High Performance Storage System Design Using Emerging Storage Technologies

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Jinfeng Yang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. David J. Lilja

December, 2021

Acknowledgements

I would like to express my heartfelt gratitude to my advisor, Prof. David J. Lilja, for his guidance, support, training, and patience throughout my entire Ph.D study at the University of Minnesota. His research methodologies and critical thinking educated me in both research and life. As my advisor, he taught me how to think research issues from high level and also resolve the detailed problems. Prof. David J. Lilja played an important role in my educational development. I'm always proud, lucky, and grateful to be his student.

I also would like to appreciate my defense committee, Prof. David Du, Prof. Kia Bazargan and Prof. Ulya Karpuzcu at the University of Minnesota, for their comments, advice, and feedback to my Ph.D thesis.

As a member of the Center for Research in Intelligent Storage (CRIS), I would like to thank my industry mentor Peng Li for his supports and suggestions in each of my research project. I also would like to thank my group members and friends of CRIS, Bingzhe Li, Yaobin Qin, Zhichao Cao, Hao Wen, Fenggang Wu, Baoquan Zhang, Hebatalla Eldakiky, Chia-Wen Hsieh, Yixun Wei, Huibing Dong, Wenlong Wang, for their helps and suggestions in the discussion.

This work was supported in part by the Center for Research in Intelligent Storage, which is supported by National Science Foundation grant no. IIP-1439622 and member companies. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Dedication

To my parents and my wife.

Abstract

In the past few decades, data volume increases exponentially. Smart devices, social media, and e-business generate an extremely amount of data everyday. While big data is promising and has been successfully applied in many areas such as machine learning, financial market, and healthcare, it still faces many problems. One main challenge for large scale data storage system is how to store and retrieve data efficiently. In this thesis, we approach the above challenge by developing high performance storage systems using emerging storage technologies (i.e., non-volatile memory and non-volatile memory block device).

First, we start by characterizing an emerging storage device - non-volatile memory (NVM) block device. With replacing the NAND flash chip inside the Solid State Drive (SSD) into NVM media, NVM block device delivers substantial performance improvements compared to NAND-based storage systems. However, its performance characteristics have not been well studied. In this study, we carry out intensive experiments and propose multiple custom-design micro-benchmarks to extract the intrinsic performance behaviors of the NVM block device, including the basic I/O performance behavior, advanced interleaving technology, performance consistency under highly intensive I/O workload, the influence of unaligned request size, the elimination of write-driven garbage collection, read disturb issues, and the tail latency problem. The performance is compared to that of a conventional NAND SSD to indicate the performance difference of the NVM block device in each scenario. In addition, by using an online analytical benchmark, a database system's performance is studied on our target storage devices to quantify the potential benefits of the NVM block device to a real application. Finally, the performance impact of hybrid NVM block devices and NAND SSD storage systems on a database application is investigated.

Second, based on the understanding to the performance characteristics of NVM block device and the bottlenecks of database system we studied on the above, we present a hybrid storage database system, called HeuristicDB, that uses a non-volatile memory (NVM) block device as an extension of the database buffer pool. To consider the unique performance behavior of NVM block devices and the block-level characteristics of database requests, a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority are proposed. To support the system implementation of the proposed rules, four programs, including *a query profile queue*, *an eviction and demotion (EV) program*, *a table access pattern detector*, and *a page placement controller* are developed. Using online analytical processing (OLAP) and online transactional processing (OLTP) benchmarks, both trace-based examination and system implementation on MySQL are carried out to evaluate the effectiveness of the proposed design. The experimental results show HeuristicDB delivers up to 75% higher performance than existing systems.

Third, in the setting of NVM is going to replace DRAM to server as a main memory in the future, we introduce a machine learning based cache replacement algorithm, named ExpCache, to improve the NVM-based system performance. By considering the non-volatility characteristic of the NVM devices, we split the whole NVM into two caches, including a read cache and a write cache, for retaining different types of requests. The pages in each cache are managed by both LRU and LFU policies for balancing the recency and frequency of workloads. The online *Expert* machine learning algorithm is responsible for selecting a proper policy to evict a page from one of the caches based on the access patterns of workloads. Moreover, a read-write discriminated eviction program is developed to eliminate the number of dirty pages written back to the storage. In experimental results, the proposed ExpCache outperforms previous studies in terms of hit ratio and the number of dirty pages written back to storage.

In summary, a NVM block device performance characterization, a hybrid storage database system, and an online learning based cache replacement algorithm for NVM are proposed in this thesis and allow large scale data storage system to deliver high performance.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Emerging Memory and Storage Devices 3

1.1.1 Non-Volatile Memory 3

1.1.2 Non-Volatile Memory Block Device 4

1.2 Hybrid Storage Database System Using NVM Block Device 5

1.3 Online-Learning Based Cache Replacement Policy for NVM 6

1.4 Summary of Contributions 7

1.5 Thesis Outline 9

2 Exploring Performance Characteristics of the NVM Block Device **10**

2.1 Introduction 10

2.2 Background 12

2.2.1 NAND Flash Memory 12

2.2.2 3D XPoint Memory 13

2.2.3 SSD Internal Architecture 14

2.2.4 NAND SSD Write Issues 14
2.2.5 Read Disturb Error 16
2.3 Related Work 16
2.4 Experimental Setup 18
2.5 Performance Studies 19
2.5.1 Basic I/O Performance Behaviors 19
2.5.2 Advanced Interleaving Technique 22
2.5.3 Performance Consistency Under Highly Intensive I/O Workload Pressure 25
2.5.4 Performance Influence of Unaligned Request Size 28
2.5.5 Performance Degradation of Write-Driven Garbage Collection . . 30
2.5.6 Read Disturb Issue 32
2.5.7 Tail Latency 36
2.5.8 Real Trace Implementation 39
2.6 Implications for a Real Database System 40
2.6.1 TPC-H Benchmark and RDBMS 41
2.6.2 TPC-H Performance Evaluation 42
2.7 Investigations of the Hybrid Storage System for a Database Application 46
2.7.1 Hybrid System with Table Placement Algorithm 48
2.7.2 Hybrid System With Caching Algorithm 52
2.8 Conclusions 55

3 HeuristicDB: A Hybrid Storage Database System Using a Non-Volatile Memory Block Device 57

3.1 Introduction 57
3.2 Backgrounds and Related Work 60
3.2.1 NVM block device 60
3.2.2 Related Work & Limitation 60
3.3 Design Principles & Heuristic Rules 62
3.3.1 Characteristics of Database Block Requests 62
3.3.2 System Design Principles 63
3.3.3 Heuristic Rules for a Query 64

