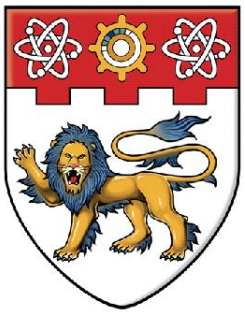

NANYANG TECHNOLOGICAL UNIVERSITY

PERFORMANCE ENHANCEMENTS IN LARGE SCALE STORAGE SYSTEMS

RAJESH VELLORE ARUMUGAM

SCHOOL OF COMPUTER ENGINEERING

2015

PERFORMANCE ENHANCEMENTS IN LARGE SCALE STORAGE SYSTEMS

RAJESH ARUMUGAM

2015

PERFORMANCE ENHANCEMENTS IN LARGE SCALE STORAGE SYSTEMS

RAJESH VELLORE ARUMUGAM

School Of Computer Engineering

A thesis submitted to the Nanyang Technological University in partial fulfilment of the requirement for the degree of Doctor of Philosophy

2015

Acknowledgements

I would like to thank my supervisors Asst. Prof. Wen Yonggang, Assoc. Prof. Dusit Niyato and Asst. Prof. Foh Chuan Heng for their guidance, valuable and critical feedback throughout the course of my investigations and research work. I am also grateful to Prof. Wen Yongang for guiding me in writing this thesis and some of the conference papers which were outcome of this research work. His valuable inputs have improved the quality of the writing and technical content significantly.

The major part of the thesis are the outcome of the Future Data Center Technologies thematic research program (FDCT TSRP) funded by the A*STAR Science and Engineering Research Council. I am grateful to all my friends/colleagues in A*STAR Data Storage Institute who provided me the guidance and support in executing the project (titled 'Large Scale Hybrid Storage System') to a successful completion. Special thanks to co-principal investigators of the project Asst. Prof. Ivor Tsang and Assoc. Prof. Ong Yew Soon from NTU. Without them this project might not have been a success.

I also extend my gratitude to A*STAR and A*STAR Data storage Institute, Singapore, for sponsoring my PhD study through their scientific staff development award. I also thank my Division manager Mr. Yong Khai Leong and Assistant Divisional manager Dr. Khin Mi Mi Aung of A*STAR Data storage institute for supporting my research in the Data Center Technologies Division. Last but not least, I would like to thank my family for bearing with me and for their continuous support throughout my PhD work.

Abstract

Data center storage systems of the future in Petabyte and Exabyte scale require very high performance (sub millisecond latencies) and large capacities (100s of Petabytes). The evolution both in scale (or capacity) and performance (throughput and I/O Per Second) is driven by the ever increasing I/O demands from the current and Internet scale applications. These large scale storage systems are basically distributed systems having two primary components or clusters. The first component is the storage server cluster which handles the primary I/O or data I/O for the applications. The second component is the meta-data server (MDS) cluster which manages a single global namespace and serves the meta-data I/O. In this thesis, we look in to the problem of performance deficiencies and scalability of these two components in a multi tenanted mixed I/O (sequential and random I/O) workload environment. To overcome the limitations of the conventional storage system architecture, the thesis proposes a 3-tier hybrid architecture utilizing next generation Non-volatile memory (NVM) like Phase change memory (PCM), Hybrid drives and conventional drives. NVM is used to absorb the writes to the NAND Flash based SSD. This improves both the performance and lifetime of the SSD. Hybrid drives are used as a low cost alternative to high speed Serial attached SCSI (Small computer system interface) or SAS drives for higher performance. This is achieved through a light-weight caching algorithm on the Flash inside the drive. On the storage server, we consider the problem of cache partitioning of next generation NVM, data migration optimization with placement across tiers of storage, data placement optimization of Hybrid drive's internal cache and workload interference among

multiple applications. On the Meta-data server, we consider the problem of load balancing and distribution of file system meta-data across meta-data server cluster that preserves namespace locality.

The following are the major contributions of this thesis to address the primary I/O and meta-data I/O performance scalability in large scale storage systems. A heuristic caching mechanism that adapts to I/O workload was developed for a hybrid device consisting of next generation NVM (like Phase change memory) and SSD. This method called HCache can achieve up to 46% improvement in I/O latencies compared to popular control theory based algorithms available in the literature. A distributed caching mechanism called VirtCache was developed that can reduce I/O interference among workloads sharing the storage system. VirtCache can reduce the 90th percentile latency variation of the application by 50% to 83% under a virtualized shared storage environment compared to state-of-art. An Optimized migration and placement of data objects across multiple storage tiers was developed that can achieve up to 17% improvement in performance compared to conventional data migration techniques. We propose new data placement and eviction algorithms on the Hybrid drive internal cache based on the I/O workload characteristics. It reduces the I/O monitoring meta-data overhead by up to 64% compared to state-of-art methods. The algorithm can also classify hot/cold data 48% times faster compared to existing methods. While these solutions address the performance scalability on the storage server, for the meta-data server scalability we developed the DROP meta-data distribution. The DROP mechanism based on consistent hashing preserves locality and near uniform distribution for load balancing. The hashing and distribution mechanism can achieve up to 40% improvement in namespace locality compared to traditional methods.

Table of contents

Table of contents vii

List of figures xiii

List of tables xvii

Acronyms xxi

1 Introduction 1

1.1 Background 1

1.2 Problem Statement 2

1.2.1 Performance Deficiencies In Storage Server 3

1.2.2 Performance Deficiencies In Meta-data Server 4

1.3 Thesis Statement 4

1.4 Contributions 5

1.5 Organization Of The Thesis 7

1.6 Overview Of Large Scale Storage Architectures 8

1.6.1 SAN/NAS Scalability 8

1.6.2 Clustered And Distributed File Systems Scalability 9

1.7 Problems With Current Storage System Architecture 13

1.8 Proposed Storage Architecture 17

2 Literature Review **23**

2.1 Introduction 23

2.2 Performance handling techniques in the Data path 23

2.2.1 Techniques and algorithms for HDD based systems 24

2.2.2 Storage Tiering and Data migration techniques 26

2.2.3 Performance enhancement through Caching 29

2.2.4 Workload prediction and prefetching 30

2.3 Performance handling in Metadata path 32

2.3.1 Metadata partitioning and distribution 32

2.3.2 Metadata pre-fetching and caching 33

2.4 Next Generation NVM Technologies 34

3 Storage System Performance Enhancement With Cache Optimizations **37**

3.1 Introduction 37

3.2 Next Generation NVM As Cache For SSD 38

3.3 Background On Cache Allocation And Replacement Algorithms 40

3.3.1 Application Working Set 42

3.4 Dynamic Cache Allocation Through HCache 45

3.5 Performance Evaluation Of HCache 52

3.6 I/O Workload Interference In Shared Storage Server 58

3.7 VirtCache Architecture 59

3.7.1 Workload Interference detection 60

3.7.2 VirtCache I/O Handler 63

3.7.3 Data Logger 64

3.8 VirtCache Prototype Evaluation 65

3.8.1 Exchange Workload on HDD 66

3.8.2 TPCC workload on HDD 68

3.8.3 Exchange workload on SSD 69
3.8.4 Consolidation of Exchange workload with Webserver workload on HDD 69
3.8.5 Effect of number of VMs 71
3.9 Summary 72

4 Improving Storage System Performance With Optimized Storage Tiering 75
4.1 Introduction 75
4.2 Multi-tier Hybrid Storage System 78
4.2.1 Data Collection 79
4.2.2 Data Monitoring Module 80
4.2.3 Data Migration Module 81
4.3 Multiple-stage Dynamic Programming For The DAP 81
4.3.1 Greedy Algorithm 82
4.3.2 Multiple-Stage Dynamic Programming Algorithm 85
4.4 Results 88
4.4.1 Simulation Instances 88
4.5 Summary 94

5 Storage System Performance Enhancements Utilizing Hybrid Drives 97
5.1 Introduction 97
5.2 Background and Motivation 99
5.3 Hybrid Cache Manager 103
5.3.1 I/O Monitoring Data: Interval Trees 105
5.3.2 Eviction Algorithm 107
5.3.3 Cache Placement Algorithm 108
5.4 Performance Evaluation 111

5.4.1 Evaluation Setup . . . 111
5.4.2 Performance Metrics . . . 112
5.4.3 Reduction In Caching Meta-data Overhead . . . 112
5.4.4 Cache Write Reduction . . . 114
5.4.5 Performance Improvements . . . 115
5.5 Performance Evaluation On Enterprise Storage System . . . 117
5.5.1 TPCC and Exchange Traces Performance . . . 117
5.5.2 Performance Improvements compared to commercial Hybrid drives 118
5.5.3 Database Performance Improvement . . . 119
5.5.4 Energy Savings . . . 120
5.6 Related Work . . . 122
5.7 Summary . . . 123

6 Metadata Placement And Load Balancing 125

6.1 Introduction . . . 125
6.2 MDS Cluster . . . 127
6.3 Locality Preserving Hashing . . . 129
6.4 Capacity Load Balancing With DROP . . . 131
6.4.1 Histogram based Load Balancing . . . 131
6.5 Request Load Balancing With Distributed Meta-data Cache . . . 133
6.5.1 Meta-data Cache System Model . . . 134
6.5.2 Load Shedding and Stealing . . . 135
6.6 Evaluation . . . 137
6.6.1 Namespace Locality And Capacity Load Balancing . . . 137
6.6.2 Request Load Balancing in C^2 . . . 139
6.7 Related Work . . . 141
6.8 Summary . . . 144

7 Conclusion And Future Work **149**

7.1 Conclusion 149

7.2 Future Work 151

7.2.1 Developments in Next Generation NVM 151

7.2.2 Developments in Non volatile memory (NVM) interfaces 152

7.2.3 Developments in Hybrid Drives 153

7.2.4 Meta-data Management 154

7.3 Publication Contributions By Thesis Author 155

7.3.1 Journal Publications 155

7.3.2 Conference Publications 155

7.3.3 Patent Publications 156

References **157**

List of figures

1.1 Network Attached Storage with NFS 9

1.2 SAN Architecture 10

1.3 Lustre Architecture [1] 11

1.4 Ceph Distributed File System Architecture [122] 12

1.5 HDFS Distributed File System Architecture [113] 13

1.6 GlusterFS System Architecture 14

1.7 Clustered Storage System Architecture 15

1.8 Multi-tiered storage using SSD, SAS/High speed drives and SATA 16

1.9 Evolution of Solid state storage 18

1.10 Storage server with hybrid devices 19

1.11 System components of the Proposed Hybrid Storage system 21

3.1 Hybrid device with SSD and non-volatile memory like STT-MRAM/PCRAM 39

3.2 Heat map of LRU cache with Financial1 trace 43

3.3 Hit rate Histogram for Financial1 trace 44

3.4 Heat map of LRU cache with Exchange trace 45

3.5 Hit rate Histogram for Exchange trace 46

3.6 Heat map for web search trace 47

3.7 Hit rate Histogram for web search trace 48

3.8 HCache: The high level overview of the HCache cache control system . . . 48

3.9 Figure showing the LRU log in Non-volatile memory 49
3.10 Financial (OLTP) trace showing Hit rates and cache size variation over time for HCache and PID control. 53
3.11 Microsoft exchange trace collected using Loadgen showing hit rates and cache size variation over time for HCache and PID control. 53
3.12 MSR trace showing hit rates and cache size variation over time for HCache and PID control. 54
3.13 GlusterFS architecture with VirtCache components 60
3.14 High level control flow between components in VirtCache. 64
3.15 VirtCache caching for Exchange workload on HDD (Latency and Queue depth at the Storage server) 66
3.16 VirtCache caching for TPCC workload on HDD (Latency and Queue depth at the Storage Server) 67
3.17 VirtCache caching for Exchange workload on SSD (Latency and Queue depth at the Storage Server) 68
3.18 VM2 (Webserver) disk latency in consolidation with Exchange (VM1) with VM2 chosen for caching 69
3.19 VM1 (Exchange) disk latency in consolidation with Webserver (VM2) with VM2 chosen for caching 70
3.20 Queue depth for VM1 (Exchange) and VM2 (Webserver) consolidation . . 70
3.21 Latency deviation from non-consolidated case as the number of VM instances are increased 71

4.1 The overall MTHS system framework 78
4.2 The data processing flow for the DAP. 79
4.3 Comparisons of benefit values between GA, HRO and MDP 92
4.4 Comparison of results between HRO and MDP using using large data range 93

5.1 Schematic of a Hybrid Drive 100
5.2 Comparison of conventional SSD caching (left figure) with Hybrid drive caching (right figure). 102
5.3 Hybrid Cache Manager Components 104
5.4 Interval Tree nodes with hot LBA region tracking information 106
5.5 LBA address range record (LRR) structure 106
5.6 Interval tree node meta data structure 106
5.7 Disk hot regions or zone estimation 111
5.8 Caching meta-data savings for different traces compared to hash based mapping 113
5.9 Cache meta-data update savings for different traces compared to hash based mapping 113
5.10 Cache write reduction compared to LRU and similar algorithms 114
5.11 Cache Hit rate drop compared to LRU and similar algorithms 114
5.12 Hybrid drive IOPS improvement per drive for TPCC workload compared to baseline HDD 116
5.13 Hybrid drive IOPS improvement per drive for Microsoft Exchange workload compared to baseline HDD 116
5.14 Hybrid drive IOPS improvement per drive for TPCC workload on Storage System prototype (RAID5 configuration) 118
5.15 Hybrid drive IOPS improvement per drive for Microsoft Exchange workload on Storage System prototype (RAID5 configuration) 119
5.16 Prototype Hybrid Drive IOPS comparison with popular commercial drives . 120
5.17 SAS drive (7.2K) Power Consumption per drive for TPCC workload (RAID5 configuration) 121

5.18 Hybrid drive (5.4K) Power Consumption per drive for TPCC workload (RAID5 configuration) 121

6.1 MDS cluster showing the DROP distributed Meta-data management 128
6.2 Meta-data server components with DROP and C^2 for capacity load balancing and request load balancing respectively 129
6.3 Encoding of file system paths 130
6.4 CDF of file paths for file system traces 130
6.5 Zipf like distribution for the Windows trace 135
6.6 Zipf like distribution for the Harvard trace 135
6.7 DROP namespace locality for Linux trace. 138
6.8 DROP namespace locality for Microsoft trace. 139
6.9 DROP namespace locality for Harvard trace. 140
6.10 DROP meta-data load distribution for Linux trace 141
6.11 DROP meta-data load distribution for Microsoft trace 142
6.12 DROP meta-data load distribution for Harvard trace 143
6.13 Load factor as a function of increasing servers in C^2 with Linux Trace . . . 144
6.14 Load factor as a function of increasing servers in C^2 with Microsoft Trace . 145
6.15 Load factor as a function of increasing servers in C^2 with Harvard Trace . . 145
6.16 Meta-data request load balancing in C^2 with Linux Trace 146
6.17 Meta-data request load balancing in C^2 with Microsoft Trace 146
6.18 Meta-data request load balancing in C^2 with Harvard Trace 147

7.1 Summary of thesis contributions 150

List of tables

1.1 Comparison of Storage technologies 17

3.1 Average latency and latency violations for different workloads 55
3.2 Percentage reduction in Log size for the different workloads 56
3.3 Control mechanism comparison with respect to HCache 57

4.1 Simulation instances for 2-tiered storage system with different types. 90
4.2 Comparisons of total benefit values between GA, HRO and MDP. 91
4.3 Comparisons of total benefit values between HRO and MDP with variable number of tiers 93

5.1 Traces Used In The Evaluation Of Hybrid Drive Cache Manager 112
5.2 HammerDB Benchmark Tests On The RAID5 (5 drives) Storage System Prototype 120

6.1 Example meta-data load balancing in DROP 132
6.2 Traces used in the evaluation of DROP 137

Acronyms

ARC Adaptive Replacement Cache

ATA Advanced Technology Attachment

DRAM Dynamic Random Access Memory

DROP Dynamic Ring Online Partitioning

HDD Hard Disk Drive

HDFS Hadoop Distributed File System

HPC High Performance Computing

IOPS I/O Per Second

LBA Logical Block Address

LFU Least Frequently Used

LRU Least Recently Used

MCKP Multiple Choice Knapsack Problem

MDP Multiple-stage Dynamic Programming

MDS Meta-data Server

MLC Multi-level Cell

NAS Network Attached Storage

NFS Network File System

NVM Non-Volatile Memory

NVRAM Non-Volatile Random Access Memory

OLTP On-line Transaction Processing

OSS Object Storage Server

PCM Phase Chage Memory

PCRAM Phase Change Random Access Memory

PID Proportional Integral Derivative

POSIX Portable Operating System Inteface

QoS Quality Of Service

RAID Redundant Array Of Independent Disks

ReRAM Resistive Random Access Memory

RPM Revolutions Per Minute

SAN Storage Attached Network

SAS Serial Attached SCSI

SATA Serial Advanced Technology Attachment

SLC Single Level Cell

SNIA Storage Networking Industry Association

SSD Solid State Drive

STT-MRAM Spin Torque Transfer - Magnetic Random Access Memory

TPCC Transaction Processing Performance Council

VM Virtual Machine

Chapter 1

Introduction

1.1 Background

Storage systems are one of the major components of any data center and have evolved from very simple storage arrays to complex large scale distributed/clustered systems over the past few decades. The evolution both in scale (or capacity) and performance (throughput and IOPS) is driven by the ever increasing I/O demands from the current and Internet scale applications. The demand for storage capacity is largely attributed to the exploding growth of both human generated and machine generated data. Human generated data comes from applications like e-mail, data bases, e-commerce, social networks and various micro blogging applications. Machine generated data comes from various sensors (e.g. weather, seismic data) and health care applications (like CT scans and X-rays etc). As the capacity scales to peta bytes and more, it becomes challenging to keep up with the expected performance from these applications in terms of I/O per second and throughput (Megabytes per second). Moreover, in a mixed environment like in a data center, the storage system is shared among multiple applications. This results in managements issues like storage capacity and QoS (Quality of service) provisioning for the different applications. To summarize, the main characteristics of future generation large scale storage systems are the following:

1. Scales in capacity to Peta bytes and Exabyte of storage. The capacity scaling has to be achieved with reasonable costs. Hard disk drives (HDD) satisfy the capacity requirement for the best Cost/GB. But they suffer from poor performance due to their mechanical nature. Though it is possible to employ solid state technologies like NAND flash based SSD (Solid state drives), the cost rapidly grows for larger capacities. Also as will be explained in detail later, NAND flash based system suffer from endurance problems.

2. Meet the various application performance demands in terms of IOPS (I/O per second) and throughput (MB/s). This involves both data and meta-data I/O performance. Meta-data here refers to the storage meta-data required to locate the original data (like inodes in file systems). For example as the system grows to Petabytes capacity the meta-data alone grows to several Terabytes. For good response times we need fast and efficient methods to store and look-up meta-data.

3. Provides a shared storage environment that meets the capacity and performance demands for the various applications using the system. The performance characteristics of the storage devices in the system depend largely on the I/O characteristics of the workloads [71]. Therefore, it becomes challenging to satisfy the performance demands of the various applications in such a mixed workload environment. The storage system also had to handle the I/O interference among the applications which impact their performance.

1.2 Problem Statement

Data center storage systems of the future in Petabyte and Exabyte scale require very high performance (sub millisecond latencies) and large capacities (100s of petabytes) [12] [89]. These large scale storage systems are basically distributed systems having two primary com-

ponents or clusters. The first component is the storage server cluster which serves the primary I/O or data for the applications. The second component is the meta-data server cluster which manages a single global namespace and serves the meta-data traffic. In this thesis, we look in to the problem of performance deficiencies of these two components in a multi tenanted mixed I/O (sequential and random I/O) workload environment.

1.2.1 Performance Deficiencies In Storage Server

Large storage capacities can be achieved only through disk drives since they provide high capacities (in Terabytes) at low cost. Utilizing HDD alone leads to heavy disk seeks when there are concurrent accesses from 1000s of clients with mixed workloads. This could dramatically reduce the throughput of the system. As an alternative SSDs are used along with HDD as the tier 1 storage or as a cache or both in some of the distributed architectures. This can improve the performance of the system significantly. But this solution has the following problems:

1. Asymmetric I/O performance of NAND flash leads to inconsistent performance for mixed workloads having both reads and writes. (Typical latencies for read are 200 microseconds and write requires a few milliseconds when the flash blocks needs to be erased.)

2. SSD have poor write performance due to the erase before write operation. The erase cycle involves several hundred millisecond latencies.

3. SSD also require complicated wear leveling algorithms to extend their life cycles due to the poor endurance life cycles of the NAND Flash cells.

4. Tiered storage with combination of SSD and HDD require complex data migration and caching techniques. This involves fine grain I/O monitoring, placement and eviction algorithms that adds large overhead to the storage system.

To overcome these problems the thesis proposes a 3-tier hybrid architecture utilizing next generation NVM, Hybrid drives and conventional drives. NVM is used to absorb the writes to the NAND Flash based SSD. This improves both the performance and lifetime of the SSD. The Hybrid drives are used as a low cost alternative to high speed SAS drives for higher performance through a light weight caching algorithm on the Flash inside the drive.

1.2.2 Performance Deficiencies In Meta-data Server

Large meta-data growth in the order of 100s of Terabytes leads to inefficient meta-data management when using conventional approaches. The large number of concurrent client access will be a major bottleneck resulting in poor scalability and performance at the MDS. It has been found [89] that meta-data performance like directory lookups and file creation times does not scale linearly in petabyte scale file systems. This requires uniform distribution of meta-data in a cluster and new techniques to achieve faster meta-data look up times. The thesis proposes methods of distributing meta-data uniformly without loss of locality of the namespace.

1.3 Thesis Statement

The main statement of the thesis is that by utilizing a hybrid storage device architecture and algorithms for managing the storage resources with efficient placement of data/meta-data, we can enhance the performance of the storage system significantly compared to traditional systems. Traditional storage systems are those which utilizes a 3-tier architecture with SSD, High speed drives (SAS) and SATA drives. While some of the problems described in section 1.2.1 has been addressed by other research work for traditional 3-tier storage systems, the thesis differs in the following areas:

1. The performance scaling techniques would need to be re-looked when new/emerging

technologies like next generation NVM (STT-MRAM, PCRAM) and Hybrid devices (Flash with STT-MRAM, Flash with HDD) are introduced in these architectures. The thesis proposes new storage allocation strategies when these hybrid and next generation devices are employed. It also proposes methods for reducing I/O interference among workloads in a multi-tenanted Datacenter environment which affects the performance of the applications.

2. The thesis proposes a 3-tier hybrid architecture for the storage server which includes combination of next generation NVM, SSD, Hybrid drives and conventional drives. This differs from the traditional 3-tier architecture and therefore requires new methods for storage resource allocation and data migration to improve performance.

3. The thesis proposes a method of meta-data distribution that preserves locality of storage namespace while providing near uniform distribution to improve meta-data I/O capacity load balancing. Other state of art methods trade-off locality for uniform distribution or vice versa and therefore suboptimal in meta-data load balancing.

1.4 Contributions

We analyze the problems mentioned in section 1.2.1 and propose methods for improving the performance of storage and meta-data nodes in a distributed storage system under a mixed workload multi tenanted environment. A hybrid device architecture is proposed where storage elements incorporate next generation NVRAM (Non-volatile memory like STT-MRAM, PCRAM etc.) and hybrid drives in addition to traditional SSD and HDD. NVRAM is used for handling deficiencies in solid state devices like NAND Flash based SSD used in conventional systems. NVRAM is used as both a persistent cache and as a storage tier.

The contributions from this thesis are the following:

1. A heuristic caching mechanism that adapts to I/O workload was developed for a hybrid device consisting of NVRAM and SSD. This method called HCache [104] adapts the NVM cache size to the dynamically changing workload characteristics in a multi tenanted storage system. We show that this method can achieve up to 46% improvement in I/O latencies compared to popular control theory based algorithms available in the literature. Chapter 3 describes this work in detail.

2. A distributed caching mechanism called VirtCache [90] was developed. VirtCache handles I/O latency variation under consolidated Virtual machine workloads as in a Datacenter cloud environment. VirtCache can pro-actively detect storage device contention at the storage server and temporarily redirect the peaking virtual disk workload to a dynamically instantiated distributed read-write cache. We show through experiments on our prototype 50% to 83% reduction in the 90th percentile latency deviation from average compared to previous work as we move from low load conditions to peak non uniform consolidated Virtual machine (VM) workloads. This complements the HCache work under a muti-tenanted distributed storage system. Chapter 3 describes this work in detail.

3. Migration mechanisms between the different tiers had to be changed to consider the device characteristics of NVM, SSD and hybrid drives. An Optimized migration [96] of data objects across different tiers was developed. This addresses the problem of maximizing the performance (IOPS/throughput) by distributing hot data objects among the different tiers. With this data migration mechanism we can achieve up to 17% improvement in performance compared to state-of-art data migration technique. This work is explained in detail in chapter 4.

4. Hybrid drives have a small capacity NVM (Flash) inside them to be utilized as a cache. In Chapter 5 we propose new caching method that performs data placement

and eviction on this internal drive cache based on the I/O workload characteristics. It uses minimum meta-data for tracking I/O and classifies hot/cold data faster than state-of-art methods. The I/O monitoring meta-data overhead can be reduced up to 64% compared to state-of-art methods. The algorithm can also classify hot/cold data 48% faster compared to conventional methods.

5. A meta-data distribution mechanism called DROP ([130],[131]) based on consistent hashing that preserves locality and near uniform distribution for load balancing was developed. The hashing and distribution mechanism can achieve up to 40% improvement in namespace locality compared to traditional methods. We extend this work to optimize meta-data placement in a distributed cache and have developed a method called Cloud Cache (C^2) [132].

The above developed components were integrated in a distributed storage architecture to handle the performance deficiencies on both the Primary data path and the Meta-data path. These optimization components work together to improve the overall performance of the entire distributed storage system. The performance deficiencies in the Primary data path is handled by the Storage server which includes the cache optimization components HCache, VirtCache, Hybrid drive caching and Optimized Data migration. The Meta-data path is handled by the Meta-data server or MDS which consists of DROP and C^2. Figure 1.11 in section 1.8 shows the overview of the various optimization components across the storage system to enhance data and meta-data performance.

1.5 Organization Of The Thesis

The following sections 1.6 and 1.7 describes the background on large scale storage architectures and the current problems respectively. This is followed by the description of the proposed architectural changes and problems to be solved with hybrid storage devices and

next generation NVM (section 1.8). Chapter 2 provides the literature survey on methods of performance enhancements in storage systems. Chapter 3 describes next generation NVM based caching to improve the performance of the storage nodes in the cluster. It also describes a distributed caching mechanism to reduce I/O workload interference in a shared Virtual machine environment. Chapter 4 describes the work done on data migration and optimization in multi-tiered storage server. Chapter 5 describes a new method of caching utilizing the Flash inside a Hybrid drive. Chapter 6 describes the meta-data distribution mechanism in a meta-data server cluster to improve performance. Finally in Chapter 7 we conclude with summary of the results and potential future directions of research.

1.6 Overview Of Large Scale Storage Architectures

1.6.1 SAN/NAS Scalability

Traditional shared storage systems like NAS (network attached storage) and SAN (Storage attached network) provide a fair amount of capacity scaling and performance. These systems have scalability and management issues when the capacity exceeds 100s of terabytes.

Figure 1.1 shows the typical components of a NAS using NFS. Normally NAS runs on top of the IP network. The NFS system consists of the NFS server and client. The server is attached with storage (typically a HDD RAID array) and provides the file service to the clients. The main problem with this architecture is that as the number of clients accessing the storage system grows, the server becomes a bottleneck. Moreover the server has to handle both the meta-data and data requests concurrently.

Figure 1.2 shows the architecture of the typical SAN. Here, the storage exists in a dedicated network and provides a block interface to the application servers. The application servers will run their own file system to access the storage. The same problems as in NAS exist for SAN also as the capacity runs into several hundreds of terabytes. Even though

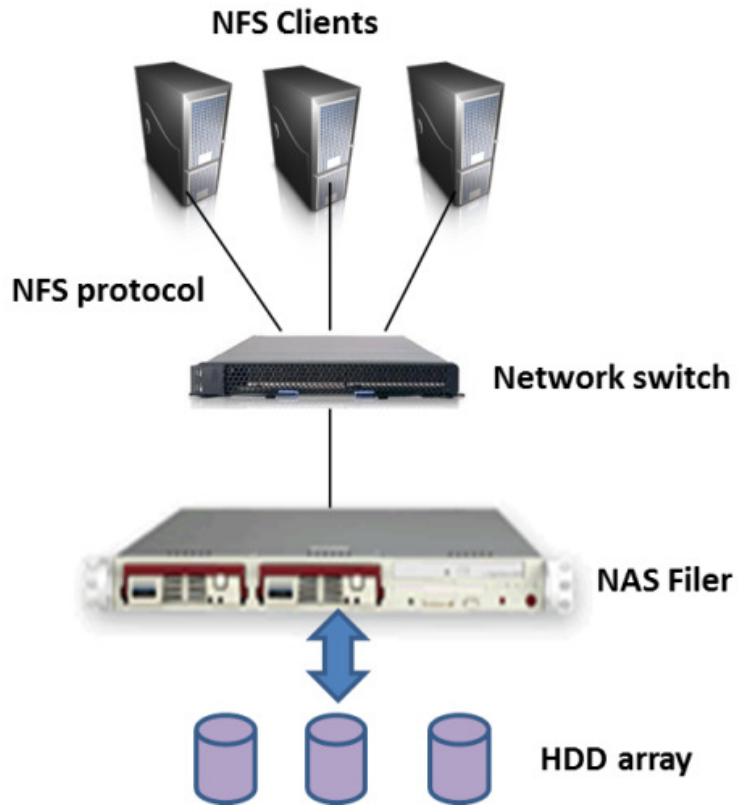

Fig. 1.1 Network Attached Storage with NFS

SAN provides a higher performance (through dedicated fiber optic networks), the system becomes complex to manage in terms of block space partitioning and meta-data management among the clients.

It has also been found [59] that 50% of storage traffic is meta-data I/O which interferes significantly with the data traffic from the clients in both the NAS and SAN systems. This impacts the performance of the system as the servers spend most of the time looking up meta-data instead of serving data to the various clients.

1.6.2 Clustered And Distributed File Systems Scalability

The very high performance requirement of High performance computing (HPC) has motivated the design and development of large scale clustered storage systems like Lustre [1],

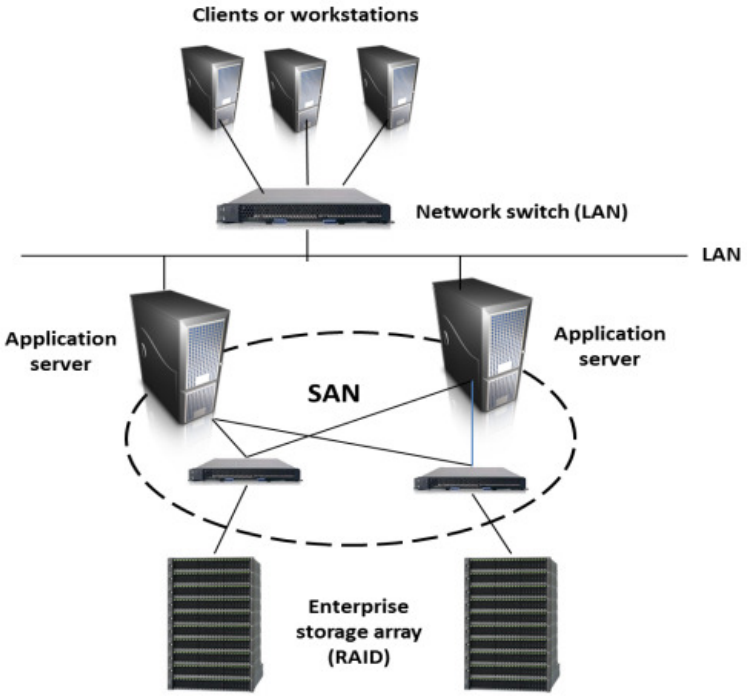

Fig. 1.2 SAN Architecture

Ceph [122], and Panasas [124]. These systems can provide a better degree of scalability in terms of 10s to 100s of Petabytes and aggregate throughputs in the range of 100s of GB/s. For these type of large scale systems, hard disks (HDD) have played a major role in satisfying the large capacities for the best cost/GB. The performance is achieved by employing multiple spindles of the HDD in parallel in a RAID (Redundant array of inexpensive disks) like fashion. These systems also handle scalability by separating the meta-data path and data path.

Lustre: Lustre is a high performance clustered storage system mainly used for HPC applications. Deployments in the range of 10s of petabytes are being employed for some of the HPC applications. The main components (Figure 1.3) of Lustre are the object storage servers (OSS), meta-data server (MDS) and the clients where the application runs. Large scale installations [30] have 100s of OSS, 1000-10,000s of clients and one meta-data server. Lustre is an object based storage system and therefore stores data as objects in the storage servers. File level striping (like in RAID) is the major mechanism used to achieve the high

performance of the system. A file is striped as objects across multiple OSS. Even though the server stores data as objects, the applications residing on the client sees a POSIX (Portable operating system interface) interface. As in other clustered storage systems, Lustre architecture has a separate data and meta-data path. All meta-data and namespace management is handled by a dedicated meta-data server.

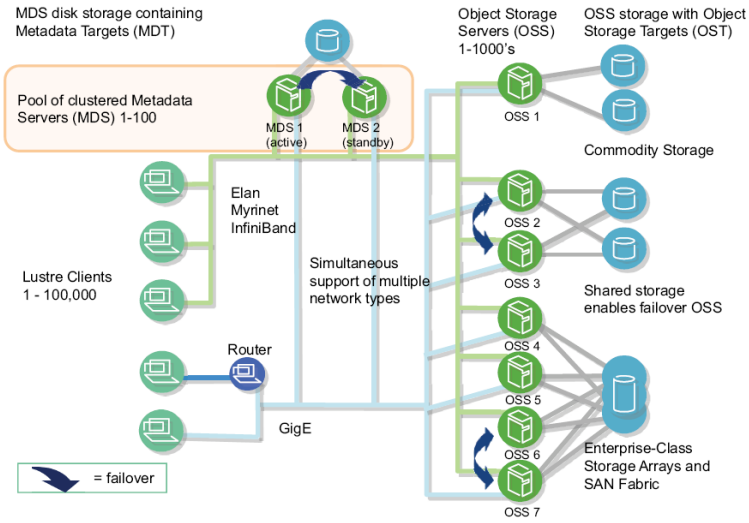

Fig. 1.3 Lustre Architecture [1]

In order to perform I/O on a file, the client issues a request to the MDS first. The MDS returns the location of the file data by providing the list of OSS and the object identifiers stored in the OSS. The client later uses this information to directly perform read/write on the objects located on the OSS. With such a mechanism, the object server (OSS) is freed from handling any meta-data I/O. The clients can therefore utilize the full network bandwidth to perform I/O directly on the OSS.

The main issue in Lustre is the MDS which is both a bottleneck as well as a single point of failure. If the MDS fails then the client cannot locate file data and system becomes unusable. Lustre mitigates this by providing both active-active and active-passive fail-over configurations [1] of the MDS. But performance is still a problem with the single MDS serving the entire cluster.

Ceph: Similar to Lustre, [122] uses a distributed architecture with separate meta-data and data path. As figure 1.4 shows, clients perform file I/O directly with the Object storage server cluster. The file system namespace and meta-data management is handled by the meta-data server (MDS) cluster. The major difference is that Ceph uses a cluster of MDS instead of a single MDS for serving meta-data to the clients. Ceph is a pure object based system and therefore provides an object interface to the applications residing on the clients.

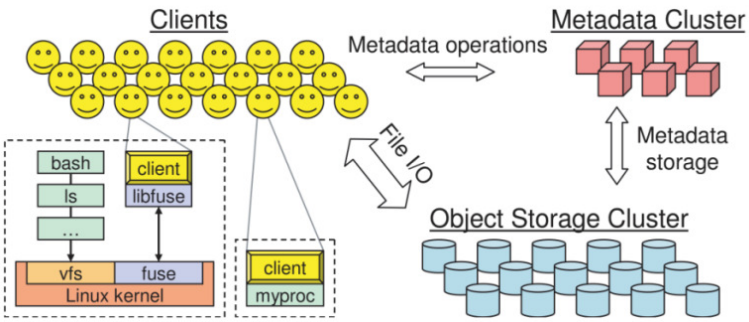

Fig. 1.4 Ceph Distributed File System Architecture [122]

HDFS: The Hadoop [113] distributed file system (HDFS) was developed for specific applications that require large volumes of data to be processed in parallel. As seen in Figure 1.5, HDFS also uses multiple storage nodes or data nodes for storing file data. The data nodes are called chunk servers since file data are stored as large blobs of data typically of size 64MB. A dedicated node called the name node is used for storing meta-data and file system namespace. Due to the specific nature of the applications running on it, (like analytics) HDFS does not support overwriting of files. It also has the problem of single point of failure at the Name node.

GlusterFS: GlusterFS [111] like Hadoop is used mainly in a cloud environment. The main difference between the other architectures is that it does not use any dedicated meta-data server (as in Lustre, HDFS) or meta-data server cluster (as in Ceph). The meta-data is stored along with file data among the different data nodes. GlusterFS makes use of striping (through RAID) at the data nodes. It uses a hash based lookup to resolve file/directory

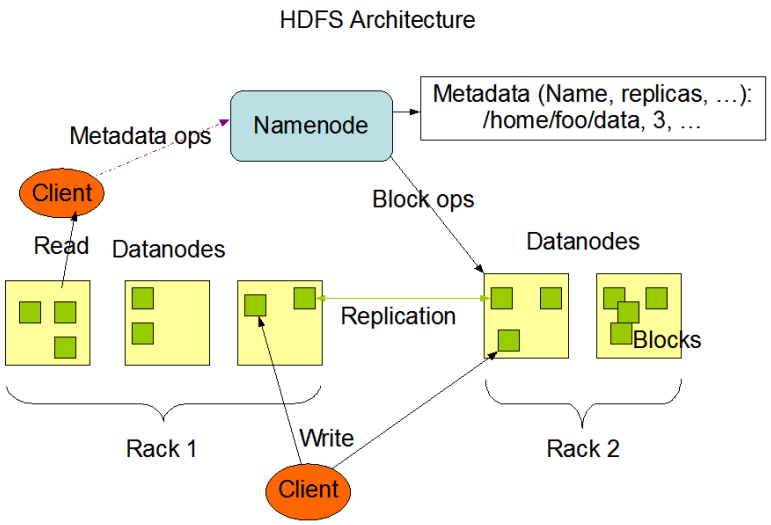

Fig. 1.5 HDFS Distributed File System Architecture [113]

names to data locations on the data nodes.

1.7 Problems With Current Storage System Architecture

The general architecture common to all the distributed and clustered storage systems explained in the previous sections (Lustre,Ceph,HDFS,GlusterFS) is shown in Figure 1.7.

