Flash Storage Management Algorithm
for Large-Scale Hybrid Storage Systems

by

Abdullah Hasan Aldhamin
B.Sc. (Honours), King Fahd University of Petroleum and Minerals, 2007

A Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Abdullah Hasan Aldhamin 2014
SIMON FRASER UNIVERSITY
Fall 2014

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Abdullah Hasan Aldhamin

Degree: Master of Science

Title of Thesis: Flash Storage Management Algorithm for Large-Scale Hybrid Storage Systems

Examining Committee:

Dr. Joseph Peters
Chair

Dr. Mohamed Hefeeda,
Senior Supervisor, Professor

Dr. Raed Al Shaikh,
Supervisor, IT Systems Specialist, Saudi Aramco

Dr. Jiangchuan (JC) Liu,
Internal Examiner, Professor

Date Approved: November 27, 2014
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the non-exclusive, royalty-free right to include a digital copy of this thesis, project or extended essay[s] and associated supplemental files (“Work”) (title[s] below) in Summit, the Institutional Research Repository at SFU. SFU may also make copies of the Work for purposes of a scholarly or research nature; for users of the SFU Library; or in response to a request from another library, or educational institution, on SFU’s own behalf or for one of its users. Distribution may be in any form.

The author has further agreed that SFU may keep more than one copy of the Work for purposes of back-up and security; and that SFU may, without changing the content, translate, if technically possible, the Work to any medium or format for the purpose of preserving the Work and facilitating the exercise of SFU’s rights under this licence.

It is understood that copying, publication, or public performance of the Work for commercial purposes shall not be allowed without the author’s written permission.

While granting the above uses to SFU, the author retains copyright ownership and moral rights in the Work, and may deal with the copyright in the Work in any way consistent with the terms of this licence, including the right to change the Work for subsequent purposes, including editing and publishing the Work in whole or in part, and licensing the content to other parties as the author may desire.

The author represents and warrants that he/she has the right to grant the rights contained in this licence and that the Work does not, to the best of the author’s knowledge, infringe upon anyone’s copyright. The author has obtained written copyright permission, where required, for the use of any third-party copyrighted material contained in the Work. The author represents and warrants that the Work is his/her own original work and that he/she has not previously assigned or relinquished the rights conferred in this licence.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2013
Abstract

As the computing platforms have evolved over the years, the associated storage requirements have also followed a rapid change in terms of performance, cost, availability and scalability. In addition, computing elements, mainly the CPU, are continuing to scale and develop at a higher pace compared to storage systems. Flash-based solid-state drives (SSDs) have led to significant innovations in storage systems architecture. However, due to their special design and architecture characteristics, they are not considered as cost-effective and immediate replacement of traditional hard-disk drives for large-scale storage systems. Thus, how we can best utilize this technology to build an efficient hybrid storage system remains a research challenge. We propose a real-time dynamic programming algorithm, called Flash Storage Management (FSM) algorithm, to address this challenge. The FSM algorithm can run in two modes: online and offline modes. We implement the proposed FSM algorithm in an event-driven simulator. To compare the FSM algorithm, we implement a simulator for the closest algorithms in the literature, which is Hystor. Our evaluation results indicate that the proposed algorithm outperforms Hystor, especially for read-intensive workloads. For example, the online FSM algorithm achieves a hit ratio of 75% when using SSD size of 30% of the workload, which outperforms Hystor by more than 20%.

Keywords: large-scale hybrid storage, solid-state drive, flash storage management;
To my holy love, my wife Zinab and my daughters, Fatimah and Maryam
“He who travels in the search of knowledge, to him God shows the way of Paradise.”

— Prophet Mohammad
Acknowledgments

Foremost, I would like to express my sincere gratitude to my senior supervisor, Dr. Mohamed Hefeeda, for mentoring me in the past two years with his patience and immense knowledge. Also, I would like to express my gratitude to Dr. Raed Al Shaikh, my supervisor, and Dr. Jiangchuan (JC) Liu, my thesis examiner, for being on my committee and reviewing this thesis. I also would like to thank Dr. Joseph Peters for taking the time to chair my thesis defense. I would also like to express my deepest thanks to Dr. Tamir Hegazy for his time and immense knowledge. For all I have learned from him and for his continuous help and support in all stages of this thesis.

I would like to thank all my colleagues at Network Systems Lab for their support and help. It was my honor to work with these talented people. I would also like to thank my friends for their encouragement and constant support. A big thank-you goes to my friend Christopher Morris and his family for the support and encouragement they have given me.

I am indebted to my family for their love, encouragement, and endless support. I owe my deepest gratitude to my parents and my parents-in-law. I owe my loving thanks to my wife, Zinab, and my daughters Fatimah and Maryam. They have lost a lot during my graduate career at SFU, and more especially during my illness. Without their encouragement and understanding it would be impossible for me to finish this thesis. I will never forget what I owe them, and my eternal gratitude to them cannot be expressed by words. This thesis is dedicated to them.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Partial Copyright License</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Dedication</td>
<td>vi</td>
</tr>
<tr>
<td>Quotation</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>Contents</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement and Thesis Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Organization</td>
<td>4</td>
</tr>
<tr>
<td>2 Background and Related Work</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Storage Systems</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Solid-State Drive: Architecture</td>
<td>10</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Gordon technical specifications. ........................................ 16
3.1 List of symbols used in the thesis. ................................. 30
List of Figures

1.1 Integration of SSDs in storage systems. ................................. 3
2.1 Architecture of the Storage Area Network (SAN). ...................... 7
2.2 Architecture of the Network Attached Storage (NAS). .................. 8
2.3 Logical architecture of flash-based solid state drives. ................. 10
2.4 Hystor architecture. ...................................................... 24
3.1 Main architecture. ......................................................... 28
3.2 FSM Flow from input into performance measurement output. ........... 29
3.3 Online flash storage management algorithm flow chart. ................. 34
3.4 Offline flash storage management algorithm flow chart. ................. 36
3.5 Database management system components. ................................ 43
4.1 Cost Function Convergence for the Flash Storage Management (FSM) Algorithm using read-intensive workload .............................. 48
4.2 Cost Function Convergence for the Flash Storage Management (FSM) Algorithm using write-intensive workload .............................. 49
4.3 FSM hit ratio for read-intensive workload: Online vs. Offline .......... 50
4.4 FSM hit ratio for write-intensive: Online vs. Offline ................. 51
4.5 FSM hit ratio for composite workload: Online vs. Offline ............. 51
4.6 FSM data swapping for read-intensive workload: Online vs. Offline .. 53
4.7 FSM data swapping for write-intensive workload: Online vs. Offline .. 54
4.8 FSM data swapping for composite workload: Online vs. Offline .......... 54
4.9 Block Table Data Structure used in Hystor .............................. 56
4.10 Hit ratio comparison for a read-intensive workload: Hystor vs. Online FSM . 57
4.11 Hit ratio comparison for a write-intensive workload: Hystor vs. Online FSM 57
4.12 Hit ratio comparison for a composite workload: Hystor vs. Online FSM . . . 58
4.13 Data swapping for read-intensive workload: Hystor vs. Online FSM . . . . . 58
4.14 Data swapping for write-intensive workload: Hystor vs. Online FSM . . . . . 59
4.15 Data swapping for a composite workload: Hystor vs. Online FSM . . . . . . 59
Chapter 1

Introduction

1.1 Overview

Storage systems are essential components of the computing hierarchy, ranging from mobile devices and personal computers all the way through high performance computing (HPC) and cloud computing. Similar to other computing elements, storage technology has enjoyed considerable growth since the first drive was introduced in 1956 [34]. This is in part facilitated by the steady evolution of storage interfaces, e.g., Small Computer System Interface (SCSI) and AT Attachment (ATA). Since the first disk, disks have grown by over six orders of magnitude in density and over four orders in performance. As the computing platforms have evolved over the years, the associated storage requirements have also followed a rapid change in terms of performance, cost, availability and scalability.

In addition, the computing environments have also changed. As organizations grow in business, a new transformation is needed to ensure a cost-effective computing environment is available to meet the new computational challenges. Furthermore, it has become essential to share data among different entities within and outside organizations. Thus, options for connecting computers to storage systems have increased dramatically in a short time, driven by the rapid change in applications development. Storage networking offers significant capabilities and flexibilities not previously available. Also, there is no single networking approach that solves all problems or optimizes all variables. There are trade-offs in cost, ease-of-management, performance, distance and maturity, to name a few of these variables. Thus, multiple storage network alternatives coexist within the same organization to serve different needs. It is important to note that the different storage architectures are the results
from different storage interfaces. That is, the key principal that differentiates between storage architectures is the interface between the storage and the computing system, which determines the functionality supported by the devices.

Storage media devices have evolved drastically from being large trunks with the capacity to hold a few kilobytes of data, to microchips able to hold a few terabytes of data. Hard disk drives (HDDs) have evolved in both size and I/O speed. However, due to their mechanical design nature they have reached levels where it cannot exceed without compromising other factors such as power consumption or performance. Thus, storage devices continue to be a concern as processors get more advanced and data acquisition rate is increasing. Solid-state drives (SSDs) are a newer type of storage that contains no moving parts allowing data to be stored on flash memory technology. But, SSDs come in smaller sizes compared to HDDs and they have more complicated design internals that brings several concerns to adopt this new technology at large-scale levels.

1.2 Problem Statement and Thesis Contributions

Solid-state drives have revolutionized the storage technology with their performance characteristics, such as their random read/write I/O performance, low energy consumption and compact size. They are used in consumer devices, such as smart phones, tablets and personal computers. In recent years there has been a great interest in adopting this technology in large-scale storage systems to improve the storage I/O performance, and hence enhance applications’ runtime.

With all the attractive features of this technology, replacing the conventional hard-disk drives with SSDs may not be a feasible option for large-scale storage systems. This is mainly due to their small capacity, limited lifetime, and their relatively high cost. Thus, a more practical solution is to use SSDs in a hybrid system, such that the features of this technology are best utilized. The key challenges are to decide what role should the SSD have in the storage hierarchy, and what data should be stored in it.

Figure 1.1 summarizes the possible scenarios to integrate SSDs in storage systems. Should we implement a flash-only storage or consider a hybrid solution that takes advantage of the huge capacity of the HDD and the performance advantages of the SSD? If we consider a hybrid approach, then what role should the SSD be given? In the case of a hybrid solution, it can be used as the major storage media to store all data and serve all I/O requests from
The problem addressed in this thesis can be stated as follows: given a hybrid storage system composed of HDDs as the primary storage media and SSDs as the secondary storage media. Data is stored in HDDs and can be cached in the SSD. Design an algorithm to allocate or cache the data in SSDs such that it maximizes data accessed from SSDs in order to improve the performance of the storage system.

To solve this problem, this thesis makes the following contributions:

a) we propose a pre-allocation flash storage management algorithm based on real-time dynamic programming technique for large-scale hybrid storage systems, where SSDs are used as a secondary storage device to improve the storage I/O performance. Our algorithm efficiently uses a pre-collected I/O trace log of the target application to train the storage system by: (a) extracting the most performance critical data chunks; and (b) ensuring that the most performance critical data are accessed from SSDs, and consequently improving the application performance.

b) we implement an event-driven simulator to assess the performance of our proposed algorithm,
c) we conduct extensive experiments with different I/O workloads. Our simulation results show that our proposed algorithm consistently improves the performance with different system parameters,

d) we compare our proposed algorithm to the closest work in the literature, i.e., Hystor. Our experimental results confirm that our algorithm outperforms Hystor.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides some background on flash-based solid-state drives architecture, characteristics and features, and their applications, especially hybrid storage systems. It also summarizes the related work in the literature. In Chapter 3, we describe the proposed flash storage management algorithm and the considered system model. We discuss and analyze the algorithm complexity and consider possible applications for our algorithm. Evaluation using simulation of the proposed algorithm is given in Chapter 4. We conclude the thesis in Chapter 5.
Chapter 2

Background and Related Work

This chapter first provides background on storage systems, architecture of SSDs, SSDs salient features and their implications, I/O characteristics of applications. We then describe the Markov Decision Process, which is used in solving the considered problem. Finally, we survey the related work in the literature.