3.3.4 Heuristic Rules for Concurrent Queries 67
3.4 HeuristicDB Overview 68
3.4.1 Metadata Management 68
3.4.2 System Overview 70
3.5 System Implementations 71
3.5.1 Query Profile Queue 71
3.5.2 Eviction and Demotion Program 72
3.5.3 Access Pattern Detector 74
3.5.4 Page Placement Controller 75
3.6 Trace Based Evaluation 76
3.6.1 Complex OLAP Workload 78
3.6.2 TPC-H Workload 79
3.6.3 TPC-E Workload 80
3.6.4 Performance Gain Analysis 80
3.7 Performance Evaluation on a Real System 82
3.7.1 Implementation Details 82
3.7.2 Experimental Results 84
3.7.3 Internals on the OLAP Workload 84
3.8 Conclusion 86

4 ExpCache: Online-Learning based Cache Replacement Policy for Non-Volatile Memory 88

4.1 Introduction 88
4.2 Background and Related Work 91
4.2.1 Non-volatile Memory 91
4.2.2 Online Expert Selection 91
4.2.3 Related Work & Limitations 93
4.3 ExpCache Algorithm 94
4.3.1 ExpCache Design Principle and Internal Architecture 94
4.3.2 ExpCache Working Processes 97
4.3.3 Online Expert Weights Update - Time Dependent Expert 99
4.3.4 Read-write Discriminated Eviction Policy 100

4.4 System Evaluation 102

4.4.1 Results Analysis: Hit Ratio 103

4.4.2 Results Analysis: Write Count 105

4.4.3 Results Analysis: Phase Change of ExpCache 107

4.4.4 Results Analysis: Impacts of the ToleranceFactor 108

4.4.5 Space and Computation Overheads Analysis 110

4.5 Conclusion 110

5 Conclusion and Future Work 112

5.1 Conclusion 112

5.2 Future Work 113

5.2.1 NVM Oriented Database System 114

5.2.2 Database on Warm Storage With Emerging Storage Devices . . . 115

5.2.3 Hierarchical Storage System Using NVM 115

5.3 Summary 116

References 117

List of Tables

1.1 Characteristics of NVM, NVM block device, and other types of memory and storage devices. 3

2.1 Tail latency comparison 37

2.2 MSR trace configuration 39

2.3 Database tables' information of TPC-H benchmark 41

2.4 TPC-H access pattern. Note, there is no random write for all queries . . 44

2.5 TPC-H repeat access rate 44

2.6 Characteristics of Q3 51

4.1 Characteristics of experimental workloads 103

List of Figures

2.1 Solid State Drive Internals. 13

2.2 NAND SSD Write Issues. 15

2.3 Basic I/O performance behaviors. 20

2.4 Interleaving technique. 23

2.5 Performance consistency under highly intensive I/O workload. 26

2.6 Performance influence of different request sizes. 29

2.7 Write-driven garbage collection. 31

2.8 Internal buffer size. 33

2.9 Read disturb issue. 35

2.10 NVM block device, only access the target page repeatedly. 36

2.11 MSR trace implementation. 40

2.12 TPC-H queries Result. X-axis represents query name. 43

2.13 Hybrid system with table placement algorithm. Step 1: While processing a database query, the block cache is first consulted. If the target tables have been cached, read them from the block cache. Step 2: Otherwise, block the query, migrate target tables into the block cache, and resume the query. Step 3: If there is no free space on the block cache to place target tables, read target tables from the storage device. 46

2.14 Hybrid system with caching algorithm. Step 1: While processing a database query, if the required page has been cached in the host, read it from the database buffer pool. Step 2: Otherwise, read the required page from the block cache. Step 3: If the required page is still not found, fetch it from the storage device. Step 4: Meanwhile, the required page has also been a cache candidate to be buffered into the block cache. . . 47

2.15 Query clause of Q3, Q10, and Q16. 49
2.16 The execution duration for the hybrid table placement algorithm. Note, the abbreviation of each table's name and its corresponding size are listed in Table 2.3. 50
2.17 The execution duration for the hybrid caching algorithm. 53
3.1 HeuristicDB overview. The left part shows the internal architecture of HeuristicDB, and the right part explains its working processes. 69
3.2 Query profile queue. 72
3.3 Internal organizations of cached pages and EV operations demonstration. 73
3.4 Experimental results of trace based evaluation - Complex OLAP Workload. 77
3.5 Experimental results of trace based evaluation - TPC-H Workload. . . . 79
3.6 Experimental results of trace based evaluation - TPC-E Workload. . . . 81
3.7 The speedup of HeuristicDB and other baselines to database workloads over the case without NVM buffer pool. The execution time(in sec) and transaction rate(TpsE) for complex OLAP workload, TPC-H workload, and TPC-E workload of the case without NVM buffer pool are 34993.22sec, 20507.77sec, and 59.3 TpsE, respectively. 83
3.8 The speedup of different replacement algorithm to each query inside the OLAP workload. The NVM buffer pool is set to 30GB. We record the execution time(in seconds) of each query of the case without NVM buffer pool buffer below each figure. 85
4.1 NVM-based System Architecture. 89
4.2 ExpCache Architecture. 94
4.3 Hit Ratio comparison between ExpCache and other baselines. The values inside each cell are computed using formula: *ExpCache hit ratio - baseline hit ratio.* The red color indicates ExpCache outperforms the baseline. The blue color indicates ExpCache has worse performance than the baseline. The white color indicates there is no significant performance difference. 104

4.4 Write Count comparison between ExpCache and other baselines. The value inside each cell records the write count difference by formula: *baseline write count / ExpCache write count*. The red color indicates ExpCache has less write count other the baseline. The blue color indicates ExpCache has higher write count than the baseline. The white color indicates there is no significant performance difference. 106

4.5 Phase change of ExpCache and different caching policies over time. The upper one shows the variations of two weights in ExpCache. The lower one shows the variations of hit ratios for different caching policies. . . . 108

4.6 ExpCache performance variation as varying the value of *ToleranceFactor*. (10% as the reference to compute the difference. 109

Chapter 1

Introduction

Coming into the 21st century, the data volume increases in an unprecedented speed. With rapidly development of social media, e-commerce, and Internet of Things (IoT), a hug amount of data is generated every day. While big data holds promise and has been widely applied into many areas such market analysis, healthcare, and information technology, it is not without its challenges. A research shown that the data volume is doubling every two years [1]. Then storing and retrieving those large volume data efficiently become to more challenging.

Industry and academia have developed a variety of modern storage systems, such as relational database system [2, 3, 4, 5, 6, 7], key-value store [8, 9, 10, 11, 12], and file system [13, 14, 15, 16, 17] to fulfill rapid data growth. A recent trend in storage system is to move towards cloud [18, 19, 20, 21, 22] and warm storage (i.e., computation and storage decoupling) [23, 24] for high usability and accessibility.