This architecture decouples the data path from the meta-data path of the storage system to achieve scalability. The meta-data is handled separately by a meta-data server component and the data is served by a Storage server component. This eliminates most of the NAS/SAN server bottlenecks in a NAS filer or a SAN server in the conventional system. It also allows the application to have direct access to the storage devices. Applications can now make parallel requests to these devices by striping files thereby improving the overall aggregate I/O throughput of the system. Such a system can scale horizontally both in capacity and performance by adding more storage server and meta-data server components. The architectures described before uses HDD extensively for providing the high capacity scaling. Though hard disks provide the capacity scaling with the best cost per gigabyte, they

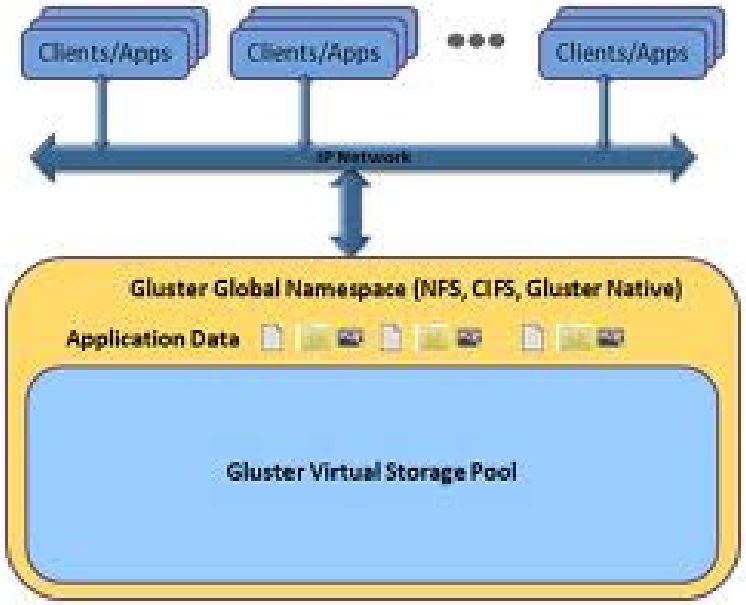

Fig. 1.6 GlusterFS System Architecture

suffer from the mechanical characteristics of the rotating media and the read/write head. This results in latencies in terms of several milliseconds (8-10ms typically). Particularly the performance becomes worse for non-sequential I/O which involves physical movement of the read/write heads. Recently, some of these architectures use SSD along with HDD to achieve the higher IOPS for non-sequential workloads. This is the idea behind multi-tiered storage management solutions which has been around for some time in the enterprise storage domain.

As shown in the Figure 1.8 above, in multi-tiered storage, we have different tiers or layers of storage with first tier normally being high performance expensive storage like SSD, tier 2 moderate cost/performance storage and tier 3 with cheap low performance high capacity devices like SATA or tape archives. The main principle behind this architecture is to strike a balance between cost and performance of the entire storage system based on the I/O demands. Frequently accessed data is moved or migrated to the higher tiers and less accessed data is moved down to lower tiers. This ensures that the high performance higher tiers are utilized effectively for very hot data and at the same time less accessed data

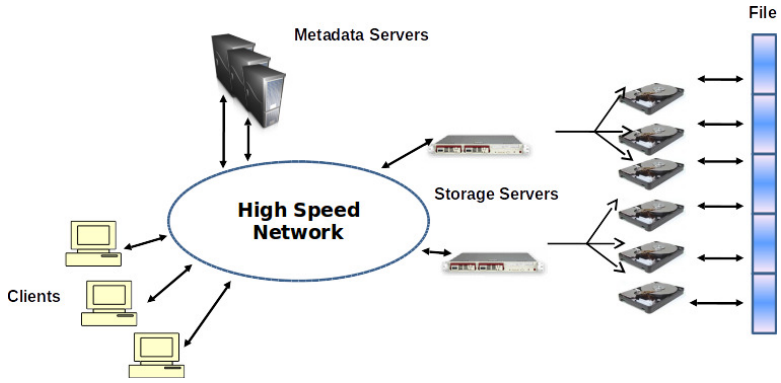

Fig. 1.7 Clustered Storage System Architecture

is moved to the low performance cheaper device.

Even such an approach runs in to problems when the system scales to large capacities with multiple application access. NAND Flash based SSDs though offering superior performance compared to Hard drives, still have problems like poor write performance and endurance since the flash cells wear out [74] [72]. The poor write performance of NAND flash being attributed to the erase before write requirement of the memory cells [60]. Though several performance improvement techniques have been proposed ([15],[22],[43],[58]) for handling this erase-before-write problem, these are limited by the inherent characteristics of the Flash device. To overcome these limitations of SSD and to improve the I/O performance, new storage technologies like STT-MRAM [19] and PCRAM [57] are emerging. These devices have very high throughputs and nanosecond latencies. They approach the performance of system memory and are expected to close the long standing gap between storage and computation. Table 1.1 shows the comparison of various storage technologies including enterprise HDD, Hybrid drives based on performance and cost. We also utilize hybrid drives due to their superior performance compared to HDD and comparable performance of SSDs (for non-sequential I/O).

We can see from the table that next generation NVM technologies like STT-MRAM and ReRAM can offer performance close to that of DRAM. By utilizing these next generation storage devices, we can achieve tremendous improvement in performance through efficient

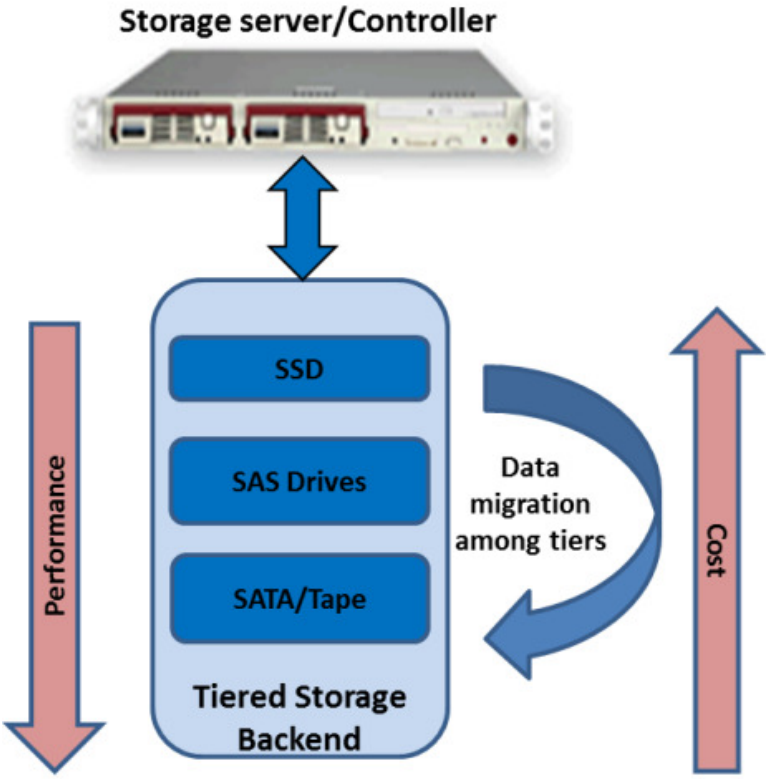

Fig. 1.8 Multi-tiered storage using SSD, SAS/High speed drives and SATA

caching or tiering or both. NVM can also solve the poor write performance of SSD. Even though these next generation NVM technologies offer superior performance, they are unlikely to replace flash entirely due to current limitations in process technologies, fabrication costs and densities [101]. Figure 1.9 shows the time line of the evolution of solid state technologies from the past 3 decades.

The adoption of stolid state storage has been progressing rapidly with the decreasing cost of NAND flash based SSD and their high performance compared to HDD. We can see from the table that next generation NVM technologies like STT-MRAM and ReRAM can offer performance close to that of DRAM. By utilizing these next generation storage devices, we can achieve tremendous improvement in performance through efficient caching or tiering or both. NVM can also solve the poor write (table 1.1) performance of SSD. Even though these next generation NVM technologies offer superior performance, they are un-

Table 1.1 Comparison of Storage technologies (See [107], [19],[57],[68],[109]). For Hybrid drives the latencies shown are average values for I/O exhibiting hits to the internal NVM cache

Technology	**Read**	**Write**	**Endurance Cycle**	**Density (area in F^2)**	**Cost($/GB)**
DRAM	<10ns	<10ns	10^{16}	6-10	11-14
SLC Flash	25μs	200μs/1.5ms (Program/Erase)	10^5	4-8	0.8-1 (SSD with SLC)
HDD (15K RPM)	6000μs	6000μs	NA	NA	0.05
Hybrid HDD (5.4K RPM)	100μs-200μs (average)	100μs-200μs (average)	NA	NA	0.07-0.1
ReRAM	10-50ns	10-50ns	10^8	4-8	High
PCRAM	60ns	60ns/120ns (Write/Erase)	10^8	8	Low
STT-MRAM	2-20ns	2-20ns	10^{15}	10-30	Highest

likely to replace flash entirely due to current limitations in process technologies, fabrication costs and densities [101]. The below figure shows the time line of the evolution of solid state technologies from the past 3 decades. The adoption of stolid state storage has been progressing rapidly with the decreasing cost of NAND flash based SSD and their high performance compared to HDD. Even though the new technologies like PCM and STT-MRAM have improved performance by several orders compared to SSD, they are still expensive and are similar to the state of the earlier SSD technologies. Some of these next generation NVM technologies (STT-MRAM, PCRAM) also suffer from lower densities (bits/area) compared to SSD thereby increasing their cost/bit as seen from the Table 1.1.

1.8 Proposed Storage Architecture

Pertaining to the problems stated in the previous section (1.7), we need a hybrid architecture that makes use of both SSD and next generation NVM to provide performance in sub mil-

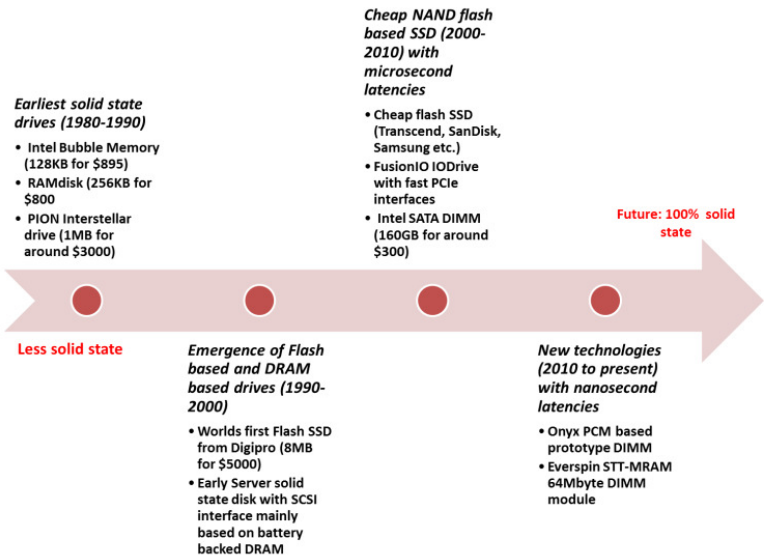

Fig. 1.9 Evolution of Solid state storage

lisecond latencies with reasonable cost. This work therefore proposes the following hybrid architecture:

1. NVRAM as a high performance cache and tier: Combination of SSD and next generation NVM as the tier 1. This will handle very small I/O and provide sub millisecond latencies.

2. Hybrid drives as tier 2: It has been found that hybrid drives can be used to improve the write latency by up to 70% [16]. This is a significant improvement in performance compared to traditional HDD. This is due to the small NAND flash capacity embedded in the drive. This small capacity Flash is used as a caching device for small I/O to the drive and therefore avoids the latency sensitive seeks.

3. Conventional Hard disk drives (HDD) at tier 3: To meet the large scale capacity scaling we still need to employ the conventional drive to store nearline or cold data.

Figure 1.10 shows the tiered architecture of the storage server with hybrid devices. Next generation NVM (like STT-MRAM) along with NAND Flash based SSD is utilized as the

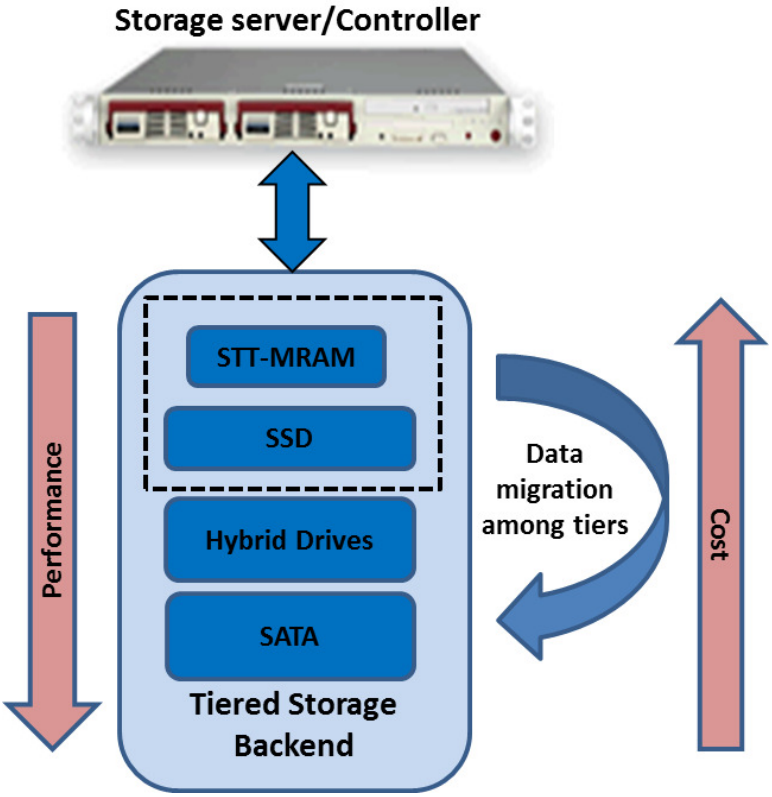

Fig. 1.10 Storage server with hybrid devices

first tier, Hybrid drives replaces the high speed SAS drives in the conventional architecture. As in the other clustered and distributed architectures described before, clients access the storage servers directly for performing file or object I/O. The MDS handles the meta-data I/O like namespace traversals and lookups. The following are the problems addressed with this architecture:

1. NVM can act as a persistent cache to the SSD, thereby providing high performance for small write intensive I/O. But this requires careful allocation of the expensive cache space to different applications using the storage system.

2. While the NVM/SSD hybrid device can improve the small write I/O performance for individual applications, in a large scale shared storage system I/O workload interference still exists. We propose a dynamic distributed cache instantiation and I/O redi-

rection mechanism to reduce the interference in both HDD and SSD thereby reducing the application perceived I/O latencies.

3. Data migration is a general problem in any tiered storage system. Here we explore different state-of-art methods in I/O monitoring and migration in the literature. We have arrived at a dynamic programming algorithm that maximizes the utilization of the tiers based on the popularity of the data.

4. The placement and eviction of data on the Hybrid drive's internal cache requires fast identification of hot data regions on the magnetic platter. The identification should be both fast and accurate with minimal I/O tracking meta-data. An algorithm utilizing Red-Black trees was developed for this purpose and we show its benefits compared to state-of-art methods used for general caching.

The major issues to address on the meta-data server cluster are the following:

1. Load balancing on the meta-data server requires uniform placement of meta-data across the MDS cluster. But this destroys locality of the namespace that is required for certain file system operations. An algorithm utilizing locality preserving consistent-hashing was developed for this purpose.

2. In addition, we also need to look into the problem of optimizing cache space for a distributed meta-data cache to improve overall meta-data cluster performance.

We look in to the details of the solution to these problems in subsequent chapters. Several performance optimization components were developed as shown in the figure 1.11 for both the storage server or data nodes and meta-data server. The respective chapters shown in the figure covers the corresponding work in detail.

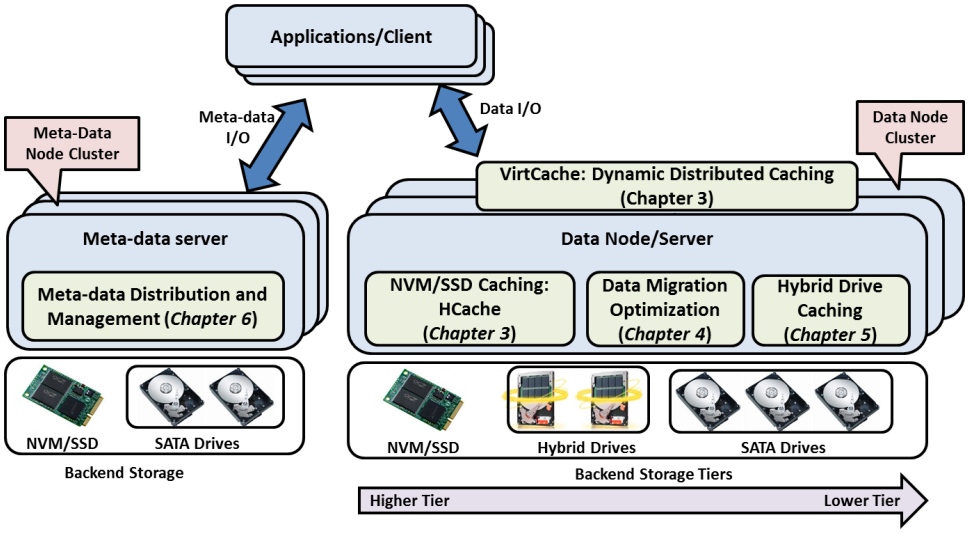

Fig. 1.11 System components of the Proposed Hybrid Storage system

Chapter 2

Literature Review

2.1 Introduction

In this chapter we briefly take a look in the State-of-art techniques in improving performance of Storage systems in general. We only look at the major work in this area at a high level and cover the specific work related to the thesis in the respective chapters. Research works to improve storage system performance can be categorized broadly into techniques for improving performance on the data path (or primary data I/O) and meta-data path (or meta-data I/O). Subsequent sections will describe in detail on some of the major research in these areas.

2.2 Performance handling techniques in the Data path

In this part we look in to some of the work in improving the performance on the Data path. The data I/O performance improvements can be classified as follows:

1. Techniques and algorithms for HDD based systems

2. Storage tiering and data migration techniques

3. Performance enhancement through Caching

4. Workload prediction and prefetching techniques and mechanisms

2.2.1 Techniques and algorithms for HDD based systems

While the commodity hard disks provide adequate storage volume at low prices, the disk access latency of using hard disk remains a major problem. This leads to the gap between the high speed CPU computation and low response time of storage system. The fact of the delay, compared with the CPU computation time, is several orders of magnitude slower [106]. The worst case results when the disk access contains a large percentage of random accesses leading to the disk heads busy moving from one track to another to seek data [64]. In order to improve the system performance and overcome the disk access latency with random accesses, several previous state-of-art work proposed different methods. Bhadkamkar et.al used block level disk reorganization by using a dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition, thus significantly reducing seek and rotational delays [14]. In FS2 [45], multiple replicas of data based on disk access patterns in file system was used to reduce latency. Most previous work on solving storage performance problems on disks were focused on throttling and manipulating I/O streams by prefetching or scheduling but do not consider migrations of data between different storage devices. In [37] the past file access patterns were used to predict the future file system requests, so that the data can be prefetched in advance before the request. In DiskSeen [31], disk prefetching policy at the level of disk layout was adopted to improve the sequential access of disk and overall prefetching performance.

Hybrid disks [17] utilizes a caching architecture by using non-volatile memory to cache read and write requests to HDD, and where the total capacity is equal to the capacity of the magnetic media. Combo drive [84] is a storage media device that combines flash memory as well as magnetic memory in a single storage device. By using the hybrid disks or combo

drives, the overall storage cost is kept low while the overall performance of the hard drive is improved compared to hard disk.

Even though these techniques based on HDD provide some improvements in performance, they are still limited by the inherent physical characteristics of the device. The rotational and seek latency impacts the overall performance of the drive. Hybrid drives are a promising alternative to the conventional HDD since they improve the performance significantly. Though SSD can substitute the conventional drives, they cannot entirely replace them due to the high cost. The advantage of Hybrid drives relative to SSD is evident from the table 1.1 from Chapter 1 (Lower cost and comparable performance). In this thesis, we introduce Hybrid drives as one of the storage tiers. While the Hybrid drive has been introduced for Desktop and Laptop environments to improve performance and save energy, little or no work has considered practical application in main stream Enterprise or large scale storage. The closest that we can come across is the work done by Timothy Bisson et al. in [16]. In this paper they utilize the hybrid disk's non volatile cache to improve the write performance or latency. Even though they can improve the latency by around 70%, the evaluation was done for Desktop workloads and they do not consider improving read IOPS through caching. In chapter 5, we focus on caching algorithms for optimal utilization of the Non-volatile cache inside the Hybrid drive. Although there are several caching algorithms in practice like LRU, LFU, ARC [69] and LIRS [47], these are mainly used for main memory (DRAM) caching. These methods cannot be used for caching inside the Hybrid drive since they increase the cache writes to the Flash in addition to the normal or foreground I/O as we have shown in our results. This severely effects the endurance of the NAND Flash as it has limited endurance. There are several works that try to improve the endurance of SSD through techniques like using disk based write caches [97], using buffering techniques with next generation NVM [101], and improving caching/tiering algorithms [134]. These techniques are applicable to hybrid storage with separate SSD and HDD devices. The Hystor

system [23], FlashCache [112] and FlashTier [94] makes use of SSD as a cache for HDD. But they suffer from huge overhead to maintain mapping tables external to the drive. Even if we ignore this overhead as this is handled by the Hybrid drive itself, the overhead in the block level cache mappings is still higher. The work in [78] tries to reduce the hot data monitoring and identification overhead by using random sampling of the I/O. But this is applicable only to hybrid storage systems with separate SSD and HDD.

2.2.2 Storage Tiering and Data migration techniques

As previously explained tiered storage and data migration techniques have existed for a long time both in research and in many of the commercial products from popular storage vendors. This is to mitigate the performance issues of HDD. The basic idea is to keep highly accessed or hot data in high performance devices like SSDs and infrequently accessed or cold data in low performance inexpensive devices like HDD.

The following are the main areas that are addressed in any data migration system [64] [78] [136] [39]:

1. Differentiating between Hot and Cold data to be moved between the tiers. This also involves differentiating between sequential and non-sequential I/O. Data with non-sequential I/O pattern is migrated to SSD for maximum performance.

2. Minimize the impact of the migration on application workload. This is the overhead involved in the migration itself. The algorithm or scheme should have minimum impact on the foreground I/O to the storage system.

3. Minimize the overhead on monitoring meta-data (frequency counts) and computation overheads. This can reduce the amount of system memory needed to keep this history data.

4. Resolve contention for higher tiers (like the NVM/SSD space) when hot data exceeds the capacity of the higher tiers. This requires optimization to place the data objects in such a way that the tiers are efficiently utilized.

Hot random offloading [64] is one of the recent works in this area that migrates files between SSD and HDD. It consists of three components: the data collector, randomness calculator and a migrator. The data collector performs the I/O monitoring and collects the usage/access statistics of files. This is input to the randomness calculator which determines whether the file is being accessed sequentially or non-sequentially. This addresses the first problem of identification of hot and cold data as well as the differentiation between sequential and random access. It uses a benefit value for the migration based on the randomness and frequency of the access to the files. Randomness is measured by counting the number of contiguous data segments accessed within a given file over a given interval. For example, if this count is one, then the file access pattern is completely sequential. The benefit value is calculated based on the following simple equation:

$v_i = f_i/s_i \times R_i$

where v_i is the benefit value, f_i is the access frequency, s_i is the size of the file and R_i is the file random access count calculated based on the method explained before. This value therefore favors frequently accessed small files with high degree of random access. Though this method seems to be efficient, it can only be applied at the file system level. This method cannot be used for the block level migration since we do not have a notion of a file. Moreover, this work formulates the Data allocation problem (DAP) as a knapsack problem. Since in our architecture, we need to handle multiple tiers, the same method cannot be utilized. The HRO method also leads to polynomial-time approximation greedy algorithm which is not able to find global solutions while we use dynamic programming to find global optimal solutions for the DAP. Dynamic programming is a pseudo polynomial algorithm. In our work described in chapter 4, we propose a multiple-stage dynamic programming (MDP)

to solve the Multiple Choice Knapsack Problem (MCKP) in multiple stages.

Hot data trap [78] though is a caching mechanism can as well be used for data migration. It addresses the problem of identification of hot and cold data with the help of bloom filters and sampling. It also reduces the computational overhead and memory space (problem 3) through its sampling based mechanism to determine whether data needs to be cached in SSD or not. This method works at the block interface level and therefore transparent to any specific file system.

An adaptive data migration approach in [136] pro-actively migrates data extents based on heat values. It migrates hot data extents in advance through estimation of an optimum look ahead window or time that maximizes the benefit for the future workload. This is based on computation of utility value of the migration for the future workload. Based on this utility value, the algorithm arrives at the optimal value of the look ahead time to initiate the migration for the future workload with minimal impact on the current workload.

Extent based dynamic tiering [39] is another recent work on dynamic tiering and data migration in a tiered storage system consisting of SSDs with SAS/FC and/or SATA drives. It uses an extent based monitoring and migration mechanism called EDT (Extent based dynamic tiering). The work describes two components of the system. EDT-CA (EDT configuration advisor) helps in arriving at a tiered storage configuration (capacity for each tier) to satisfy a given expected workload. EDT-DTM (dynamic tier manager) responds to the changing workload and dynamically migrates data between tiers accordingly. The primary goals of EDT-CA and EDT-DTM are to provide better performance at reduced cost compared to a SAS only storage system. The system also determines non-sequential data as those I/O having LBA (Logical block address) distances larger than 512KB from previous access.

In [105], a reinforcement learning algorithm is used to tune migration policies that move hot and cold data between tiers. The objective is to improve the average response time of

the system with the optimal migration policies tuned based on the recent access patterns.

Our work has some similarities to previous work described in Extent based tiering [39], Adaptive data migration [136] and Reinforcement learning based policies [105]. While these focus on data migration between only HDD and SSD, the approach in this thesis differs from these work by providing a general optimization solution for placement of data across multiple storage tiers (with more than 2 tiers) that maximizes their utilization based on the I/O workload.

2.2.3 Performance enhancement through Caching

Several caching algorithms have been proposed for storage, CPU memory and web servers in the past like the LRU, LFU, LIRS [47], CLOCK and ARC [69]. Each of these algorithms has their own benefits and behaves differently for each type of workload. The main component of any caching algorithm is the replacement policy employed when a cache is full. The most popular of them LRU, uses a simple policy of replacing the least recently used data in the cache. Generally, the common method of evaluating the efficiency of these caching algorithms is the hit rate. This determines how much of the I/O goes to the cache compared to the total I/O for a given interval of time. It is well known that the hit ratio for a given algorithm depends on work load characteristics and the cache size. For example while LRU is a simple algorithm, its efficiency drops due to cache pollution caused by a workload which performs a sequential scan. ARC [69]] is an adaptive algorithm which reacts to the varying workload based on not only the recency (like LRU) but also to the frequency of the access. BPLRU [50] explores buffer management algorithms which adapt modified versions of LRU and CLOCK algorithms respectively for SSDs. In spite of the efforts to improve the replacement policy, we find that all these algorithms do not provide any increase in performance after a particular cache size is reached for a given workload. Certain workloads does not react much to any further increase in cache sizes since most of the hits lands on the top

of the LRU queue. Other types of workloads can benefit in the increase in cache sizes that will result in increase in the hit ratio. This behavior is exhibited even within workloads as a function of time. We exploit this behavior to develop new cache control mechanisms that will be explained in chapter 3 of this thesis.

2.2.4 Workload prediction and prefetching

While performance gains can be achieved through data migration and caching, another common method is prefetching. Prefetching can improve performance by making the data available in high speed storage devices before it is accessed. This requires some form of predicting the access in advance. The following are the different categories of prefetching:

1. Heuristics based prefetching

2. Application aided prefetching

The prefetching algorithms and mechanisms also either fall under Block based or file based categories

Heuristics based prefetching: This is the simplest prefetching mechanism where the system predicts future accesses based on stable access patterns. One example of a simple access pattern is sequential access that is exploited by modern operating systems and/or file systems. Linux read-ahead [34] uses file offsets and request sizes to detect sequential accesses. It performs a read ahead of blocks from the disk before the application issues the actual request. When the actual request arrives the blocks will already be in memory and therefore the application sees a faster response time. Block Correlation mining [62] is a block based approach where the system detects correlation between disk blocks in the I/O stream. It also determines the sequence of blocks accessed. The assumption is that if a block is accessed all the related or correlated blocks will also be accessed. Therefore, it pre-fetches those correlated blocks before it is accessed. Through this method it achieves

7-25% reduction in I/O response times compared to other prefetching schemes. Collective prefetching [24] is similar to Block correlation mining [62] but is used in a HPC environment and the prefetching is done collectively across a storage cluster. The reason for using a collective approach is that individual storage nodes might not know the collective I/O pattern of the parallel application running on the multiple computation nodes. Parallel I/O prefetching [20] uses past pattern signatures (instead of traces) stored in a data base to detect application access patterns. The detected pattern is then used in predicting future accesses for prefetching to memory. This system assumes the access patterns are stable as expected in a HPC application.

Application aided prefetching mechanisms: In application aided pre-fetching, the application itself provides hints to the storage system to help in prefetching. Another method is an intrusive method that modifies the application (in the source code) itself by inserting probes that hint application accesses. In informed prefetching [82], the application provides future resource demands (like cache and pre-fetch buffer) and access patterns for the system to control the amount of prefetching. The algorithm uses a cost-benefit mechanism to allocate the amount of prefetching buffers for different applications based on the provided hints. Through this method, the paper claims to achieve around 20-83% reduction in execution times of applications like data base queries, scientific visualization etc. Sprint [91] uses an execution engine that runs in parallel with the application to access data in advance before the application issues the request. This is done by probing the source code of the application. New source code is created that issues just the I/O with all the processing stripped from the application. This code is run in parallel with the application through the execution engine. This method can achieve 2 to 15 times speedup compared to the normal execution of the application. In application directed prefetching [103],the algorithms made effective use of system primary memory by aggressively fetching as much data as fits in available memory.

Prefetching mechanisms are optimal only if there are stable access patterns or partially sequential. They cannot be utilized in large scale or enterprise workloads that have highly random access behavior and non-sequential characteristics.

2.3 Performance handling in Metadata path

So far we have seen techniques on improving the performance on the data path. The other part of the storage system is the Meta-data and Namespace management in distributed or clustered file systems. Compared to the performance improvement techniques for data, the techniques in this area are relatively few. The following are some of the techniques in this area:

1. Metadata partitioning and distribution

2. Metadata pre-fetching and caching

2.3.1 Metadata partitioning and distribution

We explored several large scale storage systems in section 1.6. In many of these storage systems the file data is stored in separate nodes from the meta-data. The meta-data in turn is stored on a single node as in Lustre [1] or distributed across a cluster of nodes as in Ceph [122]. These Metadata storage nodes have to handle operations that involves mapping from file names to corresponding data block or chunks in the data nodes, lookup of file attributes (like owner, creation/modification times) and checking of access permissions. When there are several clients accessing the system, these operations will overwhelm the meta-data nodes if the meta-data items are not properly distributed. Systems like Lustre have a single meta-data storage node to handle all the meta-data operations. This leads to a serious bottleneck and results in a single point of failure. But this is the most simple design.

It avoids many of the problems like maintaining consistent meta-data updates and concurrency issues. The distributed approaches like that in Ceph are more resilient to failures and provides better scalability. But this design leads to more complexity in terms of maintaining consistency, handling concurrent updates and also requires mechanisms of distribution of the meta-data across the cluster. Though the design is more complex, the distributed approach is more favored due to better scalability particularly when the system has to handle petabytes of data storage. Many of the early systems such as NFS [83], AFS [75] and Coda [93] utilize a static partitioning of the meta-data namespace in to specific servers. Though simple in design, this technique suffers from skewed meta-data load distributions. The other common method of placement and lookup of meta-data in a cluster of servers is Hashing. Recent systems like Ceph perform hashing at the directory levels and distributes the meta-data items accordingly based on the hash values. The HBA system [139] tries to reduce the complexity of meta-data lookup using Bloom filters to map filenames to their corresponding meta-data location in a meta-data server cluster. HBA also avoids the problems of directory hashing based schemes as in Ceph. In the hash based schemes, the system had to rehash and redistribute directories to different nodes when there is a failure of a particular node. The hash based distribution in these work can maintain optimal distribution across the meta-data cluster. But they also lead to lose of locality of the meta-data items. The meta-data locality is required for certain file system operations like entire directory lookup. Therefore, we need a meta-data distribution mechanism that can provide uniform distribution while trying to maintain the locality. We propose such a locality preserving meta-data distribution in chapter 6.

2.3.2 Metadata pre-fetching and caching

Some of the techniques for pre-fetching primary data also applies to meta-data. Since the size of the meta-data items are small, the techniques for meta-data perform aggressive

prefetching. The Nexus system [38] utilizes a relationship directed graph to pre-fetch meta-data items before it is requested. The relationship graph is obtained dynamically through access patterns in the application workloads. The meta-data items are pre-fetched based on predecessor-successor relationships in the graph. Since the penalty for a mis prediction is far lesser for meta-data than data, the algorithm performs aggressive prefetching. The reason being the size of the meta-data is in the order of few kilobytes (typically 4KB). The AMP system [63] utilizes the affinity between file accesses to perform Metadata prefetching. The affinity between files is obtained by mining history data from past access patterns. Using these methods, the system pre-fetches and caches these items on the client side thereby significantly improving performance of future meta-data requests. These prefetching methods can improve the meta-data performance for stable access patterns like that exhibited in Desktop workloads or small scale applications. The same techniques may not be applicable for large scale storage systems shared across multiple applications. Therefore, in this thesis we propose a solution that distributes meta-data across a server cluster that can handle the mixed meta-data workload characteristics of a large scale shared storage system.

2.4 Next Generation NVM Technologies

As NVM technologies are emerging, it has been a hot topic in systems architecture research to utilize them at different levels of computer memory/storage architecture. They are used as cache, as a replacement for the main memory and in storage system architecture in several research works. The work in [32],[100],[128],[137] explore the advantages in terms of performance and energy consumption in using PCM and MRAM technologies in the CPU caches (L1/L2 and main memory). In [57],[88],[138], NVM as an alternative to DRAM has been explored in terms of advantages in reduced energy consumption, higher density and improved performance. In this thesis we consider NVM on the storage stack or storage system architecture as in [52],[79]. These work mainly focus on the future of replacement

of the current Flash technologies with NVM. In this thesis we provide a hybrid solution where a combination of NVM with Flash can significantly improve performance while being practical in terms of cost.

Chapter 3

Storage System Performance Enhancement With Cache Optimizations

3.1 Introduction

In this chapter we look into methods of optimizing cache allocation to multiple applications sharing the storage system. In particular, we address the problem of dynamic partitioning of high performance next generation NVM device used as a cache for the conventional SSD. We show through simulation experiments that such a hybrid combination of next generation NVM and SSD can improve performance of applications significantly. Complimenting this work in section 3.6, we consider the problem of workload interference among different applications sharing a distributed storage system. Specifically we consider this problem in the context of a virtualized environment running multiple applications sharing a distributed storage system with Virtual Disks (VM disks). Through our proposed distributed caching and I/O redirection approach we show how the interference among competing VM disk workloads can be reduced. This is described in detail in section 3.7

3.2 Next Generation NVM As Cache For SSD

Emerging storage media such as STT-MRAM and PCRAM are likely to soon replace current NAND Flash technologies due to their very high throughput and low latencies. NAND Flash based SSDs though offering superior performance compared to hard drives, still have problems like poor write performance and endurance. This poor write performance of NAND Flash is due to the longer erase time required before writing to the memory cells. The next generation NVM like STT-MRAM or PCRAM have very low latencies close to that of DRAM. Unlike Flash, these devices support in place writes/updates and endure longer because of their higher write cycles. But these new storage media are unlikely to replace Flash entirely in the near future due to limitations in current process technologies, fabrication costs and densities [101]. Therefore, recently some of the research work [101] are exploring a hybrid combination of SSD and NVRAM as an alternative solution which could significantly improve the performance of the storage system.

In [101], PCRAM was used as a log buffer to the data on NAND Flash. Small updates to the data on Flash are absorbed by the log buffer and are done in place in the PCRAM buffer. In this work, we focus on a hybrid caching strategy for multiple applications with a small NVM (PCRAM or STT-MRAM) backed by large Flash capacity. Since the capacity of the current PCRAM or STT-MRAM technology is very limited, our main idea here is the method that adapts a minimum size of the NVM cache for each application in a shared environment. This size is adjusted dynamically based on the changing workload pattern and application's desired latency configured as a QoS (Quality of service) parameter. The latency in turn translates to a target cache hit rate achieved through a efficient cache size control scheme. The motivation behind our design is that through preliminary workload analysis, we have found that each workload has specific cache hit characteristics or histogram. The cache hit rate histogram [67] is the frequency of the hits to blocks at a given depth in the LRU queue. This varies both within and across different application workloads.

The control scheme works based on determining the hit rate histogram to a virtual shadow cache and dynamically adjust the physical NVM size based on the target or goal hit rate. The goal hit rate being determined by the latency requirement of the application workload. While the techniques presented here is applicable for a DRAM cache, the other main advantage is that writes/updates in the cache can be persisted for longer periods. This improves the reliability of the NVM and Flash Hybrid storage. It should also be noted that the control scheme and mechanism described in this work are independent of the type of NVM technology (either STT-MRAM or PCRAM) used. We assume a certain device performance characteristics for our simulation ([107], [19],[57]). Based on this, we perform simulation study of our control schemes and analyze our results. Through our HCache control scheme we can handle application workloads at $30\% - 50\%$ reduced cache sizes compared to static log partitioning schemes with LRU. HCache achieves this by adapting the cache or log size through careful monitoring of the application working set with the help of a virtual shadow cache. HCache uses NVM as a read/write log for the Flash pages. Throughout this chapter we use the terminology read/write log and cache interchangeably.

As shown in Fig. 3.1, by caching Flash pages in NVM we can drastically improve the read/write performance of the Flash based SSD. The hybrid device can also provide better reliability since small writes to the Flash pages can be retained in the NVM for a longer period of time compared to a DRAM cache.