2.1 Storage Systems

There are three key types of storage networking models, namely: direct attached storage (DAS), network attached storage (NAS), and storage area network (SAN) [44, 21, 13]. The most traditional architecture of storage is the direct attached storage. As the name suggests, the storage media, usually a disk or a tape drive, is directly attached through a cable to the computer system. It is generally restricted to access by a single host. This architecture is optimized for a single isolated computing, and usually resides inside the server enclosure. In addition, it provides users with block-level access to data through the SCSI protocol. This architecture, however, does not seem to be a viable option for today’s enterprise computing environment, where there is huge amount of data to be shared among users. Thus, several standards are used to enable these capabilities through networking the storage system.

One of the most used network architecture is the storage area network, SAN, which is a dedicated network for storage devices and the processors that access those devices [19, 37]. SANs, as illustrated in Figure 2.1, are usually built using Fiber Channel (FC) technology. Similar to DAS, SANs also provide block I/O access to storage through the SCSI protocol. In order to allow SCSI I/O commands to be sent over TCP/IP based network, the Internet
CHAPTER 2. BACKGROUND AND RELATED WORK

SCSI, commonly known as iSCSI, protocol is used to facilitate this feature. Storage devices in the SAN system are managed through a SAN server. SAN provides several advantages to computing environment. First, it improves the storage utilization through consolidating the scattered storage devices into fewer devices. Second, it enables sharing among users. Third, one of the most important features of SAN is that it provides scalability, since multiple SAN devices can be made available as a single pool of storage to all processors on the SAN. Additionally, a centralized management for the whole SAN minimizes the overhead for the system administrators. On the other hand, there are certain aspects of the SAN that need careful consideration to ensure a cost-effective implementation. Since SANs use a specialized network the initial cost to implement SAN will generally be higher than other options. Furthermore, SANs require specialized hardware and software to provide many of their potential benefits. Also, an organization must add new skills to manage this sophisticated technology. Yet, some analysis [44, 20, 8, 43] suggest that SANs have lower total cost of ownership (TCO) in the long-term compared to the alternative connectivity approaches.

Another widely used connectivity approach for storage systems is the network-attached storage, commonly known as NAS [44, 37, 13]. A NAS device, as illustrated in Figure 2.2, consists of an integrated processor called controller or the NAS head and disk storage. The NAS device is connected to a standard TCP/IP network and provides access to users through file-access/file-sharing protocols such as network file system (NFS) or the common Internet file system (CIFS). The controller in the NAS device manages the files located on disks, and issues block I/O requests to the disks to fulfill the file I/O read and write requests it receives. There are few distinctions between SAN and NAS approaches. First, the main difference is the storage interface; SAN enables block I/O access whereas NAS provides file I/O access, which is a higher-level request that specifies the file to be accessed, an offset into the file, and a number of bytes to read or write beginning at that offset. This, in turn, suggests that file I/O has no knowledge about disk volume or disk sectors. Second, unlike SAN, NAS device resides on a network that might be shared with non-storage traffic.

There are several benefits that NAS brings giving it the advantage for widespread adoption among other options [37, 44]. Probably the most notable feature is its ease of installation. It does not require special hardware, or specific skills for installation and management. Additionally, it can be installed on an existing LAN/WAN network. Another salient feature of NAS appliances is the snapshot backup, which makes backup copies of data onto
Figure 2.1: Architecture of the Storage Area Network (SAN).
Figure 2.2: Architecture of the Network Attached Storage (NAS).
tapes, for example, while minimizing application downtime. Similar facilities are available for SAN as well, but they require specific customization and special storage management packages. Furthermore, NAS allows capacity within the appliance to be pooled. That is, the NAS device is configured as one or more file systems, each residing on a specified set of disk volumes. All users accessing the same file system are assigned space within it on demand. This is more efficient than buying users their own disk volumes, as in DAS scenario, which often leads to some users having too much capacity and others too little. On the other hand, however, because NAS pooling resides within a NAS appliance, there is little if any sharing of resources across multiple appliances. This raises costs and management complexity as the number of NAS nodes increases. In contrast, an advantage of a SAN is that all devices on a SAN can be pooled. Another NAS capability is file sharing. Although it can be configured on SAN, it requires further specialized software. Both SAN and NAS utilize RAID systems that are connected to an interconnect network. The choice of RAID scheme is dependent on target security and redundancy level. RAID 6, known commercially as RAID DP, is probably the most widely used scheme on networked storages. It provides block-level striping with double distributed parity, which makes it very efficient to provide fault tolerance of up to two disks in the same RAID.

Other computing elements, mainly the CPU, are continuing to scale and develop at a higher pace compared to storage systems [6]. Even as storage bandwidth increased (but slower than compute speed), storage performance improved only marginally. This has added even more challenges to close the gap between the processor and the storage in order to ensure a scalable computing infrastructure, especially when dealing with today’s complex computational problems. Storage media devices, such as hard disk drives, are the core part of any storage system. Due to the nature of their mechanical design, improving their performance requires speeding up their rotations, which, in turn, would require more mechanical power. Flash-based solid-state drives have led to significant innovations in storage systems architecture. They have been instrumental in modern consumer devices such as personal computers, smart phones, as well as tablets. This is mainly due to their design, which is based on electronic chips, and small size. They use significantly less power, and provide higher performance. Furthermore, in recent years SSDs have been introduced in large-scale storage systems. However, the challenging problem is how to integrate this interesting technology in the storage hierarchy in a way that best utilizes its features.
2.2 Solid-State Drive: Architecture

Among the necessities of large-scale storage systems is to deliver high and reliable performance especially for the type of applications that require high IO per second (IOPS) while at the same time maintaining the cost associated as low as possible. One of the direct performance bottlenecks in these systems is the hard disk drives. Solid state drives bring solutions to solve the performance challenges, e.g., their performance for the random access, but they also bring some weaknesses to the storage systems. In addition, they support the same interfaces as conventional hard disk drives, both physically, e.g., SATA interface and logically, e.g., Logical Block Address, which makes switching to SSDs intuitive.

Flash-based SSDs are typically built on an array of flash memory packages [42], as shown in Figure 2.3. As logic pages are stripped over flash chips, high bandwidth can be achieved through parallel access. In addition, a serial I/O bus connects the flash memory package to a controller. The controller receives and processes requests from the host through connection interface and issues commands to transfer data from/to the flash memory array. When a read request arrives, the data is first read from flash memory into the register of the plan, then the data is shifted via the serial bus to the controller. A write request is processed with similar steps but in a reverse order. In some designs, SSDs are equipped with an external RAM buffer to cache data or metadata [28].

Flash SSDs are made of semiconductor chips. Based on the chip size, flash SSDs are classified into three types: Single-Level Cell (SLC), Multi-Level Cell (MLC) and Triple-Level Cell (TLC). The choice of the cell size imposes several design decisions, namely:
density, reliability and performance. Hence, SLCs chip can store 1-bit, MLCs 2-bit, and TLCs 3-bit each. More bits per chip increases the capacity of the device. However, this has a drawback concerning the device reliability. That is, more bits per chip leads to more frequent garbage collection and hence the device tends to wear out faster. In addition, performance is impacted by the number of bits per cell, due to the increased overhead to manage data within cells [25].

The core component of the SSD is the Flash Translation Layer [5, 28], which is implemented in the SSD controller to emulate a hard disk and exposes an array of logic blocks to the upper level components. Its significant role in SSD design has led to several sophisticated mechanisms to optimize the SSD performance and lifetime. Flash Translation Layer performs three major tasks in SSDs. First, logical block mapping, to organize write operations mapping from logical block address to a physical block address. Unlike conventional hard disk drives, writes in SSDs cannot be performed in place and each write of a logic page is actually conducted on a different physical page. Different approaches have been implemented for address mapping using different granularities, e.g., block vs. page [26]. In most cases, however, Flash Translation Layers are designed to use a hybrid approach, such that block-level mapping is used to map block as data blocks and page-level mapping to manage small set of logical blocks. This choice is efficient as a buffer to accept incoming write requests.

Second, Flash Translation Layer facilitates garbage collection operations. In SSDs, a block must be erased (programmed) before it is reused, which makes SSDs usually over-provisioned with a certain amount of clean blocks as an allocation pool. In case of writes, previously occupied physical page is invalidated by updating the metadata, and new data can be appended to new clean block from the pool, without having to synchronously performing erase operation. When running out of clean blocks in the pool, a garbage collector scans the log and recycles invalidated pages. Third, another important task of Flash Translation Layer is the wear leveling. Due to the locality in most workloads, writes are often performed over a subset of blocks. Thus, some flash memory blocks may be frequently overwritten and tend to wear out earlier than others. Flash Translation Layer implements wear-leveling algorithms to even out writes over flash memory blocks. The efficiency of the algorithms used directly impacts the lifetime of the flash SSD.
2.3 Solid-State Drives: Characteristics and Implications

The design nature of the flash-based solid-state drives introduces several layers of challenges [23, 5]. Thus, an important question that needs to be addressed before making a switch from HDD to SSD is where and how the SSD can bring a cost-effective solution to improve the performance of existing storage systems, considering the I/O characteristics discussed in Section 2.4.

Feng Chen et al [17] provide a comprehensive study on SSDs performance for various workloads trying to answer several questions and verifying common notions about this technology. Their work indicates that SSDs, as expected, show huge performance improvement for random reads over HDDs: their results show that SSDs achieve 31 times higher throughput. Interestingly, sequential reads performance on SSDs had also overtaken HDDs. Moreover, SSDs show a non-uniform read access latencies, an indication that the performance is correlated with workload access pattern. In contrast, write accesses to SSDs are almost independent of workload access pattern. This is associated with the fact that write requests will write to on-drive cache, regardless of the access pattern. Similar to HDDs, SSDs on-drive cache plays a significant role on SSDs design to boost the write performance. In addition, internal drive-specific operations can significantly impact the performance of the SSD. Most of these internal operations, such as garbage collection and wear leveling, are caused by write requests. Their results demonstrate that read operations are affected more than writes, since reads will most likely be served directly from the flash rather than the cache. For example, after inserting 10ms interval between writes, reads with lower latencies increases from 57% to 74% in one case and from 78% to 95% in another. This shows how much impact internal operations could have on the foreground jobs. Also, as the workload randomness increases, the write performance of SSDs suffers a significant degradation, which may lead to further implications on reads performance as well. Furthermore, the actions performed by file systems, e.g., journaling and metadata synchronization, worsen the flash write amplification problem [33], because it produces extra write operations that contribute to the wear of the SSD.

Several important implications could be drawn from these results and observations. First, read access latency on SSDs is not always uniform and correlated to the workload access pattern. Second, write performance on SSDs with on-drive caches imposes a uniform behavior regardless of the workload access performance. Third, internal drive-specific...
operations driven by incoming write requests can have significant impact on the overall performance of SSDs, especially on reads. Fourth, internal fragmentation could also affect the storage performance. Fifth, the more the frequency of write operations to SSDs produce extra write requests by the file system which increases the problem of write amplification and thus impact the wear of the SSD [33]. These implications show that SSDs may not necessarily win a performance comparison against HDDs. SSDs have a strong performance lead over HDDs when dealing with random read workload, but the performance gap gets narrower with sequential workloads.