In addition, a variety of types of emerging storage devices have been launched to meet the requirements in different usage scenarios. By blending the high performance characteristics of Dynamic Random Access Memory (DRAM) and the persistence property of conventional storage devices, Non-Volatile Memory (NVM) such as 3D Xpoint [25, 26, 27], Phase-Change Memory (PCM) [28, 29, 30, 31], and Conductive Bridging RAM (CBRAM) [32, 33, 34] has been becoming critical importance for modern systems. It is either used alone as a persistent memory or worked corporately with DRAM to offer large capacity and data persistency. One of the trends of Hard Disk Drive (HDD) is to evolve towards high capacity using new recording technologies such as Shingled Magnetic Recording (SMR) [35, 36] and Assisted Magnetic Recording (HAMR) [37, 38] to balance performance and cost. NAND-based Solid State Drive (SSD) [39, 40, 41, 42, 43] becomes prevalent in the past decades for the low random access latency. Besides, Non-volatile memory (NVM) block device (e.g. Optane SSD) [44, 45, 46] is released recently. By employing advanced interleaving technology and replacing NAND flash memory within the solid state drive to non-volatile memory chip, NVM block device delivers orders of magnitude higher bandwidth than NAND-based SSD.

The main challenge of employing those emerging storage devices is how to integrate and manage them in existing systems efficiently for maximizing performance gains. For instance, as NVM block device delivers unprecedented performance compared with conventional storage devices, building a hybrid storage database system that uses a NVM block device as a block cache for low-end storage devices becomes a cost-efficient way to improve database performance. However, as modern database systems become increasingly complex, designing an efficient data placement schema is more challenging. For another example, in the setting of NVM is going to replace DRAM to serve as a main memory in the future, it is the time to re-consider the caching replacement policy design. However, as NVM exhibits different characteristics with traditional DRAM, it is challenging to develop a proper caching policy for NVM to fit its unique proprieties.

The first part of the thesis focuses on the performance evaluation of the emerging storage device (i.e., NVM block device). We start by characterizing the NVM block device using multiple custom-design micro benchmarks and methodologies. The implications of NVM block device to database system are also investigated. Inspired by the deep understanding to the performance characteristics of NVM block device and performance bottlenecks of database systems, in the second part, we propose a hybrid storage database system, called HeuristicDB, which uses an NVM block device as an extension of the database buffer pool. HeuristicDB achieves significantly performance accelerations to database system meanwhile with limited number of page replacement. Finally, as NVM is going to replace DRAM to serve as a main memory in the future, in the third part, we develop an online-learning based caching replacement policy, named as ExpCache, for NVM based computer system. The proposed ExpCache outperforms previous studies in terms of hit ratio and the number of dirty pages written back to

	DRAM	NVM	NVM block device	NAND SSD	HDD
Read latency	20-60ns	100-300ns	10us	20us	20ms
Write latency	20-50ns	100ns	10us	100us	20ms
Addressability	byte	byte	block	block	blocks
Persistent	No	Yes	Yes	Yes	Yes
Interface	Memory bus	Memory bus	PCIe	PCIe	SATA
Endurance	10^{16}	$10^7 - 10^9$	$10^7 - 10^9$	10^6	10^{16}
Capacity	GB	GB-TB	TB	TB	TB

Table 1.1: Characteristics of NVM, NVM block device, and other types of memory and storage devices.

storage.

1.1 Emerging Memory and Storage Devices

1.1.1 Non-Volatile Memory

With DRAM-like performance and disk-like persistency and capacity, non-volatile memory is becoming critical importance for various data-intensive applications. The advent of NVM bridges the huge gap between high performance DRAM and laggard storage devices. Table 1.1 lists the performance characteristics of NVM and other types of memory and storage devices. Compared with DRAM, NVM not only delivers similar read-write bandwidth but also has a much higher density. Most significantly, all data resided in NVM is persistent. Thus, there is no need to write updated dirty pages back to storage to prevent data loss when power is off. Compared with conventional storage devices connected to an external block-addressable PCIe [47, 48, 49] or SATA [50] interface, NVM communicates with processors more efficiently using a byte-addressable memory bus. In addition, by employing high-performance memory cells, NVM presents much higher read-write bandwidth than conventional storage devices. Because of those performance advantages, NVM is expected to replace DRAM to serve as the main memory in the near future.

1.1.2 Non-Volatile Memory Block Device

NAND-based SSD has become prevalent in the past few decades thanks to its fast random access latency. While it provides 10X higher bandwidth than conventional HDD, the hug performance gap between high-performance DRAM and slow storage devices has not changed. Besides, NAND-based SSD performance degrades dramatically under some specific scenarios.

By replacing the NAND-flash memory within the solid state drive to NVM chip and employing advanced interleaving technology, the emerging storage device, NVM block device, is released recently. The NVM block device (i.e., Optane SSD) has several advantages. First, compared with conventional NAND SSDs, the NVM block device delivers orders of magnitude better performance on both read and write operations [26]. Second, because NVM supports in-place update, there is no need for garbage collection [51, 52, 26] which usually causes significant performance degradation for the NAND SSD. Third, the NVM block device inherits the PCIe interface developed for the NAND SSD and can be directly controlled by all existing systems without requiring any hardware updates. In contrast, the NVM works only when a specific processor is provided [53, 54, 55].

Due to these performance advantages, the NVM block device is expected to be a good candidate to accelerate the slower storage subsystem in the future. However, few research studies have been presented to discuss the unique characteristics of the NVM block device. Previous publications [44][45] and product specifications [46] either focused mainly on the potential performance impacts of NVM block device to real applications or gave limited information about its intrinsic characteristics. This motivates us to conduct a comprehensive performance analysis of NVM block device and investigate its implications to data-intensive applications such as Relational Database Management System (RDBMS).

1.2 Hybrid Storage Database System Using NVM Block Device

A relational database management system (RDBMS) was first introduced in 1970 by E. F. Codd [56]. It is constructed based on the relational model [57, 58, 59, 60, 61] which allows users to identify and access data in relation to other pieces of data inside the database. Most of RDBMSs use Structured Query Language (SQL) to store, retrieve, query, and update the data within the database. Today, many types of relational database systems have been developed and applied in different areas, such as financial market, e-business, and social media. Based on the internal data origination, the relational database management systems can be separated into two categories, including the row-oriented database systems (e.g. MySQL [3] and PostgreSQL [6]) , and the column-oriented database systems(e.g. Redshift [62], BigQuery [63], and Snowflake [64]).