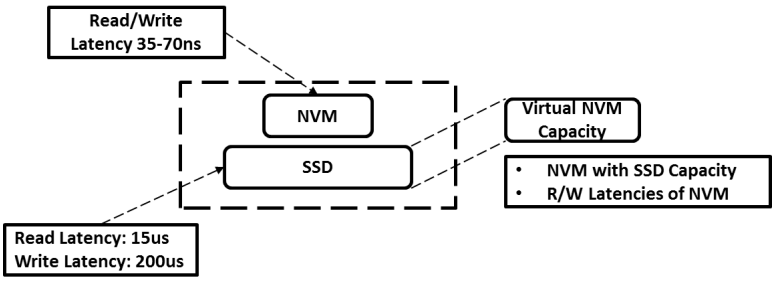

Fig. 3.1 Hybrid device with SSD and non-volatile memory like STT-MRAM/PCRAM

The advantages of such a hybrid caching scheme becomes significant in random small

write intensive workloads like that exhibited by OLTP (Online transaction processing). Our simulation results based on the OLTP traces [121] shows that we could achieve almost the same hit rates as that of standard caching algorithms at $30-50\%$ reduced cache sizes. The main contribution from this work is the HCache control algorithm. The algorithm uses a feedback mechanism through a virtual shadow cache to adjust the size of the NVM dynamically based on the changing working set of the application.

3.3 Background On Cache Allocation And Replacement Algorithms

Several caching algorithms have been proposed for storage, CPU memory and web servers in the past like the LRU, LFU, LIRS [47], CLOCK and ARC. Each of these algorithms has their own benefits and behaves differently for each type of workload. The main component of any caching algorithm is the replacement policy employed when a cache is full. The most popular of them LRU, uses a simple policy of replacing the least recently used data in the cache. Generally, the common method of evaluating the efficiency of these caching algorithms is the hit rate. This determines how much of the I/O goes to the cache compared to the total I/O in a given interval of time. It is well known that the hit ratio for a given algorithm depends on work load characteristics and the cache size. For example while LRU is a simple algorithm, its efficiency drops due to cache pollution caused by a workload which performs a sequential scan. ARC [69] is an adaptive algorithm which reacts to the varying workload based on not only the recency (like LRU) but also to the frequency of the access. Debnath et al. [28] and Kim et al. [50] explore buffer management algorithms which adapt modified versions of LRU and CLOCK algorithms respectively for SSDs. In spite of the efforts to improve the replacement policy, we find that all these algorithms do not provide any increase in performance after a particular cache size is reached for a given workload.

Certain workloads does not react much to any further increase in cache sizes since most of the hits lands on the top of the LRU queue. Other types of workloads can benefit in the increase in cache sizes that will result in increase in the hit ratio. This behavior is exhibited even within a single workload as a function of time. The main idea behind this work is to adapt the NVM cache log size to this varying work load behavior. This is done using a efficient control mechanism that works on the basis of how well an application reacts to changes in the cache size. For example, the control algorithm does not allocate more cache space to those types of workloads which do not react well to the increase in the size. This control is done both within and across applications sharing the hybrid device.

Our work is related mainly on some of the research work on write buffer management in SSD. The works by Debnath et al. and Kim et al. [28, 50] explore the write buffer management in DRAM inside SSD. They are basically modified versions of the general LRU and CLOCK algorithms adapted for SSD devices. [50] explores the idea of using the internal RAM in SSD for improving the random write performance of Flash. [28] follows a similar approach but adapts a different caching scheme which considers the recency and block level space utilization of the Flash device.

Our work utilizes a non-volatile memory technology as a read/write buffer instead of DRAM. This facilitates longer storage of the NAND Flash pages in the non-volatile memory when the pages are hot. Multiple writes to the hot pages are absorbed in the NVM space for a longer period of time without sacrificing on the reliability. [101] follows the same approach and use the NVM space as a log buffer for writes to the Flash pages. Even though these work have analyzed different techniques of write log buffering for NAND Flash none of these has addressed the problem of cache sizing for different types of workload. Our work primarily differs from these on how we manage the limited NVM space shared across multiple application workloads. We achieve this through a control mechanism which utilizes a hard reference (the shadow cache) as a control component. The write log buffer for

each application workload is chosen based on the application specification which indicates how much the hit rate to the cache can differ from the virtual shadow cache. The work presented by Sehgal et al. [95] addresses the QoS problem through SLA specification based on latencies for each workload. Based on this the size of the cache is adjusted through a control loop. But we show in the next section that the hit rate to the cache itself depends on the nature of the workload and latency cannot be used as an absolute parameter in the control loop. In our case we use the hit rate with respect to a virtual reference as the control component. Our algorithm adapts to the changing workload within and across applications thereby efficiently distributing the limited non-volatile cache space considering the hit rate deviation from the reference cache.

3.3.1 Application Working Set

To understand the working set data variation for each type of workload we took snapshots of a simple LRU queue with published and generated traces. The size of the LRU queue was set to $10MB$. Figures 3.2 shows the heat map of blocks for an LRU queue with the SPC OLTP trace [121]. In the heat map, dark regions denote very high hits to that region of the cache where left of the figure represents top of the LRU queue. Accompanying graph 3.3 shows the data points for cache depth at which 90% of the total hits is achieved. Similarly figures [3.4,3.5] and [3.6,3.7] show corresponding heat maps/histograms for the Microsoft exchange and Web search trace respectively. From the heat map and the hit rate histogram for the financial and the exchange trace we can see that most of the time 90% of the hits are registered within the top 50% of the LRU queue. For the financial trace we find that most of the hits are registered within 30% depth of the cache. In the web trace case the hits are more spread out throughout the cache space. From this observation arrive at the result that the financial and exchange workload does not react much to increase in cache size whereas the web trace will definitely benefit from increasing cache sizes. Furthermore this behavior

varies as a function of time.

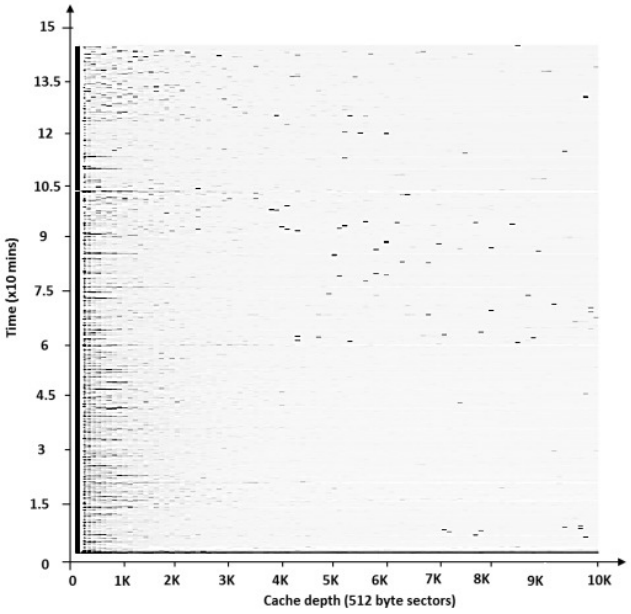

Fig. 3.2 Heat map of LRU cache with Financial1 trace

The following are the main observations from this:

1. At any point of time, most of the working set of the application resides in the top of the LRU queue. The working set includes both the new data (new writes) and old data (reads and updates). For example for the financial trace, roughly 90% of the working set gets a hit in the top 30% of the LRU queue.

2. This working set changes as the application progresses. When the cache is full, part of the working set in the current time period which is not in the cache will evict part of the working set in the previous time period. This in turn depends on the I/O pattern of the workload. For example when there is a repeated access pattern to the same blocks the working set remains almost the same for the current and previous time epochs.

3. Access patterns which exhibit smaller inter access time between accesses to the same block leads to most of the hits at the top of the LRU queue.

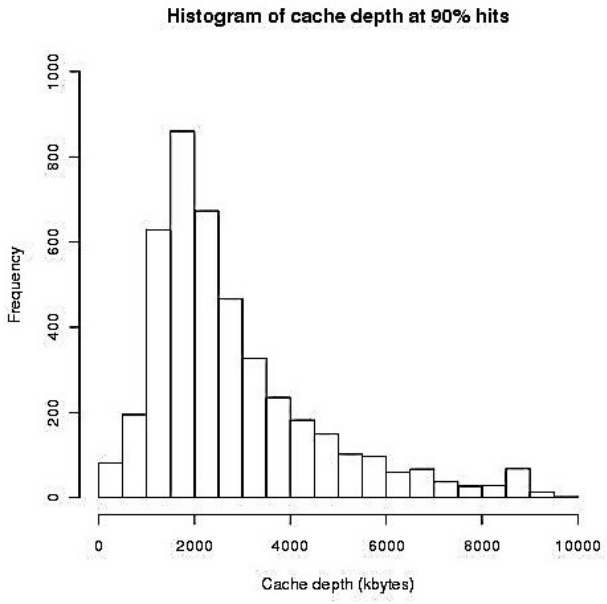

Fig. 3.3 Hit rate Histogram for Financial1 trace

4. Within a workload as the working set changes the cache size can be increased if the application benefits from the increase during that period of time. If it does not benefit as a consequence of a constant working set we can reduce the size of the cache.

We show that by closely tracking the hit behavior on a LRU queue for each application we can adapt an optimum cache size to the work load variation. This optimum cache size varies both across applications and within application as a function of time. By utilizing such optimization mechanism we benefit both in application performance improvement and efficient usage of the scarce NVM. In addition, the algorithm only allocates what is required by the application based on the current hit rate histogram and the goal hit rate. From our evaluation in Section 3.5 we have found that for certain workloads the cache can be operated at a lower size compared to static partitioning. We therefore partition the very limited size of the NVM cache among different applications based on the desired QoS and track the current workload pattern. The main challenge here is the dynamic tracking of the changing working set of the application. This is where we use the HCache feedback control scheme

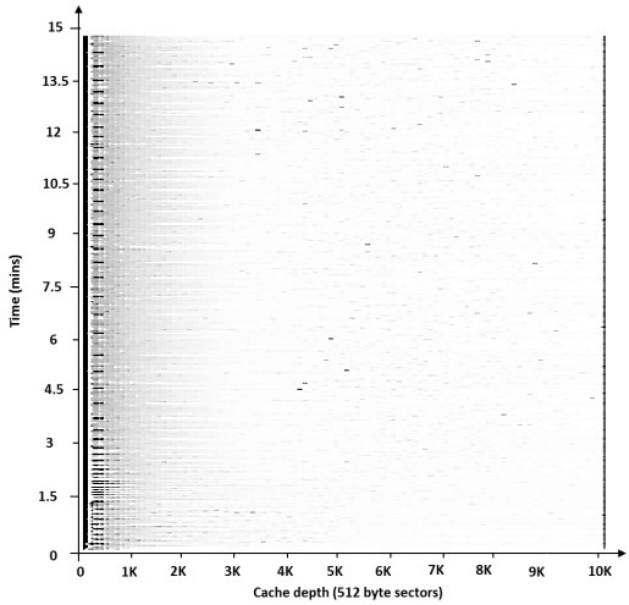

Fig. 3.4 Heat map of LRU cache with Exchange trace

to dynamically adapt the NVM cache size that works best for the current working set of the application. Since the virtual cache hit rate histogram reflects the access pattern of the application, HCache can achieve this dynamic adaptation.

3.4 Dynamic Cache Allocation Through HCache

Fig. 3.8 shows the overview of the HCache architecture. Since NVM like PCRAM and STT-MRAM is not readily available in the market, we simulate with DRAM since its performance is close to that of these types of memory. The NVM buffer acts as both a read and write log for small writes to the Flash region. The combined NVM-Flash hybrid device can act as a storage tier to a back-end array of hard disks. This work deals only with controlling the size of the NVM buffer and therefore we adapt the techniques used by Guangyu et al.[101] for the log management. Unlike [101], where the focus was on improving the lifetime and energy consumption of the Flash region, we use the log for improving read/write performance or latency based on careful NVM allocation. The NVM buffer is shared across

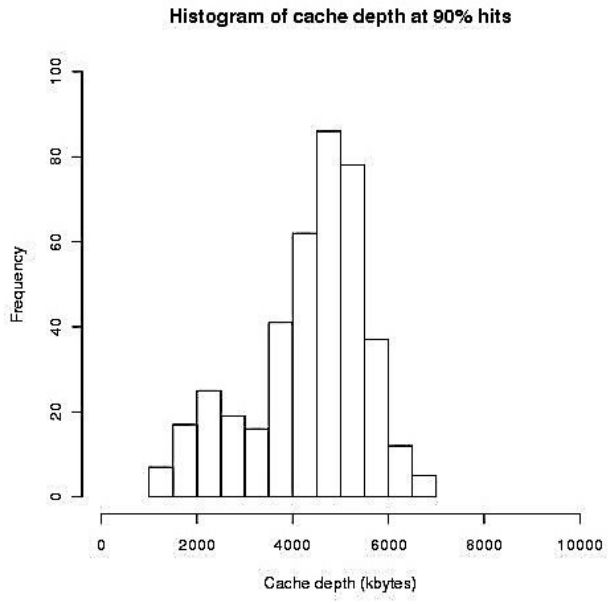

Fig. 3.5 Hit rate Histogram for Exchange trace

the application workloads W_1, W_2, W_3. The buffer stores data from Flash in terms of pages in LRU order. The size of these logs is adjusted dynamically by HCache control algorithm. Any update to the Flash page sector is updated in the NVM and kept in a log structured format. A table in the NVM keeps track of the mapping between the pages in the LRU log and the Flash pages. The Flash page size of 4KB is assumed. We maintain a LRU queue of linked list of pointers to the pages. A hit to a page moves the corresponding pointer to the head of the list. We also maintain a bit vector of size 16 for each page to mark a sector within a page as clean or dirty. This is later used by the eviction mechanism to decide whether a page can be evicted to memory. A page with all the 16 bits set to zero is simply invalidated without any update to the Flash. In addition to the primary LRU queue the main component in our scheme is the shadow log [69] which stores just the addresses of the pages and the hits. The size of this shadow LRU queue is set to the physical size of the NVM available in the system. For our experiments we assume a physical size of the NVM of 256MB.

At a high level the control algorithm works as follows:

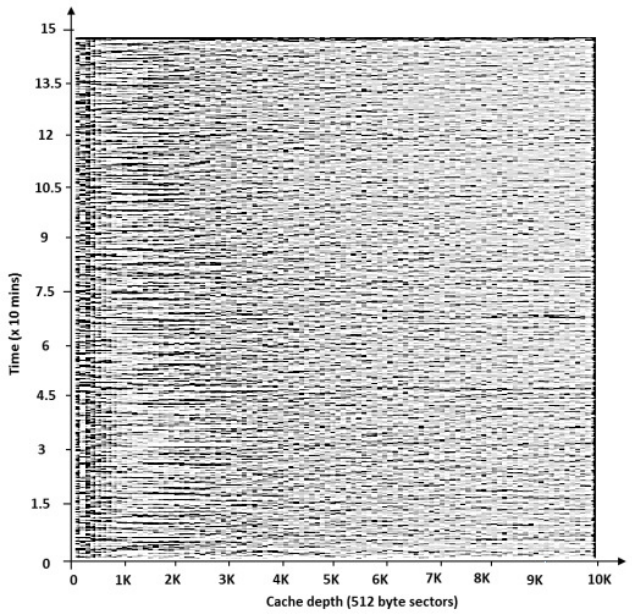

Fig. 3.6 Heat map for web search trace

1. The shadow cache acts as a virtual cache operating at the full available capacity in the system. Hit data from this cache serve as a reference to tune the maximum size of the actual LRU log.

2. The shadow log is partitioned into a number of LRU queues each with 10% of the physical size of the NVM in the system (around 25MB in our case). Each partition only holds the page id, sector bitmap and does not store the pages. Each partition also independently registers hits to that part of the log.

3. For each time epoch a simple counter is used to measure the hits to each of the shadow cache and real cache. For some of the workloads (as in the financial trace) most of the time the hits go to the top of the LRU log and therefore the top queues register more hits. In other types of workload the hits can go to the tail end of the log in which case the bottom LRU queues in the chain registers more hits.

4. The cache size either ramps up or down at the tail end of the LRU log by a fixed amount proportional to the hits registered in the chain of queues. For example if the

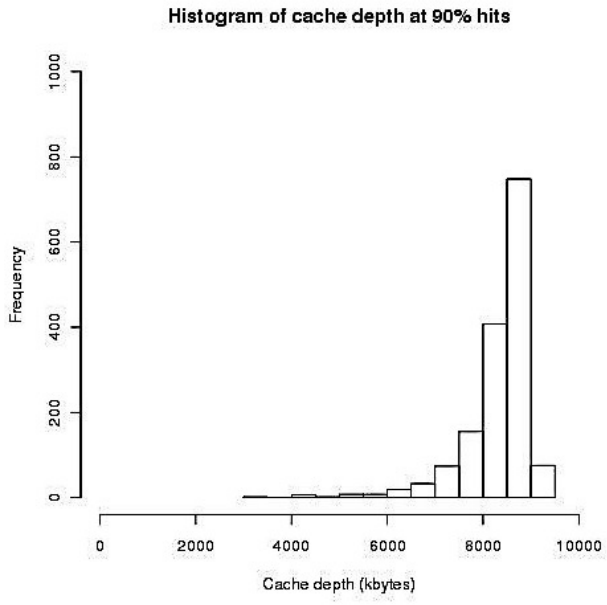

Fig. 3.7 Hit rate Histogram for web search trace

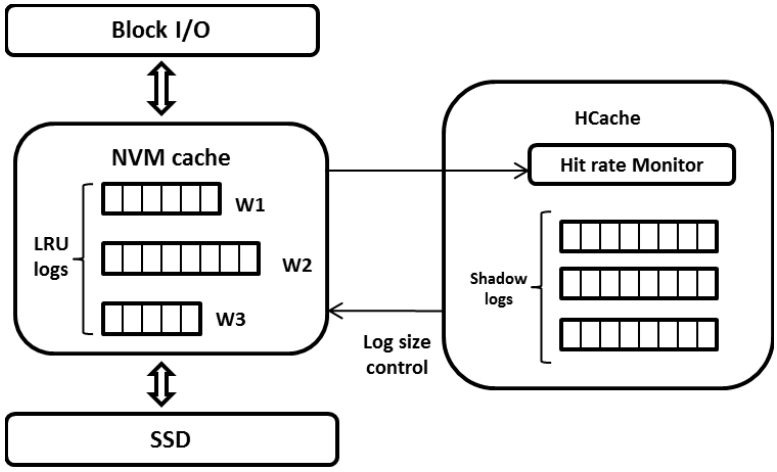

Fig. 3.8 HCache: The high level overview of the HCache cache control system

desired hit rate is achieved within 20% of the shadow queue depth (or the first two LRU queue in the chain), then the physical log size is ramped down to that size. On the other hand, if the desired hit rate can be achieved only at 60% of the queue depth than the physical log size is increased to that value. The desired hit rate is computed based on the latency value from 3.1 for a specified target application latency (l_t in the equation). During a reduction in cache size only those pages that have been modified

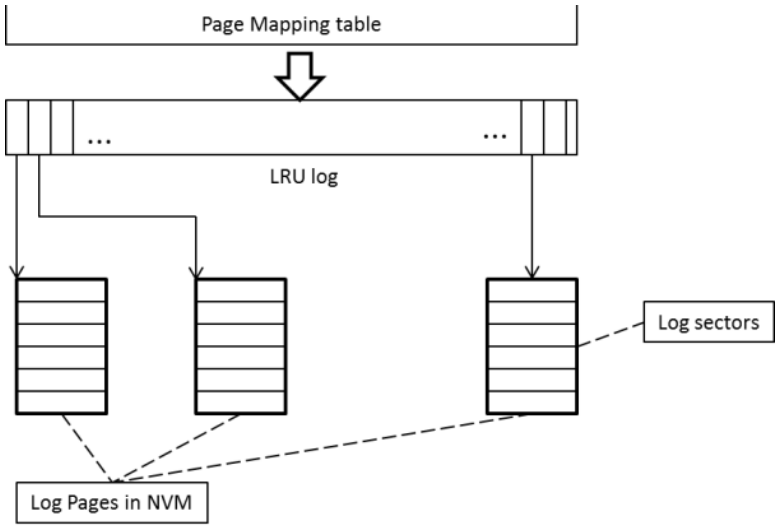

Fig. 3.9 Figure showing the LRU log in Non-volatile memory. A page size of 4KB and sector size of 512 bytes is assumed. The log contains pointers to the pages in NVM. A mapping table in the same non-volatile memory resolves logical address to physical location in the NVM

are written back to the Flash. This is identified based on the bitmap maintained in the physical LRU queue. Those pages which are cached for read are simply invalidated. The Freed log space is added to a free list pool for re-allocation and re distribution to other workloads.

The cache size for a given goal latency/target hit rate for each application is obtained based on the following:

1. For each workload since the latency of a given workload is directly related to the hit rate to the NVM log, we determine the target hit as follows:

$$h_t = (l_{ssd} - l_t)/(l_{ssd} - l_{nv}) \tag{3.1}$$

where l_{ssd} and l_{nv} are the latencies of SSD and NVM respectively, and l_t is the application target latency. In a given time interval, the control algorithm adjusts the physical cache size based on the depth of the virtual shadow queue at which the desired num-

ber of hits h_t is achieved. The hit rate values are the maximum that can be achieved for that respective workload. The desired hit rate cannot be greater than the achievable hit rate for that specific workload characteristics. If the desired hit rate cannot be achieved HCache tries to maintain the maximum of the desired and the achievable hit rate.

2. The hits registered in the shadow LRU queues are added from the top to bottom for each LRU queue in the chain.

$$H_t = \sum H_n \tag{3.2}$$

where H_t is the total hit from the top to the nth queue in the LRU chain.

3. Now this value of the H_t should satisfy the following:

$$H_t \geq h_t \times IO_{total} \tag{3.3}$$

Where IO_{total} is the total number of IO in the previous time period.

4. The final physical size of the LRU queue is obtained based on where in the chain the above criterion is satisfied. If at the Nth queue we obtain the desired number of hits than the queue size will be

$$S_p = \sum [S_t \times W_n] \tag{3.4}$$

where S_p is the physical LRU queue size and S_t is the size of each LRU queue in the shadow log. The total size of the chain of LRU queues is the same as the total available NVM size. W_n is the weight of the nth queue in the chain. We chose a linear decrease in the weight from the top to bottom of the queue. The need for a linear decrease in the weight is that we assume that a hit to the top end of the LRU queue is likely to get another hit compared to the lower end of the queue. This is based on the expected behavior of workloads in a LRU queue. Instead of directly using the calculated cache

size in the current period, we use a moving average of the past values, so that we avoid large fluctuations that might happen for certain workload [95].

Example Calculation: If we assume that the total IO in the current evaluation period is 1000 and assuming a desired hit rate of 80%*, if the first two queues register a total of 800 hits, then the desired queue size at that instant will be the total of the size of the first two queues in the shadow log chain. This will be roughly around 50MB in our case as the size of each queue in the shadow log is approximately 25MB (*10% *of the size of the NVM size of 256MB).*

Based on our initial analysis of workloads in section 3.3, we found that the depth at which we can obtain the desired hit rate varies both within and across workloads. We make use of this chain of virtual shadow queues to obtain the histogram of the hits to the LRU queues. This provides a reference for the algorithm to adjust and adapt the real queue accordingly to the changing workload pattern. In common cases the hit-ratio is a monotone function of the cache size according to the inclusion property of LRU [67]. The LRU queues utilized should capture many workload pattern which is mostly monotonic in the hit rate behavior. Non-monotonous workload characteristic should not be common at least for storage. Such characteristics may exist for non-storage workloads like that in a CPU cache.

Algorithm 1 shows the basic control mechanism for the cache size for each workload.

```
1  foreach ith workload do
2      Obtain h_t from equation 3.1;
3      foreach queue q_i do
4          while H_t < h_t × IO_total do
5              H_t = H_t + H_i
6          end
7      end
8      S_p = ∑[S_t × W_n] ;
9      Obtain running average of S_p for last n samples ;
10     C_i = S_a
11 end
```

Algorithm 1: Control algorithm to adjust cache size of various workloads.

Where H_i is the hit rate at the ith queue, S_p is the physical cache size, S_t is the size of the LRU queue in the shadow log (as in equation 3.4), S_a is the running average of S_p, C_i is the final cache size for the ith workload and the other parameters are from equations 3.2, 3.3,3.4. The algorithm therefore tries to maintain the hit rate h_t for the desired latency. The I/O is serviced from the NVM at this hit rate h_t. The following describes the scenarios in which I/O is served by the backing Flash storage:

1. Any 512 byte block read from the file system tries to fetch a page from the NVM. If there is no mapping page found, the entire page in which the block is present is fetched from Flash to the NVM and cached in the LRU log. We assume that spatial locality in the data is maintained by the backing Flash storage and adjacent sectors of the same page are always likely to be accessed together. All further writes to the same sector and its neighboring sectors in the same page will now be directed to the NVM LRU log.

2. The pages containing the updates/writes to sectors in the tail of the LRU queue are flushed to the Flash storage when the workloads cache partition is full or the maximum log size is adjusted. It has to be noted that the rate at which the dirty data is written back to the Flash will be almost the same as the miss rate of the workload which is close to that of the shadow log cache. Therefore introducing HCache control does not increase the write back rate to the backing Flash.

3.5 Performance Evaluation Of HCache

Evaluation of the system was performed with a simulation of the LRU log with the HCache control algorithm. We used combinations of workloads to analyze the effectiveness of the algorithm in tracking the workload pattern and the corresponding cache optimizations. Here we chose these 3 workloads in parallel to show how the control algorithm tracks the cache

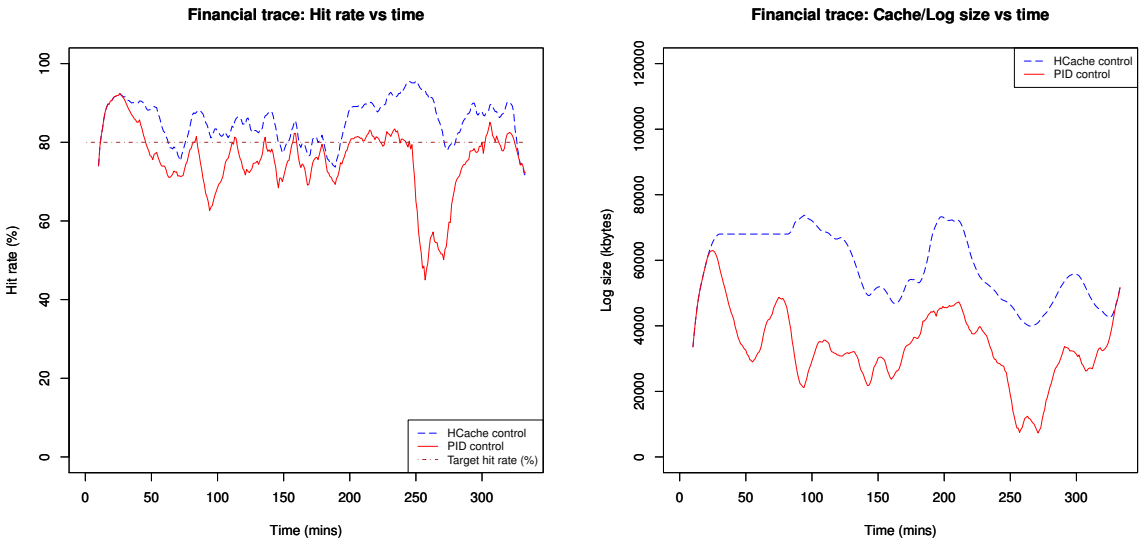

Fig. 3.10 Financial (OLTP) trace showing Hit rates and cache size variation over time for HCache and PID control.

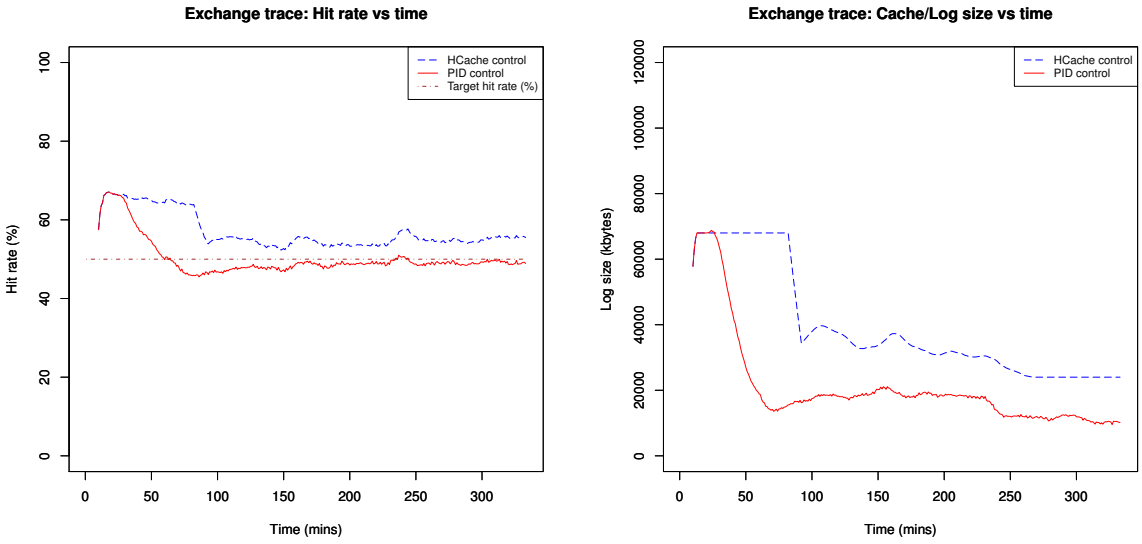

Fig. 3.11 Microsoft exchange trace collected using Loadgen showing hit rates and cache size variation over time for HCache and PID control.

demand of each application. The number of parallel workloads can be extended for the general case. We selected the Financial (OLTP) traces from UMass repository [121], MSR trace [119] from Microsoft and Microsoft exchange traces collected from the Loadgen software

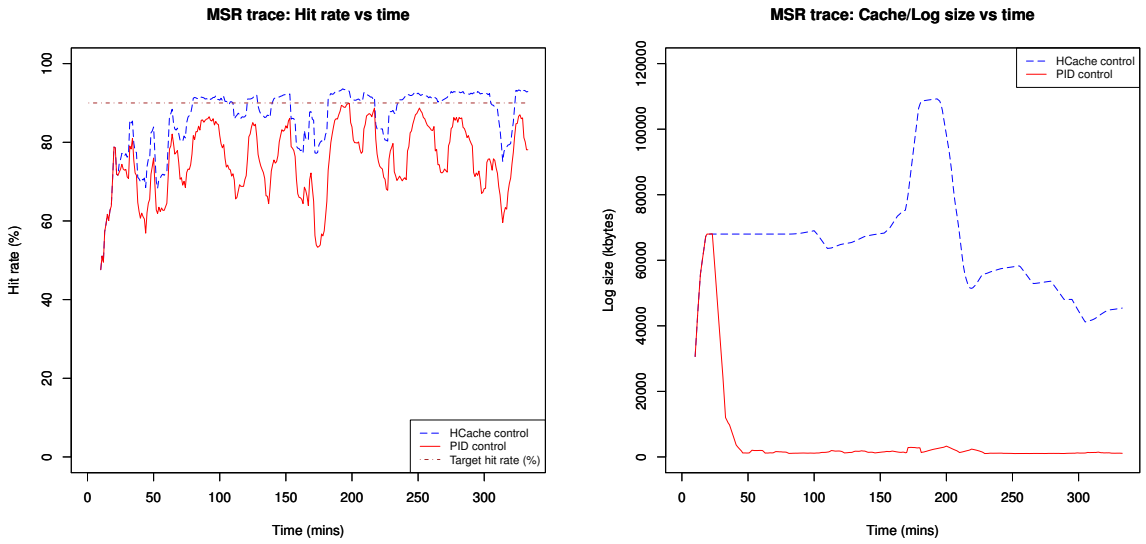

Fig. 3.12 MSR trace showing hit rates and cache size variation over time for HCache and PID control.

on Windows. The hit rate targets for each workload are set to 90%, 50% and 80% respectively for each of these 3 workloads (in that order). We compare our control algorithm to the PID (Proportional, Integral and Differential) control generally used in many other works [36, 54] as a mechanism to control the size of a storage cache to enforce a given QoS. The control mechanism described in [95] uses error in the desired latency as a feedback parameter to the proportional controller. In [36, 54] the error in hit rate is used as the control parameter to adjust the cache size.

The first set of graphs (depicted in Fig. 3.10)) shows the results from running the Financial or OLTP trace from the UMass repository. The first plot shows the hit rate variation as a function of time. It can be seen that most of the time HCache (dashed line) can meet the desired hit rate (the horizontal dashed line). Even though there are oscillations in both cases, HCache can re-adjust and stay close to the goal hit rate. The PID control can also meet the target hit rate, but in many cases the hit rate dips to more than 10% below of the desired value. At one point it even dips to nearly half or 40% of the desired rate. The second

Table 3.1 Average latency and latency violations for different workloads

Workload	**Latency (microseconds)**		**Latency reduction in HCache(%)**	**Latency violations (% of time)**	
	PID [36, 54, 95]	HCache		PID	HCache
Financial	write: 45 Read: 3.3	Write: 24 Read: 1.8	46%	63%	23%
Exchange	write: 98 Read: 7	Write: 85 Read: 6	14%	60%	1%
MSR	write: 49 Read: 3.6	Write: 27 Read: 1.9	45%	82%	33%

plot in the graph shows the variation in the size of the cache over time. We can see that both algorithms try to adjust the cache size based on the workload demands. The PID control can operate at a lower log size but this is due to the specific control parameters chosen. This leads to early evictions of the pages that are needed in the future. HCache can adapt the cache size to the workload variation because of the knowledge of the hit rate histogram in the virtual or shadow cache.

The second set of graphs (given in Fig. 3.11) shows the results of running the exchange workload. Here for the chosen control parameters, the PID control does not have many oscillations in the hit rate. But most of the time, the hit rate is slightly below the desired value. The PID control can therefore operate at a lower average value of the NVM cache size. The third set of graphs is for the MSR workload. PID control for the chosen control parameters performs worst for this workload. There are large oscillations in the hit rate due to the PID control mechanism settling in on a very low size of the cache. Even though cannot be seen in the graph, there are large oscillations around this small value of the cache. HCache can adapt to the variation in the workload pattern better and tries to remain close to the target rate. But even for HCache at one point the hit rate drops to 10% below the desired level (the period between 150 to 200 minutes) but the algorithm responds by increasing the cache size to allow for more hits.

Table 3.2 Percentage reduction in Log size for the different workloads

Workload	**PID [36, 54, 95]**	**HCache**
Financial	57%	28%
Exchange	73%	51%
MSR	92%	20%

Table 3.1 compares the performance of the PID and HCache control mechanism. The first column shows the average read and write latency measured for the three traces. We can see that HCache can achieve around 14% to 46% reduction in average latencies over the PID control. The third column in the table shows the percentage of time that there is a latency violation over the entire period of the run (around 5 hours). HCache has the least number of violations since it can adapt the cache size more efficiently through a real reference (through the shadow cache). The table 3.2 shows the cache size reduction compared to static partitioning. In all cases, HCache operates at a higher average cache size compared to the PID control. However the PID control mechanism settles on the wrong values for the size of the cache and this leads to a large number of violations. Even if we modify the PID control parameters, it cannot be set to the same value for all workloads. This is evident in the worst performance for the MSR trace where it artificially settles to a very low value for the size of the log.

Although the PID control technique is effective in tracking the changes in the application behavior, it requires careful tuning of the control parameters (K_p, K_i, K_d).). Moreover these parameters are sensitive to workload characteristics and therefore require adjustment for each workload type. This will complicate the design and implementation of the system. Goyal et al. [36] have shown the case of large oscillations in the cache size if the parameters are not tuned properly for both the proportional and PID controller. HCache completely avoids this problem and works on a simple algorithm that computes the cache size with respect to a virtual shadow cache.

Table 3.3 shows the comparison of our method HCache with some of the feedback con-

Table 3.3 Control mechanism comparison with respect to HCache

Control algorithm	Time complexity	Meets goal?	Accuracy
PID [54, 95]	O(1)	No	Yes
Retrospective Queue [36]	O(N)	Yes	Yes
HCache	O(1)	Yes	Yes

trol techniques [36, 95] for the desired characteristics. [36] tracks the evicted pages in a LRU and the access frequencies of past accessed blocks. It uses this past access history to increase or decrease the size of the physical cache to meet the specified QoS. But this method needs to scan the entire blocks in the Retrospective or history queue to identify blocks that are frequently accessed. It also provides decay on the counters to track the aging of the evicted blocks. Even though it is not mentioned in the paper, this might take $O(N)$ or at least $O(logN)$ time based on the size of the history. [95] uses a proportional controller to enforce the desired hit rate to the cache. It continuously adapts the size of the cache based on the error between the observed hit rate and the desired hit rate. It has been shown in [36] that the stability of a PID control for cache sizing is poor and can lead to large oscillations. This requires careful tuning of the PID control parameters or PID constants. These constants might also need to be tuned for each type of workload. HCache uses a realistic reference through a shadow/virtual cache and uses the hit rate to this cache to control the physical size of the log. This completely avoids the problem of tuning constants like those needed for a PID control. HCache dynamically adapts to the changing workload through the virtual reference LRU queue. The meta-data overhead (roughly around 1MB for a 256MB NVM cache) required for maintaining the queue is insignificant since we only keep track of the sector or block numbers. Compared to the retrospective control mechanism described in [36], HCache can run in $O(1)$ or constant time since we do not scan the entire shadow cache to determine the size of the physical log. A simple computation on the hit densities to each LRU log in the chain of virtual cache is performed to arrive at the physical cache size.

3.6 I/O Workload Interference In Shared Storage Server

While the NVM/SSD hybrid device can improve the application performance in a shared environment, we still need to address I/O interference among these competing workloads at the storage server. Several techniques for managing performance in shared storage and virtualized environments have been studied and explored before. These techniques include performance isolation [108], proportional sharing of storage resources [40], caching [36], proportional allocation of SSD resources [95], VM migration [41] and dynamic instantiation of host side virtual cache appliances (VCA) [10]. Even though SSDs are being extensively used as a cache in some of these work for their high random I/O performance, they too have high variability in performance especially for writes. Many of these papers deal with meeting SLO (Service level objectives) goals like IOPS (I/O per second) and average I/O latencies under VM consolidation. But little work has been done to address the problem of high variance in I/O latencies (like the 90th percentile latency variation from average) as applications are moved from non-consolidated environments to virtualized environments especially during periods of peak loads. The major cause of I/O latency variance in multi-tenanted environments is due to the I/O interference among the workloads sharing the storage system. This interference effect has been shown to exist both for HDD as well as SSDs [61]. For non-sequential workloads in HDDs, the latency contribution comes from both the seeks and during heavy loads due to I/O request queuing at the device [42]. For SSD, the variance in the latency mainly occurs when writes interferes with reads. Under write intensive periods of an I/O workload, this variance in latency is dominated by the program-erase cycles and garbage collection (GC) activity. For example when a program/erase cycle is running in the SSD, subsequent read I/Os had to wait till it completes [127].