Narayanan et al. [35] provide a cost analysis model of the SSD/HDD trade-off for different large data center servers workloads. Their results show that replacing HDDs with SSDs is not a cost-effective solution for most of the workloads due capacity of SSDs per dollar. Although the cost of SSDs continues to decline, but we still think that for enterprise workloads this remains a concern towards building an all-SSD storage solution. On the other hand, the benefits of hybrid storage systems can also be very limited and less efficient if not used the right way considering target workloads and performance correlation with access patterns.

### 2.4 Understanding Application I/O Characteristics

It is important to understand the target application I/O characteristics in order to be able to optimize its performance. There are several important I/O characteristics that need to be considered. First, we need to understand the amount of I/O load, i.e., how much I/O the application is doing and how it is changing over time. Second, we need to consider the I/O request size. There are applications that do small block I/Os, while others do large streaming I/O in which megabytes of data are transferred at a time. It is important to understand the I/O request size because it affects other optimization metrics. For example, applications with large I/O may impact the latency if large I/O holdup the storage port for longer periods. Third, we need to identify the access pattern. That is, is it a read workload, write workload, or a mixed read/write workload? In addition, we need to understand the locality of access of the application I/O. There is a significant compounding factor that is dramatically accelerating the demands on storage performance, which is the increasing randomization of I/O operations. The primary cause of this increased randomization trend is consolidation, which means that as more applications and systems are consolidated and
virtualized, they no longer enjoy the sense of dedicated storage systems and disks. In fact, the move toward improving data center efficiency through consolidation has been ongoing for the past decade [11].

2.5 Markov Decision Process

Markov decision process (MDP) is a discrete time stochastic process designed to provide a mathematical framework to model decision-making problems that are partially random and/or partially under control [40]. It is widely used when studying optimization problems solved with dynamic programming and/or reinforcement learning. At each time step, the process is in a certain state $S_i$, and the controller can perform an action $a$ available for that state. The process, in turn, responds to the action by moving into a new state $S_j$, and resulting in a corresponding cost. Thus, there is a probability for the process to move into $S_j$, and is influenced by the chosen action.

The Markov decision process framework is defined by a 4-tuple process $(S, A, T, C)$, such that, $S$ is the state space, $A$ is the action space, $T$ is a 2-D transition matrix, and $C$ is the cost incurred of moving from state $S_i$ into state $S_j$ giving by applying action $a(S_i)$. Each element, $P_{ij}$, of the matrix expresses the probability for the process to move from state $S_i$ at time $t$ into state $S_j$ at time $t+1$.

The ultimate target of solving a Markovian decision process is to find an optimal action, or policy. A policy is a function that guides the system to apply the best possible action given the current circumstances. An action is optimal if, with respect to the space of all possible actions, it minimizes the expected discounted total cost from all states. There are different approaches to solve a Markovian decision process: linear programming and dynamic programming. In this thesis, we take the dynamic programming approach to solve our optimization problem for a hybrid storage system.

2.6 Related Work

There are several proposals and commercial implementations already in place that utilize SSDs in storage systems [7]. For example, in 2008 two leading on-line search engine service providers, google.com and baidu.com, both announced their plans to migrate existing hard disk based storage systems to a platform built on SSDs, though it is not clear whether they
have completely migrated all HDDs into SSDs or they built hybrid systems [18, 45]. In the following, we describe several storage systems proposed in the literature.

2.6.1 Gordon High Performance System: SSD-only Cluster

Motivated by the need for a data-centric and power-efficient high performance computing for the massive growing amount of data, Gordon [14, 15] is designed with SSD-only storage system for data-intensive computing. Gordon uses a total of 300TB of flash-based SSD storage [4, 3]. It is designed with three important goals to achieve, mainly: 1. reduce the performance gap between processing and storage I/O in a large-scale data-intensive computing, 2. significantly improve the computing performance, 3. improve power efficiency in such environment. The Gordon cluster consists of 1024 compute nodes and 64 I/O nodes connected through QDR InfiniBand interconnect switches, in which each switch connects 16 compute nodes and one I/O node [2]. Table 2.1 summarizes the technical specifications of Gordon HPC cluster.

Gordon’s SSD-based storage system is the vital component to achieve those objectives. In order to utilize SSDs to build this system, a major consideration to the design has to take place. Thus, one of the major design aspects of Gordon is the Flash Translation Layer that manages the flash storage. Flash Translation Layer manages the wear-leveling operations as well as maintaining the logical block address. In addition, it schedules the accesses to the storage array to provide a high-performance access. In Gordon, a new Flash Translation Layer design is implemented in order to exploit as much parallelism as possible to improve the performance. Gordon’s Flash Translation Layer is implemented in the flash array controller, which provides hardware interface to the array. The original Flash Translation Layer design that Gordon has extended performs the program operations at a write point, which is a pointer to a page of the flash memory, within the array. A major limitation of this Flash Translation Layer implementation is that it only allows for a single write point, i.e. no parallel operations can be done.

The authors implement three approaches to overcome this limitation. The first is to force dynamic parallelism between accesses to flash array. To achieve this, Gordons Flash Translation Layer supports multiple write points, where each write point is assigned with a sequence number to maintain operations order. So, once a logical block address is written to a particular write point, future writes much also go to the same write point, or to the next write point with larger sequence number.
## System Component Configuration

<table>
<thead>
<tr>
<th>System Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Nodes</strong></td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Flop Speed</td>
<td>333 Gflop/s</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>64 GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>85 GB</td>
</tr>
<tr>
<td><strong>I/O Nodes</strong></td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>64 GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>85 GB</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>4.8 TB</td>
</tr>
<tr>
<td><strong>Full System</strong></td>
<td></td>
</tr>
<tr>
<td>Total Compute Nodes</td>
<td>1024</td>
</tr>
<tr>
<td>Total Compute Cores</td>
<td>16384</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>341 Tflop/s</td>
</tr>
<tr>
<td>Total Memory</td>
<td>64 TB</td>
</tr>
<tr>
<td>Total Memory Bandwidth</td>
<td>87 TB</td>
</tr>
<tr>
<td>Total Flash Memory</td>
<td>300 TB</td>
</tr>
<tr>
<td><strong>QDR InfiniBand Interconnect</strong></td>
<td></td>
</tr>
<tr>
<td>Topology</td>
<td>3D Torus</td>
</tr>
<tr>
<td>Link Bandwidth</td>
<td>8 GB (bidirectional)</td>
</tr>
<tr>
<td><strong>I/O Subsystem</strong></td>
<td></td>
</tr>
<tr>
<td>File Systems</td>
<td>NFS, Lustre</td>
</tr>
<tr>
<td>Storage Capacity (usable)</td>
<td>1.5 PB</td>
</tr>
<tr>
<td>I/O Bandwidth</td>
<td>100 GB/s</td>
</tr>
</tbody>
</table>

Table 2.1: Gordon technical specifications.
The second is to manage flash array at larger granularity by combining physical pages from several dies into super-pages. They examined three ways to create super-pages: horizontal striping, vertical striping and a combination of the two called 2D striping. In horizontal striping, a physical page in a super-page is mapped to a separate bus, which allows access to physical pages to progress in parallel. In this case, the number of buses determines the size of the super-page. In vertical striping, physical pages in a super-page are mapped to separate bus. In this scenario, number of dies in each bus limits the size of the vertical super-page. The 2D striping, in turn, combines both horizontal and vertical in order to create even larger super-pages. Thus, in this case, the flash array is divided into rectangular sets that forms a part of the same horizontal and vertical striping, allowing the Flash Translation Layer to assign one write point for each set. The 2D striping will provide better management overhead as well as minimal memory requirement to store the logical block address table.

However, because of the strict design, which is bounded by the target application, choosing a large super-page may not result in better performance for all workloads. This is especially noticeable when choosing a large super-page size for small sequential reads. A bypassing mechanism is implemented in the Flash Translation Layer to merge incoming read requests with pending requests to the same page.

To evaluate Gordon, they used various benchmarks that use MapReduce [1] for parallel computation. An important finding concerning the power consumption is that using SSDs eliminates the majority of storage system’s idle power by saving over 68%, hence allowing the design to take advantage of more efficient processors. In addition, the results show that Gordon runs faster than disk-based clusters by a factor of 1.5x and its performance is 2.5x more efficient. Nonetheless, Gordons benefits come at a cost. The high cost associated with this system may prevent a widespread adoption of such design. However, the authors argue that for systems that need to store very large amount of data and build with cheap commodity hardware justifies the cost and performance trade-off.

Although some of the design aspects of Gordon might be of great interest to apply, the cost associated of building such a system might restrict the adoption, especially when considering general-purpose servers. In addition, it requires major system modification within the SSDs with new Flash Translation Layer algorithm. This adds extra overhead for maintenance or upgrade. Even if we implement Gordon with hybrid storage approach, in which HDDs are used to store file system replicas, there will be a concern related to the
recovery time of a failed file system, and whether the power efficiency will be compromised.

2.6.2 Griffin Hybrid Storage System: Flash as an End-Point

Griffin hybrid storage \cite{24} is designed with SSDs as the main storage medium and supported with HDDs. Although single-level cell (SLC) flash-based SSDs offer excellent endurance measures, usually 100K program/erase cycles or more, and excellent IOPS, the cost associated might be a big concern for adoption. Thus, the authors suggest using the multi-level cell (MLC) SSDs instead. MLCs, in addition to their cheaper price compared to SLCs, offer larger capacity. However, a key disadvantage with this type of SSDs is their low endurance, averaging 10K program/erase cycles. Thus, a clear trade-off has to be considered, cost vs. endurance (and capacity), especially when considering SSDs as the permanent store. Thus, in order to solve the endurance issue without compromising the cost, Griffin utilizes HDDs as a write buffer for the SSDs to minimize the number of writes going to SSDs, and hence extends the SSDs lifetime.

Griffin appends writes into a log-structured HDD cache and periodically flushes it to SSD, preferably before subsequent reads. The log-structured brings two key advantages. First, it is known that HDDs are slow to handle writes especially when performing random writes. However, HDDs operate at their best when handling sequential workloads. Thus, the log-structured cache will exploit the potential of the HDDs. Second, this aggregation will enhance the lifetime not only by minimizing the number of writes, but also by minimizing the write-amplification. Write-amplification is a hidden phenomenon affecting the SSDs performance as well as their endurance by introducing further write operations in order to synchronize the file system metadata and more device-level write operations.

An important challenge faced by Griffin design is to be able to extend the SSD lifetime without compromising the read performance from SSD. That is, to be able to extend the lifetime of the SSD while ensuring data are ready in SSD for read requests. Thus, Griffin’s design has to address two things in order to retain the performance: data has to be kept in HDDs as long as possible to buffer overwrites, and at the same time data must be flushed from HDD in order to avoid expensive reads. In other words, the caching policy has to answer two questions: what data to cache and how long to cache it for. The choice of a policy is directly dependent on the characteristics of the workloads.

Based on the analysis they performed on various desktop and server workloads collected from previous works, the following conclusions are made:
1. Desktop and server workloads encounter high degree of overwrites, which makes an idealized HDD write cache achieve significant write savings.

2. There is a high degree of spatial locality in block overwrites, which helps achieving high write savings while writing fewer blocks, and in turn reduces the possibility of read penalty, i.e., reads from the HDD.

3. Blocks that are most heavily written, rarely receive reads.

4. There are two important things that help to determine data retention duration in HDD: intervals between writes and subsequent overwrites are typically short for desktop workloads; the time interval between a block write and its consecutive read is large. This suggests that data can be retained in HDDs for long enough period.