Compared with a NoSQL database [8, 9, 10, 11], an RDBMS has multiple advantages, including being ACID (atomicity, consistency, isolation, durability) compliant [65] for transactions, using a standardized SQL language, and supporting complex query functions. Because of these advantages, RDBMS technology is prevalent and will co-exist with NoSQL databases for a long time. However, RDBMS performance is usually limited by laggard storage devices [66, 67, 68]. Especially for complex Online Analytical Processing (OLPA) queries, the I/O time can be 80-90% of the total execution execution time, which degrades database performance significantly. Current research [26, 69] shows that if an NVM block device is used as persistent storage for RDBMS, I/O time is no longer a bottleneck in many cases. However, this is an expensive solution. Because of the high cost of NVM block devices, it might be impossible to provide an adequate number to preserve the entire database in real time. In this circumstance, building a hybrid storage database system that uses an NVM block device as a block cache for conventional storage devices becomes a cost-efficient way to improve RDBMS performance.

However, there are still many challenges before we build the hybrid storage database system using NVM block device. First, the NVM block device has various performance-related properties [26, 25, 53], such as much higher read-write bandwidth, the elimination of write-driven garbage collection [51, 52] compared to conventional storage devices.

Miss using those properties could lead the developed system unable to yield the best performance for NVM block device. Second, the database block requests exhibit multiple unique characteristics. Then, how to leverage those characteristics properly to build an efficient hybrid storage schema becomes to a problem. Finally, as modern database systems become increasingly complex, handing the dynamically changed database workloads is more challenging.

1.3 Online-Learning Based Cache Replacement Policy for NVM

Today, Non-Volatile Memory [28, 30, 31, 32, 34] is becoming critical importance for data-intensive applications and is expected to replace DRAM to serve as a main memory in the future. As the first type of commercial NVM, named as Optane DIMM [45, 55, 70], was released recently, it is the time to consider the NVM-based computer system [71, 72, 73, 74] design.

Compared with DRAM-based computer system, NVM-based one has multiple advantages. First, NVM provides competitive performance as DRAM meanwhile has higher density. As a result, NVM-based system offers much higher memory capacity than DRAM-based one. Second, because all data reside in NVM is persistent, there is no need to write updated pages (i.e., dirty pages) back to storage for preventing data loss while power failure, which significantly increases the system performance and simplifies the system design.

Caching is one of main techniques to optimize NVM-based system performance thanks to the access localities of workloads. However, as modern applications become complex, designing a proper cache replacement policy for NVM-based system faces many challenges. First, previous cache replacement algorithms were originally designed for DRAM-based systems without considering the non-volatility of NVM. So, they are unlikely to deliver excellent performance improvement for NVM-based system. Second, past studies in this area make a replacement decision based on certain pre-defined rules instead of the real access patterns of a workload. Since these pre-defined rules are not adapted to the dynamically changed workload access patterns, prior cache replacement schema [75, 76, 77, 78] suffer severe performance degradation as the access pattern of a workload changed.

Machine learning (ML) algorithms have been well studied in the past and have been introduced in various areas for system modeling, access pattern detection, and performance improvement, and making predictions [79, 80, 81, 82, 83, 84]. Motivated by this, it has a large potential to employ machine learning techniques in cache algorithm designs for adapting to dynamically changed workloads to improve the system performance. However, integrating machine learning algorithm into the caching algorithm design properly is more challenging.

1.4 Summary of Contributions

This thesis makes the following contributions to meet the challenges and research issues of the emerging storage device (i.e., NVM block device), the hybrid storage database system design, and the caching challenges of NVM-based computer system.

NVM Block Device Characterization. We carry out an in-depth performance evaluation of NVM block device (i.e., Optane SSD). Several basic performance behaviors of the NVM block device, including I/O access latency, internal parallelism, and performance consistency under highly intensive I/O workload, are evaluated at the first. In addition, we propose multiple custom-designed micro-benchmarks and methodologies to explore the further details of the intrinsic characteristics of the NVM block device, including byte addressability, the elimination of write-driven garbage collection, the read disturb issue, and the tail latency problem.

Besides, we also investigate the implications of the NVM block device to a relational database management system using an Online Analytical Processing benchmark. By analysis of the collected results, it is noticed that the NVM block device delivers asymmetric performance accelerations to different types of queries. The potential reasons for these results are explored.

Finally, we implement a real database application on a hybrid NVM block device and a NAND solid state drive storage system and evaluate its performance. Several key factors that can impact this hybrid system's performance significantly are studied.

Hybrid Storage Database System Using a NVM Block Device. We investigate the prior works of hybrid storage database system [85, 86, 87, 88, 89, 90, 91] and find past studies only provide limited performance improvement due to miss understanding the characteristics of database block requests. We also point out that without an efficient demotion mechanism for cached pages, previous works suffered severe performance degradation when the database workload access pattern is changed. Besides, we study past researches of hybrid table placement algorithm [92, 93] for a relational database system and find prior works encounter misplacement issues while the query execution plan tree miss-predicts the access pattern of database tables during a query.

To resolve the limitations of prior studies, we develop a new hybrid storage database system named as HeuristicDB. We comprehensively study the performance behaviors of the NVM block device and the characteristics of database block requests at the first. We then propose a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority to guide HeuristicDB in managing database block requests more efficiently. Finally, to support the system implementation of the proposed rules, we develop four assistant programs, including *a query profile queue*, *an eviction and demotion (EV) program*, *a table access pattern detector*, and *a page placement controller*.

Online-Learning Based Cache Replacement Policy for NVM. We study existing works of DRAM-based cache replacement algorithms [94, 75, 95, 77, 96] and find they are unable to deliver excellent performance improvement for NVM-based systems due to ignore the influence of write operations. Besides, we point out prior works deliver limit performance improvement while the access pattern of a workload mismatch with their pre-defined placement rules. Finally, we show integrating the online-learning algorithm with cache replacement schema improperly delivers even worse performance compared with conventional cache polices in some scenarios.

We propose a online learning-based cache replacement algorithm, called ExpCache, for NVM-based systems. The goals of ExpCache are improving both read and write hit ratios of NVM meanwhile eliminate the number of dirty pages written back to the storage. In the system, the real cache (i.e., NVM main memory) is divided into a read cache and a write cache for retaining different types of requests. Two experts, including an LRU policy and an LFU policy, are employed to manage the data elements within each of the real caches to balance both recency and frequency factors. In addition, two ghost caches are built to store the evicted pages from the real caches and emulate the LRU and LFU policies independently. Finally, a read-write discriminated eviction policy is developed to reduce the number of dirty pages written back to the storage.

1.5 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 presents the findings of the NVM block device characterizations and studies the implications of NVM block device to relational database management system. Chapter 3 introduces a hybrid storage database system, named as HeuristicDB, which uses NVM block device as an extension of database buffer pool. Chapter 4 presents an online-learning based cache replacement algorithm for NVM-based computer system. Chapter 5 concludes the thesis and discusses future works.