To address the problem of large I/O latency variation, we propose a dynamic I/O redirection and caching mechanism called VirtCache. VirtCache can pro-actively detect storage device contention at the storage server and temporarily redirect the peaking virtual disk

workload to a dynamically instantiated distributed read-write cache. We have implemented this system on top of the GlusterFS file system [111] which is widely used as a backing store in OpenStack [117]. We show that by dynamically redirecting workloads from contended storage devices to server caches across the cluster we could reduce the variance in 90th percentile latency under peak workloads. While the caches are always there to be utilized across the server cluster, the peaking I/O load is not uniformly distributed. We address this by redirecting the concentrated load from virtual disk files on specific storage servers to other server's less utilized caches. The following are the main contributions from this work:

1. A distributed storage side caching mechanism called VirtCache to minimize I/O latency variation during peak VM workloads

2. Mechanism to Pro-actively detect storage device contention, I/O tracking and redirection to the distributed cache

3. Implementation of VirtCache in a popular distributed file system (GlusterFS) and detailed evaluation showing the benefits.

3.7 VirtCache Architecture

The VirtCache architecture was designed around the existing GlusterFS storage stack. The modified GlusterFS stack with VirtCache is shown in the Fig. 3.13. We modified the IO-threads translator in GlusterFS to incorporate our caching mechanisms. The VirtCache system consists of the VirtCache I/O handler, Workload interference detector and data logger. We also modified the Qemu-Gluster block driver to incorporate the I/O redirection and dynamic allocation of memory space in the cluster. The following sections describe in detail each of the various components in VirtCache.

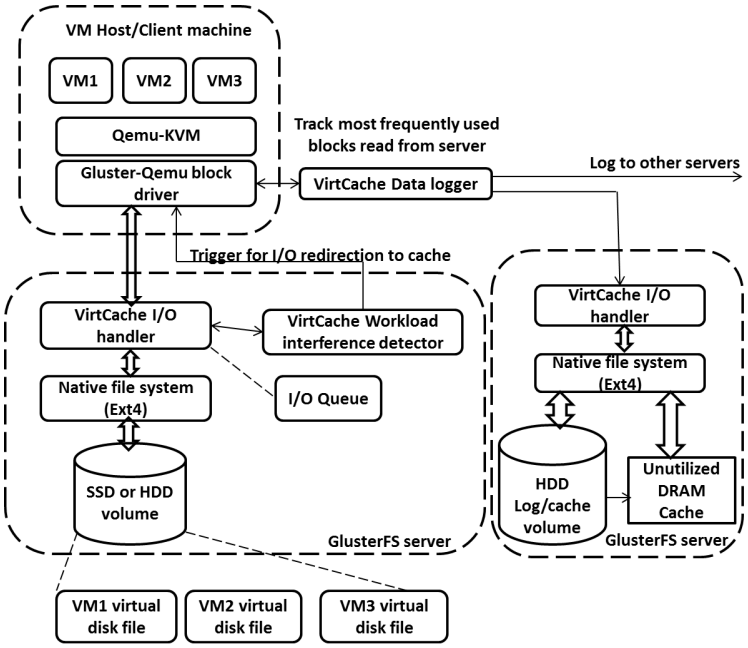

Fig. 3.13 GlusterFS architecture with VirtCache components

3.7.1 Workload Interference detection

The interference across VM workload can be considered as contention for the device I/O queues. The papers on storage performance models in [55] and [42] use queuing theory based models to predict latency and throughput based on the outstanding I/O at the device. The work done in [55] analyses contention of competing VM workloads on the same storage device. It models the VM consolidated performance as a function of arrival rates and response times when the workloads were run in isolation. The paper on storage performance management in [42] uses Little's law to model the storage latency as a function of outstanding I/O at the storage device. But these models require that characteristics of the device and workload to be profiled before the models can be used to predict performance like latencies and throughput. In our case we need a model that can be used at run time to pro-actively detect interference and contention among VM workloads. We borrow the product form queuing model studied in [13]. We use this to predict the latency expected from the device by simulating arrivals of I/O requests from the VM based on the observed

arrival rate on the storage server. The arrival rate to compute the latency from the model is set at around 10% higher than the observed to detect the contention earlier. We observed that there might be slight peaks in observed arrival rates in the workload. The 10% value was empirically chosen using workload profiling so that the system does not trigger a false cache activation. The response time or latency R_i for the ith VM at the storage device is given by:

$$R_i = d_i/(1 - \sum[\lambda_i/n] \times d_l) \tag{3.5}$$

where d_i is the device response time for the ith VM, λ_i is the arrival rate for the ith VM. n is the number of parallel I/O that can be issued to the storage device. For both the RAID5 array and SSD we set this to be 16. We observed that for the SSD we used the latency increased after we reach a queue depth of around 16. This is because even though the number of outstanding I/O can be increased beyond this value (to get higher throughput or IOPS) the latency for individual I/O still increases. Therefore we keep this value at 16. The device latency d_l is the average latency without any queuing or outstanding I/O at the device. The values d_i and d_l can be measured during non-peak workloads. The 10% higher arrival rate provides enough headroom for the I/O redirection to be initiated before the actual performance deviation occurs. In the beginning, VirtCache obtains the average latency and 90th percentile latency for each VM from the current observed value during low I/O loads. Low I/O loads are identified as the period when the average I/O queue depth is below a configured threshold (we chose 4).

VirtCache constantly monitors the observed and predicted values of response times for every time window for each VM over small time intervals. From these measurements it obtains the average latency and the 90th percentile latency from both the observed and the values obtained through the model. When there is a deviation in the difference between the average latency and the 90th percentile latency from the low load case, it triggers the I/O

redirection. Since the control simulation is done at a higher arrival rate than the observed rate, the performance variation is detected in advance to provide a small headroom. Once VirtCache detects the performance variation from low load case, it selects a VM for I/O offloading based on two parameters. The first parameter is the maximum latency deviation from the low load case. The second parameter is the cacheability requirement. VirtCache chooses the VM which has the maximum latency deviation and the maximum cache hit for a given cache size. For VMs with same cache hits the VM with higher latency deviation is chosen. For this, a hit rate curve is maintained by the VirtCache I/O handler. We do not explore on how to obtain the hit rate curve here as this is beyond the scope of this paper. Once the load in the system goes back to normal, the I/O is redirected back from the cache to the primary VM disks. For this we monitor the I/O rate at the block driver on the host machine. The pseudo code of the I/O detection of performance variation is shown in algorithm 2

```
1  foreach ith VM do
2      foreach timePeriod do
3          get λ_i^o;
4          λ_i^s = λ_i^o + 0.1 × λ_i^o;
5          R_i = d_i / ∑[λ_i^s / n] × d_l;
6      end
7      Get R_i^m from all R_i collected in timePeriod ;
8      if (R_i^m − R_i^l) > threshold;
9      Trigger VM selection and I/O redirection;
10 end
```

Algorithm 2: Algorithm to detect 90th percentile latency violation.

Here λ_i^o and λ_i^s are the observed and simulated values of the I/O arrival rates for the ith VM workload. The values R_i^m and R_i^l represent the 90th percentile latencies during the current observation period and the low load period respectively. The value *threshold* is set at 5% of R_i^l. Lines 2 to 7 obtains the 90th percentile latency R_i^m for each ith VM workload through the simulation for each *timePeriod*. Line 8 checks if this exceeds the value *threshold*. If the threshold exceeds then the VM selection and IO redirection is triggered in

line 9. The next section explains how VirtCache tracks block regions on the VM disk files for caching.

3.7.2 VirtCache I/O Handler

The VirtCache I/O handler module is a modified version of the IO-threads translator in GlusterFS. Gluster uses the name "translator" since in addition to normal file I/O it also implements other operations on files (like stat, getattr etc). The VirtCache I/O handler intercepts all read/write I/O to the back end storage device. It can identify the different VM disk files based on the global file id. The VirtCache I/O handler also tracks and provides block level I/O statistics for the VM disk files to the Gluster client. Note that since all translators in Gluster work at the file level, the VirtCache I/O handler provides a translation of file offsets to block identifiers in the logical VM disk file. The Gluster client uses the block level statistics collected by the VirtCache I/O handler to log frequently accessed data. This is done as follows. For every configured time window (typically 10 minutes), the handler records the blocks that were accessed from the client in a LFU (Least frequently used) cache. The handler only stores the block identifiers and not the actual data. At the end of each configured time interval, the handler constructs a Bloom filter [18] of the top N frequently accessed items in the LFU (Least frequently used) cache that can fit inside the given cache size ($16GB$). The Bloom filter provides a compact and efficient mechanism for recording the I/O block access. Instead of passing the entire LFU cache meta-data, the I/O handler passes the Bloom filter to the Gluster client (host machine) over another extended attribute set on the VM disk file. This extended attribute is periodically read by the client. This period is same as used by the server to refresh the Bloom filter. Equipped with the last N block access record at the storage server the client performs the data logging on the log volume. This is done by the data logger on the client whose function is explained in the next section.

3.7.3 Data Logger

We modified the Qemu-KVM block driver to implement the data logging. The data logger uses the Bloom filter provided by the VirtCache I/O handler through a extended attribute on the VM disk file. Whenever the block driver sees a block id in the response from the server matching a id in the Bloom filter, it writes this block on the log volume available in the cluster. Since this set represents the frequently used blocks, it is likely that this will be accessed when the I/O to the primary volume where the VM disk file resides is offloaded to the other servers containing the dynamic cache. When the I/O to the VM disk is offloaded, this set of blocks can be read back from the log volume to warm up the distributed caches quickly. Since this set is refreshed periodically by the VirtCache I/O handler, it also represents the recently used frequently accessed blocks.

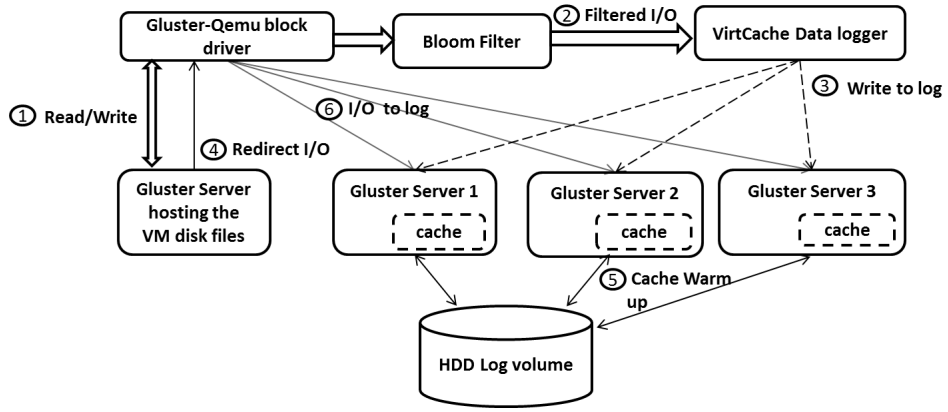

Fig. 3.14 High level control flow between components in VirtCache. The bloom filter is used to filter I/O that needs caching. Control flow numbered 1 to 3 represent activity before the I/O redirection. Control flow numbered from 4 to 6 represent activity during I/O redirection to the distributed cache

We assume that the size of the file system cache inside the VM itself is not sufficient to hold this entire set of blocks. Our assumption is valid since this is the reason there was an I/O to the server in the first place. By only logging the most recently and frequently used data set we can reduce the space required on the log volume to store this data and as well

as reduce the I/O to the caching servers. During the I/O redirection period, this logged data is read from the servers if there is a cache hit to the Bloom filter. For false positives (which are rare)the I/O is retried on the primary server having the VM disk image. The schematic in Fig. 3.14 shows the overall control flow between the different components on the client and server side during normal I/O and I/O redirection. Note that the I/O offloading is for both read and write. The offloaded writes are persisted on the distributed log volume and later moved to the primary VM disk file when the normal load returns in the system.

3.8 VirtCache Prototype Evaluation

The Evaluation Gluster storage servers used was a Xeon Server with 4GB RAM configured with RAID5 with 4 SATA drives (500GB capacity) for HDD storage. We used two real workload traces. The first one was a Microsoft exchange trace collected in our Data center with Loadgen software. We emulated an environment with 1000 user mail boxes with 100MB each with an average of around 500 emails sent/received per day from each user. The other workload was the TPCC trace downloaded from SNIA trace repository [119]. The TPCC trace were publicly available trace captured at Microsoft Research on their enterprise servers. The Exchange workload was around 30% read and 70% write, whereas the TPCC workload was around 66% read and 33% write. Both have a cache hit rate of 60% and 33% respectively for a 8GB cache. The VM2 workload was used to simulate a low load and high load scenario. At low load the VM2 fio workload was kept at 200 IOPS and for high load was kept at 400 IOPS (close to maximum IOPS for 4 drives). Similarly for SSD, we used 1000 IOPS and 2000 IOPS as low load and high load cases respectively. The distributed cache size in all our experiments were fixed at 16GB. The size of the log volume was also set to this value for each VM.

The Legend in all the graphs indicates scenarios with and without VirtCache (low load and high load). As shown in the Legend, VirtCache Read caching refers to caching and I/O

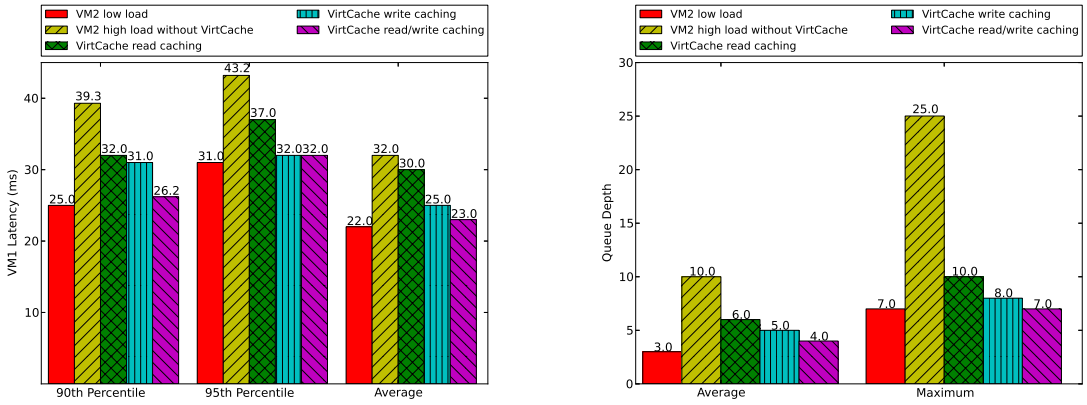

Fig. 3.15 VirtCache caching for Exchange workload on HDD (Latency and Queue depth at the Storage server). VM1 runs replayed trace of an Exchange workload. VM2 runs FIO with IOPS varied from 200 (low load) to 400 (peak load).

redirection only for reads, VirtCache write caching refers to I/O redirection only for writes similar to the Everest system [77], VirtCache read/write caching refers to I/O redirection and caching for both reads and writes. The vertical axis refers to the VM1(VM2) latency/queue depth against the different scenarios indicated in the Legend.

3.8.1 Exchange Workload on HDD

The set of Fig. 3.15 show the 90th percentile, 95th percentile and Average latencies for the cases of VM2 low load, VM2 high load and VM1 with Exchange workload without VirtCache, consolidation of VM1 and VM2 high load with read only caching, write caching and read/write caching. We also include the 95th percentile latency in the results. Since in some of the cases VirtCache can also reduce the variation in the 95th percentile from the average. From the graph we can see a huge variation in the 90th percentile latency as we move from the low load scenario to high load one. The performance variation is around 80% in the high load case (VM1 and VM2) compared to that under low load. With read caching alone VirtCache can reduce this variation to below the low load case (2ms compared to 3ms). But the average latency reduction is still small compared to the case without VirtCache. This

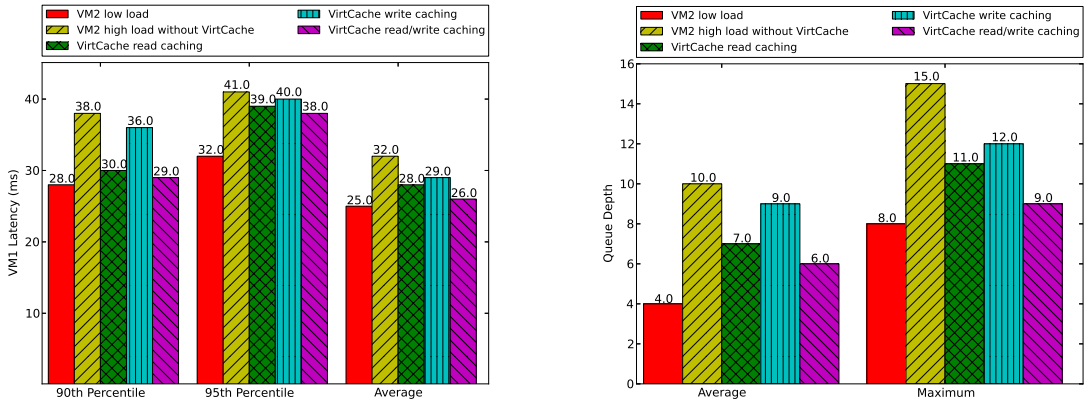

Fig. 3.16 VirtCache caching for TPCC workload on HDD (Latency and Queue depth at the Storage Server). VM1 runs replayed trace of TPCC workload. VM2 runs FIO with IOPS varied from 200 (low load) to 400 (peak load).

is because this particular workload is more write intensive and could not provide significant improvement with a read cache alone. The write only case can absorb most of the writes in the distributed cache and therefore provides a significant reduction in average latency. But the 90th and 95th percentile latencies are still large compared to the average. The read/write cache can almost bring the average latency to the low load case. It can also bring the 90th percentile latency within 7% of the low load case. This is because most of the reads and writes are offloaded from the VM disk file to the distributed cache servers. A fraction of the load with cache misses still has to be served from the VM disk file. Since VM2 generates a significant rate of I/O, the average latency of VM1 is slightly higher than that of the low load case. VirtCache can also bring the 95th percentile latency to levels same as the low load scenario. We can also see that the read/write caching can achieve around 50% reduction in variation of the 90th percentile latency from average compared to the write offloading/caching alone. Compared to the case without VirtCache during peak workload the latency variation is reduced to around 56%.

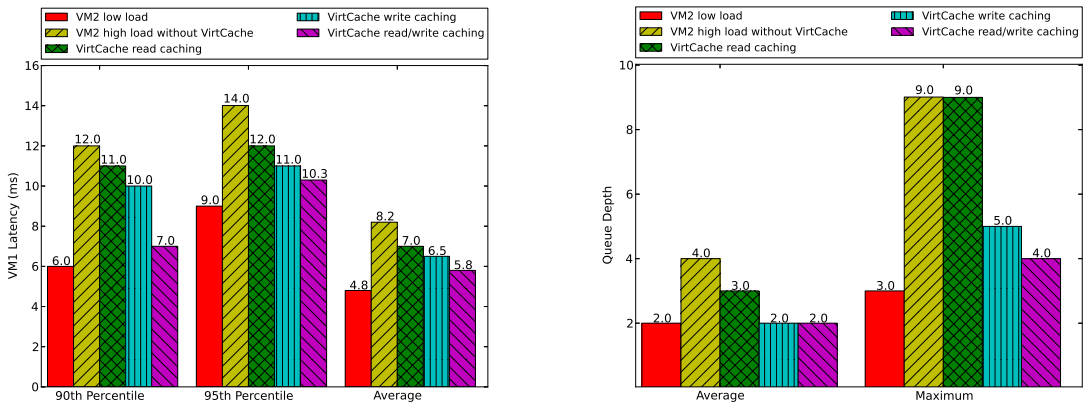

Fig. 3.17 VirtCache caching for Exchange workload on SSD (Latency and Queue depth at the Storage Server). VM1 runs replayed trace of an Exchange workload. VM2 runs FIO with IOPS varied from 1000 (low load) to 2000 (peak load).

3.8.2 TPCC workload on HDD

For the TPCC workload, the performance variation for the peak conditions without VirtCache increases by twice the amount in the low load case. We can also see from Fig. 3.16, that the maximum I/O queue size increased to around 15 that is twice to that during low loads. Since this workload has more reads than writes we get a significant improvement in both the average latency and the 90th percentile latency variation. The 90th percentile latency variation from average in the high load scenario is in fact lesser than the low load case. With just the VirtCache write caching both the average latency and the 90th percentile latency variation from the average is higher than the low load case. With VirtCache read/write caching, we can bring down the average latency levels close to the low load case. The variation in the 90th percentile latency is the same as the low load case with VirtCache. The variation in this case is around 50% smaller compared to that without VirtCache during peak loads. The 95th percentile latency of this workload is higher than for that of Exchange workload. This is due to the lower cache hit rate of this workload compared to the Exchange workload.

3.8.3 Exchange workload on SSD

We also tested with Exchange workload on VM disk files on the SSD and the results are shown in Fig. 3.17. We could not test for the TPCC case since the block access pattern for this workload spans more than 400GB which could not be tested on a single SSD drive. For the exchange workload, even for SSD we can still see significant improvement in both the average latency and the 90th percentile latency variation compared to the average. Under VirtCache read/write caching both the 90th and 95th percentile latency were close to the low load case since we can get a very high hit rate for this workload. The variation of the 90th percentile latency was also close to that of the low load case. The VirtCache read/write caching for even SSD volumes can do better by around 65% compared to the write offloading alone.

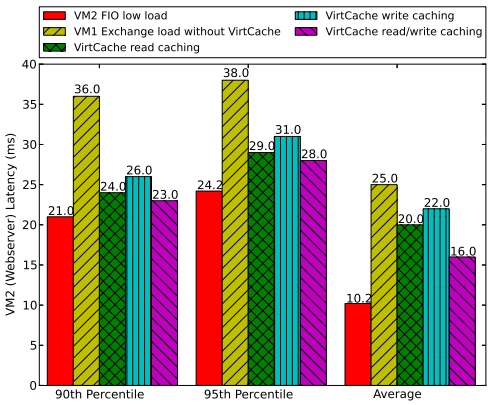

Fig. 3.18 VM2 (Webserver) disk latency in consolidation with Exchange (VM1) with VM2 chosen for caching

3.8.4 Consolidation of Exchange workload with Webserver workload on HDD

We also experimented with consolidation of the Exchange workload (VM1) with a FIO web-server workload (VM2) with different IOPS rate.The low load case was simulated with 200

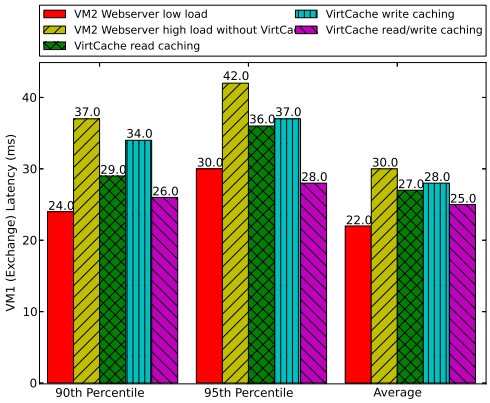

Fig. 3.19 VM1 (Exchange) disk latency in consolidation with Webserver (VM2) with VM2 chosen for caching

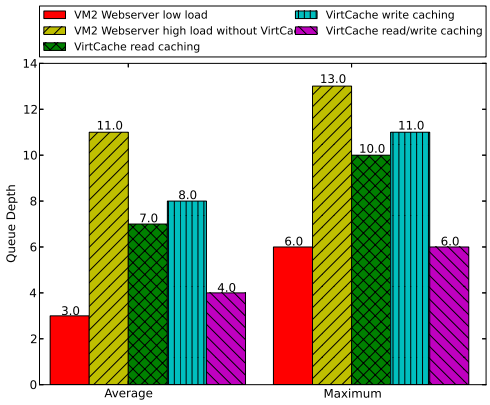

Fig. 3.20 Queue depth for VM1 (Exchange) and VM2 (Webserver) consolidation

IOPS as before and the high load was replayed with 400 IOPS. VirtCache chooses the Webserver workload to be offloaded since it has a higher cacheability with only 4GB required for the cache to get around 60% hit rate. As can be seen from the graph in Fig. 3.19 for Exchange workload the 90th percentile latency variation from average is around 3.5 times at high load compared to the low load case. This variation is reduced to less than the low load case with VirtCache read/write caching. We can also see that the VirtCache read/write caching can reduce the 90th percentile variation by around 1/6th (from 6ms to 1ms) compared to write offloading alone (around 83% reduction). We also compare the effect of the caching for the Webserver workload. For this we compare the latencies from our previous

experiment with a Webserver VM consolidated with an FIO lower load. In the Fig. 3.18 this is shown as "VM2 FIO low load". The "VM1 Exchange load without VirtCache" shows the latency values for the Webserver workload in consolidation with Exchange (VM1). The 90th and 95th percentile latency values are as high as that of Webserver since the queue depths are common for both workloads at the device (see Fig. 3.20). Due to the redirection of the I/O to the cache we can see a large reduction in the average latency under VirtCache read/write caching. The 90th percentile latency variation (from average) is also reduced to below that of the low load case.

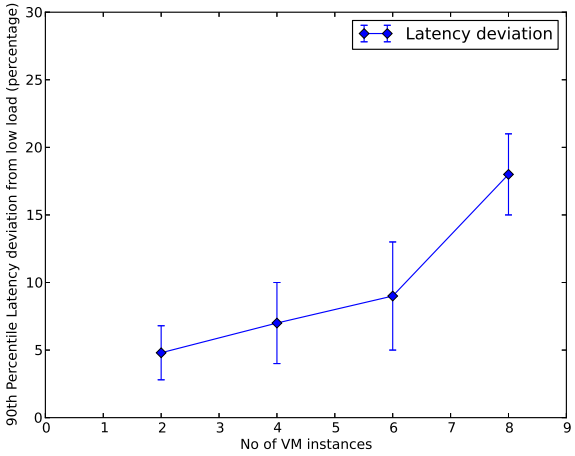

Fig. 3.21 Latency deviation from non-consolidated case as the number of VM instances are increased. The number of Gluster storage node is fixed at 4. Each VM workload exhibits a cache hit rate of 30 to 60 percentage from a synthetic workload replayed with FIO with IOPS of 300

3.8.5 Effect of number of VMs

As another experimental scenario to test the scalability of VirtCache, we varied the number of VM instances from two servers (hosts) and measured the latency variation as the IOPS is increased from each VM. We replayed a synthetic workload from these VMs and measured the deviation of the 90th percentile latency of the workloads from the non-consolidated

case. Figure 3.21 shows the deviation in percentage from the non-consolidated case as the number of VMs are increased. As a reasonable value for caching, we generated around 30 - 60 percentage hit rate in the synthetic traces. The virtual disk files for the workloads were created on a single Gluster server with SSD. The number of Gluster servers were fixed at four each with 4GB RAM (same configuration as used in the previous experiments). As shown in the figure as the number of VMs are increased from 2 to 8 the deviation increases from 5 to 20 percent from the low load case. Some of the workloads I/O were moved to the less utilized server's caches. This reduced the contention on the virtual disk file hosting server, thereby reducing the I/O queuing at the storage device.

3.9 Summary

This work explored the potential of utilizing a small Non-volatile memory device (like STT-MRAM/PCRAM) as a persistent read/write log to a Flash backed SSD. We have developed an algorithm that adapts the log size of the NVM to the workload characteristics of the application dynamically. Simulation experiments have shown that our algorithm can distribute NVM allocations based on the application that gets the maximum benefit. This is done by closely tracking the working set of the application through a novel control loop using a shadow log as a reference. Hit rate deviations with respect to the shadow cache and the desired latency are used to tune the real log size of the application. Through such a control algorithm, HCache can meet the configured latencies of the application with up to 50% reduced log sizes compared to static allocations and improve the SSD performance up to 90% in some of the workloads. We have also shown that our method can achieve up to 46% reduction in latencies compared to the popular PID control mechanisms used in other works. Our method also satisfies the latency requirement of the applications most of the time in the simulation run.

We also consider the problem of workload interference among different applications

sharing a distributed storage system specifically in the context of a virtual disk VM workloads. This problem not only exists for HDDs but also present in SSD as well. We have shown through GlusterFS prototype evaluation that by temporarily redirecting I/O to a distributed cache during peak loads we can reduce the variation in performance of VM workloads to those during normal loads. In some of the cases we could achieve average and 90th percentile latencies very close to that of the off-peak periods. Compared to write offloading alone we can achieve around 50% to 80% improvement in reducing the performance variation.

Chapter 4

Improving Storage System Performance With Optimized Storage Tiering

4.1 Introduction

In this chapter we look in to the data migration problem when the storage system consists of several tiers of storage as described in Chapter 1 in section 1.8. The storage server consists of NVM-SSD hybrid, Hybrid drives and high density SATA drives at the different tiers. We address the problem of placement of data across these tiers of storage based on the relative benefit value to improve storage server performance. While the hard disk drives provide adequate storage volume at low cost, the access latency remains a major problem. In particular, the disk access latency is significant when I/O workload exhibits random access. This results in heavy seeks or disk head movement accounting for the higher access latency [64]. In order to improve the system performance and overcome the disk access latency with random accesses, many previous state-of-art work proposed different methods. Bhadkamkar et. al [14] used block level disk reorganization by using a dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition, thus significantly reduced seek and rotational delays [14]. In FS2 [45], multiple replica of

data based on disk access patterns in file system was used to reduce latency. In [37] the past file access patterns were used to predict the future requests, so that the data can be served in advance of the request for the data. In DiskSeen [31], disk prefetching policy at the level of disk layout was adopted to improve the sequentiality of disk accesses and overall prefetching performance. In application directed prefetching [103], the algorithms made effective use of system primary memory by aggressively fetching as much data as fits in available memory.

Most previous work on solving storage performance problems were focused on throttling and manipulating I/O streams by prefetching or scheduling but do not consider migrations of data among storage devices. Recently, Lin et al. [64] proposed a data migration algorithm, called Hot Random Off-loading (HRO) to move the random accessed files to SSD so as to reduce the latency by using the SSD as the by-passable cache to the hard disk. A well known advantage of SSD unlike hard disks lie in the fact that they utilize non-volatile solid state memory chips which contain no physical or mechanical movement. Thus the read/write response times will be almost constant with the random read and write performance one to two orders of magnitude lower than hard disks. By using SSD as one of the tiers for storage of random accessed and hot data (hot stands for frequently accessed data), the overall storage system performance can be improved.

As the ratio of cost versus performance of SSD is low compared with hard disks, SSD alone as primary storage is still not the current viable solution. Hybrid storage methods which take advantages of both SSD for performance and hard disk for capacity are proposed as a alternative. We would see in the Chapter 5 new kinds of device which incorporate a small Non-volatile cache (NAND Flash) in addition magnetic platters found in conventional Hard drives. These Hybrid disks [17] utilizes the non-volatile memory to cache popular read and write requests to HDD. The storage capacity being equal to the capacity of the magnetic media. Combo drive [84] is a storage media device that combines flash memory as well as

magnetic memory in a single storage device. By using the hybrid disks or combo drives, the overall storage cost is kept low while the overall performance of the storage system is improved compared to hard disk. We adopt the idea of hybrid storage by proposing a multi-tiered hybrid storage (MTHS) system. The MTHS includes both SSD and hybrid disks as high performance storage tier and commodity hard disks as a low performance tier. By combining SSD and hybrid disks forming a multi-tier storage system, the overall system I/O performance can be improved for fast random access while minimizing the cost. In order to achieve higher overall performance, it is necessary to keep the most frequently accessed data (hot data) on the high performance storage tier, while storing the least accessed data at lower tiers [105]. The MTHS storage system consists of two components, one is the monitor module which collects the I/O access pattern and distinguishes the random accesses from sequential ones. The other component is the data migration module which schedules the migration work and moves the random hot data between lower storage tiers and higher tiers.

The difference between our MTHS and HRO [64] are as follows. First, HRO uses a single tier of SSDs in their hybrid storage system while we use multiple tiers, i.e., SSD, hybrid drive, and hard disks. We also use various tiers (up to 10 tiers) in our simulations to show the performance of our proposed algorithm. This is necessary since we have a multiple tiers of storage with NVM/SSD hybrid, Hybrid drives and SATA drives. Secondly, HRO formulate the Data allocation problem (DAP) as a knapsack problem while we formulate the problem of data allocation for multiple tiers as the multiple choice knapsack problem (MCKP). Lastly, HRO used a polynomial-time approximation greedy algorithm which is not able to find global solutions while we use dynamic programming to find global optimal solutions for the DAP. Dynamic programming is a pseudo polynomial algorithm. We propose a multiple-stage dynamic programming (MDP) to solve the MCKP in multiple stages. We compare our work with the greedy algorithm in HRO and show the benefit obtained using dynamic programming than the greedy algorithm even with smaller test cases.

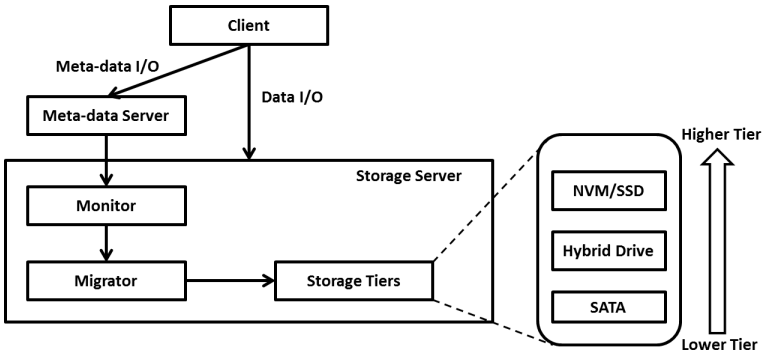

Fig. 4.1 The overall MTHS system framework

4.2 Multi-tier Hybrid Storage System

The multi-tier hybrid system as in Fig. 4.1 consists of the client, meta-data server (MDS), storage devices and data modules for data migration. In this chapter we focus only on the Storage server component which manages the data allocation on the storage devices. The client accesses file data based on the meta-data from the MDS. The client's file or meta-data operations can be monitored through the MDS. The data monitoring module can get these client access patterns and provides this information for data allocation algorithm. The DAP is then solved by algorithm run within the data monitoring module and generates the optimal data lists for migrating to different storage tiers. The storage system consist of multiple tier of storage devices including NVM/SSD, hybrid drive, and SATA drives.

Fig. 4.2 shows the data flow processing in the system. The dashed rectangle indicates the monitor module. It contains two sub-modules, the data collector, which collect the I/O traffic from the clients. The other component is the data allocation sub-module which process the output from the collector and generate the allocations of the files. From Fig. 4.2, we can see that the I/O information from the clients will be used to determine or distinguish between sequential and random accesses in the storage system. The set of random and hot data objects are identified and output to data allocation sub-module. The technique to identify sequential and random I/O is not the focus of this work. We utilize techniques like

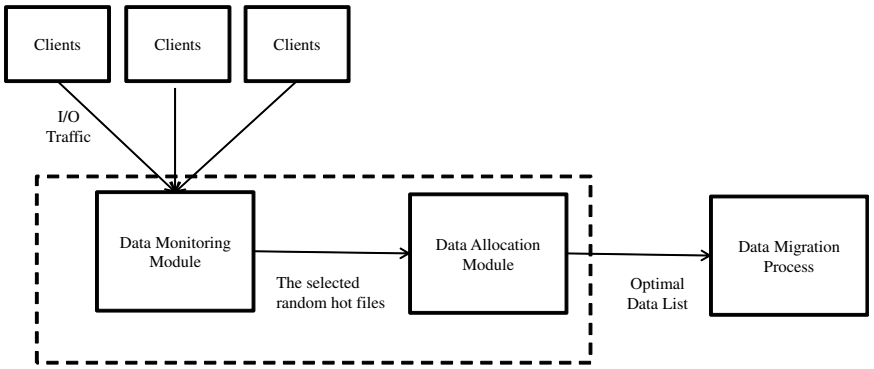

Fig. 4.2 The data processing flow for the DAP.

that in Hot Data Trap [78] to differentiate between hot and cold data as well as sequential and non-sequential I/O. The data allocation sub-module will then allocate the hot random data objects to multiple tiers so as to improve the system performance. The migration module will perform the data migration process when the client side I/O activities are low.

4.2.1 Data Collection

Data collection is a sub-module of monitor module in the MTHS system. In order to distinguish the random I/O from the sequential one, we use the following definitions:

Definition: Random I/O: random I/O is the type of read and write access to data objects in the storage server.

Definition: Sequential I/O: sequential I/O is the type of read and write access to data objects in a predetermined, ordered sequence in the storage server. The ordering is based on the object offset or the block addresses.

The method to differentiate the two types of I/O access can be similar to the one described in [64]. For the current I/O request, the sequentiality of the I/O is decided by comparing with the previous I/O. If the access type and the start address of the new I/O is the end address of the previous I/O, then the current I/O is a sequential. Otherwise, it is considered as random.

4.2.2 Data Monitoring Module

The data allocation problem is to allocate a selected set of random accessed data in lowest tier, i.e., SATA, and to other two tiers so that the total benefit value is maximized without violating the capacity constraint of disks. The DAP can be formulated as follows:

Definition: DAP: Given a set of n data objects $D = \{d_i, i = 1,2,...n\}$, each object in the set has a size $S = \{s_i, i = 1,2,...,n\}$, benefit value $V = \{v_i, i = 1,2,...,n\}$, and m tiers $T = \{t_j, j = 1,2,...,m\}$, each tier with a tier size $S^t = \{s^t_j, j = 1,2,...m\}$, allocate the set of objects in D to m disk tiers, so that the total sum benefit value for the m tiers are maximized while not violating the constraint of limitation of each tier size.

The DAP problem can be formulated as a multiple choice knapsack problem as follows [8, 85]:

$$\text{Maximize: } z = \sum_{i=1}^{n}\sum_{j=1}^{m} v_{ij}x_{ij} \tag{4.1}$$

$$\text{subject to: } \sum_{i=1}^{n}\sum_{j=1}^{m} s_{ij}x_{ij} \leq s^t_j, i = 1,2,...,n \tag{4.2}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1, j = 1,2,...,m \tag{4.3}$$

$$x_{ij} \in \{0,1\}, i = 1,2,...,n; j = 1,2,...,m \tag{4.4}$$

$$v_{ij} = \alpha * v_{ik}, 1 \leq j < k \leq m \tag{4.5}$$

$$s^t_j = 1/\beta * S_k, 1 \leq j < k \leq m \tag{4.6}$$

$$x_{ij} = \begin{cases} 1, & \text{if data object } i \text{ is assigned to knapsack } j; \\ 0, & \text{otherwise.} \end{cases}$$

where n is the number of data objects selected for data allocation, m is the number of choices, i.e., the number of tiers available for allocation for each data object. Constraint

4.2 ensures that total size of of data objects assigned to each tier is not larger than the maximal available size of each tier. Constraint 4.3 specifies the limitation that each data object can only allocated once to all the available tiers. Constraint 4.4 states that it is a 0/1 knapsack problem.