In addition, Griffin implements two caching policies as follows:

1. Full caching: this policy caches every write request. This is the default policy.

2. Selective caching: in this policy only the most overwritten blocks are cached. To implement this policy, the overwrite ratio is calculated, which is the ratio of number of overwrites to the number of writes a block receives. The block is written to the HDD if the overwrite ration exceeds a defined overwrite threshold.

### 2.6.3 HybridStore

Motivated by SSD design trade-offs, the HybridStore [31] is designed to integrate SSDs as a balancing, i.e., accelerating, storage unit to improve the performance and provide service differentiation given the cost constraints. To achieve these goals, HybridStore consists of two key elements:

1. A capacity-planning model called HybridPlan, which is a long-term resource provisioner.

2. HybridDyn that handles the performance guarantees.

**HybridPlan**

A key component of the HybridStore system is the planning model. The main objective of the HybridStore model is to minimize the deployment and operation cost, in terms of dollars
($), subject to a combination of both performance and re-deployment limits. Actions occur at coarse time-scale, i.e., months to years. In other words, the capacity-planning problem is formulated as a model to minimize the cost of acquiring and installing HybridStore while meeting the targeted workload performance and useful lifetime budget constraints. The performance budget uses IOPS as a metric for measurement, whereas the lifetime budget represents the time between successive capacity planning decisions and equipment provisioning. That said, the cost of a HybridStore can be expressed as follows:

\[
\text{Cost}_{\text{HybridStore}} = \text{Cost}_{\text{Installation}} + \text{Cost}_{\text{Recurring}}
\]

The \(\text{Cost}_{\text{Installation}}\) refers installation cost of devices, whereas the \(\text{Cost}_{\text{Recurring}}\) covers the cost of associated power, cooling and all maintenance. The constraints of performance and lifetime budget equations are classified in terms of data features, devices capacities and bandwidth as well as the SSD lifetime.

An essential part of the capacity planning formulation is the analysis of the target workload to extract and understand the hidden characteristics. Common features, such as total size, read-to-write ratio and request arrival rate, are first gathered by dividing the workloads to sub-workloads called classes. To achieve this, the entire logical address space of the workload is divided into fixed-size chunks and mapped to different classes. Then, classes are analyzed to find commonality within workload streams.

**HybridDyn**

HybridDyn is a statistical model for the performance of SSD and HDD to make dynamic request partitioning decision. In addition, it employs data management techniques within SSD. HybridDyn consists of several components: performance prediction module, which is the core component of HybridDyn, a fragmentation buster and write regulator.

**Performance Prediction Module for SSD** The core component of the HybridDyn is the performance prediction model based on a very long history. This model is used to take actions targeted to improve the SSD performance, subject to current workload status. For example, a large number of random writes might cause fragmentation over time and the resulting garbage collector invocation would degrade the performance of requests that follow. So, this component uses history of crucial workload characteristics that play major role in predicting the performance changes. These features include: a) average read to write
ratio, b) spatial locality, c) request inter-arrival time, and d) current request size. Using this 
information, the performance prediction model uses the system response time as a measure 
of the SSD performance, which is a function of the device service times and workload.

**Fragmentation Busting**  This component is responsible of maintaining the performance 
level and minimizes the affects of large random write requests to the SSD. As the number 
of random writes allowed into SSD increases, data fragmentation on the device increase 
accordingly, which leads to more frequent garbage collection invocation overhead that de-
grades the overall performance. Fragmentation busting is a flushing technique implemented 
to prevent or minimize fragmented zones. It relies on the device controller to decide on 
which page on the device is causing the fragmentation to be marked for flushing.

**Write Regulation**  Write regulation is a technique used to manage the SSD lifetime by 
handling sudden unanticipated bursts in requests. Although workload characteristics are 
part of the HybridPlan module of the system, it still imposes a challenge due its unpre-
dictability, which is related to the dynamic nature of the workload that is affected by several 
factors. The write regulator monitors the write rate of blocks and once an irregular burst 
takes place and action to re-balance is performed. It controls the writes sent to the flash 
by over-riding decisions made by the performance module by randomly choosing requests 
being sent to the SSD and directing them back to disk.

There are several important observations that can be drawn from their HybridPlan solver 
results for various workloads. First, some of the sequential read-intensive workloads can be 
satisfied with slow HDDs. This is contributed to low request arrival rate and hence low 
bandwidth. Second, similar to read-intensive case, some sequential write-intensive work-
loads can be satisfied with slower HDDs if it does not encounter a need for high IOPS. 
However, as the IOPS increases a need for an SLC class SSD becomes mandatory, according 
to HybridPlan, to satisfy the performance budget. Third, it is known that HDDs performs 
better when handling sequential requests, whereas SSDs are far more superior when handling 
random requests. That said, HybridPlan shows that there is a strong correlation between 
increased workload randomness and large number of SSDs to satisfy the performance con-
straints. Additionally, the random read-intensive workloads require about 3 times more of 
SSD storage to meet the IOPS. Thus, random workloads will be more costly compared to 
sequential ones because they will need larger number of SSD devices. On the other hand, 
the write-intensive workloads have to be considered carefully in the lifetime budget. The
HybridPlan results show that when the lifetime constraint is applied, the configuration uses SLC class SSDs over the MLC. Additionally, the costs for longer lifetime configuration are much higher. This is because the MLC, which are cheaper, may not be able to provide the required lifetime budget.

The HybridDyn simulation results show some promising results. For example, the performance prediction module is able to move requests that are subject to suffer from low performance on the SSD due to garbage collection overhead back to HDD in the Hybrid-Store which leads to better performance. Furthermore, the write regulation shows a good outcome by reducing the SSD block erase rate by 25%.

An important aspect of the HybridStore approach is that it tries to eliminate the need for system modifications in order to achieve the performance targets by correlating the workload characteristics to the available device feature configuration. This is desirable for widespread adoption. However, the overhead of obtaining very large history about the target workload, and performing such analysis might be considered expensive for today’s data center operations. This is especially important for the HybridDyn component.

### 2.6.4 Hystor

Hystor [16] is a high performance hybrid storage system that utilizes the SSD as an accelerator for both reads and writes. It integrates both low-cost HDDs and high-speed SSDs as a single block device. It isolates complicated details from other system components in order to avoids undesirable changes to existing OS kernels and applications. Hystor uses the SSD to provide two functions: a major function as a storage to retain the performance-critical blocks (read cache), and a minor function as a write-back buffer for writes. Hystor addresses three optimization goals: 1. efficiently identifying the most performance-critical blocks and fully exploiting the unique performance potential of the SSDs, 2. maintaining access history in an efficient way with low overhead that help to characterizing access history, 3. avoiding major system changes while effectively implementing the hybrid storage management policies.

As discussed in Section 2.2, integrating the SSD in the storage hierarchy is not a straightforward task. Thus, as the first step towards deciding where to place the SSD, is to understand the characteristics of the workload to be able to identify the critical blocks that have impact on the application performance, especially for random reads workloads. Hystor uses access frequency/request size as an indicator metric for a performance critical block, as it
expresses latency across multiple workloads. Several observations have led to this metric choice. First, since a large request can effectively amortize the seek and rotational latency over many blocks, the average latency per block is highly correlated to request size. Second, the request size reflects the workload access patterns. Finally, small requests tend to sustain high latency, since they are more likely to be intervened by other requests, which would cause high latencies from disk head seeks and rotations. In summary, access frequency and request size metrics describe both temporal locality and access pattern, respectively. On the other hand, the file system metadata is critical to system performance. In order to capture the semantically critical blocks and pre-allocate them into SSD and speed up file accesses, the block layer is modified to tag incoming requests for metadata.

The other optimization goal that Hystor addresses is maintaining data access history in an efficient way and with minimum memory requirements. For this purpose, Hystor utilizes a three-level tree data structure called block table [29]. The three levels are: Block Global Directory (BGD), Block Middle Directory (BMD) and Block Table Entry (BTE). Those three levels describe the storage space segmented in units of regions, sub-regions and blocks, respectively. Furthermore, each BGD or BMD entry has a 32-bit pointer field pointing to a BMD or BTE page in the next level, a 16-bit unique field tracking the number of BTE entries belonging to it, and a 16-bit counter field recording data access information. In addition, each BTE entry has a 16-bit counter field and a 16-bit flag field to record other properties of a block, e.g. it can determine if a block is metadata block.

Figure 2.4 shows the main architecture of Hystor. It works as a pseudo block device at the block layer of the system. The upper level components, such as file systems, view it as a single block device. As shown in the figure, Hystor consists of three major components, mainly remapper, monitor and data mover. In addition, a data structure, called mapping table, is used to track the original location of blocks on the SSD. The remapper maintains the mapping table. When a request arrives at the remapper, first it checks the mapping table. If the requested block is available in the SSD, the request is redirected to the SSD; otherwise it is serviced from the HDD. In addition, the remapper forwards I/O requests to the monitor, which collects I/O requests and updates the block table to profile the workload. Also, the remapper checks the incoming request tags to determine if the request is for a metadata block. In addition, the monitor periodically analyzes the data access history, identifies the block that should be remapped to the SSD sends a remapping plan to the data mover. The data mover, then, relocate data blocks across storage devices. It is important to note that
Hystor does not delete blocks from HDD once they are relocated to SSD.

In order to optimize the SSD usage, blocks are allocated in it in chunks. A chunk is basically group of blocks. By doing this, Hystor achieves two benefits. First, writes are organized more efficiently in a reasonably large request. Second, it avoids splitting requests into several small requests. This is similar in nature to RAID [39]. The reported results are based on 4KB chunk size. The choice of the chunk size is very critical. That is, a large chunk size is desirable for reducing memory overhead of the mapping table and the block table, while small chunk size is considerably better to improve the SSD space utilization because a large may contain both cold and hot data, i.e., performance-critical and non-performance-critical blocks.

On the other hand, Hystor uses the SSD as a write-back buffer to improve the performance of the write-intensive workloads. For this purpose, Hystor maintains a write-back area in the SSD. Blocks in the block write-back area are managed in two lists: a clean list and a dirty list. When a write request arrives, the SSDs blocks are allocated from the clean list. The new written dirty blocks are written into the SSD and added to the dirty list. To control the number of dirty blocks, a counter is maintained such that if the counter exceeds a configurable threshold, then a scrubber is called to write dirty blocks to the HDD until a low watermark is reached. That said, cleaned blocks are added back to the clean list.
Our work discussed in this thesis is inspired by the Hystor hybrid storage system and uses the same system design. The focus of our work is mainly on the monitor component of the system, where we propose our algorithm to improve the performance.
Chapter 3

Proposed Flash Storage Management Algorithm

In this chapter, we present our proposed algorithm to optimize hybrid storage systems. We start by describing the architecture of such systems and how our algorithm improves their performance. Then, we present the details of our algorithm.

3.1 Overview

It is crucial to introduce minimal changes to the storage system when introducing SSDs in it. Our system architecture is inspired by the work done in Hystor [16], but with some modifications to fit our proposed algorithm. Figure 3.1 illustrates the main architecture of the proposed system. The system can be implemented as a pseudo block device at the block layer. It is responsible for managing the block devices, together with hardware specific device drivers. Despite the fact that it contains several components, it is considered as one single block device. This allows users to perform regular storage tasks on the system, e.g., creating partitions and file systems. Furthermore, it can be implemented at the operating system level as a system daemon.

