storage systems," in *IEEE International Conference on Autonomic Computing*, Dublin, Ireland, 2006, pp. 297–300.

[50] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: performance isolation and differentiation for storage systems," in *Twelfth IEEE International Workshop on Quality of Service*, Palo Alto, CA, 2004, pp. 67–74.

[51] B. Seo and R. Zimmermann, "Efficient disk replacement and data migration algorithms for large disk subsystems," *ACM Transactions on Storage*, vol. 1, no. 3, pp. 316–345, 2005.

[52] C. Lu, G. A. Alvarez, and J. Wilkes, "Aqueduct: Online data migration with performance guarantees," in *Proceedings of the 1st USENIX Conference on FAST'02*. Monterey, CA: ACM, 2002, pp. 219–230.

[53] G. Zhang, L. Chiu, and L. Liu, "Adaptive data migration in multi-tiered storage based cloud environment," in *IEEE 3rd International Conference on Cloud Computing*, Miami, FL, 2010, pp. 148–155.

[54] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proceedings of the 9st USENIX Conference on FAST'11*. San Jose, CA: ACM, 2011, pp. 20–20.

[55] "VMware Flash Cache Project," https://wiki.eng.vmware.com/VFC.

[56] "EMC FAST CACHE," http://www.emc.com/collateral/white-papers/fast-cache-wp.pdf.

[57] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, San Jose, CA, 2007, pp. 61–76.

[58] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009, pp. 145–158.

[59] T. Pritchett and M. Thottethodi, "Sievestore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010, pp. 163–174.

[60] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "Ssd bufferpool extensions for database systems," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[61] "CBRC," https://wiki.eng.vmware.com/CBRC.

[62] G. Venkitachalam, F. Tian, B. Weissman, M. Vilayannur, R. Venkatasubramanian, and J. Ramnarayan, "A case for supporting low-latency direct-attached storage in the vi platform," in *Proceedings of VMware vRadio 2011*, Palo Alto, CA, 2011.

[63] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebooks distributed data store for the social graph," in *Proceedings of the 2013 USENIX Annual Technical Conference on ATC'13*, San Jose, CA, 2013, pp. 49–60.

[64] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, Bejing, China, 2008, pp. 327–338.

[65] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proceedings of the International Conference on Supercomputing*, Tucson, Arizona, 2011, pp. 22–32.

[66] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th symposium on Operating systems design and implementation*, Boston, MA, 2002, pp. 181–194.

[67] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *Proceedings of the 6th ACM symposium on Operating systems principles*, 1997, pp. 143–156.

[68] "CSIM19 development toolkit for simulation and modeling," http://www.mesquite.com, 2005.

[69] J. Zhang, N. Mi, J. Tai, and W. Meleis, "Decentralized scheduling of bursty workload on computing grids," in *IEEE International Conference on Communications (ICC)*, 2011.

[70] M. F. Neuts, *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. New York: Marcel Dekker, 1989.

[71] D. Cox and P. Lewis, *The Statistical Analysis of Series of Events*. New York: John Wiley and Sons, 1966.

[72] R. Gusella, "Characterizing the variability of arrival processes with indexes of dispersion," *IEEE JSAC*, vol. 19, no. 2, pp. 203–211, 1991.

[73] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 1094–1104, October 2001.

[74] A. Caniff, L. Lu, N. Mi, L. Cherkasova, and E. Smirni, "Fastrack for taming burstiness and saving power in multi-tiered systems," in *Proceedings of the 22nd International Teletraffic Congress (ITC'10)*, Amsterdam, The Netherlands, 2010.

[75] "VMware vCenter Server," http://www.vmware.com/products/vcenter-server/overview.html.

[76] S. Kavalanekar, B. Worthington, Z. Qi, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, Seattle, WA, 2008, pp. 119–128.

[77] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "Srcmap: Energy proportional storage using dynamic consolidation," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2010.

[78] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Warming up storage-level caches with bonfire," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2013, pp. 59–72.

[79] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage*, vol. 4, no. 3, pp. 10:1–10:23, 2008.

[80] S. B. Vaghani, "Virtual machine file system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 57–70, 2010.

[81] B. Sorav and M. S. Dharmendra, "Car: Clock with adaptive replacement," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2004, pp. 187–200.