4.2.3 Data Migration Module

In our MTHS system, the data migration operations are performed when foreground I/O activities are low so that the impact on the client application is minimized. This type of migration is considered as "off-line" type migration since the migration process is scheduled as one-time activity and does not consider the current application performance. As frequently moving the hot random files to different tiers may cause the system to be unstable, the off-line migration is relatively simple and effective while the on-line migration definitely has a higher overall performance if it does not affect the user accesses. In this work, our optimal data allocation will generate a one-time scheduled migration job, which can be called as "static" migration. We do consider on-line dynamic migration by constantly performing data migration based on the client I/O. As we will consider the feedback affects on the user's performance during migration, the data migration module should be extended or enhanced by using simple feedback control as in [65], which can be easily incorporated into our system.

4.3 Multiple-stage Dynamic Programming For The DAP

As data allocation of hot random data objects to one tier can be considered as a standard 0/1 knapsack problem, dynamic programming can be used as a effective method to find optimal solutions. However, for multiple tiers, the problem is then formulated as a multiple choice knapsack problem as mentioned in section 4.2.2. To solve the MCKP, we use a simple

heuristic, i.e., we treat the multiple-tier DAP as multiple single-tier DAP by solving the highest tier first then the rest tiers according to its priorities (here priority is defined as its storage performance in terms of IOPS).

4.3.1 Greedy Algorithm

Input: Number of data objects n, vector of benefit values v, vector of size w, constraints of maximal size W.
Output: List of selected data objects x, total benefit value V_t

```
1  foreach i in v do
2  |  //Sort vector v in decreasing order
3  |  sorted_index = sorted(v, key=v.get, reverse=True);
4  end
5  //Initializing
6  totalW = 0;
7  totalV = 0;
8  marked = [];
9  foreach i, 0 < i < n do
10 |  marked.append(0)
11 end
12 foreach i, 0 < i < n do
13 |  if totalW + w[sorted_index[i]] > W then
14 |  |  continue;
15 |  end
16 |  totalW += w[sorted_index[i]];
17 |  totalV += v[sorted_index[i]];
18 |  marked[sorted_index[i]] = 1 ;
19 end
20 foreach i, 0 < i < n do
21 |  x[i] = marked[i] ;
22 end
23 V_t = totalV ;
24 Return V_t, x ;
```

Algorithm 3: The greedy algorithm (**greedy_kp**) for the DAP.

We implement the same greedy method as in HRO [64] shown in Algorithm 3. The idea of this fast polynomial-time approximation algorithm is to sort the set of data objects based on the rank or value, which can be the benefit value or similar value to rank the objects.

Then the maximum object with size smaller than the available size in the tier is put in the tier, then the next object with maximal value is checked until the tier is full. In lines 1 to 4, the data objects are sorted in decreasing order of size. The next loop checks all the data objects whether it can be added to the tier based on the limit of available size of the tier in decreasing order. The results should contain the list of objects selected and the total benefit values for the selected objects. Besides the sorting, the greedy algorithm has a complexity of $O(N)$, where N is the total number of data objects. Though very fast as it is, the greedy algorithm is obviously not optimal.

To use the greedy algorithm HRO on the multiple-tier DAP, the multi-stage HRO is used as shown in Algorithm 4. In the multiple-stage version of algorithms (Algorithm 4 and 6), the key requirements are:

- Solve the multiple tier data allocation problem as multiple single tier problem.
- For each single tier data allocation problem, normal greedy algorithm or dynamic programming algorithm is used.
- Solve the single tier using the normal algorithm sequentially with the order of highest tier first.

Before we introduce the multiple stage version algorithms, we shall state the simulation setup first so that the algorithms can be explained clearly. We generate the simulation test cases using the methods as in [85]. We can generate 4 types of testing cases, i.e., uncorrelated, weak correlated, strong correlated, and subset sum. The output will include the number of data items, the weight and value of each item, and the maximal weight. Provided with the test cases generated using the methods in [85], we will have a benefit value and size of the low tier, e.g., SSD. To simulate the multiple tier problem, we assume that the top tier(s) with higher performance, e.g., SSDs in 2-tier problem, has a double benefit value and a size of 1.5 smaller compared with the lower tier next to it. The assumption used

Input: Number of data objects n, number of tiers m, vector of benefit values v, vector of size w, constraints of maximal size W.
Output: List of selected data objects x, total benefit value V_t

```
1  foreach i, 0 < i < m do
2      t = m - i - 1;
3      foreach j, 0 < j < t do
4          foreach k, 0 < k < n do
5              W_i[k] = W_i[k]/1.5;
6              v_i[k] = 2 * v[k];
7          end
8      end
9      [totalValue, marked] = greedy_kp(n, v, w, W); foreach k, 0 < k < len(w) do
10         if marked[k] != 1 then
11             w[k] = w_i[k];
12         end
13     end
14     foreach k, 0 < k < len(v) do
15         if marked[k] != 1 then
16             v[k] = v_i[k];
17         end
18     end
19 end
20 foreach i, 0 < i < n do
21     x[i] = marked[i] ;
22 end
23 V_t = totalValue ;
24 Return V_t, x ;
```

Algorithm 4: The multi-stage greedy algorithm for the DAP.

here is only for simulation purpose. For real application, the parameters and values may be different, but it does not affect the correctness of the algorithms.

Multiple Stage Greedy Algorithm: The multiple stage version of this algorithm basically utilizes the greedy algorithm for the single tier and applies it to the N tiers. In Algorithm 4, lines 1 to 8 are used to initialize the simulation input. Then it calls Algorithm 3 (*greedy_kp*) to solve the single-tier DAP in line 9. The used data objects are marked and the rest are used as input for next round as shown in line 10 to 18. The final results are accumulated in line 20 to 23.

Input: Number of data objects n, vector of benefit values v, vector of size w, constraints of maximal size W.
Output: List of selected data objects x, total benefit value V_t

foreach *i,* $0 < i < n$ **do**
  foreach *j,* $0 < j < W+1$ **do**
    if $W[i] > j$ **then**
      $c[i][j] = c[i-1][j]$;
    else
      $c[i][j] = max(c[i-1][j], v[i] + c[i-1][j-w[i]])$;
    end
  end
end
$currentW = len(c[0]) - 1$;
$i = len(c) - 1$;
while $i \leq 0$ *and* $currentW \leq 0$ **do**
  if $(i == 0$ *and* $c[i][currentW] > 0)$ *or* $(c[i][currentW] != c[i-1][currentW])$ **then**
    $marked[i] = 1$;
    $currentW = currentW - w[i]$;
  end
  $i = i - 1$;
end
foreach *i,* $0 < i < n$ **do**
  $x[i] = marked[i]$;
end
$V_t = c[n-1][W]$;
Return V_t, x ;

Algorithm 5: The dynamic programming algorithm (**mckp_dp**) for the DAP.

4.3.2 Multiple-Stage Dynamic Programming Algorithm

Dynamic programming (DP) algorithm has been proven to be effective to solve knapsack problems in various application domains. In [133], authors used the dynamic programming to solve the reliability and redundancy allocation in system design. They showed that the sub-problems are equivalent to one-dimensional knapsack problems which can be solved in pseudo-polynomial time with a dynamic programming approach. In [87], authors modeled the scheduling of scientific grid workflows as an extension of the multiple-choice knapsack problem and proposed a general bi-criteria scheduling heuristic called dynamic constraint algorithm (DCA) based on dynamic programming. They showed the performance over

Input: Number of data objects n, number of tiers m, vector of benefit values v, vector of size w, constraints of maximal size W.
Output: List of selected data objects x, total benefit value V_t

```
foreach i, 0 < i < m do
    t = m - i - 1;
    foreach j, 0 < j < t do
        foreach k, 0 < k < n do
            W_i[k] = W_i[k]/1.5 ;
            v_i[k] = 2 * v[k] ;
        end
    end
    [totalValue, marked] = mckp_dp(n, v, w, W); foreach k, 0 < k < len(w) do
        if marked[k]!=1 then
            w[k] = w_i[k];
        end
    end
    foreach k, 0 < k < len(v) do
        if marked[k]!=1 then
            v[k] = v_i[k];
        end
    end
end
foreach i, 0 < i < n do
    x[i] = marked[i] ;
end
V_t = totalValue ;
Return V_t, x ;
```

Algorithm 6: The multi-stage dynamic programming (**MDP**) algorithm for the DAP.

existing algorithms through practical applications. In this work, we proposed the multiple-stage dynamic programming and use it to solve the DAP.

Dynamic Programming for DAP: At a high level the dynamic programming algorithm for data allocation works based on computing a cost matrix based on the benefit value of each object and their size. The cost matrix takes into consideration the constraints or storage capacity of each tiers to place the objects. At the same time it tries to achieve the maximum benefit value. The dynamic programming algorithm is shown in Algorithm 5. The most important part of the DP is to construct the N by W cost matrix C where N is the size

number of items and W is the maximum size the knapsack can hold. As we can see from Algorithm 5, it computes the cost matrix from lines 1 to 9 and obtain the resulting total benefit values at the $C[N][W]$. However, cost matrix does not keep record of which subset of data objects gives the optimal solution. We used lines 10 to 18 to backtrack the cost matrix and get the selected data objects list. The computational complexity of the DP is $O(NW)$ and it needs $O(NW)$ space. DP is a pseudo-polynomial algorithm because the runtime is bound not only on the size of the input N, but also on the magnitude of the inputs of the problem W.

Proof: In algorithm 5, $c[i][j]$ by definition represents the optimal solution (total benefit value) for i objects with total weight j ($< W$). Here j represents the storage size of the ith object. To compute $c[i][j]$ we have only two choices (line 6 in the Algorithm 5) for the ith object:

Leave Object i from being placed in tier: The best we can do with objects {1,2,3...i-1} and storage capacity j is $c[i-1][j]$.

Place object i in tier (Only if $w[i]$ is less than W): In this case we gain a benefit value of $v[i]$, but the object storage size $w[i]$ is utilized. The best we can do with the remaining storage objects {1,2,3...i-1} and storage capacity $(j-w[i])$ is $c[i-1][j-w[i]]$. So that we get $v[i]+c[i-1][j-w[i]]$.

Note that the above applies for all values of j (inner loop in the algorithm) till the total storage capacity W of the tier (see lines 2 to 7 in Algorithm 5). Therefore we obtain the optimal solution or benefit value c[n][W] as the output of the algorithm.

Multiple stage Dynamic Programming for DAP: Similar to the multiple stage of the Greedy algorithm (Algorithm 4), Algorithm 6 calls Algorithm 5 ($mckp\_dp$) to compute the knapsack value and lists of select data objects for each tier. It marks the already utilized objects in each stage and applies the single stage Dynamic Programming algorithm to the subsequent tiers. Finally it consolidates the outputs of each tier after all the object lists for

all the tiers have been computed.

4.4 Results

4.4.1 Simulation Instances

We use the algorithm (kpGen) provided in [86] to generate the simulation instances. The software can generate different types of random cases by choosing different input parameters. The types of simulation cases can be one of the four types as below.

- Uncorrelated
- Weakly correlated
- Strongly correlated
- Subset sum

The above different simulation instances are representative of the Object characteristics of storage systems. Uncorrelated type represents random choice of object size and their value. It refers to no correlation between the size of the object and their benefit value. Weakly correlated type refers to some of the object's size having correlation with the benefit value. For example, large objects might have smaller benefit value compared to smaller objects. This could represent meta-data accesses which are smaller and accessed frequently or it could represent small random I/O. The Strongly correlated type has many of the objects having proportional benefit value to their size. The subset sum refers to all objects having a direct relationship between their size and benefit value. The generated simulation instance will be saved in text files. However, This type of simulation instance suits normal knapsack problem with only one knapsack. In order to simulate the multiple-tier data allocation

problem (i.e., the multiple choice knapsack problem), we make the assumption that the output generated using the algorithm (kpGen) in [86] is for the lowest tier (T_0) in the MTHS system. The values for the rest of the tiers are assumed as following.

- For tier i (denoted as T_i), where i is the number of tiers above the T_0, $i = 0, 1, ..., N$, the benefit values of data objects of this tier is set as the double the value of the next adjacent lower tier, i.e, $V_i = V_0 * 2^i$, where V_0 is the benefit value for T_0 which is generated using kpGen. This is to emulate the higher benefit value for objects that can be placed on higher tiers. Instead of having a linearly increasing benefit value, we use an exponential increase.

- For tier i (denoted as T_i), where i is the number of tiers above the T_0, $i = 0, 1, ..., N$, the maximum size of this tier is assumed to be 1.5 smaller than the next adjacent lower tier, i.e., $mW_i = mW_0/(1.5^i)$, where mW_0 is the maximum size for T_0 which is generated using kpGen. The 1.5 times smaller value is to simulate real world case where smaller objects with higher benefit value (heat value) are stored in higher tiers.

- The size of the data objects generated using kpGen is kept the same for every tier.

We first generate the simulation instances with 2 tiers using range of coefficient as 50, and number of tests in the series as 1000. The generated simulation instances using kpGen are shown in Table 4.1. There are total 20 test cases with 4 groups. Each group is of the same type and has different number of input data objects ranging from 100 to 10000. The maximal sizes of T_0 are also listed.

We compare the results based on the total benefit values for MTHS system. We use the term **improvements** (P) as the performance metric which is defined as:

$$P = (V_{mdp} - V_{pre})/V_{pre} \tag{4.7}$$

Table 4.1 Simulation instances for 2-tiered storage system with different types.

Instance	No. Data Objects	No. Tiers	Max Size (Tier 0)	Data Range	Type
1	100	2	51	50	1
2	500	2	51	50	1
3	1000	2	76	50	1
4	5000	2	384	50	1
5	10000	2	764	50	1
6	100	2	51	50	2
7	500	2	51	50	2
8	1000	2	76	50	2
9	5000	2	384	50	2
10	10000	2	764	50	2
11	100	2	51	50	3
12	500	2	51	50	3
13	1000	2	77	50	3
14	5000	2	382	50	3
15	10000	2	764	50	3
16	100	2	51	50	4
17	500	2	51	50	4
18	1000	2	77	50	4
19	5000	2	382	50	4
20	10000	2	764	50	4

where V_{mdp} is the total benefit value obtained using our proposed *MDP* and V_{pre} is the value from previous methods.

The results of comparison between greedy algorithm *HRO* and the proposed algorithm *MDP* are shown in Table 4.2. From this table, we can see that the biggest improvements can be obtained for the type 1 datasets, while both methods can find the optimal for the type 4 datasets. This is because datasets in type 4 is of special type which have the same value for benefit value and size for each data object. From this table we can see the following.

Table 4.2 Comparisons of total benefit values between GA, HRO and MDP.

	GA	HRO	MDP	P (vs GA)
1	584	718	735	0.26
2	647	1726	1753	1.71
3	979	2913	2952	2.02
4	2150	14518	14555	5.77
5	3850	29151	29209	6.59
6	134	141	165	0.23
7	143	211	242	0.69
8	196	344	391	0.99
9	994	1762	1988	1
10	1965	3610	4052	1.06
11	148	326	359	1.43
12	159	590	629	2.96
13	239	1019	1059	3.43
14	1090	5553	5590	4.13
15	2162	11208	11222	4.19
16	118	115	119	0.01
17	119	116	119	0
18	179	176	179	0
19	890	887	890	0
20	1782	1779	1782	0

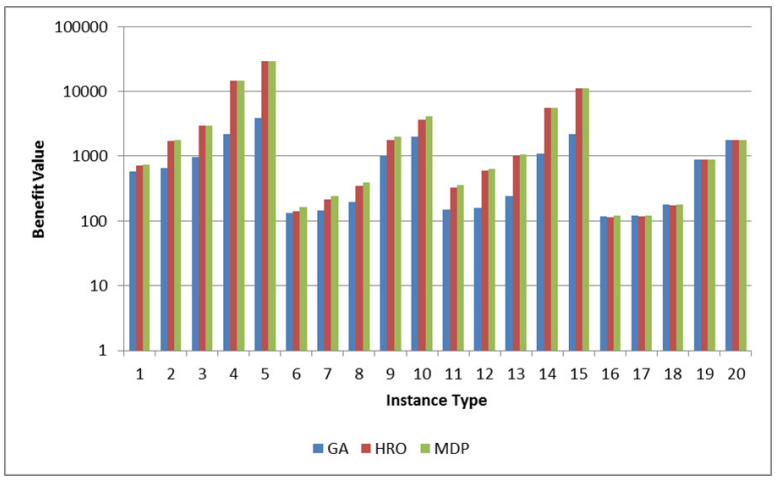

Fig. 4.3 Comparisons of benefit values between GA, HRO and MDP from table 4.2. The benefit value is the total heat value of all the objects in the different Tiers.The vertical axis is in logarithmic scale. Horizontal axis is the simulation instance types from table 4.1 for Uncorrelated, Weakly correlated, Strongly Correlated and Subset sum instances (5 in each type).

1. The improvement of the total benefit value is increasing with the number of input data objects (expect for the special case in simulation instance type 4).

2. The correlation between the two inputs, i.e, benefit value and size of each data object has impact on the performance. The uncorrelated type has the best performance than the rest.

Figure 4.3 shows the comparisons of benefit values between GA and MDP for various instances in Table 4.1. We can see that the MDP can outperform GA in all simulation instances except the instance in subset sum type, as in this type the benefit value and size are equal. This type of instance makes the HRO difficult to choose optimal solutions because HRO sorts the objects based on the weighted value (V_{ij}/W_{ij}). We can see that for type 4 instances (instance 16 to 20) HRO is the worst among all the methods.

In order to show the performance compared with HRO, we generate weakly correlated instances with big data range $(0-500)$. The results are shown in Fig. 4.4. From this figure, we can see that MDP can achieve up to 17% performance compared with HRO.

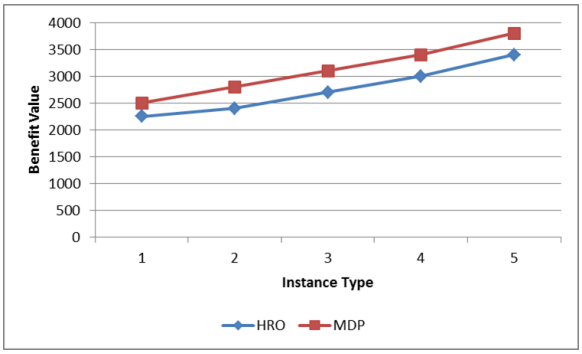

Fig. 4.4 Comparison of results between HRO and MDP using using large data range. The benefit value is the total heat value of all the objects in the different Tiers. Horizontal axis is the instance type for weakly correlated instances.

Table 4.3 Comparisons of total benefit values between HRO and MDP with variable number of tiers. Instances are generated using type 1 and with data range of 100 (each with 1000 data objects).

	No. Tiers	GA [1]	HRO [2]	MDP	P (vs. [1])	P (vs. [2])
1	1	887	1506	1741	96.28%	15.60%
2	2	2132	3362	3869	81.47%	15.42%
3	3	3719	6207	6827	83.57%	9.99%
4	4	5919	10463	11094	87.43%	6.03%
5	6	13667	25247	26171	91.49%	3.66%
6	8	36867	56215	58041	57.43%	3.20%
7	10	90134	125375	129137	43.27%	3.00%

To show the scalability of the MDP, we generate test cases with various tiers to show the impact of number of tiers on the performance besides the test cases with 2-tiers as in Table 4.1. Table 4.3 shows the results for tiers ranging from 2 to 10 with 1000 data objects.

From Table 4.3, we can see that MDP can achieve much better solution than GA and HRO. However, the P is slowing down with the increasing of the tiers. With 10 tiers, the performance gain is down from 15.6% to 3% for comparison with HRO. The performance drop can be explained as follows.

1. The drawback of the greedy algorithm by taking the data objects with maximal benefit value can be leveraged through multiple stages.

2. The multiple-stage scheme in MDP which tackles the knapsack problem in each stage is a approximation algorithm which is not global optimal solution as it solves each knapsack problem separately.

The major part of this work is on the algorithm to optimally assign the set of data objects to multiple tiers in hybrid storage system. The output of the list of data objects can be used by the data migration process to move data among the different storage tiers. We had provided the simulation results using the simulated datasets generated using KpGen software. Since it is hard to get real workloads with large datasets (Objects) we utilized simulation to validate our algorithm. The future work will be focused on using real workload and link the data allocation and data migration together to show the performance of the whole system.

4.5 Summary

The storage performance is improved by optimal allocation of data objects across the storage tiers. We consider this as a data allocation or distribution problem (DAP) among the different tiers based on the hotness of the data and the performance characteristics of the different tiers of storage. We proposed a multiple stage dynamic programming to solve the DAP problem on the storage server. The problem is critical for optimal data migration and overall system performance when the I/O workload contains random accesses. By migrating the hot and random accessed data from low tier to high tiers constantly, the overall latency can be greatly reduced. Due to the size and cost limitation of the available high performance tiers, the number of data objects to be allocated should be optimally chosen so that the total benefit value is maximized. The problem of the DAP can be reduced to the multiple choice knapsack problem and we used a pseudo polynomial algorithm, i.e., multiple-stage dynamic programming (MDP) to solve the problem. The simulation results with various input data

objects and various number of tiers show that the proposed MDP can achieve improvements up to 6-fold compared with the existing greedy algorithm.

Chapter 5

Storage System Performance Enhancements Utilizing Hybrid Drives

5.1 Introduction

Hybrid drives are introduced as one of the Tiers of storage in our proposed Hybrid storage server architecture described in the Introduction section 1.8. In this chapter we consider the problem of optimal utilization of the internal non-volatile cache inside the Hybrid drive to improve system performance. Storage servers used in large scale or enterprise storage systems utilize high RPM (Revolutions per minute) drives to provide higher performance or IOPS. Typically these are SAS (Serial Attached SCSI) drives with 10K and 15K RPM rotational speeds. They have average read/write latencies of around 2*ms* and seek times of around 3*ms* to 5*ms* with IOPS in the range of 175 to 210 [29]. There are two major problems associated with these high performance or Enterprise drives. They are:

1. Higher Performance or IOPS require higher RPM drives. Enterprise Hard drive manufacturers have hit physical limitations in increasing the RPM of disk spindles beyond 15K RPM [126].

2. High RPM drives consume more power to drive the spindles to these high speeds. Research work like that by Gurumurthi et al. [44] have shown that the power consumption of drives increases as a function of RPM.

In this chapter, we address the above problems by introducing a new kind of storage device called a Hybrid drive that combines existing rotational magnetic media with a small amount of Non-volatile memory (NVM) cache consisting of NAND Flash chips. Originally introduced for consumer devices like Laptops and Desktops, Hybrid drives were employed to speed up Operating System (OS) boot times and application performance. For example, the Windows OS used a new feature called ReadyDrive [110] to leverage the Flash inside the Hybrid drive to reduce boot time. While the rotational speeds of the Hybrid drive are relatively low (7.2K or less), by utilizing the embedded NVM cache the performance or IOPS achievable can be close to or greater than high speed Enterprise drives. This eliminates the need for increasing the RPM of the drive, thereby avoiding the first problem (1) of hitting the 15K RPM limit without sacrificing performance. The later problem (2) is also addressed as a additional benefit since low rotational speeds reduces the power consumption. Even if the lower speed and the presence of NVM cache are characteristics of the device, solving the first problem without compromising on performance requires cache management algorithms catered to the need of utilizing the internal NVM cache efficiently. To this end we have developed cache management algorithms to control the placement/eviction of data to/from the internal NVM cache. Through experiments with large scale storage workloads, we show that our algorithms can improve the average IOPS by around 2 to 6 times compared to a baseline SATA drive. For certain workloads it can outperform the IOPS of Enterprise SAS drive. The potential for replacement of SAS drives with Hybrid drives is evident from this result. Our algorithms also have the advantages of minimizing memory and processing overheads of cache management by up to 64% and 48% respectively compared to conventional methods. In order to improve the endurance of the NAND Flash NVM, our placement

algorithm also reduces the write I/O to the cache by up to 150%. To summarize, our main contributions in this work are:

1. Design of Cache monitoring meta-data structures to track hot data on the Hybrid drive.
2. Design of Hybrid drive cache manager with placement and eviction algorithm
3. Development of the Hybrid storage system to demonstrate the advantages of the cache manager

The following section discusses the background and motivation behind our work. Then the design and implementation of the hybrid drive cache manager along with the placement(eviction) algorithm are presented. The detailed experimental results with a functional prototype is explained in the subsequent sections.

5.2 Background and Motivation

In this section we first present a short background on Hybrid drives and then describe how our architecture with Hybrid drives differs from existing conventional SSD/HDD hybrid storage system. Later we explain why current caching solutions are not suitable for Hybrid drives.

Hybrid Drive Components: The schematic of the components inside a typical Hybrid drive is shown in Figure 5.1. The main components are the rotating magnetic media which serves as the primary storage medium as in any other Hard drive and NVM with NAND Flash which acts as a persistent cache. The DRAM component acts as a small temporary read cache typically with capacities of a few Megabytes. The size of the NAND Flash ranges from 16GB to 32GB. The ATA-8 specification [120] provides commands for controlling data placement/eviction to/from this internal Flash. The data can be served from either the NVM during a cache hit or from the magnetic media during a cache miss. While not shown

in the figure, the Hybrid drive also maintains a mapping table to map block regions on the magnetic media to the corresponding location or block address on the Flash for all those cached data. The drive utilizes this during every I/O to locate data either on the Flash or magnetic media.

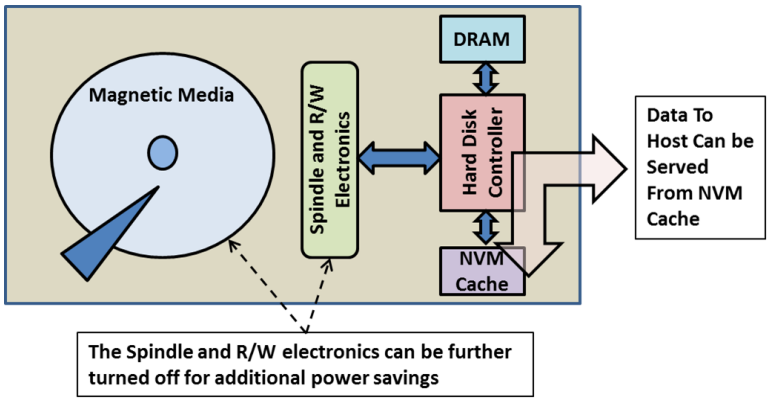

Fig. 5.1 Schematic of a Hybrid Drive

Cache Management in HDD/SSD system vs Hybrid drive system: In Hybrid storage system with separate solid-state cache, or NAND flash SSD [53][39][23][94][112][64], the cache management system must maintain block address mapping between flash and disk. The address mapping is basically a table that contains the translation of Logical block address (LBA) of the block on the HDD to the corresponding cached location on the Flash. With the use of Hybrid disk, the need for such a mapping table is eliminated, as this is pushed into Hybrid drive. However, conventional cache management methods used in system level caching still retain the complexity, particularly the per-block based address mapping, which is a huge overhead in terms of cache lookup performance and efficiency as the number of drives in a RAID [81] configuration increases. For example, for each drive in the RAID configuration, the storage controller has to maintain the mapping between LBA address and the cached block in Flash. Figure 5.2 shows comparison of the conventional storage system consisting of separate SSD and HDD with a Hybrid drive system. In conventional system the storage controller has to maintain the cache mapping table at the individ-

ual block level, whereas I/O from application can be of any size crossing block boundaries. Since the controller had to keep track of all dirty data in the cache that needs to be flushed to the HDD during eviction, it has to manage cache mapping table at the block granularity level. In addition to this, the controller also has to manage data migration between SSD and HDD during cache placement or eviction. In the Hybrid approach (on the right side of the Figure 5.2), the controller complexity is much reduced as both the cache mapping table and the data migration is handled within the drive. The controller does not need to manage cache mapping tables and cache migration (placement/eviction). Moreover when a large array of drives are handled as in a RAID configuration, the overhead of the mapping tables and data migration in conventional approach is increased. Whereas in Hybrid drive cache management, the processing overheads are distributed across the drives. The controller processing is therefore reduced to handle only the primary I/O and the cache control. Another major advantage is that while in the conventional system the placement(eviction) to(from) the cache leads to data movement through the controller, the Hybrid approach eliminates this unnecessary flow. This reduces the interference of the primary or foreground I/O with the cache data migration.

Problems in utilizing existing caching solutions for Hybrid drives: Though there are many caching algorithms that have been previously developed (e.g. like LRU, ARC[69], 2Q [48]), these are mainly applicable only for the first level caching inside an Operating system buffer cache or in an application. Any given caching solution requires continuous monitoring of the I/O workload, maintenance of monitoring meta-data or I/O statistics (like hard disk block cache hits, hit frequency). The following are the problems that were identified related to these requirements:

1. There are no known methods for determining monitoring granularity (or I/O size on disk) of the workload. Conventional system fixes monitoring sizes: E.g. 4KB, 64MB, 1MB (used in research work like [39]). Cold data regions might be cached along with

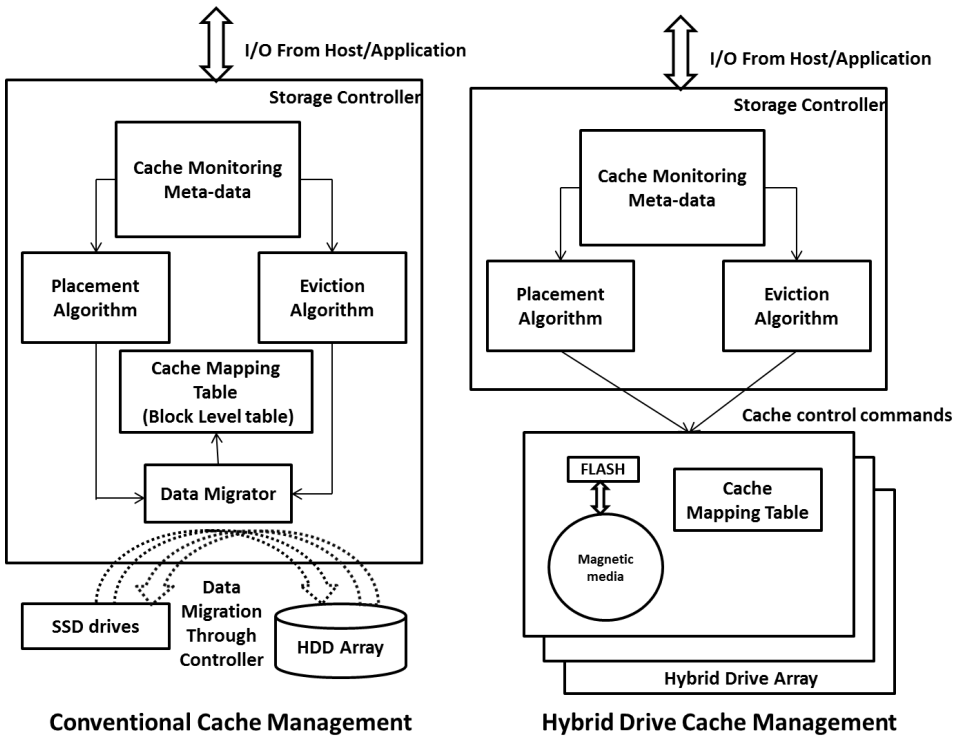

Fig. 5.2 Comparison of conventional SSD caching (left figure) with Hybrid drive caching (right figure).

hot data in the NV cache

2. Hybrid drive NVM cache is limited and should be used only for non-sequential or random I/O. Since hard disk drives provide the best performance for sequential I/O, we do not need to cache data that would be accessed sequentially. For random I/O requests, conventional methods does not consider seek times to make decisions on NV caching.

3. When granularity of the data is small (e.g. 4KB blocks) the amount of monitoring meta-data is large. More time is spent in updating I/O statistics for each block (due to larger search space).

4. Increased write to the NV cache since caching incurs additional I/O [134] due to placement or migration of data from the magnetic media to the Flash based NVM.

Since the capacity of the NAND Flash inside the drive is limited, we need a placement algorithm that selects only hot data and minimizes the writes to cache. The NAND Flash inside the drive might become unusable if the number of cache writes (due to placement) is not controlled.

To address the above problems, a method for hot data region dynamic estimation and selection was developed. This can reduce the amount of monitoring meta-data by 64% compared to a simple set associative hash table based mapping used in caching solution utilized in FlashCache [112]. It can also reduce the cache processing time by up to 48% and improve the IOPS by around 2 to 6 times per drive compared to baseline SATA drive and in some workloads can outperform high RPM SAS drives. The placement algorithm also reduces the number of writes to the internal cache by up to 150% compared to LRU based algorithms. In next section we explain the details of the cache manager design, I/O monitoring data structure and algorithms for cache placement/eviction.

5.3 Hybrid Cache Manager

The Cache manager controls the placement and eviction of drive's hot data on the internal NAND Flash of the Hybrid drive. For this it needs to maintain history of the I/O access to the individual blocks on the drive. The history data relates to the recency and frequency of access to the individual blocks on disk. We call this cache monitoring meta-data. Conventional methods fix the monitoring size or granularity of the data on disk. This can vary from a single block of 4KB blocks (fine granularity) to a large number of blocks in range of Megabytes (course granularity). The granularity determines the accuracy of the classification between hot and cold data. The accuracy in turn determines the effectiveness of utilization of the cache space. Fine granularity tracking leads to accurate classification and thereby leads to optimal cache space utilization. But this increases the monitoring overhead

or meta-data size. Course granularity leads to placement of hot data along with cold data in the cache and thereby leads to poor utilization of the available cache space.

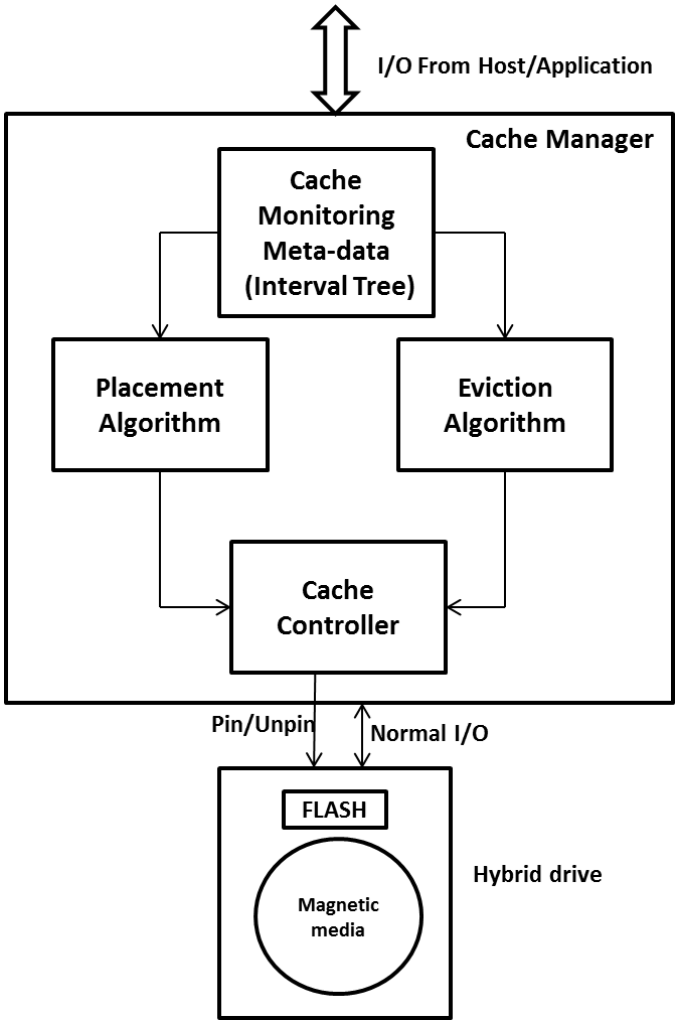

Fig. 5.3 Hybrid Cache Manager Components

We approach this monitoring granularity problem by dynamically determining the size based on the I/O workload. To this end, we utilize an Interval Tree data structure that keeps track of the monitoring meta-data. We keep track of hot data on the cache through LBA ranges or intervals. Instead of fixing the granularity of the monitoring regions, the cache management algorithm adapts the data monitoring size to the I/O workload. The Figure 5.3 shows the components of the Cache Manager. The Cache placement algorithm determines LBA regions that needs to be cached based on hot zones or block ranges on disk. The

Eviction module uses Interval tree and hotness value estimation to select LBA regions to be evicted from the cache. The following sections describe these components in detail.

5.3.1 I/O Monitoring Data: Interval Trees

The Interval Tree is basically a Red-Black tree [11] indexed with LBA ranges. Each node in the tree keeps track of a single hot data region on the disk with the corresponding LBA start and end address. It also stores the hotness value which is used to determine whether the LBA region corresponding to that node has to be evicted when there is contention for cache space. Hot data are tracked in terms of intervals since the cache control commands work in terms of LBA regions. In order to move data from the magnetic media to the NV cache, we need to 'pin' the LBA region or interval to the cache. Likewise we send a 'unpin' command to remove or evict a LBA interval from the cache. The interval tree therefore tracks all those regions or LBA intervals that are pinned to the cache. The figure 5.5 shows the structure of the LBA address range record containing the LBA range entry (LRE) or regions that contains the LBA start address and the range length in terms of disk sectors. This is followed by the time stamp at which this entry was inserted in the tree. The data hotness value or *dhv* which is the next field in the structure is defined using Equation 5.1a. This is the heat value of the block regions on the disk. Frequently and randomly accessed block regions have large *dhv* values. The figure 5.4 shows the structure of the Interval Tree with the Red-Black nodes containing the caching information like the minimum hotness value (*mhv*). The *mhv* of a particular node holds the minimum *dhv* of all the subtree nodes below that level in the tree. This value is the main factor behind the very fast identification of cold data in the entire cache. Later in the evaluation section 5.4 we will see that by tracking data on disk in terms of intervals we can reduce significant amount of the monitoring meta-data overhead compared to Hash based mappings used in traditional SSD based caching solutions. By incorporating *mhv* value in each nodes of the tree we can speed up the searching of the cold

data by up to 6 times. At the same time, the O(logN) search time of the Interval tree (same as a Red black tree [11]) helps in fast updates or book keeping of the I/O statistics counters of each LBA region in the cache based on the incoming workload.