The proposed hybrid storage system can be used with different SSD types. We discussed in Section 2.2 that SSDs are classified into three types, i.e., SLC, MLC, and TLC, and our system does not depend on the specifications of any of those types. This makes the system more flexible and can adapt to whatever device type being used.
CHAPTER 3. PROPOSED FLASH STORAGE MANAGEMENT ALGORITHM

The system consists of three major components: dispatcher, data mover, and data server. The data server’s main function is to answer the coming I/O request, such that if the requested block exists in the SSDs, then the request is served from the SSDs. Otherwise, the block is serviced from the HDDs. The dispatcher keeps track of the incoming requests. At the designated time period, the dispatcher calculates the action of the current observed state of the storage system. Once the action is calculated, it sends the action plan to the data mover. The data mover, in turn, relocates data blocks across storage devices based on the action plan. In particular, it sends I/O commands to the block interface to update the storage devices and it updates the mapping table to reflect the recent changes. Furthermore, the data mover decides which chunks to swap data with to accommodate for the incoming new chunks in the SSDs. It is important to note that the data moved to SSDs are not deleted from HDDs.

The system integrates multiple HDDs and SSDs and exposes them to upper level as an array of logical blocks. Logical blocks are directly mapped to physical blocks in the HDD and logical block addressing scheme is used to index blocks. A mapping table is maintained in order to keep track of the logical blocks mapped to the SSD. The mapping table is maintained in a statically specified location and synchronously written in the non-volatile memory. For example, we can assign the first few MBs of the SSD to maintain this table. It is important to note that the data remapped to SSDs are not deleted from HDDs. This provides simplicity to the system implementation. For example, moving back blocks from SSDs to the HDDs results in high cost I/O operations. However, when blocks are duplicated we can drop the copies as long as they are clean. The mapping table can also be implemented in a B-tree data structure in the main memory. Gupta et al. [26] propose useful techniques, which maintain the most frequently accessed mapping entries, for the dynamic mapping table that can improve its implementation efficiency and reduce its in-memory size. SSDs are used to store hot data blocks identified by our algorithm. In addition, they can be used to remap file system metadata blocks, which can be identified through available semantic details in operating system kernel.

The key component of the system is the dispatcher. Unlike other components of the system, the dispatcher can be implemented in either kernel mode or user mode. In the latter case, it is implemented as a user-level daemon. The dispatcher monitors the coming I/O requests, which are temporarily maintained in a small buffer in the kernel memory. Those requests are periodically passed over to the dispatcher to calculate the next action
Figure 3.1: Main architecture.
plan for the system to take. The dispatcher can be integrated with the blktrace tool \cite{38} to benefit from using the existing infrastructure to record I/O traces. This is useful when running the algorithm in online mode, which we describe later.

![Diagram showing the flow of input into output](image)

**Figure 3.2:** FSM Flow from input into performance measurement output.

Our proposed flash storage management algorithm improves the performance of the dispatcher. The algorithm can be deployed in two modes: offline and online. In the offline mode the algorithm runs as described in Algorithm 1 using a pre-collected I/O traces. The I/O traces contain I/O requests made by the target application over a period of time. Each I/O request is identified by a timestamp, the starting block offset and the request size. We use this as an input to the algorithm to run it in offline mode. Then, the outputs of the algorithm, which are the state and action tables, are fed to the dispatcher component. The dispatcher uses these tables to periodically instruct the data mover with the appropriate action to update the mapping table and the SSD with candidate chunks based on the observed state of the storage system. Figure 3.2 illustrates the algorithm flow from the storage system input into performance measurement output that the dispatcher uses to control the storage system. In practice, this means that the algorithm takes the collected I/O traces and performs the computations as described in Algorithm 1. The dispatcher then uses the outputs of the algorithm to guide the storage system behavior.

In the online form, the dispatcher runs the algorithm in real-time, i.e., while the system is working, in which it takes the input state from incoming I/O requests. In this case, the dispatcher wakes up periodically to process the collected I/O requests and observe the current state of the storage system and then calculates the action to be performed by the
system. The dispatcher does not need to keep collected I/O requests after they are processed. That is, when the dispatcher wakes up and finishes processing the collected I/O requests of that period it can safely delete the stored I/O requests and reclaim the memory to use it for the next period.

### 3.2 Proposed Flash Storage Management (FSM) Algorithm

In this section, we present our proposed flash storage management (FSM) algorithm. The basis of our algorithm is a class of stochastic optimal problems called Markov decision processes. We introduced the basics of Markov decision process in Chapter 2. Thus, we start by defining the Markov decision process parameters in the context of our hybrid storage system problem. As we discussed in Section 2.5, a Markov decision process consists of four tuples: state space, actions, transition matrix, and cost function. In our problem, a state is determined by two parameters. First, the most frequent data chunk the storage system has accessed over the last time period \( t \). Second, the content of the SSD. Thus, a state \( S_i \) is defined with two vectors: 1. \( \bar{S}_i \) indicates which hot data chunk the system is currently at, and 2. \( \bar{\bar{S}}_i \) indicates the status of the SSD. Furthermore, our action space \( A \) is defined with two types of actions: 1. stored action, and 2. random action. An action, \( a \), is defined as a vector of binary codes, in which 1 means move the corresponding block to SSD, and 0 means do nothing. The transition matrix is used to describe the transitions from one hot data chunk to another in the storage system.

#### Table 3.1: List of symbols used in the thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>state space</td>
</tr>
<tr>
<td>( S_i )</td>
<td>system state at ( i ) time interval</td>
</tr>
<tr>
<td>( \bar{S}_i )</td>
<td>first vector of the system state</td>
</tr>
<tr>
<td>( \bar{\bar{S}}_i )</td>
<td>second vector of the system, i.e., SSD state</td>
</tr>
<tr>
<td>( P_{ij} )</td>
<td>state transition probability</td>
</tr>
<tr>
<td>( T )</td>
<td>transition matrix</td>
</tr>
<tr>
<td>( A )</td>
<td>action space</td>
</tr>
<tr>
<td>( \aleph )</td>
<td>discount factor</td>
</tr>
<tr>
<td>( r )</td>
<td>random number</td>
</tr>
<tr>
<td>( A(S_i) )</td>
<td>action list applicable to state ( S_i )</td>
</tr>
<tr>
<td>( C_{S_i}(a) )</td>
<td>cost of action ( a ) when applied to state ( S_i )</td>
</tr>
</tbody>
</table>
Algorithm 1 shows the pseudocode of the FSM algorithm. The algorithm can be run in two modes: offline and online. In the offline mode, we offload the computation from the storage system. That is, we run the computation outside the system, and then we provide the dispatcher engine with the required input, i.e., the state and action tables. For this mode to work, we use a simulation technique to direct the algorithm to which state to go to instead of evaluating all states in every iteration. The algorithm requires pre-collected I/O trace of the target application. As we discussed in Section 3.1, the trace file contains I/O requests that were made by the target application over a period of time. Each I/O request is identified by a timestamp, the starting block offset and the request size. The I/O trace file is used to extract two data items. First, we determine the hot data chunks accessed by the application in each period of time. Second, we compute the transition matrix $T$, such that $P_{ij}$ is the fraction of time that chunk $j$ is accessed immediately after chunk $i$. For better performance and memory optimization, the transition matrix is maintained in a 2-D sparse matrix data structure.

The algorithm maintains three main data structures: state table, action table, and cost table. Each element of the state table represents a two-vector state: $\bar{S}_i$ and $\bar{\bar{S}}_i$. The first vector, $\bar{S}_i$, indicates which hot data chunk the system is currently at. That is, periodically the algorithm observes what the current state is, and sets the position of that state in vector to 1, and the rest are set to 0. The second vector, $\bar{\bar{S}}_i$, represents the status of the SSD. That is, it indicates what data chunks are currently present in the SSD. Both vectors are initialized to some starting state of our choice. For example, we can initialize $\bar{S}_i$ to a random chunk and $\bar{\bar{S}}_i$ with the first five chunks. We further discuss the importance of this initialization in more details in Section 3.3. The state at position $k$ in the state table has a corresponding action at position $k$ in the action table. The cost of that action is stored in the cost table at same position, $k$. In our algorithm, the action space list is defined with two types of actions:

- **Stored action**: is the best-known action for a given state. We retrieve this from the action table. That is, the stored action for state $S_i$, where $i$ is the position of that state in the state table is the action at position $i$ in the action table. A null value means that this is a new state, and has never been evaluated before.

- **Random action**: is a random action created on demand.

Once the initialization is done, the algorithm starts computing the Bellman optimality
Algorithm 1 Flash Storage Management (FSM) Algorithm

INPUT: I/O trace
OUTPUT: State Table and Action Table

1: compute transition matrix $T$ using input I/O trace
2: initialize STATE_TABLE, VALUE_TABLE, ACTION_TABLE
3: initialize $S_0$ and $\bar{S}_0$ to a starting state

4: while true do
5:     immediate_cost = $\bar{S}_0 \ast \bar{S}_0$ $\triangleright$ scalar product of two vectors forming the current state of the system
6:     for each action in the action list do
7:         cost = min[immediate_reward + $\alpha \sum C(S_i) \ast P_{ij}$]
8:     end for
9:     update the corresponding action for current state with the action with minimum cost;
10:    apply the selected action to the SSD vector $\bar{S}_i$;
11:    if cost function converges to threshold then
12:        break;
13:    else
14:        next_state_prob = $T \ast S_i$ $\triangleright$ compute next state probability vector
15:        cumVector = cumulative probability of next_state_prob vector
16:        $\bar{S}_j = \text{SIMULATENS}(cumVector)$ $\triangleright$ call SimulateNS() function to simulate next state
17:        continue;
18:    end if
19: end while

20: function SimulateNS(cumVector)
21:    initialize $r$ a random number between 0 and 1
22:    initialize $\bar{S}_j$ next state vector
23:    initialize $k$ index pointer to iterate through cumVector
24:    while $r < \text{cumVector}(k)$ do
25:        $k = k + 1$
26:        continue;
27:    end while
28:    $\bar{S}_j(k) = 1$
29:    return $\bar{S}_j$
30: end function

31: output state and action tables
equation, or the cost function:

\[
Cost = \min_{a \in \text{ActionList}} \left[ \text{immediate\_cost} + \alpha \sum C(S_i) * P_{ij} \right]
\] (3.1)

The cost function consists of two parts. The first part is the immediate cost, which is the cost received of the transition from current state to the next state. It is computed by the scalar product of the two vectors \( \bar{S}_i \) and \( \bar{S}_j \). It is important to note that the immediate cost is common no matter how many actions we have because it is not influenced by the action list. The second part is the discounted cost, which is the sum of all costs of all states if the action \( a \) was to be applied. To compute this, we apply the action to the SSD state vector, \( \bar{S}_i \), and retrieve the cost of each state from the cost table. Once the total cost is computed for each action, we compare them and select the action with the minimum cost. This action will be the best-known action for that state at this stage of computation. Thus, we store that action in the action table at the position of the current state, and apply the action on the SSDs. Before we proceed to the next state and compute its action and cost, we check for the cost function convergence to see if we reached the convergence threshold that we defined earlier. We compute the function convergence by calculating the absolute difference in the cost table after many state evaluations, e.g., every 1000 state evaluations we perform this. If the function converges to the threshold, then we break from the algorithm and write the final output, i.e., the state and action tables. Otherwise, we guide the algorithm to which data chunk, i.e., state, it should go to by simulating the next state and starting to perform the same computations for the next state. The next state simulation is computed by first calculating the state probability vector. This vector is the result of the multiplication of the transition matrix and the first vector \( \bar{S}_i \). Then, we compute the cumulative probability vector of that state probability vector. In order to determine the next \( \bar{S}_i \), we generate a random number \( r \) between 0 and 1, inclusive, and set the position of the chunk in \( \bar{S}_i \) at the position where the value of the random number is greater than or equal to the cumulative probability value in the cumulative probability vector.