Part of the works in the thesis appear in the proceeding of conferences and journal publications [69, 26, 97].

Chapter 2

Exploring Performance Characteristics of the NVM Block Device

2.1 Introduction

Solid-state storage drives (SSD) based on NAND flash technology can deliver several orders of magnitude better performance than conventional hard-disk drives (HDD). Most notably, the NAND SSD can directly access the physical location of the data without mechanical arm movement. This emerging innovation not only saves the seek time that usually existed within conventional rotating disks, but reduces power consumption. Because of these performance advantages, the NAND SSD has rapidly become one of the most prevalent persistent storage devices during the last few decades and has been widely discussed in both industry and academia [98, 99, 100, 101, 102, 103, 104].

Although the NAND flash-based SSD has achieved significant performance improvements, the huge performance gap between high-performance processors and slow storage devices has not changed. In addition, NAND SSD performance will degrade dramatically under some specific situations. For instance, the random read latency can be 10X slower than sequential read due to the pre-fetching failures of the internal read-ahead mechanism; the write performance degrades significantly when the request size is not aligned on a multiple of the page size [99]; and the update operation requires longer completion time because of garbage collection issues. To pursue fast and stable performance, numerous new technologies have been developed, including Phase Change Memory (PCM) [28], Conductive Bridging RAM (CBRAM) [32], Ferroelectric RAM (FeRAM) [105], and others.

As one of these emerging non-volatile memories, Intel recently released 3D Xpoint [44] technology and integrated it into the solid-state driver to replace conventional NAND flash cells. This newly developed storage device is known as Non-Voltile Memory (NVM) block device (i.e., Optane SSD [106]) and can deliver fast I/O access latency and high throughput. Most notably, 3D Xpoint memory can support functionalities that NAND flash cannot, such as byte addressability and in-place update. Due to these performance advantages, the NVM block device is expected to be a good candidate to accelerate the slower storage subsystem in the future. However, few research studies have been presented to discuss the unique characteristics of the NVM block device. Previous publications [44][45] and product specifications [46] either focused mainly on the potential performance impacts of the NVM block device to real applications or gave limited information about its intrinsic characteristics.

In this study, by designing various types of I/O workloads, intensive experiments are conducted on NVM block device, and the collected results are analyzed. The purpose of this chapter is to provide an understanding of the unique performance behavior of NVM block device rather than to carry out a performance competition with the conventional NAND SSD. This chapter makes the following contributions:

1. Several basic performance behaviors of the NVM block device, including I/O access latency, internal parallelism, and performance consistency under highly intensive I/O workload, have been evaluated.

2. Based on multiple custom-designed micro-benchmarks and methodologies, further details of the intrinsic characteristics of the NVM block device are explored, including byte addressability, the elimination of write-driven garbage collection, the read disturb issue, and the tail latency problem.

3. The implications of NVM block device for a real database system are studied using the TPC-H benchmark [107]. By analysis of the collected results, it is noticed that

the NVM block device delivers asymmetric performance accelerations to different types of queries. The potential reasons for these results are explored.

4. A database application is implemented on a hybrid NVM block device and a NAND SSD storage system and its performance is evaluated. Several key factors that can impact this hybrid system's performance significantly are pointed out.

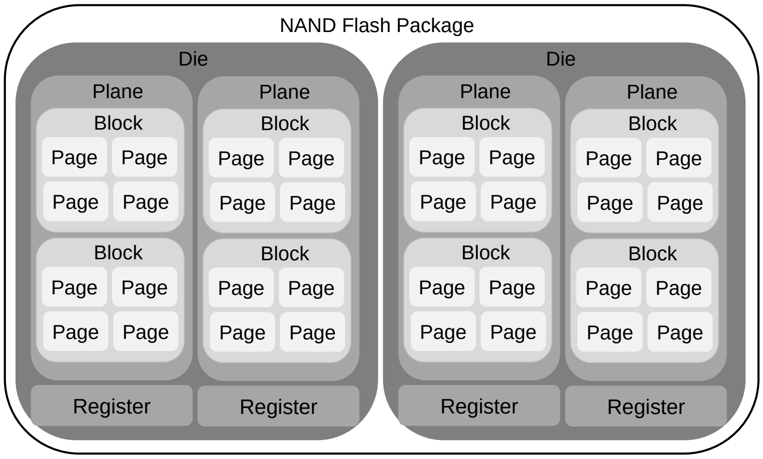
The rest of this chapter is organized as follows. Section 2.2 introduces background information. Section 2.3 discusses related work. Section 2.4 describes the experimental environment and tools. Section 2.5 presents the custom-designed micro-benchmarks and the analysis of the collected results. Section 2.6 describes a study of the performance of a database application on both NVM block device and NAND SSDs and an investigation of the implications of the NVM block device for a real application. Section 2.7 investigates the key factors that could impact database query performance for a hybrid NVM block device and NAND SSD storage system. Section 2.8 concludes the chapter.

2.2 Background

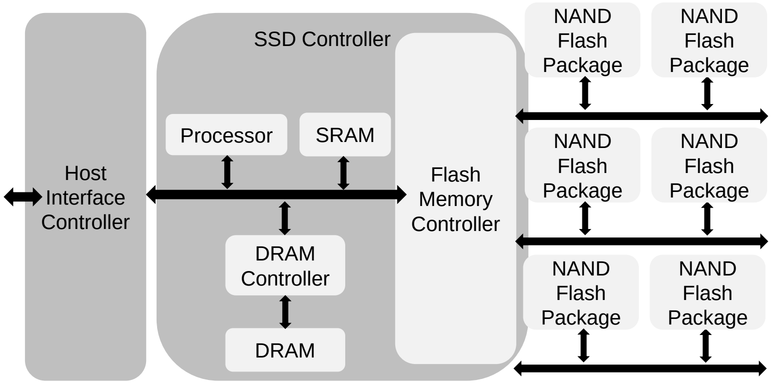
2.2.1 NAND Flash Memory

NAND flash memory is a type of memory medium that is widely used in solid-state drives. As described in Figure 2.1a, a *NAND flash package* typically consists of a number of *dies* (or chips). Each die is composed of one or more *planes* and a number of *registers* for buffering I/O. A single plane contains multiple *blocks*. A block is the smallest *erase* unit inside NAND flash memory. Each block consists of 128 or 256 *pages*. A page is the smallest unit on which normal *read* and *write* operations can be performed. A typical page contains a 4KB (or larger) data area and a 128-byte meta-data area for error-correction coding (ECC) [98].