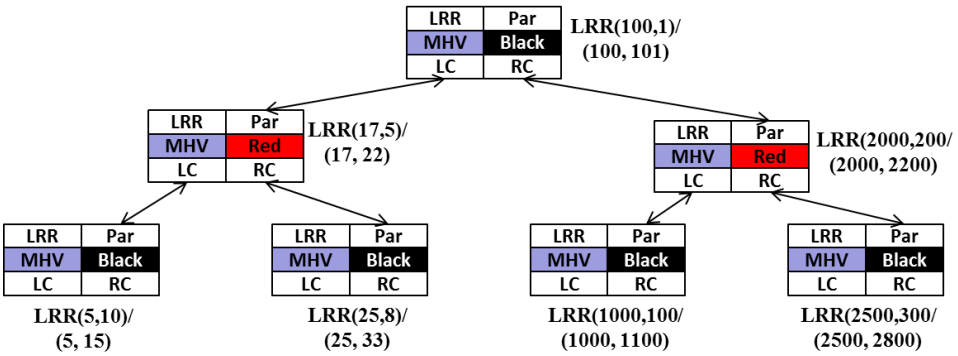

Fig. 5.4 Interval Tree nodes with hot LBA region tracking information

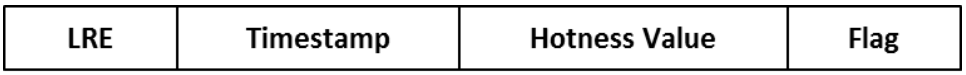

Fig. 5.5 LBA address range record (LRR) structure

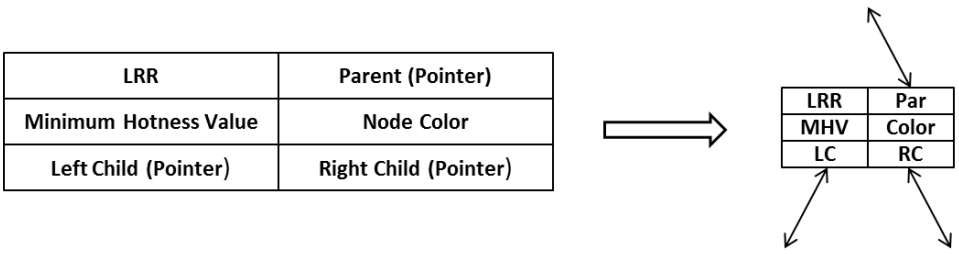

Fig. 5.6 Interval tree node meta data structure

Hot data Estimation: The hotness value of a specific region is estimated based on the Randomness of the I/O to the block, frequency and recency of the access. This value is maintained in each node of the Interval tree representing the LBA region on disk. In addition, it also incorporates the randomness of the I/O based on the LBA access pattern to

the same region. Equation 5.1a is used to estimate the data hotness value (*dhv*):

$$dhv = n \times r + afv \tag{5.1a}$$

$$afv = afv + 1 \text{(when a old node is evicted)} \tag{5.1b}$$

$$r = d_{avg}/S_{lba} \tag{5.1c}$$

In Equation 5.1a, n is the frequency of access to the LBA region and afv is the aging factor value incremented when old nodes are evicted. The afv value is the running counter for each eviction of a block region from the interval tree. Therefore it captures the recency of the access to the blocks. Any new node when added in the tree gets a afv value that is larger than any nodes which already exist. This favors LBA regions that are recently added in the cache to be placed in higher levels of the tree. The value r (5.1c) is the randomness in the request to this LBA, obtained based on a exponential moving average (d_{avg}) of the last LBA distance from the current accessed LBA interval. It is also inversely proportional to the size (S_{lba}) of the block region (similar to [64]). These two values together add to the hotness value of a given region on disk.

5.3.2 Eviction Algorithm

Equipped with Interval tree, the eviction mechanism shown in Algorithm 7 is a simple tree search to find the cache node with the smallest *mhv* value. Since each node stores the *mhv* value which is the minimum of the *dhv* under the subtree, the algorithm traverses the tree by looking in the left and right nodes containing the *mhv* equal to the current nodes value. Then it branches to the node containing the same *mhv* and recursively repeats the search until it finds the node containing the same value of *mhv*. This node is then selected for eviction from the drives cache. Whenever the hotness value of a node is updated due to a cache

hit, the minimum hotness value (*mhv*) is propagated from that node all the way to the root node. This preserves the correctness of the *mhv* value at each nodes which aids the eviction mechanism.

Data: Interval Tree with LBA range address nodes
Result: Cache Node with minimum Hotness Value
1 *currentNode* = *rootNode* ;
2 **while** *currentNode mhv value not equal to dhv* **do**
3 | **if** *mhv of left child equal to mhv of currentNode* **then**
4 | | Assign *currentNode* left child to *currentNode*;
5 | **end**
6 | **else**
7 | | Assign *currentNode* right child to *currentNode*;
8 | **end**
9 **end**
10 **return** *currentNode*

Algorithm 7: Interval Tree Search for minimum hotness value.

5.3.3 Cache Placement Algorithm

Existing caching algorithms like LRU,LFU,ARC [69] focus on identifying data for replacement, but they fail to consider initial placement of selection of hot I/O to be moved in to the cache. This becomes critical considering the limited endurance of the NV cache inside the drive. For example, LRU uses a cache-all mechanism in which every miss leads to the data being moved in to the cache. Other algorithms follow a similar mechanism. This leads to heavy writes to the cache which reduces the life time of the internal NAND Flash. The algorithm proposed here tries to minimize the cache writes based on a hot data selection method. The following is the general principle behind the hot data selection algorithm. The whole disk region is split in to fixed regions (typically 1GB) and based on the I/O the region hotness value is estimated. The per region hotness value is used to decide whether a chunk or region of data need to be moved in to the internal NV cache. The following section explains this method in detail.

Hot zones estimation: We utilize two methods of I/O hot data LBA range selection for moving (or pinning) data in to the internal cache. One is to identify zones that have high I/O access or frequency. New cache nodes within these regions are chosen based on whether the zone level hotness value is greater than a specific threshold. These are called hot zones and any I/O that falls in to these regions are selected for caching. The other method is to identify a specific area within a cold zone that has a higher frequency of access within that zone. The former method performs placement of hot data a little less aggressively as it does not consider small hot regions within the cold zones. The later method is more aggressive in the selection but still tries to select only I/O that fall under specific hot regions within the cold zones. The algorithm utilizes two counters for tracking and estimating the hotness of the regions. I/O cumulative counter $zCIO_k$ tracks the frequency of access within the kth zone. The $gCIO_{avg}$ tracks the average access frequency of the entire disk. Each of these counters are updated based on a active window period to track temporal intensity of the I/O. To determine the hot disk regions within a zone the equation 5.2 is utilized. The $Zone_{size}$ in the equation is the size of the disk zone (fixed as 1GB) and AIO_{span} is the size of the active or hot region within the zone. The exponent of 2 is to make the calculation of the value simple by using bit shifts. The hot region within a cold zone is determined by roughly how many I/O falls in a narrow region determined by the exponential factor. If more I/O falls in a specific region its $zCIO_k$ value will be higher and the hot region will be less narrow. Intuitively we can see that when $zCIO_k$ is almost equal to the $gCIO$ the hot region is almost half that of the size of the zone ($500MB$ for a $1GB$ zone size). For higher $zCIO_k$ value, the region increases to the whole zone size and therefore the placement is done based on the entire hot zone calculation as in the first method.

$$AIO_{span} = Zone_{size} \times 2^{-\left\lfloor \frac{gCIO_{avg}}{zCIO_k} \right\rfloor} \tag{5.2}$$

The algorithm 8 gives the outline of the cache placement decision. It utilizes the two

Data: LBA of the current I/O access
Result: Hot data region Cache Node inserted in the Interval Tree
Obtain z_k zone corresponding to input *lba* in the I/O ;
if $zCIO_k$ *greater than* $gCIO_k$ **then**
    Prepare Interval tree node with LBA address range record ;
    Update *dhv* value of the Record ;
    Insert in to the Interval Tree;
end
else
    Obtain AIO_{span} from equation 5.2 ;
    Obtain Average LBA $zLBA_k$ within the z_k;
    $hotRegionStart = zLBA_k - \frac{AIO_{span}}{2}$;
    $hotRegionEnd = zLBA_k + \frac{AIO_{span}}{2}$;
    if *lba greater than hotRegionStart or lesser than hotRegionEnd* **then**
        Prepare Interval tree node with LBA address range record ;
        Update *dhv* value of the Record ;
        Insert in to the Interval Tree ;
    end
end
return

Algorithm 8: Placement algorithm of LBA intervals based on hot zones and regions.

methods explained based on whether the access to the disk is inside a hot zone or the cold zone. The value $zLBA_k$ tracks the average LBA region within the kth hot or cold zone for the current active time window. This is used along with the value AIO_{span} to determine the LBA range within a cold zone that is active or hot. Figure 5.7 shows an example of how the hot data regions are estimated based on the equation 5.2. The narrow triangles within the region represent the hot data regions that are estimated based on the exponential component of the equation. As the I/O to a specific region varies based on the workload pattern the hot regions within the zone or the shape of the triangles also changes. Any I/O access falling within this hot region is selected for placement in the cache and a new tree node is inserted in to the Interval tree to track I/O to this region.

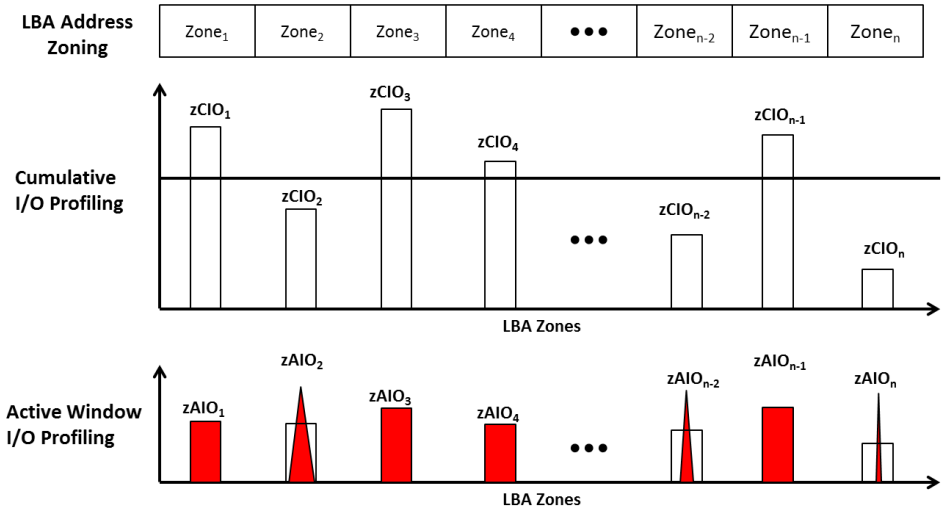

Fig. 5.7 Disk hot regions or zone estimation

5.4 Performance Evaluation

5.4.1 Evaluation Setup

The hybrid cache manager was implemented on Linux as a block device mapper kernel module. The Prototype Hybrid drive used had up to 32GB of internal NV cache space. We control the utilized size of this cache in our experiments. This prototype drive supports the NV cache command set of the ATA-8 specification [120]. The command set describes the SATA feature set to 'pin' and 'unpin' disk sectors to/from the Hybrid drive's internal NVM cache. The hardware used was an Intel Xeon system with 4GB RAM. The testing was done for a single disk configuration. For the workload, published traces were used from the SNIA block traces repository at [119] and the UMass repository at [121]. The table 5.1 shows a short description of the different traces used in the experiments. The block traces were replayed through a user space program.

Table 5.1 Traces Used In The Evaluation Of Hybrid Drive Cache Manager

Trace	**I/O requests in trace**	**Trace Type**
Web Proxy [119]	168,638,964	Webserver Proxy
Project Server [119]	23,639,742	Software development/build machine
Exchange Trace [119]	7,450,837	Email server trace
TPCC [119]	2,458,938	TPC-C Database transactions
Financial1 [121]	4,899,020	OLTP
Financial2 [121]	3,698,864	OLTP

5.4.2 Performance Metrics

Systems like FlashCache, FlashTier [94], HyStor [23] and MixedStore [53] all use hash based mappings to map HDD blocks to corresponding Flash blocks. Therefore we evaluate our Interval-tree based cache meta-data management against a generic hash based management. For the performance or IOPS improvement we compare against both a baseline drive without the NVM cache and with a enterprise or high RPM drive. In line with the hybrid drive caching design goals, the evaluation metrics considered are caching meta-data overhead savings (memory and update times), cache write reduction and IOPS.

5.4.3 Reduction In Caching Meta-data Overhead

Graph 5.8 shows the percentage reduction in the number of caching meta-data nodes in the hybrid caching scheme. This is compared against a hash table based method used in FlashCache. The reduction is largely due to the monitoring granularity determined by the I/O sizes in the workload. While the methods in FlashCache and other caching system need to map each block (typically the File system block of 4KB) to the corresponding blocks in the cache, the Interval-tree approach stores a large LBA range. This leads to less number of tracking meta-data as an entire range of blocks (several Kilobytes) is monitored. The range

maintained in each of the tree nodes is also adjusted based on the access pattern. Therefore we get reduced meta-data utilization without losing the accuracy of classifying hot and cold data. The reduction in meta-data can be up to 65% as shown in the graph.

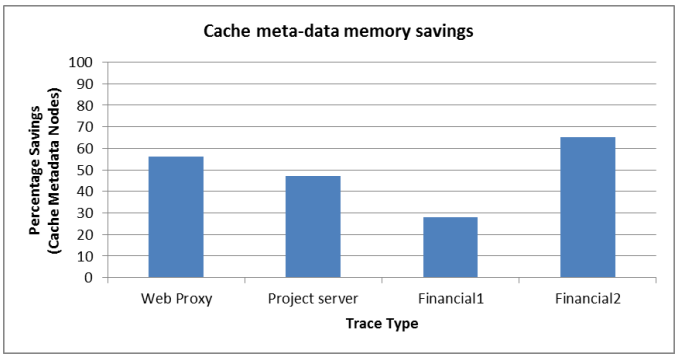

Fig. 5.8 Caching meta-data savings for different traces compared to hash based mapping

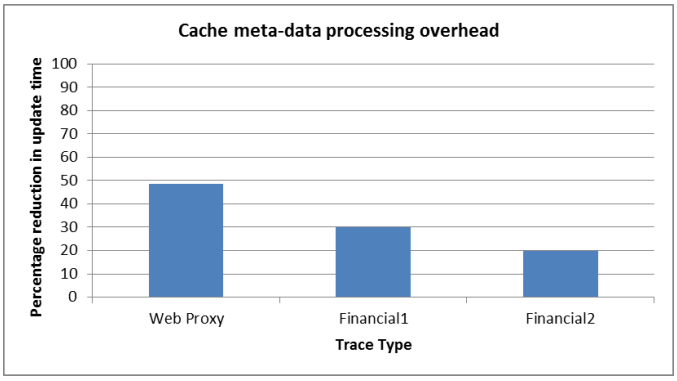

Fig. 5.9 Cache meta-data update savings for different traces compared to hash based mapping

We can also see that the overhead or processing time for cache meta-data updates (like frequency and hotness value) is also reduced by up to 48% as shown in Figure 5.9. The trace for Project server trace gave more than 100% savings in update times and therefore not shown in the graph. Even though a hash table lookup is faster than a Interval-tree, as the number of cache meta-data nodes are large the hash table leads to collisions. In a chained hashing implementation these collisions ends up with a linear search of the meta-data nodes. In contrast, the Interval-tree approach always gives a O(logN) search time to update the nodes since the tree is always balanced.

5.4.4 Cache Write Reduction

The second criteria we evaluate is the reduction in cache writes. This is critical since the endurance of the NAND Flash inside the drive is limited. The hot I/O dynamic selection algorithm explained in section 5.3.3 prevents frequent movement of data into the cache thereby reducing the writes. Figure 5.10 shows the reduction in writes to the Hybrid drive's internal cache as the size of the cache is increased from 2GB to 16GB for the Exchange trace. The corresponding drop in hit rate is shown in Figure 5.11. Both of these plots are compared with respect to a LRU like algorithm as used in FlashCache.

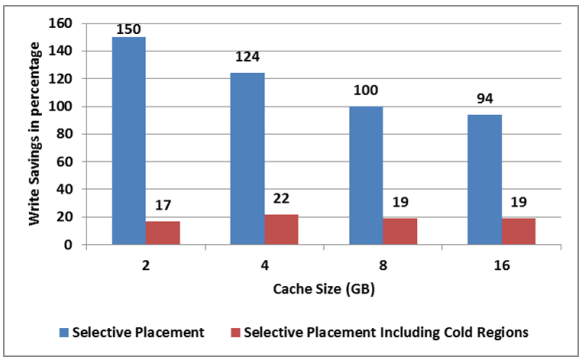

Fig. 5.10 Cache write reduction compared to LRU and similar algorithms

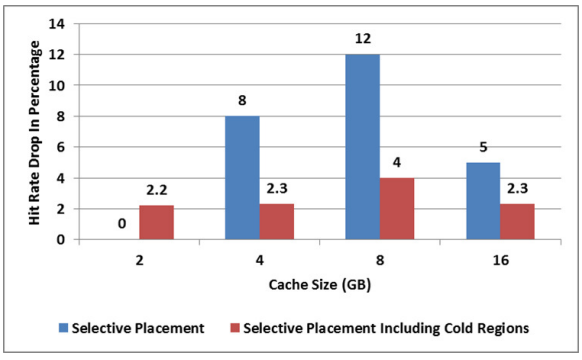

Fig. 5.11 Cache Hit rate drop compared to LRU and similar algorithms

The hit rate for the Exchange trace varies from 27% to 57% as the cache size is varied from 2GB to 16GB. Figure 5.11 shows a slight drop in the cache hit rate as the algorithm selectively places data in the cache unlike LRU. This results in some hot data not being

placed into the cache earlier as would have been for LRU. The decision to move into the cache is done later after these data become relatively hot. The hit rate is close to the LRU case when the cache size is around 16GB. At this cache size, the selective placement can reduce 94% writes to the cache with almost the same cache hit rate as LRU. We also evaluate the more aggressive placement which selects even I/O that falls within small hot regions in the cold zones. The hit rate drop is around 5% in this case compared to LRU. With this slight drop in the cache hit rate we can still get around 20% savings in the cache writes. Whereas the first approach trades-off the cache hit rate for nearly 94% cache write savings, the second approach trades-off write savings for cache hit rates close to that of LRU.

5.4.5 Performance Improvements

Finally we compare the performance of the Hybrid caching solution against the raw disk performance. Specifically the I/O per second (or IOPS) is measured for the TPCC and Microsoft exchange server under a single disk Hybrid drive configuration. These are representative traces of large scale server storage workloads collected. The experiments were done for 8*GB* and 16*GB* cache sizes. From Figure 5.12 we can see that the Hybrid caching can achieve more than twice the performance of the raw disk for the TPCC workload. For the Exchange workload the IOPS can reach up to 5 times the baseline performance as seen from Figure 5.13. There are certain regions in the trace where there is poor cache hit and therefore the performance drops to near the baseline. This is due to the nature of this particular trace having poor temporal locality in the access pattern in that region. Since this is an intrinsic characteristics of the workload any other algorithm cannot achieve better performance than this. Though the performance drops in a specific region, on the average we can get an IOPS close to around 2 times higher than the baseline for the Exchange trace. It should be noted that these increase in IOPS can be achieved with a lower rpm drive (5.4K rpm) and a small NAND Flash cache. The major advantage of our approach is that this increase in perfor-

mance can be achieved with reduced caching I/O to the internal Flash thereby improving its life time. While LRU and similar algorithms can achieve close to this performance, they incur huge overheads in meta-data updates, caching I/O and maintenance of mapping tables as we have seen from the evaluation results in the previous sections (5.4.3), (5.4.4).

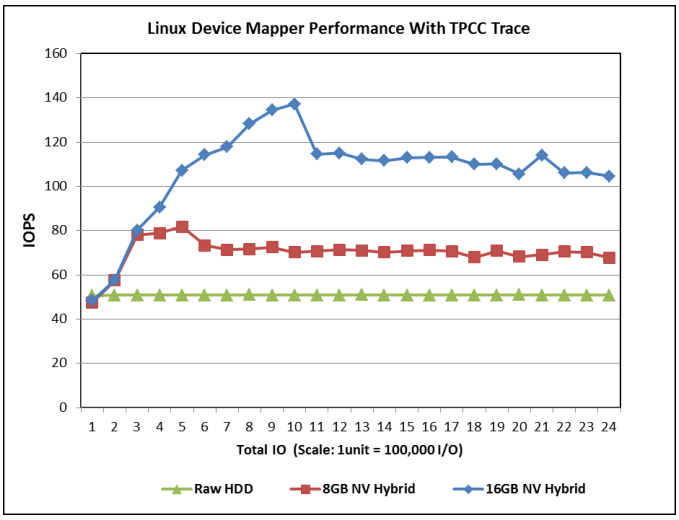

Fig. 5.12 Hybrid drive IOPS improvement per drive for TPCC workload compared to baseline HDD. X-axis Scale: 1 unit = 100,000 I/O operations

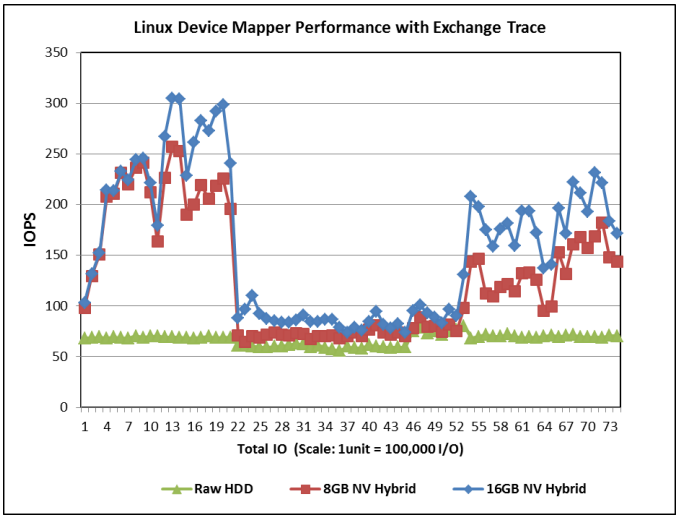

Fig. 5.13 Hybrid drive IOPS improvement per drive for Microsoft Exchange workload compared to baseline HDD. X-axis Scale: 1unit = 100,000 I/O operations

5.5 Performance Evaluation On Enterprise Storage System

In addition to the Linux prototype, we also implemented our dynamic cache management inside the storage controller on the Fujitsu Eternus Enterprise storage system [3]. The system runs an embedded OS with minimal memory (DRAM) and CPU resources. Our efficient Interval-tree based cache meta-data management and update mechanism proved to be well adapted for such an environment. Our evaluation setup was a Fujitsu Enternus storage array populated with Hybrid drives with our dynamic cache management module running in the storage controller. A RAID5 volume was created on the Eternus system connected to the host computer over Fiber Channel interface. The traces were replayed on the host using a user-space program. The traces used were the same as that used for the Linux prototype evaluation. We also benchmarked the system using HammerDB [114] benchmarking tool run on the host computer. Finally, we also measured the energy savings of Hybrid drive storage system compared to that of a system with SAS drives. The following sections describe our experimental results in detail.

5.5.1 TPCC and Exchange Traces Performance

Since the user-space program used on the host for replaying the traces was able to issue only synchronous I/O to the drives, this was equivalent to measuring IOPS on a per drive basis. In other words, the figures [5.14, 5.15] show graphs of IOPS per drive that are similar across all the five Hybrid drives used in the RAID5 configuration. Measurements were made for both 4GB and 8GB internal NV cache size configurations for all the 5 Hybrid drives. We find results similar to that of the Linux prototype as evident from the shape of the curves. But for the TPCC workload we find a huge improvement in the IOPS of over 3 times compared to the baseline HDD. Since the TPCC trace did not have more than 2.4 million I/O, we

could not see the IOPS improvement beyond this point. The exchange trace shows similar pattern as before with a drop in IOPS in the center of the plot. But the peak IOPS can reach more than 6 times the baseline values. The results also show that for the Exchange trace the IOPS can exceed that of a high speed SAS drive with IOPS of around 175 to 210. It also exceeds the IOPS of an Enterprise drive from Seagate [4] that we evaluated having average of 200 IOPS for the Exchange and TPCC workload. We can achieve this with a relatively low speed 5.4K rpm drive with a modest cache size of 8GB. With such a drive we can drastically reduce the power consumption without much loss in performance. Later in section 5.5.4, we show through experiments the power consumption reduction compared to high speed SAS drives.

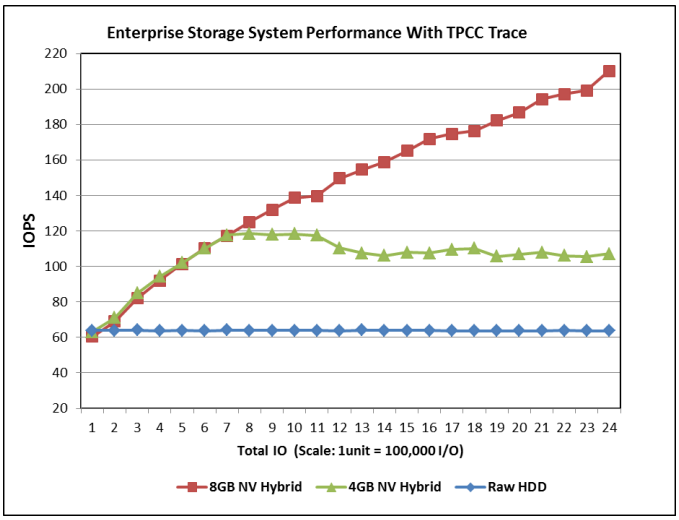

Fig. 5.14 Hybrid drive IOPS improvement per drive for TPCC workload on Storage System prototype (RAID5 configuration)

5.5.2 Performance Improvements compared to commercial Hybrid drives

As another experiment, we compared the performance of the prototype drive with our caching algorithm with some of the commercial Hybrid drives. We chose two of the popular models from Seagate and Toshiba which had an internal Flash cache of size 8GB. Our prototype drive was also configured to use 8GB of cache. The caching algorithm used inside

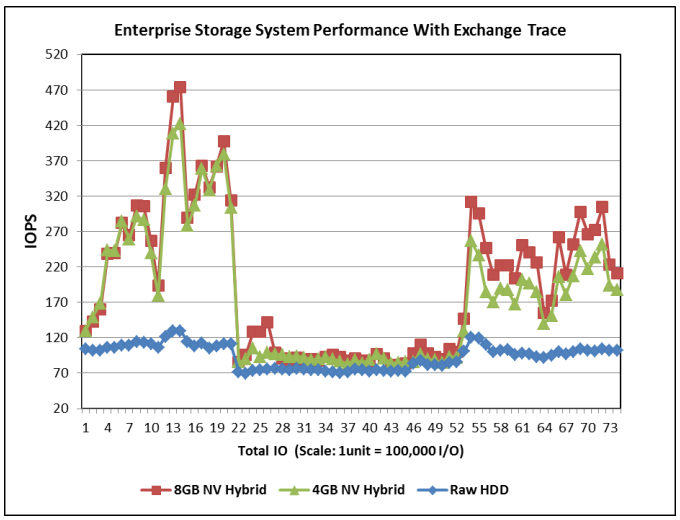

Fig. 5.15 Hybrid drive IOPS improvement per drive for Microsoft Exchange workload on Storage System prototype (RAID5 configuration)

these drives are hidden and proprietary. Therefore we treated it as a black box and compared it against our caching method. As a representative enterprise workload, the TPCC [119] trace was replayed on these drives and the IOPS was measured. It should be noted that even though a single drive was used, the same IOPS per drive can be achieved with a RAID-5 configuration with multiple drives (for example with 10 drives). We can see from these tests that our Prototype drive with the new caching scheme can achieve more than twice the read IOPS of the Seagate/Toshiba drives. Similar improvement of twice the IOPS in performance can be seen for the write I/O. The main reason for the increased performance is that our algorithm has prior knowledge (based on statistics) of hot regions on disk and uses this to make decisions on selective placement/eviction of blocks to/from the cache.

5.5.3 Database Performance Improvement

We used the HammerDB [114] benchmarking tool to evaluate the performance of our cache management implementation in the Fujitsu storage system. We used warehouse database benchmarking with the standard TPCC profile. The profile used 45% new orders, 43% payment transactions, 4% order delivery, 4% order status queries and 4% stock level sta-

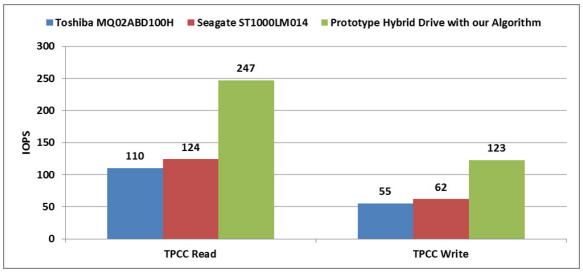

Fig. 5.16 Prototype Hybrid Drive IOPS comparison with popular commercial drives. The IOPS was measured using replay of TPCC trace

Test	**TPM**	**NOPM**
Without Cache Management	15,308	234
With Hybrid Drive Cache Management	37,618	575

Table 5.2 HammerDB Benchmark Tests On The RAID5 (5 drives) Storage System Prototype. TPM represents transactions per minute and NOPM represents new orders per minute

tus queries. The size of the database used was 88GB with 1000 warehouses. The number of concurrent users was set at 100. The measurement metrics used was the Transactions per minute (TPM) and new orders per minute (NOPM). The table shows the comparison between the Hybrid Drive storage system with and without our cache management implementation. In both cases RAID5 configuration was used with 5 drives. We can see that our cache management can improve the TPM and NOPM by 2.4 times compared to the baseline without caching.

5.5.4 Energy Savings

To show the energy savings in using Hybrid drives over Enterprise (high RPM) drives, we did power consumption measurements when running the TPCC workload. The Enterprise drive used was a Seagate SAS drive [4]. The prototype Hybrid drive we used was similar in characteristics to a Toshiba 5.4K RPM SATA drive [2]. Both these drives were setup in a RAID5 configuration and the TPCC trace was used for benchmarking the power consump-

tion. First, we measured the power consumption of the RAID5 configuration with the SAS drives. The same experiment was repeated for the Hybrid drive RAID5 array. Both Figure 5.17 and Figure 5.18 shows the power consumption over a period of 20 to 30 seconds. We observed the same stable power consumption pattern throughout the trace run. Even though the graphs show the power consumption for a single drive in the RAID5 configuration, it is the same or similar for other four drives. For the Hybrid drive case, the power measurement was made after the cache warm up time of around 2 hours. It should be noted that the total period of the original trace was around 7 hours. The warm up time was required for the cache management algorithms to populate the Hybrid drive's internal NVM cache. From figure 5.17, we can see that the average power for the SAS drive was around 5.33 Watts with a standard deviation of 0.2 watts. Compared to this, the Hybrid drive consumes only 2.22 Watts average power. The Hybrid drive power consumption is reduced by more than half that of the SAS drive. Based on this, we can conclude that the overall RAID5 Hybrid drive array should consume half the power of the SAS disk array.

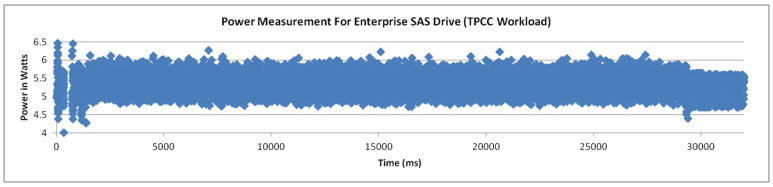

Fig. 5.17 SAS drive (7.2K) Power Consumption per drive for TPCC workload (RAID5 configuration). Average: 5.33 Watts, Standard Deviation: 0.2 Watts

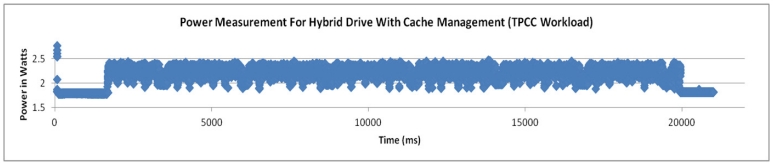

Fig. 5.18 Hybrid drive (5.4K) Power Consumption per drive for TPCC workload (RAID5 configuration). Average: 2.22 Watts, Standard Deviation: 0.09 Watts

5.6 Related Work

To our knowledge, the work described here is the first to utilize Hybrid drives in Enterprise or large scale storage systems with the internal NVM cache controlled by the storage system controller or host. Enterprise storage systems handle workloads that are I/O intensive and are server workloads with large number of concurrent clients or users (like database transactions, web servers). This is in contrast to Desktop workloads which are usually single user and less I/O intensive. While the Hybrid drive has been introduced for Desktop and Laptop environments to improve performance and save energy, little or no work has considered practical application in main stream Enterprise or server storage. The closest we came across is the work done by Timothy Bisson et al. in [16]. In this paper they utilize the hybrid disk's non volatile cache to improve the write performance or latency. Even though they can improve the latency by around 70%, the evaluation was done for Desktop workloads and they do not consider improving read IOPS through caching. The other approach for hybrid storage utilized in practice and research is the combination of HDD and SSD in large scale storage systems. Such hybrid storage systems employ techniques like automatic tiering and caching between the HDD and SSD to improve performance of frequently accessed data. In both storage tiering and caching, frequently accessed data with non-sequential access pattern is moved from HDD to faster storage like SSD to increase the IOPS or latency. This is because SSD offer lower latencies (in terms of tens of microseconds) compared to HDD (in terms of few milliseconds) for non-sequential or random access. The difference between tiering and caching is mainly in terms of the time it takes for the system to respond to changes in I/O behavior and take action like moving the data from HDD to SSD. For tiering typically the system responds in few minutes to hours whereas for caching it is done in terms of seconds or even milliseconds [10].

In our work, we focus on caching algorithms. Although there are several caching algorithms in practice like LRU, LFU, ARC [69] and LIRS [47], these are mainly used for

main memory (DRAM) caching. These methods cannot be used for caching inside the Hybrid drive since they increase the cache writes to the Flash in addition to the normal or foreground I/O as we have shown in our results. This severely effect the endurance of the NAND Flash as it has limited endurance. There are several works that try to improve the endurance of SSD through techniques like using disk based write caches [97], using buffering techniques with next generation NVM [101], and improving caching/tiering algorithms [134]. These techniques are applicable to hybrid storage with separate SSD and HDD devices. The Hystor system [23], FlashCache [112] and FlashTier [94] makes use of SSD as a cache for HDD. But they suffer from huge overhead to maintain mapping tables external to the drive. Even if we ignore this overhead as this is handled by the Hybrid drive itself, the overhead in the block level cache mappings is still higher compared to our Interval-tree approach as our results in section 5.4.3 show. The work in [78] tries to reduce the hot data monitoring and identification overhead by using random sampling of the I/O. But this is applicable only to hybrid storage systems with separate SSD and HDD.

5.7 Summary

We have developed dynamic cache management algorithms to control the placement and eviction of hot random access data on the non volatile cache inside a Hybrid drive. A prototype system on Linux was developed with our cache management and evaluated with prototype Hybrid drives supporting the ATA Non volatile NV cache command set [120]. We have shown that our method can achieve around 64% reduction in caching meta-data compared to state-of-art systems. Our algorithm can also achieve more than 48% reduction in caching meta-data updates compared to other methods that used hash map like structures. By introducing these algorithms in the Hybrid drive storage system we can achieve 2 to 6 times improvement in performance (IOPS) compared to a baseline SATA drive. For certain work loads it can outperform the IOPS of high speed drives. We have also developed a

prototype Enterprise RAID5 Storage system with Hybrid drives and have shown similar improvements in IOPS compared to an array of SAS drives. We have also shown more than 2 times improvement in performance of Hybrid drive array when running Database benchmarks with HammerDB. The other major advantage obtained from the experiments is the savings in power while using Hybrid drives. There was more than 2 times reduction in power consumption when Hybrid drives are used compared to the Enterprise SAS drives. This further strengthens our claims in this work on the advantages of integration of Hybrid drives in the large scale storage system architecture. Due to their low RPM and higher performance we envision a future of Hybrid drives replacing the conventional high speed drives in the Enterprise or large scale storage system space enabling Datacentres with large storage capacities, higher performance and lower energy consumption.

Chapter 6

Metadata Placement And Load Balancing

6.1 Introduction

In the previous chapters [3-5] we have seen various algorithms and techniques for improving performance on the primary data path or in the storage servers attached with multiple tiers of storage devices. We have seen how hybrid devices like the Next generation NVM/SSD combination and Hybrid drives can improve the performance of the storage system significantly. In this chapter, the main focus is on improving the performance on the Meta-data I/O path. Meta-data is the file system information required for locating files and the associated namespace is the directory hierarchy maintained by any typical File System. The location information are stored in data structures called inodes and the directories contain list of inodes mapping to user files. We assume file system data structures like in ext4 [66], zFS [92] and other Unix file systems. We call this collectively as Metadata and File system namespace. In this chapter we look in to the problem of distributing Metadata and file system namespace across a cluster of Metadata servers (MDS) to improve load balancing and performance. To this end, we have developed a locality preserving consistent hash-

ing scheme called DROP (Dynamic Ring Online Partitioning), that can partition meta-data uniformly across the cluster while preserving namespace locality at the sub directory level. While the partitioning serves the capacity load balancing we also need to handle request load balancing. Request load balancing ensures that meta-data operations are uniformly distributed across the cluster and no single server is heavily loaded. We have developed a cache distribution mechanism to achieve the request load balancing.

Compared to the overall data, the size of meta-data is relatively small, and it is typically 0.1% to 1% of data [73], but it is still large in Exabyte scale file systems, e.g., 1PB to 10PB for 1EB file systems. Besides, 50% of all file system accesses are meta-data I/O [59]. In order to achieve high performance and scalability, a careful meta-data distribution mechanism must be designed and implemented to avoid potential bottlenecks caused by meta-data I/O. To efficiently handle the workload generated by a large number of clients, meta-data should be properly partitioned so as to uniformly distribute meta-data traffic by leveraging the MDS cluster. At the same time, to deal with the changing workload, a scalable meta-data management mechanism is necessary to provide good meta-data performance for mixed workloads generated by tens of thousands of concurrent clients [124]. A well-designed MDS cluster should be able to achieve satisfactory storage load balancing. In addition, we have to efficiently organize and maintain very large directories [80], each of which may contain billions of files.

The following are the main contributions from this work:

1. A key-value based meta-data distribution system with locality preserving Hashing that maintains the namespace locality across the MDS cluster thereby improving the performance of certain file system operations (like listing directories)

2. Capacity load balancing mechanism using a solution to the 0-1 Knapsack problem to distribute meta-data for uniform utilization of storage capacity across the MDS cluster

3. Request load balancing mechanism using a distributed meta-data cache. We solve

a similar 0-1 Knapsack problem to achieve popularity based meta-data distribution across this distributed cache.

4. Meta-data performance evaluation on published File system traces to show the effectiveness of the above approaches compared to directory and file name based hashing (DirHash and FileHash). DirHash and FileHash are popular methods used in research work and open source/commercial clustered large scale file system.

6.2 MDS Cluster

The MDS cluster distribution is shown in Figure 6.1, where a typical standard hash table evenly partitions the space of possible hash values of file system meta-data. The file system meta-data is stored as a key-value pair, with the keys mapping path names of files/directories to the corresponding file or directory inodes/meta-data. Current hash-based mapping does not evenly partition the address space into which keys get mapped, causing some meta-data servers getting a larger portion of the keys. To address this, virtual nodes are used as a means of improving load balancing ([99], [56]), each participating independently in the DROP overlay network, thus each server's load is determined by summing over several virtual nodes. Virtual nodes not only make re-distribution become easier, but also scaling out as data grows. When scaling out, more physical MDS may be added and virtual nodes can be moved onto them seamlessly.