The online mode, which is an adaptive form of the offline mode, of the algorithm performs the same steps. However, since the algorithm runs in online mode, which means that it performs the computations while the storage system is running, we do not perform any simulation. Instead, the algorithm takes its next state from the actual observed state of the system. In other words, we refine the results of both the state table and action table using the actual state of the storage system instead of the input traces. In addition, the online
Figure 3.3: Online flash storage management algorithm flow chart.
mode can also use a trace file for initialization and starts normally as if it runs offline, but it adapts the tables based on the new I/O requests. Figure 3.3 illustrates the flow chart of the online mode.

Figure 3.4 illustrates the flow chart of our proposed algorithm running in the offline mode. We start by constructing the required data from the input I/O traces. This includes the hot chunks accessed by the target application and the transition matrix. Then, we compute the Bellman optimality equation. In each state evaluation, we compare the cost of both the stored and random actions. Based on the comparison result, we update the tables with the minimum resulted action. After that, we verify the convergence of the cost function. If it hits the convergence threshold, we terminate the algorithm and write the final output tables. Otherwise, we simulate the next state, and continue the computation.

3.3 Analysis and Discussion

Dynamic programming algorithms are known traditionally to be computationally intensive. This is because their exhaustive search nature in the state space, and expansion of all possible states. However, they are very effective in a way that heuristic algorithms are not, because they systematically update the evaluations of the states. Further, their computational steps can be arranged, or guided, in a way that leads to incorporating the algorithm in a real-time mode, i.e., running the algorithm while the system is running. In this case, convergence to an optimal evaluation function requires visiting states infinitely often, but performance improves incrementally while this being accomplished.

In our proposed system model, the real-time dynamic programming algorithm is the core of the hybrid storage, since it controls what data should be moved to SSDs. While the system is running, the dispatcher component monitors the incoming I/O request over periods of time, e.g., for every 1 minute. The choice of the time period will depend on the choice of the chunk size to make sure that is enough for the system to move the data to SSD and make use of them. At the end of each time period, the algorithm, which is running from the dispatcher, computes the current state of the system by finding the most frequently accessed chunk over that period and observing the current SSD state. The algorithm uses this information to guide the computation. After that, the algorithm evaluates this state using the cost function in Equation (3.1). Once the result is stored in the action table, they are also applied into the system as the dispatcher sends the action plan to the data mover.
CHAPTER 3. PROPOSED FLASH STORAGE MANAGEMENT ALGORITHM

Figure 3.4: Offline flash storage management algorithm flow chart.
in order to move data between storage devices. The algorithm continues running as long as the storage system is running. Thus, we conclude that the on-line flash storage management algorithm and the dispatcher interacts as follows: 1. actions are based on the most up-to-date information from the dynamic programming computation, and 2. the state sequence generated during action execution influences the selection of states to which the dynamic programming backup operation is applied. It is important to note that the uncertainty in the system behavior implies that a closed-loop policy produces better performance for the system. By closed-loop we mean that each action depends on current observations of the storage system. That is, the action is a function of the current state of the storage system. Thus, we do not need to store any information of the past system’s behavior. This yields a computationally efficient method for calculating each action as a function of the observed system state.

3.3.1 Optimality

As we discussed in Section 2.5, a Markovian decision process is defined in terms of a discrete-time stochastic dynamic system with a finite state set. Time is represented by a sequence of time steps. At each time step, the FSM dispatcher observes the storage system’s current state and computes an action to be applied on the system, which results in the system to move to another state. That is, if $S_i$ is the observed state, then the action is selected from a finite set of possible actions, say $A(S_i)$. When action $a \in A(S_i)$, the system’s state at the next time step will be $S_j$ with state-transition probability $P_{ij}(a)$, and the application of action $a$ in state $S_i$ incurs an immediate cost $C_{S_i}(a)$. Furthermore, For any action $a$, and state $S_i$, we define $f^a(S_i)$ to be the expected value of the discounted cost that will accumulate over time given that the system uses action $a$ and $S_i$ is the initial state:

$$f^a(S_i) = E_a[\sum_{t=0}^{\infty} \alpha^t C_t | S_0 = S_i]$$  \hspace{1cm} (3.2)

This is the expected discounted sum of all the immediate costs that will be incurred over the future starting from state $S_i$, where $\alpha$, $0 \leq \alpha \leq 1$, is a factor used to discount future immediate costs, and $E_a$ is the expectation assuming the storage system always uses action $a$. The discount factor $\alpha$ determines how expected future costs influence current actions. When $\alpha = 0$, the cost of any state is just the immediate cost of the transition from that state. As $\alpha$ increases towards one, future costs become more significant in determining
optimal actions.

An important aspect of our algorithm is the relationship between actions and evaluation functions, or the cost function. The cost function \( f^a \) corresponding to action \( a \) gives the cost for each state, assuming that the FSM always uses action \( a \). However, \( a \) does not necessarily lead to the best successor states as evaluated by \( f^a \). In other words, \( a \) is not necessarily a greedy action with respect to its own evaluation function \( f^a \). We use Watkins Q notation \([46]\) to define a greedy policy in this stochastic scenario. For each state \( S_i \) and action \( a \in A(S_i) \), let

\[
Q_f(S_i, a) = C_{S_i}(a) + \alpha \sum_{S_j \in S} P_{ij} f(S_j) \tag{3.3}
\]

\( Q_f(S_i, a) \) is the cost of action \( a \) in state \( S_i \) as evaluated by \( f \). It is the sum of the immediate cost and the discounted expected value of the costs of the possible successor states under action \( a \). Because of the stochastic nature of our storage system, each state has a probability to be visited. Thus, if \( f \) is the evaluation function for some action, then \( Q_f(S_i, a) \) gives the cost of generating action \( a \) in state \( S_i \) and thereafter following this policy. That is, an action \( a \) is greedy with respect to \( f \) if for all states \( S, a(S) \) is an action satisfying:

\[
Q_f(S, a(S)) = \min_{a \in A(S)} Q_f(S_i, a) \tag{3.4}
\]

We should note that there can be more than one greedy action with respect to \( f \) if more than one action minimizes the set of \( Q - values \) for some state. Further, a key aspect underlying dynamic programming methods is that the only actions that are greedy with respect to their own evaluation functions are optimal actions. That is, if \( a^* \) is any optimal action, then its evaluation function is the optimal evaluation function \( f^* \), and \( a^* = a^{f^*} \). In addition, any action that is greedy with respect to \( f^* \) is an optimal policy. Therefore, if \( f^* \) is known, it is possible to define an optimal action simply by defining it to be greedy with respect to \( f^* \). This leads to an important fact that \( f^* \) summarizes all information about that action, and hence saves a lot of unnecessary repeated computations every time. Another key fact is that a necessary and sufficient condition for \( f^* \) to be the optimal evaluation function is that for each state \( S_i \) it must be true that
which is our cost function used in the flash storage management algorithm. This is one form of the Bellman optimality equation which can be solved for each \( f^*(i), S_i \in S \), by a dynamic programming algorithm.

There are several techniques of dynamic programming algorithms to solve this decision problem. We use the asynchronous value iteration approach of dynamic programming. Value iteration is a successive approximation procedure that converges to the optimal evaluation function, \( f^* \). It is a successive approximation method for solving the Bellman optimality equation whose basic operation is backing up estimates of the optimal state costs.

Because of the nature of our hybrid storage system problem, we apply the asynchronous value iteration. The key advantage of asynchronous dynamic programming is that we do not back up state costs in any systematically organized fashion. This brings a great merit to utilize today’s multi-core processors, with communication time delays and without a common clock, to provide efficient computational power. It has obvious utility in speeding up dynamic programming and thus brings a practical significance. Each core can be responsible for a number of states. Further, the time at which each processor backs up the cost of its state can be different for each processor. That said, to back up the cost of its state, each processor uses the costs for other states that are available to it when it wakes up to perform a backup. The adoption of the asynchronous model is appropriate for our FSM system interaction with the dynamic programming algorithm due to the flexibility with which its stages can be defined. As a result, the dispatcher automatically uses intermediate results of the dynamic programming computation to guide its behavior. The asynchronous value iteration method has been thoroughly studied in [11][12].

### 3.3.2 Time Complexity

Asynchronous dynamic programming converges to the optimal cost function \( f^* \) provided that the cost of each state is backed up infinitely often [40]. In practice, this means that the strategy for selecting states for cost backups should never eliminate any state from possible selection in the future. When running the algorithm online, this can be assured from the fact
that our Markovian process yields nonzero probability of visiting any state no matter what actions are executed. On the other hand, if we opt to run the algorithm in offline mode, we can use a method called trial-based dynamic programming [9, 40]. In this scenario, a trial consists of a time interval of nonzero finite duration during which the dynamic programming is performed. After this interval, the system is set to a new starting state, and a new trial begins.

In addition, asynchronous dynamic programming converges if the following conditions are met [9, 40, 11]: 1. the initial cost of every state is zero, 2. there is at least one proper action, and 3. all actions that are not proper incur infinite cost for at least one state. The last condition ensures that every optimal action is proper, i.e., it rules out the probability that a least-cost action exists that never reaches the goal state. An action is proper if its use implies a nonzero probability of eventually reaching the optimal states starting from any state. Furthermore, it suffices to find an action that is optimal when restricted to relevant states because the other, i.e., irrelevant states, will never occur during the use of that optimal action. In practice, the relevant state means the most frequently accessed chunks of data. By applying the dynamic programming to just these states, we can save considerable amount of computation. These conditions are valid for both the offline and on-line version of the dynamic programming algorithm.

In conclusion, if there are \( n \) states and \( m \) is the largest number of admissible actions for any state, then each iteration, which consists of backing up the cost of each state exactly once requires at most \( O(nm) \) computational steps. This can further be optimized. For example, if we know which of the state-transition probabilities from state \( S_i \) are zero, then the amount of computation can be much less. Thus, using Equation (3.4), an optimal action can be determined and that requires \( m - 1 \) comparisons, which results in a linear time complexity.

### 3.3.3 Space Complexity

In our implementation, we explicitly store all states, actions, and their respected costs. This introduces a concern of practical importance in implementing such dynamic programming algorithm, which is how the data and cost function are represented and stored. Applying the classic version of the dynamic programming, which involves continuous states and/or actions, leads to space complexity exponential in the number of states. This phenomenon is called by Bellman [10] as a curse of dimensionality. However, in our particular problem
we need an efficient methodology to optimize the space requirement in a way that allows us to run it in online mode. There are several methods that can be used for making the tables representations more efficient. In our FSM system, we are interested in only a set of relative states and their corresponding actions and costs. These states are the most frequently accessed data chunks by the storage system. Hash table methods permit efficient storage and retrieval in our problem. Another method is to use $kd$–tree data structure. In our implementation, we use hash table since it provides efficient and easy way that preserves the integrity of the stored costs. Furthermore, costs of states rarely visited after the early stages can be over-written by costs of states on which the algorithm focuses in later stages. This makes the tables linear in size.