NAND flash memory has some unique features. First, the read-write (program) latency delivered by NAND flash memory is not uniform. The write operation typically takes a longer time than the read operation. Second, due to the asymmetric unit size between program and erase operations, a log-structure based out-of-place update and a garbage collection mechanism are required for write operations. Finally, a NAND flash memory has a limited number of program/erase(P/E) cycles, usually 1,000 - 100,000



(a) NAND flash package



(b) SSD internal architecture

Figure 2.1: Solid State Drive Internals.

cycles [100, 108] and cannot work once it wears out. Hence, manufacturers usually reserve an extra space (called over-provisioning) to replace bad memory cells and improve the lifespan of a NAND SSD.

2.2.2 3D XPoint Memory

3D Xpoint [44] is one of types of non-volatile memory media recently released by Intel. It has been integrated for the first time into a solid-state drive, the NVM block device, to replace conventional NAND flash memory. Compared with NAND flash, 3D Xpoint memory exhibits different characteristics from two main aspects. First, 3D XPoint memory is byte-addressable, in contrast to NAND flash that accesses data in a page unit. By benefiting from this byte-addressability, the NVM block device internal controller can break a 4KB I/O into several smaller data chunks and spread them across multiple channels concurrently to hide single-I/O latency, thus boosting performance. Second, 3D Xpoint memory supports the update-in-place operation, which means that the NVM block internal controller does not need to perform extra read, write, and erase operations, also known as read-modify-write, when overwrite requests are submitted. This update-in-place enabled functionality allows NVM block device to deliver fast and consistent performance during overwriting.

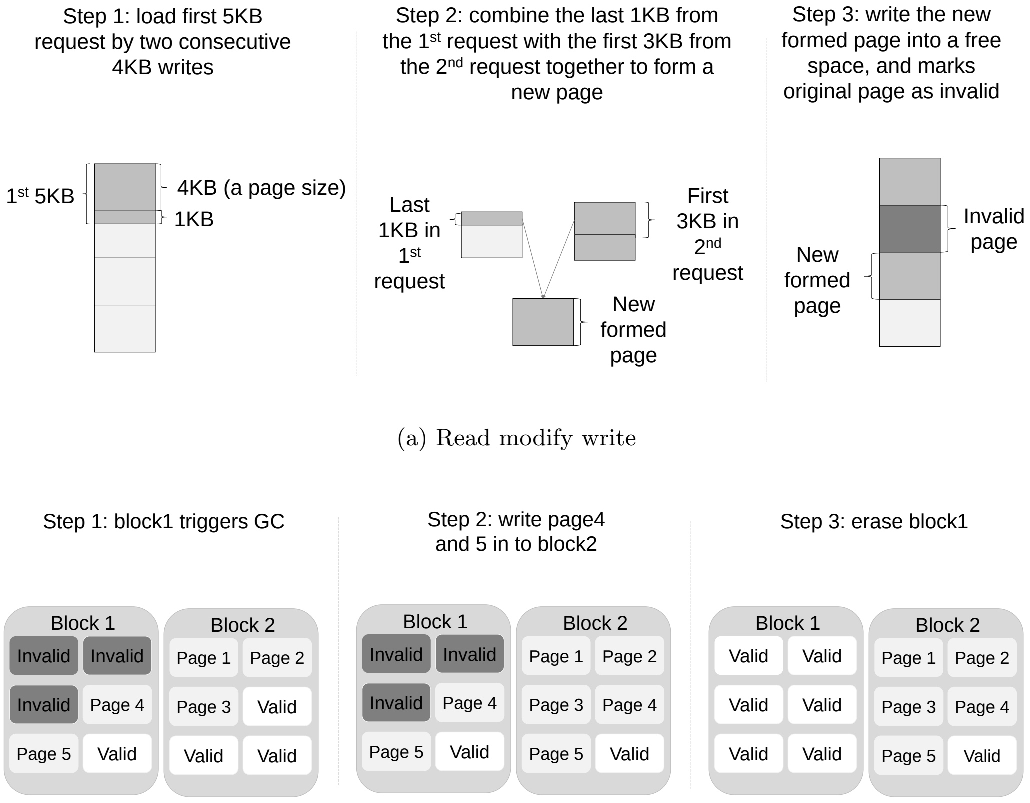
2.2.3 SSD Internal Architecture

As shown in Figure 2.1b, a conventional SSD is composed of several main components, including a host interface, an internal controller, an embedded RAM, and persistent memory media (usually NAND flash memory). The host interface establishes a physical connection between the storage device and the host. It transfers logic commands and data from/to the host through a pre-defined protocol (e.g., SCSI, SATA, or PCIe). The SSD controller manages data placements in NAND flash memory. To improve the SSD's lifetime, the internal controller is also assigned to process wear leveling and garbage collection mechanisms [98]. To deliver low I/O access latency, a high-end NAND SSD is usually equipped with an embedded RAM to buffer data temporarily for read and write operations [109].

Moreover, to improve bandwidth, SSD manufacturers usually spread flash memory cells across multiple independent channels, packages, dies, and planes to exploit SSD internal parallelism [110, 111]. By using this interleaving technique, the SSD controller can fetch/load multiple pages from/into different cells simultaneously and consequentially obtain high bandwidth.

2.2.4 NAND SSD Write Issues

Read Modify Write - As discussed before, a page is the smallest unit on which normal read and write operations can be performed within a NAND SSD. To maintain high


Step 1: load first 5KB request by two consecutive 4KB writes
1st 5KB
4KB (a page size)
1KB
Step 2: combine the last 1KB from the 1st request with the first 3KB from the 2nd request together to form a new page
Last 1KB in 1st request
First 3KB in 2nd request
New formed page
Step 3: write the new formed page into a free space, and marks original page as invalid
Invalid page
New formed page
Step 1: block1 triggers GC
Step 2: write page4 and 5 in to block2
Step 3: erase block1
Block 1
Block 2
Invalid
Valid
Page 1
Page 2
Page 3
Page 4
Page 5

(a) Read modify write

(b) Garbage collection

Figure 2.2: NAND SSD Write Issues.

performance, any requests submitted to a NAND SSD are required to align on a page size. However, in real applications, the request size may not be bounded to a page size. This scenario is defined as a subpage request [112] and results in fragmentation and endurance problems. In addition, because NAND flash memory does not allow data update-in-place, a subpage write typically causes a NAND SSD to trigger more internal operations than necessary. As described in Figure 2.2a, when a subpage request is submitted, the rest of the page (to which the subpage belongs) is read into embedded RAM and combined with the subpage request to form a full page. Then the SSD controller writes this newly formed page into a free space. Meanwhile, the old page is marked as invalid and will be recycled during garbage collection. Finally, the flash translation layer [113, 114, 115, 116] updates the mapping table. The processes just described are known as *read-modify-write* [98]. Because read-modify-write usually requires additional read, write, and update mapping table operations, it can significantly impact foreground write performance.