DROP achieves namespace locality and load balancing by allocating more virtual nodes for each physical meta-data server since meta-data IDs are not uniformly distributed. Since meta-data structures are small, they are typically not so expensive from the perspective of storage space, thus it is not a serious problem. The major problem which we may need to consider with the architecture in Figure 6.1 is the network bandwidth. In general, to maintain connectivity of the overlay network, every node frequently sends heart beat messages

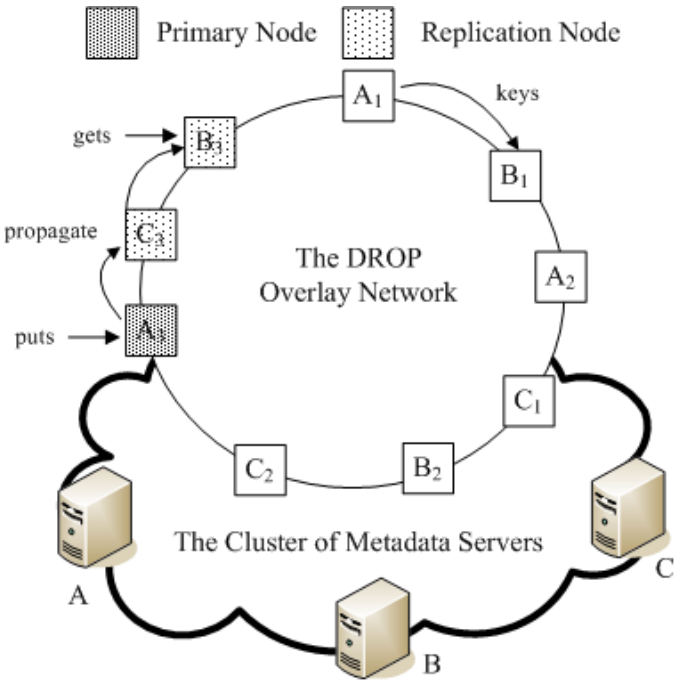

Fig. 6.1 MDS cluster showing the DROP distributed Meta-data management. The overlay network is similar to chord [99] with distributed hashing like in Cassandra [56]

to its neighbors to make sure they are still alive. The nodes are replaced with new neighbors if they are not alive any more. To maintain the DROP network, therefore involves communication overhead across the network. In our case this, since the MDS cluster resides within a data center, there will be enough bandwidth to support this communication unlike Chord [99]. Therefore we borrow the Chord like design but utilize our own meta-data distribution mechanism to namespace preserve locality as required by some of the workloads.

Figure 6.2 shows the components of the Meta-data server consisting of DROP and C^2. The DROP component handles the locality preserving hashing and dynamic capacity load balancing. C^2 component handles the adaptive request load balancing using a distributed lookup cache. The SSD/NVM key value store is utilized for storing the meta-data items. Subsequent sections explain the function of these components in detail.

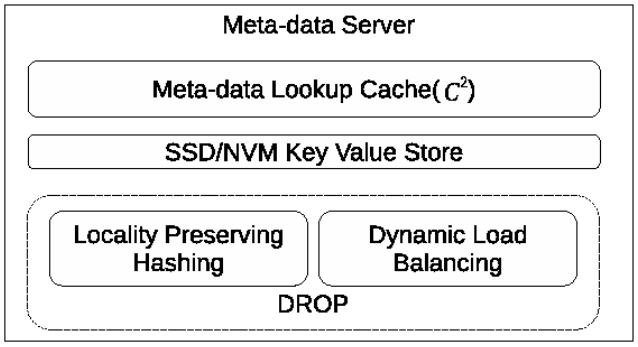

Fig. 6.2 Meta-data server components with DROP and C^2 for capacity load balancing and request load balancing respectively

6.3 Locality Preserving Hashing

Namespace locality is achieved by placing meta-data of files belonging to the same parent directories on a single MDS server in the cluster. For sub directories in the file system namespace that contains files as well as another level of directory, the entire sub directory could reside in the same MDS. This reduces the communication to a single MDS for operations like listing entire sub directories (e.g a stat on entire subdirectory contents). Instead of fetching the file system meta-data or inodes from multiple MDS server, a single MDS returns the result with additional performance boost due to a single large sequential read on the disk storing the meta-data. For e.g. the contents of $/usr/$ could reside on one MDS server and $/usr/bin$ on another. In our approach, we use a fixed size key that encodes the entire path name. Figure 6.3 shows the encoding of the file system path components in the keys. The file path is encoded with the first 40 bytes, and each directory is encoded with 2 bytes. For longer paths, the next 4 bytes are reserved for the rest of path since 40 bytes are only sufficient for 20 path levels. The traces utilized in our experiments [76], [33] (see table 6.2) contain 0.001%, 0.018% and 0.0% of files with long nested paths.

The cumulative distribution function (CDF) of path lengths in the three traces are shown in figure 6.4. We can see that the proportion of the longer paths is smaller as the level of

the directory nesting increases. For example, as shown in table 6.2, as the Harvard trace does not contain many deep paths (maximum path length 18) it's CDF peeks well below path length of around 5. The last 4 bytes of the key is allocated for the file name thereby accommodating 2^{32} files per directory. The key encoding described here provides a good trade-off between key size and file count, and it enables naming of new files and directories. In addition, a file may be moved to a different directory, and its key can be quickly changed to reflect the new path using the encoding mechanism. Furthermore, related meta-data items are organized into a group using it to preserve in-order traversal of file system, e.g., files in the same directory are related.

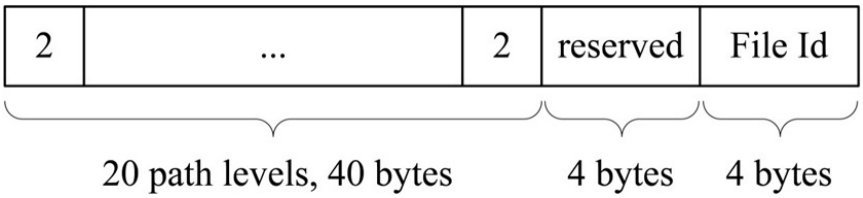

Fig. 6.3 Encoding of file system paths

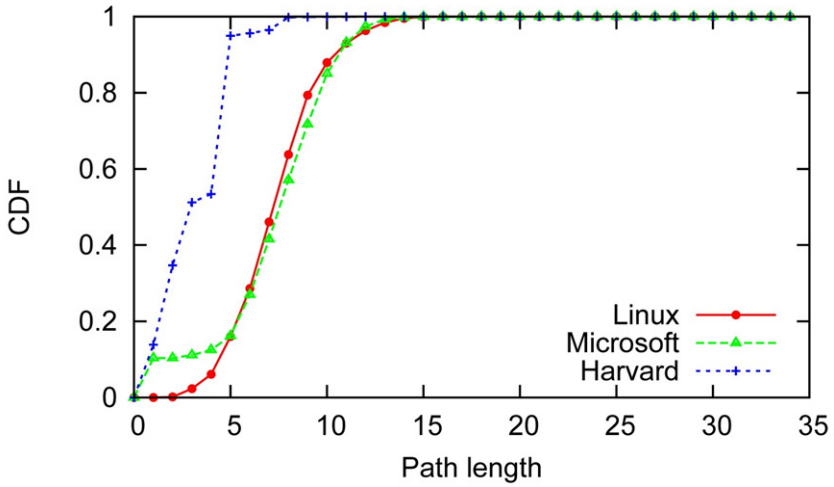

Fig. 6.4 CDF of file paths for file system traces

6.4 Capacity Load Balancing With DROP

We propose a simple load balancing algorithm that achieves meta-data placement with minimum loss of locality in the namespace. The algorithm is fully distributed and similar to other dynamic load balancing schemes [123]. The main mechanism behind the meta-data load balancing is the maintenance of key range density histogram across the MDS cluster. The range density of the keys is the number of keys maintained by a specific virtual meta-data node starting with a minimum value of the key and ending with the maximum value. This range density of the keys are exchanged between the servers to determine whether a specific MDS is overloaded (in capacity). This exchange of the histograms allows each MDS server to obtain the average load L of the system. With the average load L and the histograms from other MDS, each server can also determine the lightly loaded regions in the cluster. From this information any heavily loaded MDS can send requests to the lightly loaded MDS to take on some of the load (or keys) from its allocated key space. Thus the cluster gets dynamically balanced based on the current state of the distribution of the keys.

6.4.1 Histogram based Load Balancing

For a given set of m meta-data servers $S = \{s_i, i = 1,2,...m\}$ and a set of n virtual nodes $V = \{v_j, j = 1,2,...n\}$, each virtual node v_j has a weight w_j that represents the number of files in a range that are maintained by v_j. Each meta-data server s_i has a remaining capacity (weight) W_i that represents the difference between the average storage load (or capacity) W and the existing weight of the meta-data server s_i. We approach the load balancing problem as a 0-1 knapsack problem. The solution is to determine the reassignment of n virtual nodes

to m meta-data servers so that it minimizes the underutilized space on the MDS:

$$\text{Maximize: } z = \frac{1}{\sum_{i=1}^{m} s_i} \tag{6.1a}$$

$$\text{subject to: } \sum_{i=1}^{m} x_{ij} = 1, j \in N = \{1,2,...,n\} \tag{6.1b}$$

$$\sum_{j=1}^{n} w_j x_{ij} + s_i = W_i y_i, i \in M = \{1,2,...,m\} \tag{6.1c}$$

$$x_{ij} \in \{0,1\}, y_i \in \{0,1\}, i \in M, j \in N \tag{6.1d}$$

Where

$$x_{ij} = \begin{cases} 1 & if \text{ Virtual node } j \text{ is assigned to MDS } i \\ 0 & \text{Otherwise} \end{cases} \tag{6.2}$$

$$y_i = \begin{cases} 1 & if \text{ MDS } i \text{ is used} \\ 0 & \text{Otherwise} \end{cases} \tag{6.3}$$

$$s_i = \text{Space left in MDS } i \tag{6.4}$$

The constraint 6.1b restricts each virtual node to be assigned to only one physical server. The number of files assigned to each meta-data server should be less than or equal to its capacity. This is maintained by the constraint 6.1c. Constraint 6.1d shows it is a 0-1 Knapsack problem.For example, if there is a meta-data server A, which has three neighbors B, C and D. They include virtual nodes as shown in Table 6.1, where the number represents the load of a virtual node. There are a set of virtual nodes $V = \{3,2,7,6,2\}$ that will be reassigned to light MDS $S = \{C,D\}$, which have the remaining capacities 11 and 12 respectively. After solving the 0-1 MKP, we can see that there is a rough load balancing from Table 6.1.

Table 6.1 Example meta-data load balancing in DROP

MDS	Metadata Items	Removed Meta-data Items	Result
A	{3,2,7,12}	{3,2,7}	{12}
B	{15,6,2}	{6,2}	{15}
C	{1}	{}	{1,2,7,2}
D	{}	{}	{3,6}

6.5 Request Load Balancing With Distributed Meta-data Cache

DROP meta-data distribution takes care of the capacity load balancing during additions and removal of new servers or during non-uniform placement due to locality preserving hashing. While this addresses the problem of capacity load balancing, we still need to handle the meta-data operations or request load balancing across the MDS servers. To deal with potentially unpredictable shifts in the request workload, e.g., flash crowds [125], we introduce an adaptive request load balancing approach called C^2. C^2 periodically calculates new assignment of meta-data items to the distributed MDS cache based on dynamic meta-data workload changes. The load balancing is done based on the popularity of the meta-data items. After the new assignment, the workload gets uniformly distributed across the MDS cluster resulting in reduced latencies for the meta-data operation thereby improving the application performance. The periodic re-assignment is required to deal with shifting meta-data workload patterns across the cluster. By monitoring a running workload of requests to meta-data items, C^2 calculates a new load-balancing plan and then migrates them when their request rates are more than the request capacity of the MDS node that maintains them. C^2 utilizes a meta-data lookup cache where the *key* is the path name of a file and *value* is its

inode data.

6.5.1 Meta-data Cache System Model

A physical meta-data server might have a set of $\mathbb{N}$ virtual nodes $\mathbb{N} = \{n_j, j = 1,2,...d\}$ with a set of loads $\mathbb{L} = \{l_j, j = 1,2,...d\}$. Load is applied to meta-data servers via their virtual nodes, i.e., meta-data server S might have load $L_S = \sum_i^d l_i$. A MDS is said to be load-balanced when it satisfy Definition 1, i.e., the largest load is less than t^2 times the smallest load in the DROP system. Here the factor t could be less than or equal to 2 following Definition 1. When the value of t is equal to 1 and the inequality (in Definition 1) is satisfied the system will be in perfect load balancing as each MDS will be receiving just the average load of the entire system. According to Definition 1, a MDS has an upper target L_u ($L_u = t \times \bar{L}$) and a lower target L_l ($L_l = 1/t \times \bar{L}$). If a MDS finds itself receiving more load than L_u, it considers itself overloaded.Otherwise, it considers itself underloaded if it finds itself receiving less load than L_l . MDSs may want to operate below their capacities to prevent variations in workload from temporary overload.

Definition 1: *MDS_i is load balancing if its load satisfies* $1/t \leq L_i/\bar{L} \leq t (t \leq 2)$

File popularity follow Zipf request distributions [33]. It states that a small number of objects are greatly popular, but there is a long tail of unpopular requests. A Zipf workload means that destinations are ranked by popularity. The Zipf law states that the popularity of the ith-most popular object is proportional to $i^{-\alpha}$, in which α is the Zipf coefficient. Usually, Zipf distributions look linear when plotted on a log-log scale. Figures [6.5,6.6] shows the popularity distribution of file/directory meta-data items in the Microsoft and Harvard traces. Like the Internet, the meta-data request distribution as observed in both traces also follows Zipf distributions.

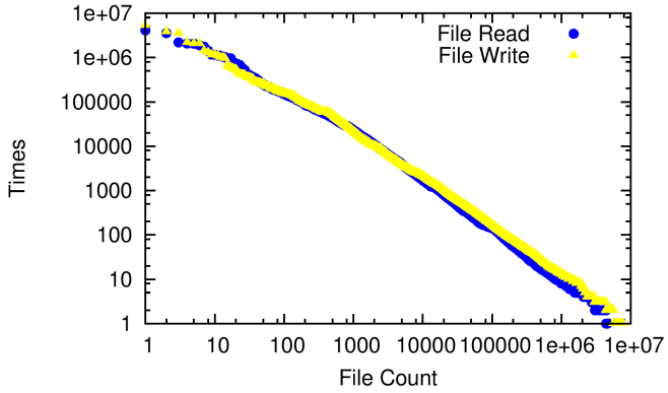

Fig. 6.5 Zipf like distribution for the Windows trace. X-axis showing file count and Y-axis shows the number of times files are seen in the trace

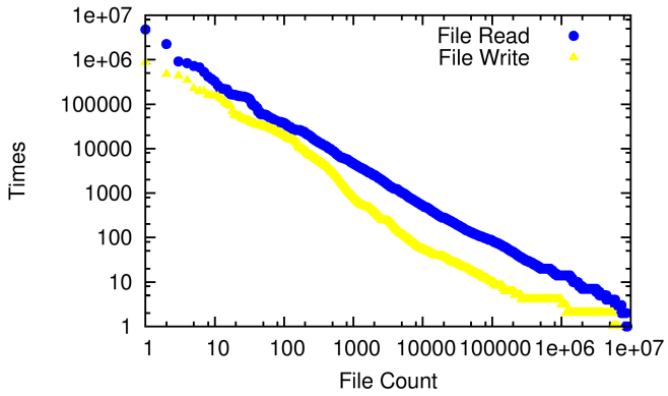

Fig. 6.6 Zipf like distribution for the Harvard trace. X-axis showing file count and Y-axis shows the number of times files are seen in the trace

6.5.2 Load Shedding and Stealing

We use two methods to balance the request load across the MDS cluster. The first method, Load shedding attempts to offload requests to one or more underloaded MDS when a node is overloaded. The other method, Load stealing seeks out to take load from one or more overloaded nodes when a MDS is underloaded.

Load Shedding: There are m meta-data items in a node, with a tuple of loads $\langle l_1, l_2, .., l_m \rangle$ and a tuple of probabilities $\langle p_1, p_2, p_3, ..., p_m \rangle$. When this node has a cache of size $c > 0$, the c most frequently requested items will all hit the cache of this node, with two tuples of positive numbers $\langle l_1, l_2, l_3, ..., l_c \rangle$ and $\langle p_1, p_2, p_3, ..., p_c \rangle$ respectively. Let L be $t \times \bar{L}$, and this

node is overloaded if $\sum_i^c l_i > L$. Therefore, it can be formulated as a 0-1 Knapsack Problem that is NP-hard, i.e., it is to determine how to reassign some items to other nodes in a way that minimizes meta-data migration from this node as follows:

$$\text{Maximize: } z = \sum_{i=1}^{c} p_i x_i \tag{6.5a}$$

$$\text{subject to: } \sum_{i=1}^{c} l_i x_i \leq L \tag{6.5b}$$

$$x_i \in \{0,1\}, i \in \{1,2,...,c\} \tag{6.5c}$$

Where x_i is the ith item from the m meta-data items in a particular node or meta-data server.

Load Stealing: There are a number of meta-data items from previous nodes with tuples of positive numbers $\langle l_1^{'}, l_2^{'}, l_3^{'}..., l_c^{'}\rangle$ and $\langle p_1^{'}, p_2^{'}, p_3^{'}..., p_c^{'}\rangle$ respectively. If $\sum_i^c l_i < L$, this node is in load balancing, and it can take some items with its cache space of $L - L_e$. Here L_e is defined by equation 6.6d. Therefore, this can also be formulated as a 0-1 Knapsack Problem, i.e., it is to determine how to take some items from other overloaded nodes in a way that maximizes the cache utilization of this node as follows:

$$\text{Maximize: } z = \sum_{i=1}^{c^{'}} p_i^{'} x_i \tag{6.6a}$$

$$\text{subject to: } \sum_{i=1}^{c^{'}} l_i^{'} x_i \leq L - L_e \tag{6.6b}$$

$$x_i \in \{0,1\}, i \in \{1,2,...,c^{'}\} \tag{6.6c}$$

$$L_e = \sum_{i=1}^{c} l_i \tag{6.6d}$$

6.6 Evaluation

The prototype system was built on top of the FAWN [7] key value store on Linux. We first evaluate DROP comparing its meta-data distribution efficiency with DirHash and FileHash. There are three traces we analyze as shown in Table 6.2. Microsoft traces represent the Microsoft Windows build server production traces [76] from BuildServer00 to BuildServer07 within 24 hours, and its data size is 223.7GB (including access pattern information). Harvard is a research and email NFS trace used by a large Harvard research group [33], and its data size is 158.6GB (including access pattern information). We implemented a meta-data crawler that performs a recursive walk of the file system using stat() to extract file/directory meta-data. By using the meta-data crawler, the Linux trace is fetched from 22 Linux servers in our data center. The file system meta-data size is 4.53GB, and data size is 3.05TB in this trace. Based on the Linux trace, we perform two estimations:

1. Storing 1 trillion files, the meta-data size is $441TB$ and the data size is 290PB by computation

2. Storing 1EB data, the meta-data size is $1.56PB$ and the number of files is 3.53 trillion files by computation

Table 6.2 Traces used in the evaluation of DROP

Trace	**Number of files**	**Metadata size**	**Maximum path length**
Microsoft [76]	7,725,928	416M	34
Harvard [33]	7,936,109	176M	18
Linux	10,271,066	786M	21

6.6.1 Namespace Locality And Capacity Load Balancing

File system namespace locality improves the performance of common file system operations like listing directories or doing regular expression search on entire subdirectory. Locality is

measured as follows: $locality = \sum_{j=1}^{m} p_{ij}$, where p_{ij} (0 or 1) represents if a subtree path p_i ($\in \mathbb{P}$) is located in MDS j. The metric represents the number of meta-data servers to which the items under a path $\mathbb{P}$ is split across. Figures 6.7, 6.8, 6.9 represents namespace locality comparisons of three level paths on the three traces using FileHash, DirHash and DROP. The figures show that DROP has better namespace locality than DirHash and FileHash for the three traces. The percentage above a box is calculated as follows: $\frac{N-S}{S} \times 100$, where S is the number of MDS using Subtree (S=1), N is the number of MDS using one of other three approaches. DROP performs close to that of static subtree partitioning except the first level paths in both the Linux trace and the Microsoft Windows trace. For first level paths DROP can only achieve suboptimal namespace locality using locality-preserving hashing, i.e., assigning keys that are consistent with the order of pathnames. For DirHash and FileHash, the order of pathnames is not considered so that namespace locality is lost. Note that we do not plot the results of static subtree partitioning since each path is maintained by only one metadata server according to its definition, but its load imbalance is severe.

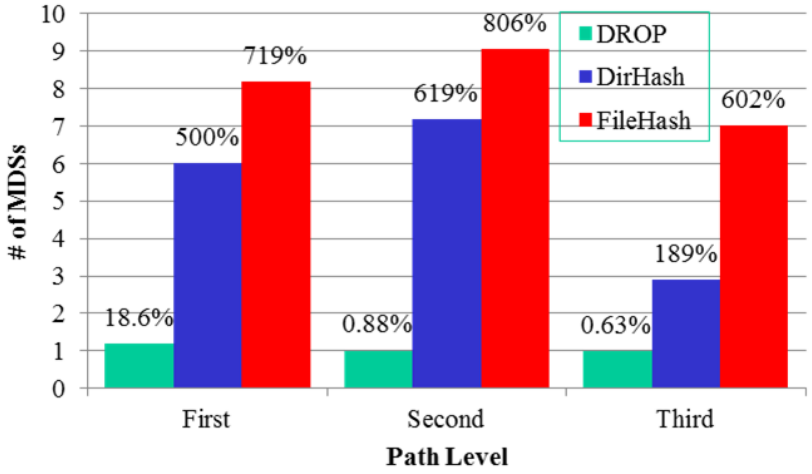

Fig. 6.7 DROP namespace locality for Linux trace.

We then show scalable load distribution with different sizes of MDS cluster. We utilize three metrics: median load, maximum load and minimum load. Note that we use median

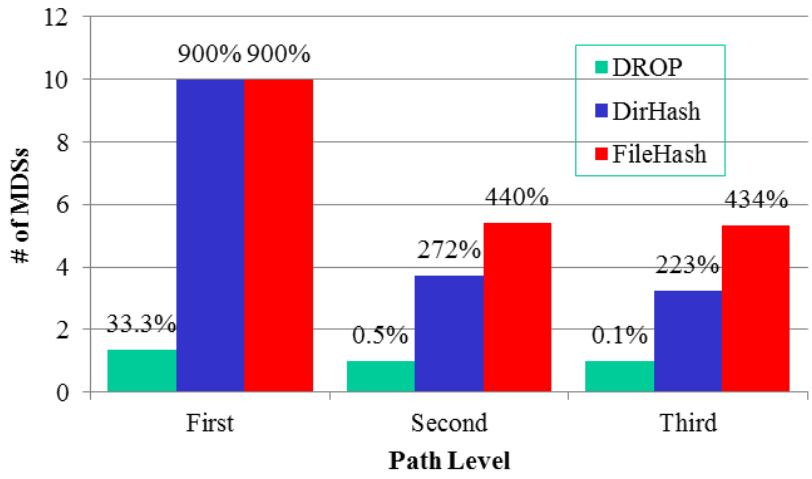

Fig. 6.8 DROP namespace locality for Microsoft trace.

load instead of average load, and we do not show the results of static subtree partitioning because its scalability is sub-optimal in load distribution. In Figures 6.10, 6.11 and 6.12, the vertical bars indicate the spread of the number of meta-data items across the different meta-data servers. Larger size of the vertical bar indicates non-uniform distribution of meta-data in the cluster. We can see that DROP's load distribution is suboptimal (large vertical bar) than DirHash and FileHash when the MDS cluster size is small. But it can achieve similar load distribution with the best namespace locality as DirHash and FileHash when the MDS cluster size is greater than around 20 servers. Therefore, DROP can preserve locality close to that of DirHash and FileHash while at the same time has uniform distribution.

6.6.2 Request Load Balancing in C^2

In this section we evaluate the request load distribution that can be obtained with our C^2 mechanism. This was done through simulation experiments with an event-driven simulator. We define *Load Factor* as follows $LoadFactor = \frac{Max.Load}{Min.Load}$. Each MDS has five virtual nodes, and *Linux* trace follows the Zipf distribution with $\alpha = 1.2$. All the simulation experiments are conducted on a Linux server with four Dual-Core AMD Opteron(TM) 2.6GHz

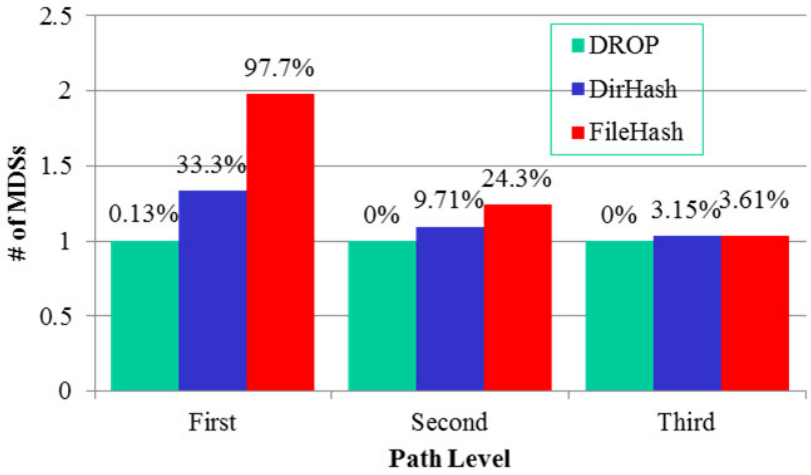

Fig. 6.9 DROP namespace locality for Harvard trace.

processors and 8GB of RAM, running 64-bit *Linux*.

We first measure request workload distribution using the three methods (DirHash, FileHash, DROP) for the three traces. Figures shows the experimental results with and without C2 in request workload distribution. Figure present the load for all the metadata servers, while the internal figures present the load for the first 50 most loaded virtual nodes in the system. The left side of the figures shows that the most loaded MDSs are the ones with the highest number of received requests. The figures confirm that most of the MDSs have roughly similar load, thus achieving good load balancing after running C2. Figure 6.16 and 6.18 illustrate that all the approaches have similar trend in both the Linux and Harvard traces because the two traces are based on the same path naming scheme. In Figure 6.17, we can see that DROP, DirHash and FileHash without C2 have big differences in request workload distribution. This is because there are only three first-level directories in the Microsoft trace, and related storage locations are frequently accessed. In the Microsoft trace, the top 10 metadata items in access frequency are from “Disk3:”. Due to the property of locality-preserving hashing, they are maintained by the same node or a few of the nodes, causing a high load on the server. Figures [6.13,6.14,6.15] show the load factor as a function

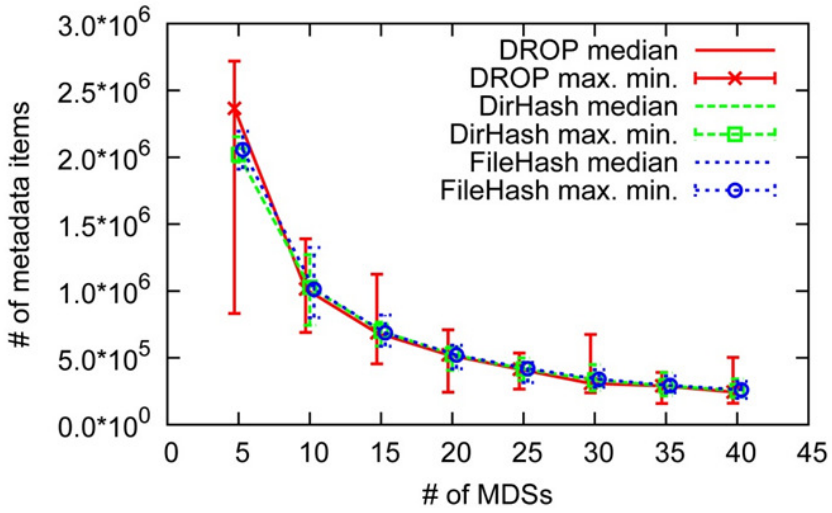

Fig. 6.10 DROP meta-data load distribution for Linux trace. The vertical bars indicate the spread of the number of meta-data items (minimum and maximum) across the different servers. A smaller spread indicates uniform distribution across the entire cluster. DROP can reach uniform distribution as the number of servers reach 25

of increasing the meta-data servers in the cluster with C^2 for the three traces.

6.7 Related Work

Single Metadata Server Architectures: While some of the clustered and distributed file systems [[1], [113]] utilize a singled MDS architecture that simplifies placement, it becomes a bottleneck and a single point of failure. Distributed file systems, e.g., Coda [93], partition their namespace statically among multiple storage servers, so most of the meta-data operations are centralized. Other distributed file systems, like GFS [35], have a single MDS, with a fail-over MDS that becomes operational if the primary server becomes unavailable. In GFS, File system meta-data is stored on a dedicated server called master, while application data is stored on data servers called chunk servers. At any time only one MDS is operational, which is a source of a potential bottleneck as the number of clients and/or files increases.

Clustered and Distributed Metadata Server Architectures: In Clustered and Dis-

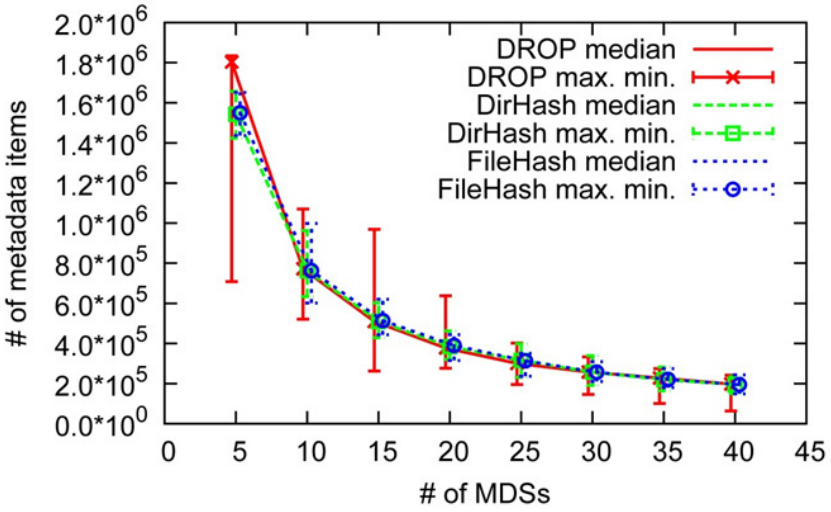

Fig. 6.11 DROP meta-data load distribution for Microsoft trace. The vertical bars indicate the spread of the number of meta-data items (minimum and maximum) across the different servers. A smaller spread indicates uniform distribution across the entire cluster. DROP can reach uniform distribution as the number of servers reach 25

tributed Metadata server architectures, meta-data and namespace are distributed uniformly across the servers. Each MDS can re-distribute the meta-data items across the cluster dynamically based on the current workload. This ensures high performance and prevents hot spots on specific MDS within the cluster. Ceph [122] has a cluster of meta-data servers using a partitioning algorithm that can dynamically adapt to changes in the Metadata access patterns. It maps Namespace subtrees to individual MDS. When a specific subtree is hot or accessed frequently it redistributes the items under the subtrees to less loaded MDS in the cluster [123]. Lustre [1] has an implementation of clustered namespace that stripes the directory contents over multiple meta-data servers. This leads to distribution of the meta-data items under a single directory in the namespace to multiple MDS, thereby improving the scalability of the system.

Hash-based Metadata Distribution: One of the common Metadata distribution mechanism is hashing of pathnames and file names for placement. The computed hash is used to both store and locate the meta-data item on a specific MDS. Vesta [27] and zFS [92] leverage pathname hashing to locate meta-data. While hashing provides a uniform distri-

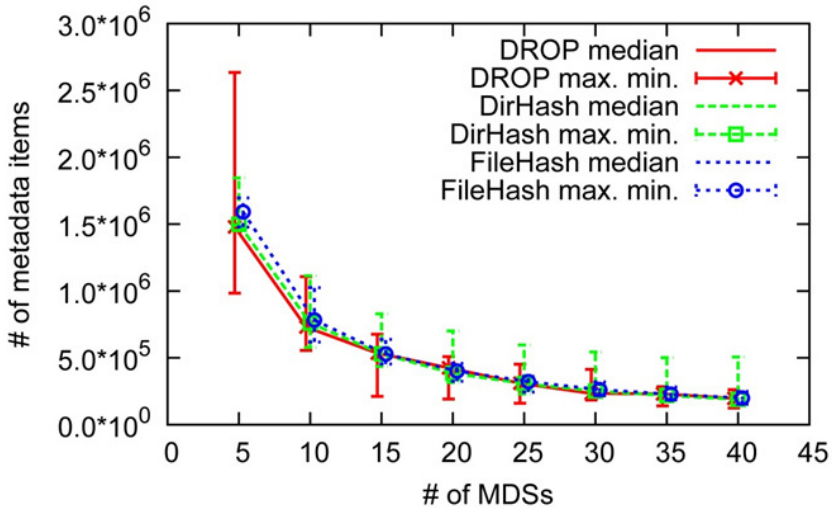

Fig. 6.12 DROP meta-data load distribution for Harvard trace. The vertical bars indicate the spread of the number of meta-data items (minimum and maximum) across the different servers. A smaller spread indicates uniform distribution across the entire cluster. DROP can reach uniform distribution as the number of servers reach 25

bution of the meta-data across a given number of servers, it destroys the inherent locality of the file system namespace. This incurs additional cost for certain file system operations. For example to verify user access permissions on a particular file, the system has to contact many servers for traversal of the parent directories contained in the full path. This leads to communication overheads for such simple operations.

Subtree Partitioning: Static subtree partitioning [124] tries to avoid the locality problem of Hash based distribution by assigning entire Subtrees in the directory hierarchy to different meta-data servers. The main disadvantage in this approach is that it scales poorly when there are hot spots on specific subtrees. This leads to a few meta-data servers to be heavily loaded impacting system performance. Dynamic subtree partitioning [122] overcomes this problem by redistributing the hot spots to lightly loaded servers.

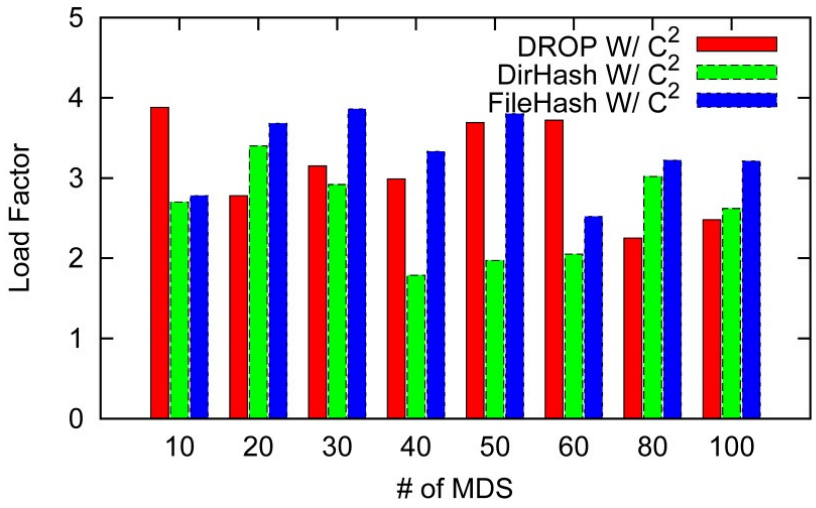

Fig. 6.13 Load factor as a function of increasing servers in C^2 with Linux Trace

6.8 Summary

We have developed the DROP meta-data distribution system prototype. Our prototype can preserve meta-data locality using locality preserving hashing. It can also dynamically distribute the meta-data among the servers in the cluster based on the dynamic load on the system. When storage load changes, it utilizes the HDLB strategy to quickly redistribute the meta-data. Even after the re-distribution, DROP tries to preserve the namespace locality. In addition to preserving locality, DROP can balance the meta-data storage load as optimal as a hash-based mapping. Compared to other distributed meta-data management techniques, DROP brings multiple advantages, such as balancing meta-data storage load efficiently, high scalability and no bottlenecks with negligible additional overhead. We also have shown that the distributed metadata caching method C^2 can achieve good request load balancing along with the capacity load balancing of DROP.