3.4 Applications of Flash Storage Management Algorithm

There is a diverse range of applications where hybrid storage systems can use the proposed FSM algorithm. In this section, we describe few cases of such potential systems to benefit from FSM. The fundamental aspect which determines whether the system will benefit from the FSM or not is the type of I/O workloads it is handling. We discussed in Chapter 2 about the characteristics of the different I/O workloads. There are two distinct I/O workload types: sequential and random. The sequential data is stored one block after the other, and accessed on that way. Sequential I/O workloads will not benefit from any caching algorithm, because they can generally be satisfied with the conventional HDD drive. Random data, however, are scattered in random blocks in the disks, which creates a major overhead to access such data from HDDs. In turn, it will impose a performance degradation in the system when handling such random I/O workloads. As discussed in Chapter 2, unlike HDDs, SSDs are characterized with a near-zero seek time feature that we use to improve the performance issues of the random I/O workloads. In our FSM design we focus in the random I/O workloads, because they are affected by the performance limitations of the HDDs. Also, they can benefit from SSDs through an efficient caching technique in a hybrid storage systems.

3.4.1 Database Management Systems

Database management systems are crucial in data centers. A key function of these systems is to support the storage of very large amounts of data, many terabytes or more, over a
long period of time, allowing efficient access to the data queries and database modifications. The data of a database resides in the secondary storage devices, i.e., HDDs. However, since data retrieval from HDDs takes much longer to complete than data retrieval from cache or main memory, that data must be in main memory. The storage manager component of the database system is responsible of managing the placement of data on disk and its movement between disk and main memory. The storage manager keeps track of the location of files on the HDDs and obtains the block or blocks containing a file on request from the buffer manager. The buffer manager, on the other hand, is responsible for partitioning the available main memory into buffers, which are page-sized regions into which disk blocks can be transferred.

Figure 3.5 shows a block diagram of a database management system component. The majority of interactions with the database system follow the path on the left side. A user or an application program initiates some command, using the data-manipulation language. This command affects the content of the database or extracts data from the database. The command is handled by two separate subsystems. First, the query is parsed and optimized by a query compiler. The resulting query plan is, then, passed to the execution engine. Next, the execution engine issues a sequence of requests for small pieces of data, to a resource manager that knows about data files and index files. The requests for data are passed to the buffer manager. The buffer manager’s task is to bring appropriate portions of the data from secondary storage where it is kept permanently to the main memory buffers. For better performance optimizations, database management systems usually use blocks as a unit of transfer between buffers and disks. To perform this task, the buffer manager communicates with a storage manager to get data from disks. The storage manager keeps track of the location of files on the disk. Further, it obtains the blocks to be transferred to the buffers based on the request from the buffer manager.

Database performance is I/O-dependent. That is, the faster the database management system can complete an I/O operation, the faster the database application will run. In practice, this means that it depends on how fast the storage manager can transfer the data between the HDDs and the main memory. However, random reads to HDDs are slow and expensive to accommodate for a growing demand for data retrieval [32]. Petabytes databases and data warehouses are becoming more common these days, and it is crucial to provide an efficient storage solution to mitigate the I/O bottleneck caused by the HDDs.

Hybrid storage for database systems is becoming popular in recent years. It offers a
Figure 3.5: Database management system components.
practical solution for this problem. The hybrid storage utilizes the SSDs to host the most frequently read blocks. In practice, the FSM algorithm can be integrated in the storage manager. This brings a performance advantage for the storage manager to speed up hot data movement from storage, i.e., SSDs, to the main memory. This can be achieved in two ways. First, we can modify the storage manager to send the performance critical data chunks from the SSD to the main memory, which will improve the performance instead from requesting the data from HDD. Second, we can minimize the required main memory storage, and serve the user queries directly from SSDs.

3.4.2 Hybrid SAN for Virtualized Environment

Many enterprises have turned to virtualization technologies for their desktops and servers in their data centers. Although virtualization is not new to the computing industry, it is gaining a renewed attention in recent years motivated by the recent technological advances in multi-core and virtualization-enabled processors and advanced memory access mechanism. Virtualization offers several benefits. A key advantage is that it improves the hardware utilization by consolidating many underutilized resources into a fewer consolidated resources. Thus, with proper consolidation, virtualization can lead to considerable cost savings and hence lower total cost of ownership at the data center. In addition, it brings faster provisioning and deployment mechanisms to the environment. Seeing the practical benefits virtualization delivers in those environments, enterprises are applying this technology to their servers as well as desktop computers.

Despite the fact that consolidation tends to offer various benefits, it introduces several challenges and concerns affecting the performance of the computing environment. A common problem is known as I/O blender effect [27]. The I/O blender effect is a phenomenon in virtualized environments that degrades storage performance. It occurs when multiple virtual machines all send their I/O streams to a hypervisor for processing. Under heavy workloads, I/O processes which might otherwise have been relatively sequential become random. Disk seek times and rotational latencies start to comprise a much higher percentage of the I/O transfer times, significantly slowing down the I/O. The effect of putting multiple virtual machines together randomizes the aggregate I/O stream going to the storage system. This randomization can occur with all virtualization platforms, i.e., virtual server and virtual desktop. The I/O blender effect can be even worse in virtual server environments because they are extremely consolidated with tens or hundreds of users often share a single storage
I/O pipe, dividing up the random I/O quotient.

At an enterprise level computing, storage controllers are responsible for coordinating read from disk arrays and write operations to the arrays. Depending on the disk configuration and array configuration, scheduling and performing these operations can have a significant impact on performance especially if large amounts of requests from disparate systems are realized. Furthermore, if we consider the fact that traditional storage arrays are typically designed for handling sequential reads and writes. This design decision made sense before having virtualization in place, because it could be expected that I/O requests were for larger contiguous amounts of data. This means that the array could handle multiple requests at the same time. With virtualization, however, each virtual server no longer writes directly to disk. It writes to the hypervisor, which then writes to the disk, giving multiple and complex I/O streams and impacting performance.

The I/O blender effect is an inevitable occurrence with virtualization. In order to deliver an acceptable application performance in such environments, we need a storage system that provide high IOPS in a cost-effective manner. Hybrid storage system brings a feasible solution to this problem without compromising the cost of the virtualized environment. The FSM can be used to improve the performance in such environments, where it put the active and more performance critical virtual servers into SSDs.
Chapter 4

Evaluation

4.1 Experimental Setup

In this section, we explain our simulation setup that we use to conduct our experiments. We simulated a storage system with essential functionalities. We implemented our simulator in Java. The fundamental data structure that we use in our implementation is the sparse matrix, which enables us to optimize the space requirements. For that purpose, we use the Colt set of open source libraries for high performance and scientific computations [22]. We use both one-dimensional sparse matrices as well as two-dimensional sparse matrices. Based on our proposed FSM algorithm presented in Section 3.2, we implemented two simulators to test both versions of the algorithm, i.e., the online and the offline. We configured both simulators to use chunk size of 50 MB. The choice of the chunk size plays an important role in our FSM algorithm, and it is influenced by several observations. First, we ensure that I/O requests are completely fulfilled. It is a common practice in data storage system to configure the chunk size in RAIDs to be large for applications with small individual I/O requests. Thus, it ensures that any I/O request is fulfilled within that chunk. Second, larger chunk sizes along with chunk access frequency are excellent metrics to represent both temporal and spatial locality as indicated in [16]. This, in turn, helps to minimize the amount of data swapping, as we show in Section 4.2. Third, the chunk size will impact the amount of memory needed to store the state, action and value tables. For example, instead of manipulating millions of individual small blocks of data, we can optimize that by manipulating few thousands, or less depending on the chunk size, of larger chunks, which makes a difference.
We have two separate simulators. First, the offline simulator, which accepts the I/O trace log file as an input and executes the FSM algorithm, as explained in Section 3.2. Once it finishes, it will output the state and action tables, which we give to the storage system Dispatcher to control the system behavior. Second, the online simulator takes the I/O traces as an input for initialization and executes the FSM algorithm online, while the storage system is running. In this case, both the state and action tables are updated instantly as a new state is evaluated.

In addition, we use the Postmark benchmark [30]. This benchmark, which is developed by Network Appliance Inc. (NetApp), is used to test small-file I/O on file systems similar to tasks endured by web and mail servers. We use this benchmark to generate several types of I/Os, namely: read-intensive, write-intensive, and a random composite of both reads and writes. We perform our experiments using those three types of I/O types.

First, we configure the benchmark with the required parameters, such as, number of files, files size, number of transactions, and the desired I/O behavior whether a read-intensive, write-intensive, or a composite of both. In our experiments, we use file size range between 4 KB and 70 KB, with total dataset size of approximately 104 GB. Next, we start running the benchmark based on that configuration. In order to collect the I/O traces, we use the Linux tool called blktrace [38]. This tool is used to collect important information about an application I/O that can be used later for the analysis, for example, I/O block size and address, and I/O type. We run this tool concurrently with the benchmark, with the preferred options.

As soon as the benchmark finishes, we stop the blktrace tool to flush the collected I/O traces into the output file. Next, we use the output I/O trace file as an input to our simulator. For each of the data points of the results below, we run the experiment three times, and we take the average of the results.

4.2 Experimental Results

We use four performance and accuracy measurements. First, the cost function convergence. Second, the hit ratio, which measures how much of the total I/O requests were successfully captured in SSD. Third, is the data swapping, which illustrates how much data swapping is performed in the device.
4.2.1 Cost Function Convergence

An important test for the stochastic dynamic programming algorithms is the Bellman Equation convergence test. A common issue with all stochastic algorithms is that there is no known reliable and implementable stopping rule [9, 40]. Thus, function convergence provides an important theoretical correctness property about the algorithm [40]. The first experiment we perform to judge our algorithm is to verify the cost function convergence. Figure 4.1 shows the cost function convergence test result of our proposed Flash Storage Management (FSM) algorithm in offline mode using the read-intensive workload. We measure the function convergence by calculating the absolute cumulative difference of each value in the value table after a number of iterations, or state evaluations. Each point in the figure corresponds the absolute difference between the current value and the one at previous point. As the figure

![Figure 4.1: Cost Function Convergence for the Flash Storage Management (FSM) Algorithm using read-intensive workload](image)
shows, the cost function convergences towards 0, which is what we are looking for. In other words, the cost function has reached the optimal value. As we explained in Chapter 3, we can restart the algorithm to the initial in order to discover new possible states. Thus, we keep the algorithm running after it converges the first time to check if it can discover new states. Then, we restart the algorithm to the initial state to discover new states. We can see that it converges faster at later runs than the initial run, which enables us to restart the algorithm to initial state several times before terminating the computations. This explains why we have the oscillation starts after the function converges at $3 \times 10^4$ iteration. The composite workload function in a similar manner and converges in a similar manner, but needed more iterations to converge.

In contrast, the cost function does not converge when applying the algorithm on different types of workloads. Figure 4.2 illustrates the cost function convergence results for the FSM algorithm using write-intensive workload. In this case, the cost function neither converges nor it shows any sign of possible convergence towards any specific value, which means that the FSM algorithm is not able to reach an optimal value for this workload.

In the next section, we will see the practical implications of these results, and how they
impact the performance of our proposed hybrid storage system.

### 4.2.2 Hit Ratio

A read request to blocks resident in the SSD is considered a hit. Thus, the hit ratio is the ratio of the read requests serviced from the SSD to the total read requests. Figures 4.3, 4.4 and 4.5 show the hit ratio results for both the online and offline FSM algorithm for the three different workloads.

![Hit Ratio Graph](image)

**Figure 4.3:** FSM hit ratio for read-intensive workload: Online vs. Offline

We consider the hit ratio as the most important performance metric for such caching algorithm, as it shows how good and efficient the actions applied by the algorithm. Also, we can see in this section how the algorithm performs on different application workloads, which helps determine where best we can fit our algorithm.