Write-Driven Garbage Collection [99] - Unlike a hard disk drive, a NAND SSD does not allow data update-in-place. When an update operation is submitted, instead of overwriting, the SSD controller writes the updated data into a free space. Meanwhile, the pages containing the old information are marked as invalid. As the number of invalid pages continues to increase, the NAND SSD may no longer have enough free space to store incoming data. As a result, the internal controller triggers the garbage collection mechanism to recycle invalid pages. During garbage collection, as described in Figure 2.2b, to prevent data loss, the valid pages within the target block must first be read out and written into a free space. The SSD controller then erases all pages within the target block to release more free space. Garbage collection typically degrades NAND SSD performance tremendously due to the extra internal operations (reads, writes and erases) and the long latency of the erase operation ($\sim 2ms$) [99].

2.2.5 Read Disturb Error

Read disturb is an intrinsic error within many types of memory media, such as NAND flash memory [117] and Phase Change Memory (PCM) [118, 119]. When a high read count is required on a given page, the analog voltages of neighboring cells could be disturbed enough to change their stored information. To prevent data loss, NAND SSD manufacturers usually set a threshold on the number of read cycles. Once the number of reads on a given page/block exceeds the threshold, the internal controller forces a refresh to copy the neighboring pages into a new location. Meanwhile, the controller marks the old pages as invalid and recycles them during garbage collection. Because the read disturb issue triggers internal data migration and garbage collection with certain probabilities, it is one of the factors that limit NAND SSD performance.

2.3 Related Work

The NAND SSD has been an active research area in recent decades. Many research studies have been conducted on SSD performance evaluation and optimization. For example, Agrawal et al. [98] provided an overview of SSD internal origination and discussed the design tradeoffs achievable by a trace-driven simulator. They found that SSD performance is highly sensitive both to internal hardware design and to software implementation. Dirik et al. [99] studied the internal parallelism inside SSDs and concluded that exploiting SSD internal concurrency would deliver significant performance improvement. Chen et al. [100, 120] explored the unique performance behavior and internal characteristics of a SATA-based NAND SSD using a number of custom-defined micro-benchmarks. Similarly, Kim et al. [108] presented a number of methodologies for extracting the key configurations inside a SATA-based NAND SSD, including cluster page size, cluster block size, and so on. By optimizing the operating system through these extracted parameters, the overall system performance could be greatly accelerated.

To improve performance, recent research has focused mainly on addressing the lack of NAND SSD data update-in-place ability through either a flash file system implementation [121, 122, 123] or a flash translation layer design [124, 125]. Josephson et al. [126] introduced a direct file system (DFS). Using a newly designed virtual flash storage layer, the DFS can access flash memory cells directly. As a result, DFS is substantially faster than other conventional file systems. Lu et al. [127] proposed an object-based flash translation layer (OFTL). With their design, storage management is assigned to the NAND SSD FTL from the file system to manage flash memory directly. Benefiting from this advanced implementation, write amplifications were reduced tremendously. Taking a different approach, Soundararajan et al. [128] developed a hybrid storage system, Griffin, which used a HDD as a write cache for a NAND SSD. During I/O processing, the HDD maintained a log-structured cache and wrote cached data periodically to the SSD. With this implementation, Griffin not only reduced the number of writes into flash memory, but also provided similar performance to a NAND SSD.

By combining high-performance 3D Xpoint non-volatile memory media with a well-optimized solid state drive, the NVM block device can outperform a conventional NAND SSD in multiple aspects. There has been significant prior work studying the NVM block device. Hady et al. [44] and Izraelevitz et al.[45] provided basic performance evaluations of NVM block device. The performance impacts of NVM block device to real applications have been studied as well. By using multiple benchmarks, Zhang et al. [129]

analyzed NVM block device performance behaviors on virtualized and non-virtualized environments, respectively. A hybrid system called MyNVM for using NVM block device as a second level cache for key-value store has been proposed in [130]. MyNVM reduces DRAM footprint as well as keeping up competitive performance. However, previous work has provided limited information about NVM block device's intrinsic characteristics. In this study, through carefully designed micro-benchmarks, multiple NVM block device's characteristics are explored while its potential benefits were confirmed using a real database application.

2.4 Experimental Setup

Experiments are conducted on a Dell Precision Tower 3620 workstation equipped with four Intel core i7-6700 3.4GHz processors and 16GB DDR4 DIMMs. The Ubuntu 16.04.4 LTS operating system and Ext4 file system are installed on the workstation. Two types of SSDs are directly attached on-board through a PCIe x16 NVMe 30 x4 interface, including an Intel 168GB DC P4800X **Optane** SSD [46] (i.e. the NVM block device) and an Intel 2TB DC P3700 NAND SSD [131]. The NAND SSD is used as a baseline device.

For these experiments, the Completeness Fairness Queuing scheduler (the default I/O scheduler for the Linux kernel) is changed to the Noop scheduler. The Noop scheduler [132] implements the simplest first-in-first-out (FIFO) algorithm, leaving I/O performance optimization to the storage device. For fast storage devices such as SSD, Noop can outperform other I/O schedulers in most cases [100]. In addition, all read and write requests are performed by direct I/O to bypass the buffer cache within the file system.

To investigate the intrinsic characteristics of the NVM block device, the Flexible I/O tester (FIO) [133] is used as a benchmarking tool. In the experiments, various I/O workloads are generated by FIO with different parameters, such as I/O type, request size, queue depth, access rate, and others. During the experiment, each workload is run three times to observe typical behavior. For random read and write requests, a random number generator is configured inside the FIO to guarantee that the read and write order is not repeated on different runs.