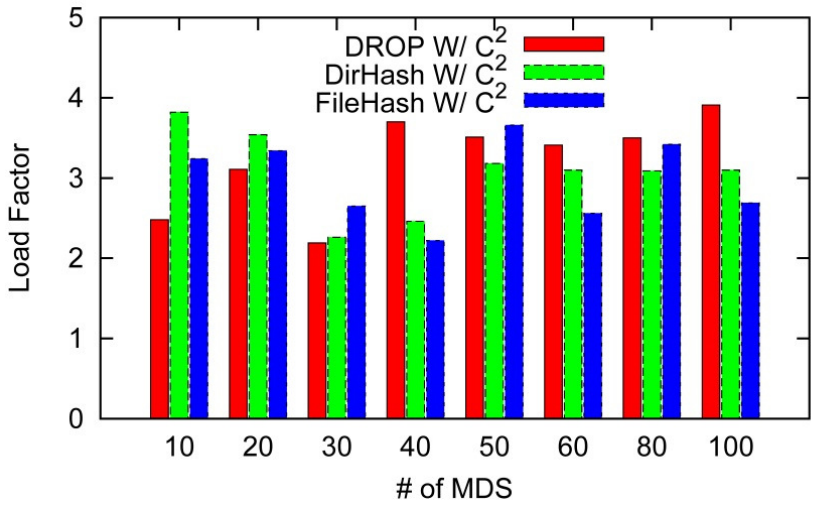

Fig. 6.14 Load factor as a function of increasing servers in C^2 with Microsoft Trace

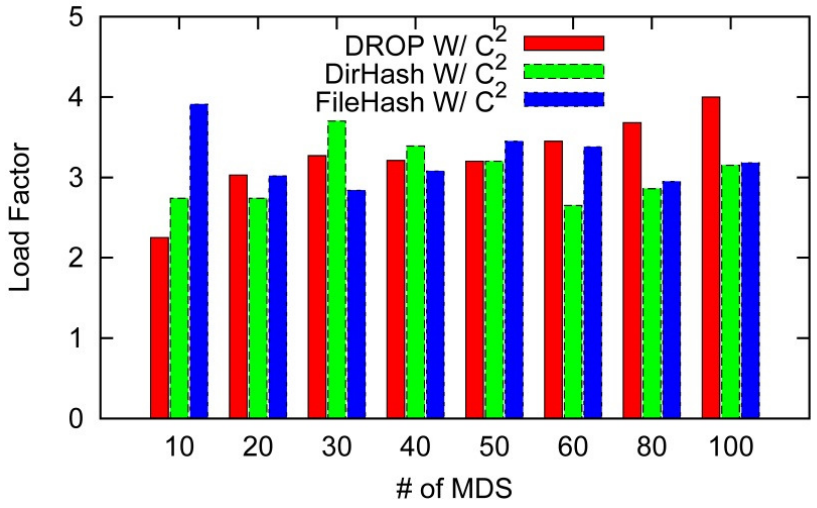

Fig. 6.15 Load factor as a function of increasing servers in C^2 with Harvard Trace

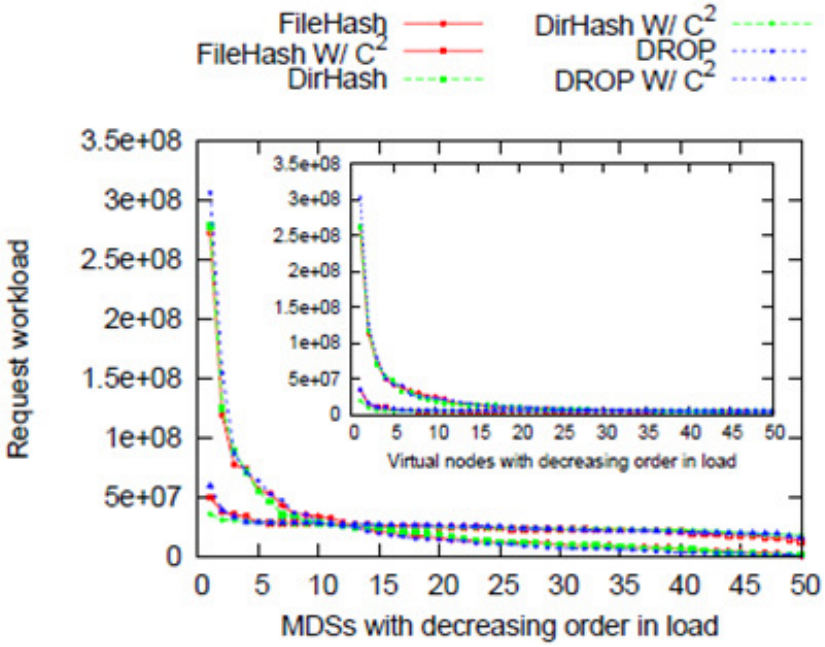

Fig. 6.16 Meta-data request load balancing in C^2 with Linux Trace for a cluster of 50 servers. Inset shows the load distribution for the top 50 VMs (running on the physical meta-data servers).

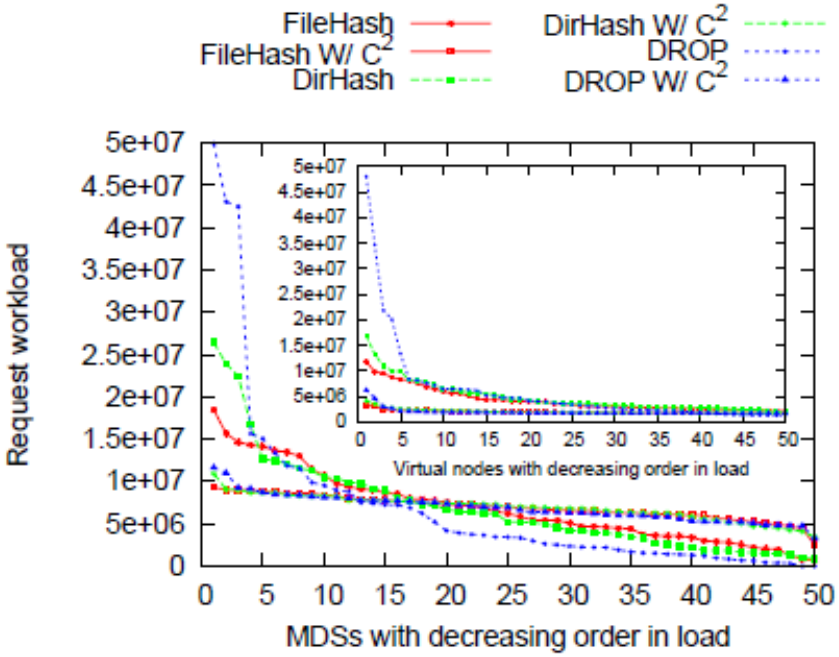

Fig. 6.17 Meta-data request load balancing in C^2 with Microsoft Trace for a cluster of 50 servers. Inset shows the load distribution for the top 50 VMs (running on the physical meta-data servers).

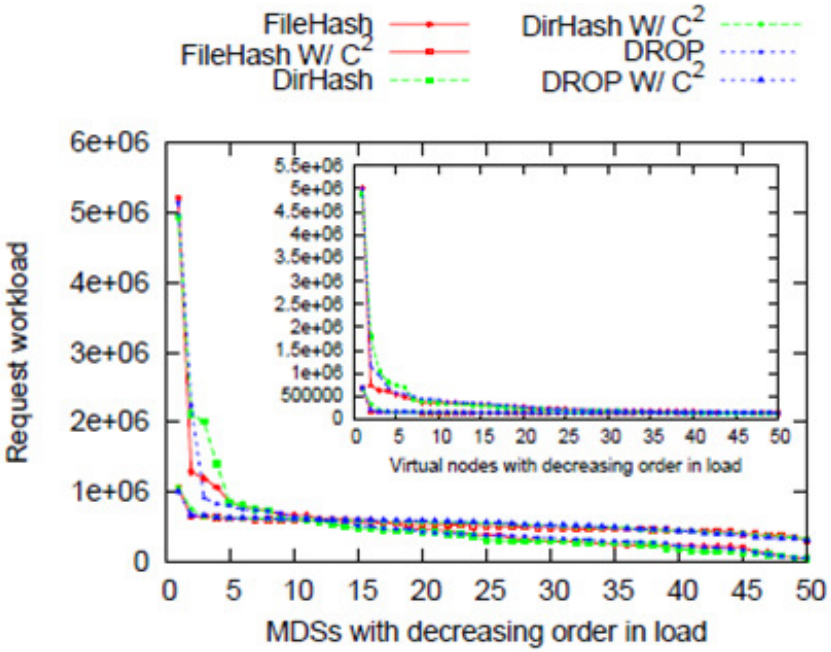

Fig. 6.18 Meta-data request load balancing in C^2 with Harvard Trace for a cluster of 50 servers. Inset shows the load distribution for the top 50 VMs (running on the physical meta-data servers).

Chapter 7

Conclusion And Future Work

7.1 Conclusion

The thesis has explored and proposed various techniques and algorithms to improve the performance of large scale storage systems. Specifically we focused on improving the performance of storage elements or components in a distributed storage system like Ceph, Luster, GlusterFS. The major storage elements or components being the Storage server for primary data and the Meta-data server for serving meta-data requests (like file creations and directory operations). On the Storage server side we have seen the architectural changes and new algorithms for managing caching and tiering when new devices like next generation NVM and Hybrid drives are introduced. For improving meta-data performance we have introduced algorithms for meta-data placement, distribution and load balancing in a cluster of servers.

These work has culminated in the development of a large scale test bed that was deployed in one of the Data centers in A*STAR Data storage Institute (Singapore) who have been providing the funding and support for major parts of the work in this thesis. The test bed deployed integrates all these components to serve as a storage platform for I/O demanding applications (Database, Web servers, Email etc). While the algorithms and storage

components described in the thesis are developed independently, the final goal was the integration of these components on the Ceph file system. We utilize Ceph as the base platform and have replaced some of the storage components/algorithms with those described in this thesis. The hybrid drive, data migration and caching components were integrated with the Ceph object store and the DROP algorithm from Chapter 6 replaces the Dynamic subtree partitioning.

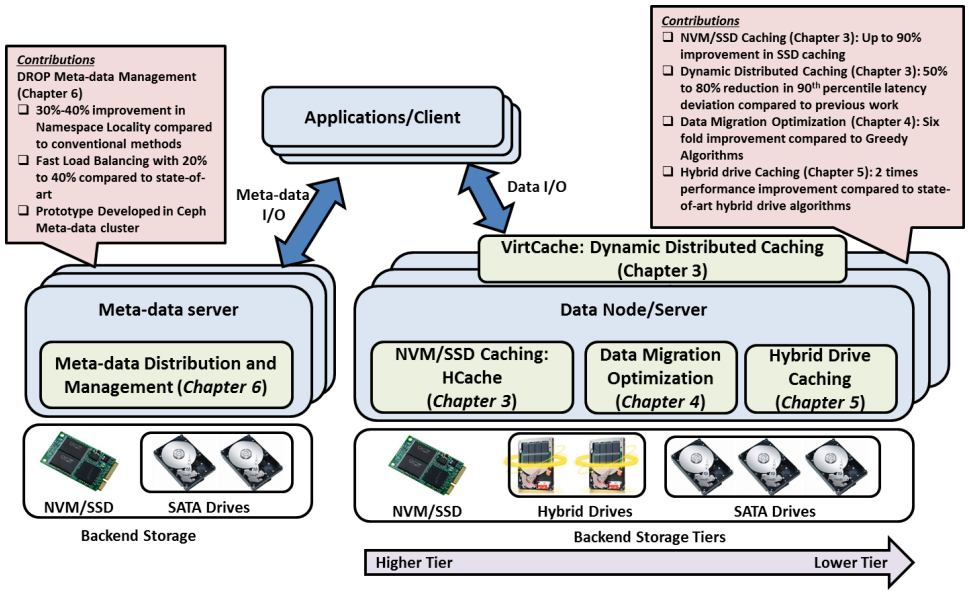

Fig. 7.1 Summary of thesis contributions and respective performance improvement components developed on the storage server or data node cluster and the meta-data server cluster. While figure 1.11 from Chapter 1 shows the high level components, here the respective contributions from the thesis is shown

As part of the future work we intend to deploy applications that exploits the high performance characteristics of the test bed. As a first step we have deployed Genome Sequencing workflows on the test bed as a real-world use case. Currently we are evaluating the performance of the integrated system to find further potential bottlenecks. Such insights would help in tuning the storage components and algorithms to practical I/O workloads thereby enhancing the performance of the whole system.

The work presented in Chapter 5 has culminated in the development of a prototype Hybrid drive storage system on one of the Enterprise storage products of a well known

storage systems manufacturer. As described in Chapter 5, the prototype with Hybrid drives showed significant improvement in performance at reduced power consumption compared to employing high speed SAS drives.

Figure 7.1 shows the contributions from this thesis work with the storage system components developed for performance optimization in the Meta-data and data paths. Though Ceph [122] and GlusterFS [111] were primarily used as the base platform to experiment on the developed ideas from the thesis, any other distributed storage system can be utilized (e.g. Lustre).

7.2 Future Work

While the ideas in the thesis are being explored/developed there were several developments in the areas of next generation NVM, SSD interfaces, and Hybrid drives. We will look into each of these developments, how these impacts the work presented in the thesis and the possible future work.

7.2.1 Developments in Next Generation NVM

We have shown in Chapter 3 that next generation NVM used as a cache can improve the performance and endurance of current NAND Flash. The work in [51] has explored use of Phase Change Memory (PCM) as storage tier as well as for storage caching. Systems like Mnemosyne [107] and NV-Heaps [25] have considered moving NVM like STT-MRAM closer to the CPU on the memory bus so that some of the storage interface speed bottlenecks are eliminated.

The work [9] explores the different system architectures when next generation NVRAM is introduced. It discusses the need for replacing or modifying entire Operating System components like Paging, File systems and incorporating new data reliability mechanisms.

The Moneta system [21] explored the need for replacing the entire storage stack (software) in Operating system kernels to remove the large software overheads involved in handling I/O from the applications. Applications can directly access the persistent storage system (NVM) without the intervention of the Operating system thereby exploiting the advantages of the very low latencies of these devices. While many of these works are ambitious in the goals, we may not see these developments of replacing mainstream DRAM memory entirely with next generation NVM in the near future. This is mainly because the cost of these devices are still very high for large volumes.

The work presented in this thesis is one of the few first steps towards a complete architectural change to incorporate next generation NVM in a storage system. We propose this as a first step towards the goal of completely replacing the current solid state technology (SSD) with next generation NVM. It is a challenging to reduce the performance to cost ratio (IOPS/dollar) through hybrid combinations of next generation NVM and Flash. Though this has been addressed on the storage side in this thesis, future work should consider combination of resource allocation both on the host or client side and storage. Recent attempts have been made to lower cost of PCM ([46],[135],[98]) with multiple level cells (MLC) for storing more than one bit per cell. In [46] the authors propose new writing schemes to improve the cell write performance. It becomes even more challenging when different technologies like PCM and ReRAM are considered in a hybrid setting. Hybrid combinations of these technologies (e.g. like in [101]) with Flash can result in reduction in cost but may require challenging placement and caching schemes to address both performance and endurance of these devices.

7.2.2 Developments in Non volatile memory (NVM) interfaces

There have been developments in improving the interface speeds of system bus connecting SSD to rest of the server or computing system. The conventional storage system intercon-

nects like SATA and SAS are too slow for the very low latencies and throughputs that can be obtained from NAND flash based SSD. This gave rise to the NVM express or NVMe [115] standard which can provide higher throughputs and very low latencies expected from current SSD and next generation NVM.

7.2.3 Developments in Hybrid Drives

Some of the major Hard disk manufacturers have recently introduced the Hybrid drive technology as an alternative to high speed enterprise drives [6]. They come up with larger NV cache (up to 32GB) and also separate persistent write caches to improve the endurance of the bigger NAND flash cache. But these devices come with closed interfaces and do not provide the access to control the internal cache of the drive. They have their own cache placement and eviction algorithms applicable in general to different types of Enterprise workloads. In Chapter 5 we utilize the ATA-8 standard to control the cache inside the drive. This gives more flexibility and control to the higher layers to implement different caching algorithms based on the workloads. The block mapping tables are still maintained by the underlying drives. But the caching meta-data is now managed by the cache manager running on the storage system or server which has more computational resource. This offloads the compute intensive cache decision making part from the drive to the storage server. Though the implementation of our work was based on the ATA-8 standard for controlling the cache, it can in general be used to other emerging hybrid mechanisms like in the SATA 3.2 specification [5]. The reason being that the placement and eviction algorithm is independent of the cache control mechanism being used.

While performance improvement with novel caching techniques for Hybrid drives was the focus of Chapter 5, another important future direction could be intelligent or smart drives. Though intelligent drives have been proposed before [70], it has gained significance recently. The Kinetic drive [118] can be connected directly to a Ethernet switch with appli-

cations directly accessing the storage. The applications can access data stored on the drive through standard interfaces like the Swift object API [116] over TCP/IP. This eliminates several layers of overhead like the OS file system, block layer/device drivers as present in the conventional storage systems. Instead of drives acting as dumb devices storing blocks of data, they have full fledged storage management (placement, allocation and management of data blocks). A hybrid device with a small capacity non-volatile storage (NAND Flash) would further improve performance and extend functionality for these types of drives. The performance improvement can be achieved by placing meta-data or small I/O on internal solid state storage while placing large sequentially accessed data on the magnetic media. In addition to this, the application can specify QoS hints for the different data objects stored on the drive. Based on this QoS hint, the drive can make decisions on whether to place the application data on the solid state storage or the magnetic media. Past work have also been done in the area of code execution or processing on the drive itself [129],[49]. Such code execution within the drive itself becomes more efficient on a hybrid device since the latencies to access data on the internal solid state storage is much lower than the magnetic media. Complex storage management processing like De-duplication, Compression, Parity computation can now be pushed in to the drive itself instead of managed by a central controller thereby improving the scalability for large number of drives. Future work can address issues in optimizing such processing across a cluster of such intelligent drives.

7.2.4 Meta-data Management

In Chapter 6 we looked in to scalability of file system meta-data storage through DROP load balancing and distribution techniques. Though our approach has shown significant improvements in meta-data distribution and performance compared to state-of-art meta-data partitioning, these were optimized for deployments within a Datacenter. We did not consider cross data center distribution or a WAN level distribution and replication. CalvinFS [102]

is a recent work in this direction where meta-data distribution and replication is managed across multiple data centers. The authors show fault tolerance for even complete Datacenter outages and significant improvement in performance.

All the research work in this thesis was focused on performance improvements and scalability of storage systems within a Datacenter. Replication, meta-data distribution, managing request latencies and replica consistency across data centers are more challenging [26] areas that can be considered for Future work.

7.3 Publication Contributions By Thesis Author

The following are the publication contributions by the thesis author.

7.3.1 Journal Publications

1. Shi, H., *Arumugam, R.*, Foh, C., and Khaing, K. (2013). Optimal disk storage allocation for multitier storage system. Magnetics, IEEE Transactions on, 49(6):2603–2609. (Thesis Author Contribution: MDP algorithm idea and development)

2. Xu, Q., *Arumugam, R.*, Yong, K., and Mahadevan, S. (2014). Efficient and scalable metadata management in eb-scale file systems. Parallel and Distributed Systems, IEEE Transactions on, PP(99):1–1. (Thesis Author Contribution: DROP Meta-data distribution core idea using Distributed hash tables with locality, implementation approach in Linux and Development)

7.3.2 Conference Publications

1. *Rajesh, V.*, Xu, Q., Shi, H., Cai, Q., and Wen, Y. (2014). Virtcache: Managing virtual disk performance variation in distributed file systems for the cloud. In Cloud Computing Technology And Science, 2014 IEEE 6th International Conference on.

2. *Vellore Arumugam, R.*, Foh, C. H., Shi, H., and Khaing, K. K. (2012). Hcache: A hybrid cache management scheme with flash and next generation nvram. In APMRC, 2012 Digest, pages 1–4.

3. Haixiang Shi; *Arumugam, R.V.*; Chuan Heng Foh; Kyawt Kyawt Khaing, "Optimal disk storage allocation for multi-tier storage system," APMRC, 2012 Digest , vol., no., pp.1,7, Oct. 31 2012-Nov. 2 2012 (Thesis Author Contribution: MDP algorithm idea and development)

4. Xu, Q., *Arumugam, R.*, Yong, K., and Mahadevan, S. (2013). Drop: Facilitating distributed metadata management in eb-scale storage systems. In Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on, pages 1–10. (Thesis Author Contribution: DROP Meta-data distribution core idea using Distributed hash tables with locality,implementation approach in Linux and Development)

5. Xu, Q., *Rajesh, V.A.*,Yong, K., Wen, Y., and Ong, Y. (2014). C2: Adaptive load balancing for metadata server cluster in cloud-scale file systems. In The 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems. (Thesis Author Contribution: Meta-data Request Load balancing algorithm development)

7.3.3 Patent Publications

1. Yong Hong WANG, *ARUMUGAM Rajesh VELLORE*, Kyawt Kyawt KHAING (2014). Method and apparatus for hot data region optimized dynamic management,

   WO2014209234A1

2. Yong Hong WANG, *ARUMUGAM Rajesh VELLORE*, Chun Teck Lim, Kyawt Kyawt KHAING, Qingsong WEI, Cheng Chen, Jun Yang (2015). Method for hot i/o selective placement and metadata replacement for non-volatile memory cache on hybrid drive or system, WO2015072925A1

References

[1] (2002). Lustre: A scalable, high-performance file system. Cluster File Systems Inc.

[2] (2014). 5,400rpm 2.5-inch sata hard disk drives. http://storage.toshiba.com/techdocs/mq01abdxxx.pdf.

[3] (2014). Fujitsu eternus, disk storage systems. http://www.fujitsu.com/global/products/computing/storage/disk/.

[4] (2014). Product manual, constellation.2 sas. http://www.seagate.com/sg/en/internal-hard-drives/enterprise-hard-drives/hdd/constellation/.

[5] (2014). Sata-io specifications. http://www.sata-io.org/.

[6] (2014). Solid state hybrid technology. http://www.seagate.com/sg/en/solutions/solid-state-hybrid/products/.

[7] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. (2009). Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA. ACM.

[8] Armstrong, R. D., Sinha, P., and Zoltners, A. A. (1982). The multiple-choice nested knapsack model. *Management Science*, 28(1):pp. 34–43.

[9] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. (2011). Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA. USENIX Association.

[10] Bairavasundaram, L. N., Soundararajan, G., Mathur, V., Voruganti, K., and Srinivasan, K. (2012). Responding rapidly to service level violations using virtual appliances. *SIGOPS Oper. Syst. Rev.*, 46(3):32–40.

[11] Bayer, R. (1972). Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306.

[12] Bell, G., Gray, J., and Szalay, A. (2006). Petascale computational systems. *Computer*, 39(1):110–112.

[13] Bennani, M. and Menasce, D. (2005). Resource allocation for autonomic data centers using analytic performance models. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 229–240.

[14] Bhadkamkar, M., Guerra, J., Useche, L., Burnett, S., Liptak, J., Rangaswami, R., and Hristidis, V. (2009). Borg: block-reorganization for self-optimizing storage systems. In *Proccedings of the 7th conference on File and storage technologies*, pages 183–196, Berkeley, CA, USA. USENIX Association.

[15] Birrell, A., Isard, M., Thacker, C., and Wobber, T. (2007). A design for high-performance flash disks. *Operating Systems Review*, 41(2):88–93.

[16] Bisson, T. and Brandt, S. (2007). Reducing hybrid disk write latency with flash-backed i/o requests. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on*, pages 402–409.

[17] Bisson, T., Brandt, S. A., and Long, D. D. E. (2007). A hybrid diskaware spin-down algorithm with i/o subsystem support. In *In Proceedings of the 26th IEEE International Performance, Computing and Communications Conference*.

[18] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[19] Buhrman, R. (2009). Spin torque mram: Challenges and prospects. In *Device Research Conference, 2009. DRC 2009*, pages 33–33.

[20] Byna, S., Chen, Y., Sun, X.-H., Thakur, R., and Gropp, W. (2008). Parallel i/o prefetching using mpi file caching and i/o signatures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12.

[21] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S. (2010). Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA. IEEE Computer Society.

[22] Caulfield, A. M., Grupp, L. M., and Swanson, S. (2009). Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGARCH Comput. Archit. News*, 37(1):217–228.

[23] Chen, F. and Koufaty, D. (2011). Hystor: Making the best use of solid state drives in high performance storage systems. In *In In Proceedings of International Conference on Supercomputing, ICS 2011, ICS '11, Tuscon, Aizona*. ACM.

[24] Chen, Y. and Roth, P. (2010). Collective prefetching for parallel i/o systems. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5.

[25] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 47(4):105–118.

[26] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2013). Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22.

[27] Corbett, P. F. and Feitelson, D. G. (1996). The vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264.

[28] Debnath, B., Du, D., and Lilja, D. (2009). Large block clock (lb-clock): A write caching algorithm for solid state disks.

[29] Deng, Y. (2011). What is the future of disk drives, death or rebirth? *ACM Comput. Surv.*, 43(3):23:1–23:27.

[30] Dillow, D. A. O., Fuller, D. O., Wang, F. O., Oral, H. S. O., Zhang, Z. O., Hill, J. J. O., and Shipman, G. M. O. (2010). *Lessons Learned in Deploying the World s Largest Scale Lustre File System*.

[31] Ding, X., Jiang, S., Chen, F., Davis, K., and Zhang, X. (2007). Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference'07*, pages 261–274.

[32] Dong, X., Wu, X., Sun, G., Xie, Y., Li, H., and Chen, Y. (2008). Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 554–559.

[33] Ellard, D., Ledlie, J., Malkani, P., and Seltzer, M. (2003). Passive nfs tracing of email and research workloads. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 203–216, Berkeley, CA, USA. USENIX Association.

[34] Fengguang, W., Hongsheng, X., and Chenfeng, X. (2008). On the design of a new linux readahead framework. *SIGOPS Oper. Syst. Rev.*, 42(5):75–84.

[35] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In

Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 29–43, New York, NY, USA. ACM.

[36] Goyal, P., Modha, D., Tewari, R., and Jadav, D. (2003). Cachecow: Providing qos for storage system caches. In *In SIGMETRICS*, pages 306–307.

[37] Griffioen, J. and Appleton, R. (1994). Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 13–13, Berkeley, CA, USA. USENIX Association.

[38] Gu, P., Wang, J., Zhu, Y., Jiang, H., Society, I. C., and Shang, P. (2010). A novel weighted-graph-based grouping algorithm for metadata prefetching. *Computers, IEEE Transactions on*, pages 1–15.

[39] Guerra, J., Pucha, H., Glider, J., Belluomini, W., and Rangaswami, R. (2011). Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 20–20, Berkeley, CA, USA. USENIX Association.

[40] Gulati, A., Ahmad, I., and Waldspurger, C. A. (2009). Parda: proportional allocation of resources for distributed storage access. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 85–98, Berkeley, CA, USA. USENIX Association.

[41] Gulati, A., Kumar, C., Ahmad, I., and Kumar, K. (2010). Basil: automated io load

balancing across storage devices. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 13–13, Berkeley, CA, USA. USENIX Association.

[42] Gulati, A., Shanmuganathan, G. S. R., Ahmad, I., Waldspurger, C. A., and Uysal, M. (2011). Pesto: online storage performance management in virtualized datacenters. In *2nd ACM Symposium on Cloud Computing*. http://www.odysci.com/article/1010113016091773.

[43] Gupta, A., Kim, Y., and Urgaonkar, B. (2009). Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 229–240, New York, NY, USA. ACM.

[44] Gurumurthi, S., Sivasubramaniam, A., and Natarajan, V. K. (2005). Disk drive roadmap from the thermal perspective: A case for dynamic thermal management. In *In Proceedings of the International Symposium on Computer Architecture (ISCA*, pages 38–49.

[45] Huang, H., Hung, A., and Shin, K. G. (2005). Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *In Proceedings of 20th ACM Symposium on Operating System Principles*, pages 263–276. ACM Press.

[46] Jiang, L., Zhao, B., Zhang, Y., Yang, J., and Childers, B. (2012). Improving write operations in mlc phase change memory. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–10.

[47] Jiang, S. and Zhang, X. (2002). Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Marina Del Rey*, pages 31–42. ACM Press.

[48] Johnson, T. and Shasha, D. (1994). 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[49] Karakoyunlu, C., Runde, M., and Chandy, J. (2014). Using an object-based active storage framework to improve parallel storage systems. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 70–78.

[50] Kim, H. and Ahn, S. (2008). Bplru: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, Berkeley, CA, USA. USENIX Association.

[51] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. (2014). Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 33–45, Berkeley, CA, USA. USENIX Association.

[52] Kim, J. K., Lee, H. G., Choi, S., and Bahng, K. I. (2008a). A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings*

of the 8th ACM International Conference on Embedded Software, EMSOFT '08, pages 31–40, New York, NY, USA. ACM.

[53] Kim, Y., Gupta, A., and Urgaonkar, B. (2008b). MixedStore: An Enterprise-scale Storage System Combining Solid-state and Hard Disk Drives.

[54] Ko, B.-J., Lee, K.-W., Amiri, K., and Calo, S. (2003). Scalable service differentiation in a shared storage cache. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 184 – 193.

[55] Kraft, S., Casale, G., Krishnamurthy, D., Greer, D., and Kilpatrick, P. (2012). Performance models of storage contention in cloud environments. *Software and Systems Modeling*, pages 1–24.

[56] Lakshman, A. and Malik, P. (2010). Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

[57] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. (2009). Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13.

[58] Lee, S.-W. and Moon, B. (2007). Design of flash-based dbms: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 55–66, New York, NY, USA. ACM.

[59] Leung, A. W., Pasupathy, S., Goodson, G., and Miller, E. L. (2008). Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical*

Conference on Annual Technical Conference, ATC'08, pages 213–226, Berkeley, CA, USA. USENIX Association.

[60] Leventhal, A. (2008). Flash storage memory. *Commun. ACM*, 51(7):47–51.

[61] Li, C., Goiri, I., Bhattacharjee, A., Bianchini, R., and Nguyen, T. (2013). Quantifying and improving i/o predictability in virtualized systems. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–6.

[62] Li, Z., Chen, Z., and Zhou, Y. (2005). Mining block correlations to improve storage performance. *Trans. Storage*, 1(2):213–245.

[63] Lin, L., Li, X., Jiang, H., Zhu, Y., and Tian, L. (2008). Amp: An affinity-based metadata prefetching scheme in large-scale distributed storage systems. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 459–466.

[64] Lin, L., Zhu, Y., Yue, J., Cai, Z., and Segee, B. (2011). Hot random off-loading: A hybrid storage system with dynamic data migration. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 318 –325.

[65] Lu, C., Alvarez, G. A., and Wilkes, J. (2002). Aqueduct: Online data migration with performance guarantees. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA. USENIX Association.

[66] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. (2007).

The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33. Citeseer.

[67] Mattson, R., Gecsei, J., Slutz, D., and Traiger, I. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117.

[68] Meena, J., Sze, S., Chand, U., and Tseng, T.-Y. (2014). Overview of emerging non-volatile memory technologies. *Nanoscale Research Letters*, 9(1).

[69] Megiddo, N. and Modha, D. (2004). Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58 – 65.

[70] Mesnier, M., Ganger, G., and Riedel, E. (2003). Object-based storage. *Communications Magazine, IEEE*, 41(8):84–90.

[71] Mesnier, M. P., Wachs, M., Sambasivan, R. R., Zheng, A. X., and Ganger, G. R. (2007). Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 37–48, New York, NY, USA. ACM.

[72] Mielke, N., Belgal, H., Kalastirsky, I., Kalavade, P., Kurtz, A., Meng, Q., Righos, N., and Wu, J. (2004). Flash eeprom threshold instabilities due to charge trapping during program/erase cycling. *Device and Materials Reliability, IEEE Transactions on*, 4(3):335–344.

[73] Miller, E. L., Greenan, K., Leung, A., Long, D., and Wildani, A. (2008). Reliable

and efficient metadata storage and indexing using nvram. http://dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf/. [Online; posted October 2008].

[74] Modelli, A., Visconti, A., and Bez, R. (2004). Advanced flash memory reliability. In *Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on*, pages 211–218.

[75] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S., and Smith, F. D. (1986). Andrew: A distributed personal computing environment. *Commun. ACM*, 29(3):184–201.

[76] Narayanan, D., Donnelly, A., and Rowstron, A. (2008a). Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23.

[77] Narayanan, D., Donnelly, A., Thereska, E., Elnikety, S., and Rowstron, A. (2008b). Everest: scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 15–28, Berkeley, CA, USA. USENIX Association.

[78] Park, D., Debnath, B., Nam, Y., Du, D. H. C., Kim, Y., and Kim, Y. (2012). Hotdata-trap: A sampling-based hot data identification scheme for flash memory. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1610–1617, New York, NY, USA. ACM.

[79] Park, Y., Lim, S.-H., Lee, C., and Park, K. H. (2008). Pffs: A scalable flash memory file system for the hybrid architecture of phase-change ram and nand flash. In *Proceedings*

of the 2008 ACM Symposium on Applied Computing, SAC '08, pages 1498–1503, New York, NY, USA. ACM.

[80] Patil, S. V., Gibson, G. A., Lang, S., and Polte, M. (2007). Giga+: Scalable directories for shared file systems. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA. ACM.

[81] Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA. ACM.

[82] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995). Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29(5):79–95.

[83] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D. (1994). Nfs version 3 - design and implementation. In *In Proceedings of the Summer USENIX Conference*, pages 137–152.

[84] Payer, H., Sanvido, M. A., Bandic, Z. Z., and Kirsch, C. M. (2009). Combo Drive: Optimizing cost and performance in a heterogeneous storage device. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH), co-located with ASPLOS*. ASPLOS.

[85] Pisinger, D. (1994). A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83:394–410.

[86] Pisinger, D. (1999). Core problems in knapsack algorithms. *Operations Research*, 47:570–575.

[87] Prodan, R. and Wieczorek, M. (2010). Bi-criteria scheduling of scientific grid workflows. *Automation Science and Engineering, IEEE Transactions on*, 7(2):364 –376.

[88] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. (2009). Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA. ACM.

[89] Raicu, I., Foster, I. T., and Beckman, P. (2011). Making a case for distributed file systems at exascale. In *Proceedings of the Third International Workshop on Large-scale System and Application Performance*, LSAP '11, pages 11–18, New York, NY, USA. ACM.

[90] Rajesh, V., Xu, Q., Shi, H., Cai, Q., and Wen, Y. (2014). Virtcache: Managing virtual disk performance variation in distributed file systems for the cloud. In *Cloud Computing Technology And Science, 2014 IEEE 6th International Conference on*.

[91] Raman, A., Yorsh, G., Vechev, M., and Yahav, E. (2011). Sprint: Speculative prefetching of remote data. In *Proceedings of the 2011 ACM International Conference on Object*

Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 259–274, New York, NY, USA. ACM.

[92] Rodeh, O. and Teperman, A. (2003). zfs - a scalable distributed file system using object disks. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 207–218.

[93] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., David, and Steere, C. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459.

[94] Saxena, M., Swift, M. M., and Zhang, Y. (2012). Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA. ACM.

[95] Sehgal, P., Voruganti, K., and Sundaram, R. (2012). Slo-aware hybrid store. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –6.

[96] Shi, H., Arumugam, R., Foh, C., and Khaing, K. (2013). Optimal disk storage allocation for multitier storage system. *Magnetics, IEEE Transactions on*, 49(6):2603–2609.

[97] Soundararajan, G., Prabhakaran, V., Balakrishnan, M., and Wobber, T. (2010). Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 8–8, Berkeley, CA, USA. USENIX Association.

[98] Stanisavljevic, M., Athmanathan, A., Papandreou, N., Pozidis, H., and Eleftheriou,

E. (2015). Phase-change memory: Feasibility of reliable multilevel-cell storage and retention at elevated temperatures. In *Reliability Physics Symposium (IRPS), 2015 IEEE International*, pages 5B.6.1–5B.6.6.

[99] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA. ACM.

[100] Sun, G., Dong, X., Xie, Y., Li, J., and Chen, Y. (2009). A novel architecture of the 3d stacked MRAM L2 cache for cmps. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 239–249.

[101] Sun, G., Joo, Y., Chen, Y., Niu, D., Xie, Y., Chen, Y., and Li, H. (2010). A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12.

[102] Thomson, A. and Abadi, D. J. (2015). Calvinfs: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, Santa Clara, CA. USENIX Association.

[103] VanDeBogart, S., Frost, C., and Kohler, E. (2009). Reducing seek overhead with

application-directed prefetching. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 24–24, Berkeley, CA, USA. USENIX Association.

[104] Vellore Arumugam, R., Foh, C. H., Shi, H., and Khaing, K. K. (2012). Hcache: A hybrid cache management scheme with flash and next generation nvram. In *APMRC, 2012 Digest*.

[105] Vengerov, D. (2008). A reinforcement learning framework for online data migration in hierarchical storage systems. *J. Supercomput.*, 43:1–19.

[106] Verma, A., Sharma, U., Jain, R., and Dasgupta, K. (2008). Compass: optimizing the migration cost vs. application performance tradeoff. *IEEE Transactions on Network and Service Management*, pages 118–131.

[107] Volos, H., Tack, A. J., and Swift, M. M. (2011). Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104.

[108] Wachs, M., Abd-El-Malek, M., Thereska, E., and Ganger, G. R. (2007). Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA. USENIX Association.

[109] Wang, K. L., Alzate, J. G., and Amiri, P. K. (2013). Low-power non-volatile spintronic memory: Stt-ram and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003.

[110] Weblink (2014a). http://technet.microsoft.com/en-us/magazine/2007.03.vistakernel.aspx.

[111] Weblink (2014b). Cloud storage for the modern data center, an introduction to gluster architecture. http://www.gluster.org/.

[112] Weblink (2014c). Flashcache. https://github.com/facebook/flashcache/.

[113] Weblink (2014d). Hadoop distributed file system. http://hadoop.apache.org/docs/r0.20.2/hdfs_design.html.

[114] Weblink (2014e). Hammerora database benchmarking tool. http://hammerora.sourceforge.net/.

[115] Weblink (2014f). Nvm express standard.

[116] Weblink (2014g). Openstack swift interface/api.

[117] Weblink (2014h). Openstack, the open source cloud operating system. http://www.openstack.org/.

[118] Weblink (2014i). Seagate kinetic hdd.

[119] Weblink (2014j). Snia block traces repository. http://iotta.snia.org.

[120] Weblink (2014k). T13 standards. http://www.t13.org/documents/.

[121] Weblink (2014l). Umass block traces repository. http://traces.cs.umass.edu/.

[122] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA. USENIX Association.

[123] Weil, S. A., Pollack, K. T., Brandt, S. A., and Miller, E. L. (2004). Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA. IEEE Computer Society.

[124] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. (2008). Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA. USENIX Association.

[125] Wendell, P. and Freedman, M. J. (2011). Going viral: Flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 549–558, New York, NY, USA. ACM.

[126] Wood, R. (2009). Future hard disk drive systems. *Journal of Magnetism and Magnetic Materials*, 321(6):555 – 561. Current Perspectives: Perpendicular Recording.

[127] Wu, G. and He, X. (2012). Reducing ssd read latency via nand flash program and erase suspension. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

[128] Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., and Xie, Y. (2009). Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 34–45, New York, NY, USA. ACM.

[129] Xie, Y., Muniswamy-Reddy, K.-K., Feng, D., Long, D., Kang, Y., Niu, Z., and Tan, Z. (2011). Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12.

[130] Xu, Q., Arumugam, R., Yong, K., and Mahadevan, S. (2013). Drop: Facilitating distributed metadata management in eb-scale storage systems. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–10.

[131] Xu, Q., Arumugam, R., Yong, K., and Mahadevan, S. (2014a). Efficient and scalable metadata management in eb-scale file systems. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1.

[132] Xu, Q., Rajesh, V., Yong, K., Wen, Y., and Ong, Y. (2014b). C2: Adaptive load balancing for metadata server cluster in cloud-scale file systems. In *The 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems*.

[133] Yalaoui, A., Chatelet, E., and Chu, C. (2005). A new dynamic programming method for reliability redundancy allocation in a parallel-series system. *Reliability, IEEE Transactions on*, 54(2):254 – 261.

[134] Yang, J., Plasson, N., Gillis, G., and Talagala, N. (2013). Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 10:1–10:11, New York, NY, USA. ACM.

[135] Yoon, D. H., Chang, J., Schreiber, R., and Jouppi, N. (2013). Practical nonvolatile multilevel-cell phase change memory. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12.

[136] Zhang, G., Chiu, L., and Liu, L. (2010). Adaptive data migration in multi-tiered storage based cloud environment. In *In Proceedings of the 3rd International Conference on Cloud Computing*, pages 148–155.

[137] Zhang, W. and Li, T. (2009). Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 101–112.

[138] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. (2009). A durable and energy efficient main memory using phase change memory technology. *SIGARCH Comput. Archit. News*, 37(3):14–23.

[139] Zhu, Y., Jiang, H., Wang, J., Xian, F., and Member, S. (2008). HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 750–763.