As explained in Chapter 3, the FSM algorithm requires I/O logs about the target application as an input. We call each entry in the I/O log as a data point. That said, the more data points we can provide to the algorithm, the more precise it can determine the application I/O pattern. Thus, we can collect more data points from read-intensive workloads than the write-intensive.
Figure 4.4: FSM hit ratio for write-intensive: Online vs. Offline

Figure 4.5: FSM hit ratio for composite workload: Online vs. Offline
CHAPTER 4. EVALUATION

Figure 4.3 compares the hit ratio performance between the online and offline FSM algorithm for the read-intensive workload. At SSD size of 5% of the workload, both are very close to each other and resulted in less than 15% hit ratio. The performance increases for both modes as we increase the SSD size, to hold more data. We can see that at SSD size of 30% of the workload, the online FSM achieves more than 70% hit ratio, while the offline goes up to almost 60% hit ratio at similar SSD size. An important factor for this performance increase in parallel with the SSD size increase is the quality of the I/O log, since read-intensive application by nature will provide such data for us.

On the other hand, for the composite workload FSM algorithm achieved less hit ratio compared to the read-intensive. Figure 4.5 shows the results of this experiment. For example, the online FSM maintained a hit ratio of more than 60% when using SSD size of 30% of the workload. In addition, the offline recorded almost 50% hit ration using the same SSD size.

In contrast with the previous two cases, we observe a clear performance degradation for the write-intensive workload. Figure 4.4 shows the hit ratio of both the online and offline FSM algorithms. As we can see, the online mode stands at 2% hit ratio when using SSD size of 5% of the workload. The offline mode, on the other hand, did not do any better and it has 0% hit ratio using the same SSD size. Even when we increased the SSD size to 30% of the workload, both hit ratio performances did not exceed 15%. The key observation behind this poor performance is the fact that we were not able to collect enough data points to feed into the FSM algorithm in order to perform the required computations. Hence, the FSM algorithm is not able to produce quality actions to move hot data chunks to SSD, and thus impacting the performance.

4.2.3 Data Swapping

The data swapping metric represents the ratio of the number chunks being replaced or swapped from SSD divided by the total number of chunks moved into the SSD. This metric is an indication of how efficient the algorithm is in managing the allocated SSD space as we are targeting less write operations into the SSD. This is required in order to maintain longer lifetime of the SSD. That said, the lower the rate is the better. Furthermore, it impacts how the system performs. That is, reducing the data swapping should reduce the amount of bandwidth required to transfer the data between both devices, i.e. HDD and SSD, which in turns reduces system contention for bandwidth with the actual running application. There
are several factors that affect the amount of data swapping, mainly the SSD size with respect to the workload size, the locality of the data as well as the choice of the chunk size.

![Figure 4.6: FSM data swapping for read-intensive workload: Online vs. Offline](image)

We present our FSM algorithm data swapping performance results using three different workloads, and for both the online and offline. Figures 4.6, 4.8 and 4.7 show the results. Figure 4.6 shows the data swapping performance results for the read-intensive workload. As we can see, the data swapping rate decreases as we increase the SSD size. When the SSD size is 5% of the total workload size, which is approximately 104 GB, we can see that both the online and offline FSM did more than 60% of data swapping. The main reason for this is the SSD size with respect to the workload size, which forces FSM to move chunks around frequently and resulting on hot data chunks not to stay on the SSD for longer periods to be accessed by the application on the next read I/O request. Thus, it impacts the hit ratio of the algorithm as we can see from Figure 4.3, where the algorithm barely achieved around 15% hit ratio at 5% SSD size. As the SSD size increases, it offers more room to keep some hot chunks in SSD and that save the system extra unneeded delete and write operations to SSD, as well as minimizes the I/O contention between various components of the system. For example, at SSD size 30% of the workload, the data swapping for the online FSM is 16%, while the offline FSM is 22%.

The composite workload data swapping results are shown in Figure 4.8. In this case,
Figure 4.7: FSM data swapping for write-intensive workload: Online vs. Offline

Figure 4.8: FSM data swapping for composite workload: Online vs. Offline
the data swapping rate is a little higher compared to the read-intensive, which is caused by the fact that this workload does not enjoy the same level of temporal locality as the read-intensive.

Finally, the data swapping results for write-intensive workload are shown in Figure 4.7. As we can see, FSM performed the worse among the different workloads. We attribute this behavior to the poor level of temporal locality of this workload as the main reason. In addition, write-intensive workloads does not provide excellent input for the FSM algorithm to use it to perform the computations, which contributes to the data swapping performance.

4.2.4 Comparison with Hystor

We compare the FSM algorithm against the Hystor algorithm [16].

Hystor hybrid storage system is one of the major hybrid storage systems that uses SSD drives for high performance storage. Hystor implements a special data structure called the Block Table to achieve two objectives: a) to represent the metric values in compact and efficient way, and b) to maintain data access history for a large-scale storage. This block table is divided into three levels, Block Global Directory, Block Middle Directory, and Block Table Entry, where each level describes the storage space divided into units of regions, sub-regions, and blocks. Further, each level consists of multiple 4KB pages, and each of which consists of multiple entries. In order to use this structure in practice, the logical block number (LBN) is broken into three components, each of which is an index to an entry in the page at the corresponding level. In addition, each entry in the Block Global Directory and Block Middle Directory has a 32-bit pointer field to a Block Middle Directory and Block Table Entry page in the next level; a 16-bit counter field recording data access information, and a 16-bit unique field tracking the number of Block Table Entry entries belonging to it. Figure 4.9 illustrates the block table structure.

Hystor algorithm is based on a technique called inverse bitmap, which is used to encode the performance metric and it consists of two parameters: request size and frequency. This technique is represented by the following equation:

$$b = 2^{\max(0, 7 - \lfloor \log_2 N \rfloor)}$$  \hspace{1cm} (4.1)

When a block is accessed by a request of $N$ sectors, an inverse bitmap, $b$, is calculated using Equation (4.1). Each entry at each level of the block table maintains a counter.
This counter in the Block Global Directory, Block Middle Directory and Block Table Entry represents the hotness of the region, sub-region and block, respectively. For example, when receiving an I/O request, the logical block number is used to traverse the block table through the three levels, and at each level the counter of the corresponding entry is incremented by $b$. Thus, the more frequently a block is accessed, the more often the corresponding counter is incremented. As a result, a block with a larger counter value is regarded as a hot block to be relocated to the SSD.

We implemented a Hystor simulator based on this algorithm. We used sparse matrices in order to represent the Block Table efficiently. We did not consider the write-back buffer, because this is a minor role for the SSD. Since Hystor is designed to run always in real-time as the target application is running, we compare its results with our online FSM algorithm.

Figures 4.10, 4.11 and 4.12 show the hit ratio performance results of Hystor compared to our online FSM algorithm for the different workloads. Both systems are competitive when using SSD size of 5% of the workload, especially in the composite workload where they both achieved the same performance results. As the SSD size increases, we start to notice a performance gap between the two. For example, at SSD size of 30% of the workload, FSM outperforms Hystor in both the read-intensive and composite workloads by more than 20% in both cases, while Hystor excels in the write-intensive workload by more than 15%. Hystor is not able to maintain a good performance for read-intensive workloads.
Figure 4.10: Hit ratio comparison for a read-intensive workload: Hystor vs. Online FSM

Figure 4.11: Hit ratio comparison for a write-intensive workload: Hystor vs. Online FSM
**Figure 4.12:** Hit ratio comparison for a composite workload: Hystor vs. Online FSM

**Figure 4.13:** Data swapping for read-intensive workload: Hystor vs. Online FSM
Figure 4.14: Data swapping for write-intensive workload: Hystor vs. Online FSM

Figure 4.15: Data swapping for a composite workload: Hystor vs. Online FSM
due to its hierarchical model imposed by the design of block table as it maintains relatively small chunk sizes compared to our FSM algorithm. On the other hand, the write-intensive workloads, which incur less read I/O requests, have benefited from this technique to improve the performance.

4.2.5 Summary

In summary, the performance our proposed Flash Storage Management (FSM) algorithm varies with different I/O workloads. The cost function convergence results gives us the theoretical indications of our FSM algorithm performance. We see that the FSM cost function converges for both the read-intensive and composite workloads, while it does not converge when applying it on a write-intensive workload. Further practical implications of these results are discussed in the hit ratio and data swapping experimental results. The results show that our FSM algorithm performs best in the read-intensive workload case. In addition, they show that write-intensive workloads do not benefit from FSM. Furthermore, the results demonstrate the efficiency of our FSM algorithm in maintaining the lifetime of SSDs by effective write operations to the devices, which is achieved through two factors. First, we move data between devices at large granularity of chunks. This will help in minimizing the fragmentation and internal write operations that might affect the lifetime of the SSD. Seconds, the algorithm achieves less data swapping as discussed in Section 4.2.3 which also contributes to maintaining longer lifetime of the SSD. On the other hand, Hystor is designed to move data at small granularity. This, as discussed in the results, creates more overhead in the system performance and the lifetime of the SSD, which makes it not suitable for large-scale storage system, but can be applicable for consumer level computing, such as, PCs.

The FSM algorithm performs two important computations with minimal cost and CPU overhead. First, generating the transition matrix, and it is computed at the start of the system using the input I/O log. It can also be updated as the system is running periodically, i.e., after a number of I/O requests, to adapt to the current system state. Second, computing the next action, i.e., deciding what data should be moved around the devices. Since the algorithm is designed with large granularity of data chunks, it significantly minimizes the number of state evaluations and hence minimizes the computations. In addition, computing the action plan does not generate further computations complexities. This is because of the nature of the random I/O workload that we are working with that results in some state
transition probability to be zero, which leads to skipping such states from the computations.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

While the solid-state drives (SSDs) offer several advantages for storage systems used in personal computers and consumer devices such as mobile phones and tablets, many challenges slow down their adoption in large-scale storage systems. These challenges include limited lifetime and high cost per GB. Several techniques have been previously suggested to tackle these challenges at different levels.

We presented the Flash Storage Management (FSM) algorithm for large-scale hybrid storage systems. The algorithm is designed using Markov decision process. To manage the storage system, the algorithm produces a state table and action table. A state describes the current data chunk the system is accessing. Each state has a corresponding action to instruct the system what data to be moved to the SSD.

We implemented two versions, i.e., online and offline, of our algorithm in a software simulator, and evaluated in terms of the cost function convergence, hit ratio and data swapping. We tested our algorithm using three different workloads: read-intensive, write-intensive and composite workloads. In addition, we compared our algorithm against the closest one in the literature, called Hystor.

Our experimental results show that the cost function we used for our algorithm converges. Furthermore, the results show that our algorithm achieves the best performance results for read-intensive workloads generated using the standard benchmark Postmark. For example, the online FSM achieves a hit ratio of 72% when using SSD size of 30% of the workload, which outperforms Hystor by more than 20%. However, the performance of all algorithms at SSD
size of 5% of the workload is close. Such results suggest that our proposed FSM algorithm is a good fit for large-scale hybrid storage systems with extensive read requirements to accelerate the I/O of the storage system.

\section{Future Work}

This work can be extended in several directions. In this thesis, the FSM algorithm defines a state as the most frequently accessed chunk of data over a period of time. One possible extension is to expand the state space definition with more complex states, such as the three most frequently accessed chunks of data over a period of time. This might help extracting some hidden characteristics of the target application I/O pattern.

Furthermore, we believe that including some heuristic actions into the action space, which can be compared with the random action and the best-known action, can produce better actions to be applied into the system. This, in turn, will produce better performance results.

Finally, our FSM algorithm is based on a first-order Markov decision process. The higher the order of a Markov decision process, the more search it does in order to compute the best action to take at any given time. We expect that some types of I/O workloads may benefit from higher orders of Markov decision process.
Bibliography


[27] Kieran Harty.