2.5 Performance Studies

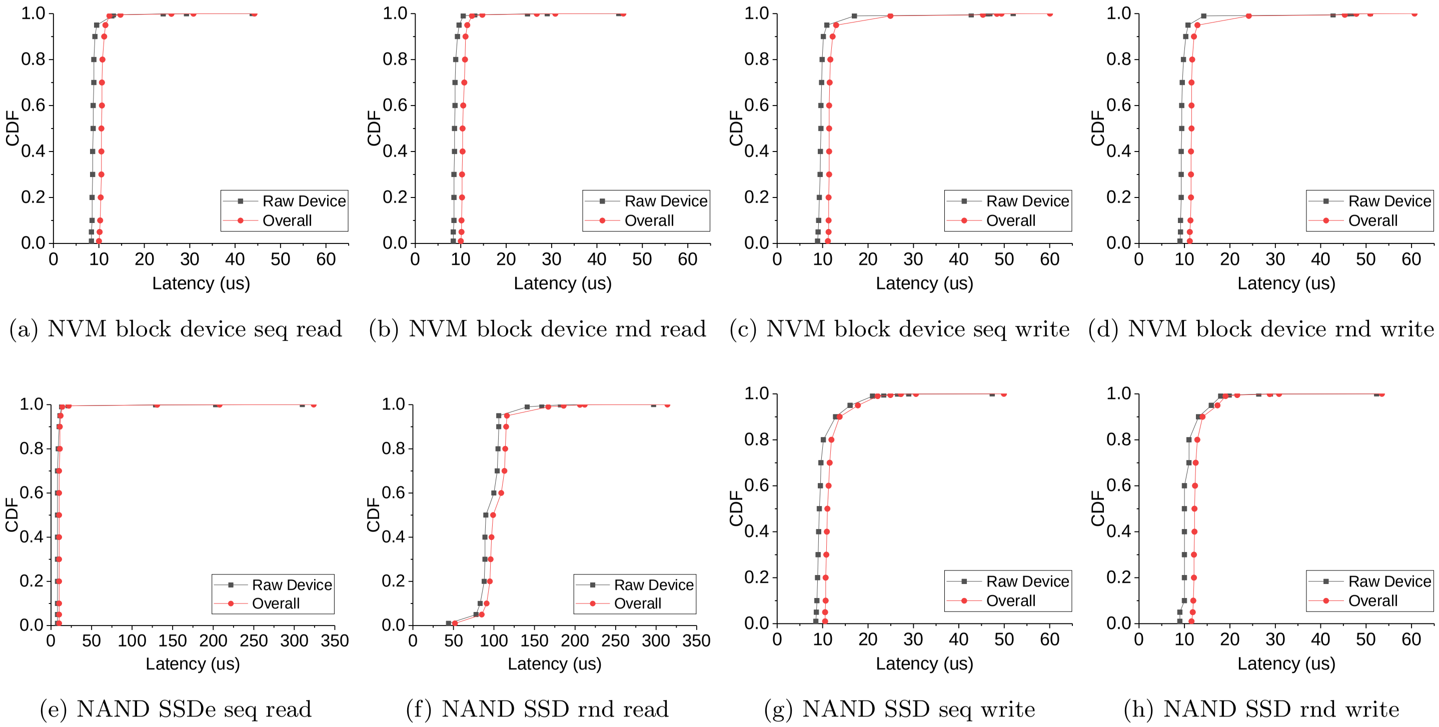
The experiments described in this section investigate the unique features of the NVM block device from multiple aspects, including basic I/O performance behaviors, advanced internal parallelism, performance consistency under highly intensive I/O workload, the influence of unaligned request size, the elimination of write-driven garbage collection, the read performance variation potentially resulting from the read disturb issue, and the tail latency problem. The detailed experimental methodologies and results analysis are discussed below.

2.5.1 Basic I/O Performance Behaviors

Because it uses high-performance non-volatile memory media, the NVM block device is expected to deliver high performance. The experiments described in this subsection examine the basic read/write latency of the NVM block device under different access patterns, using the NAND SSD as a baseline device for comparison.

The experiment involves four workloads, including *sequential read*, *random read*, *sequential write*, and *random write*, with a 4KB request size and a queue depth of 1. Two timers are invoked from two different start points to record the latency: one started recording when the I/O request is generated to measure overall I/O latency; the other start recording when the I/O request is sent into the NVMe submission queue [134] to measure raw device latency. Both timers are stopped once the completion interrupt is received. For each workload, one million I/O requests are submitted into storage, and their I/O latency distribution is recorded.

Figure 2.3 plots the cumulative distribution functions (CDFs) of the I/O access latency for four different workloads on both NVM block device and NAND SSD. The results reveal that the NAND SSD delivers nearly the same overall I/O latency as the NVM block device for the *sequential read*, *sequential write*, and *random write* workloads ($\sim 10 - 14us$). This high performance of the NAND SSD can be attributed to its read-ahead mechanism and the potential of the internal buffer. For sequential read, the internal controller pre-fetches needed pages early into the internal buffer and sends them to the host when required. For both sequential and random write operations,



(a) NVM block device seq read (b) NVM block device rnd read (c) NVM block device seq write (d) NVM block device rnd write

(e) NAND SSDe seq read (f) NAND SSD rnd read (g) NAND SSD seq write (h) NAND SSD rnd write

Figure 2.3: Basic I/O performance behaviors.

the internal controller caches pages into NAND SSD internal buffer and sends the completion interrupt to the host immediately. The cached pages are loaded later in the background into the proper location inside the NAND flash cells. Because the pages are fetched/loaded from/into the NAND SSD internal buffer through a PCIe interface, the NAND SSD delivers similar I/O latency for *sequential read* ($\sim 10us$) and *write* ($\sim 11us$) operations. This assumption can be confirmed from Figures 2.3e, 2.3g, and 2.3h. However, due to mistakes in pre-fetching, a tiny portion of the data is read from the NAND flash cells. The NAND SSD has a very wide I/O latency distribution range on sequential read, as shown in Figure 2.3e. In addition, the performance enhancement provided by the read-ahead mechanism is highly correlated with the access pattern. As shown in Figure 2.3f, under the random access pattern, because of the low pre-fetching accuracy of the read-ahead mechanism, NAND SSD random read overall latency increases to 100 us, which amounts to a 10X degradation of the sequential read pattern.

In contrast, the NVM block device delivers uniform I/O latency regardless of access pattern (either sequential or random). This result confirms that the NVM block device does not need a read buffer to boost its performance, especially for sequential read. It is also worth noting that the NVM block device has nearly the same I/O access latency between read ($\sim 10.72us$) and write ($\sim 11.95us$, a little bit lagged behind read) operations. In addition, compared with the NAND SSD, the NVM block device provides a more compact I/O latency distribution under the read access pattern. As shown in Figure 2.3a-b, the I/O latency of the NVM block device is concentrated in the $10us$ - $45us$ range for both sequential and random read. On the contrary, the NAND SSD exhibits a very wide I/O latency distribution, from $10us$ to $324us$ for sequential read, and from $52us$ to $314us$ for random read. This wide range of the I/O latency distribution for both NVM block device and NAND SSDs is known as tail latency, which will be discussed in more detail later in this chapter. Finally, by comparing overall and raw device latency on the NVM block device, it can be observed that the application level introduces 20% ($2us$) latency while processing I/Os. This could be an important factor limiting NVM block device performance, which hopefully can be fixed by further optimization.

The NVM block device's advanced performance can be attributed to two factors. First, the non-volatile memory media delivers higher performance than NAND flash. Second, to hide single-I/O access latency, the NVM block device spreads a single 4KB

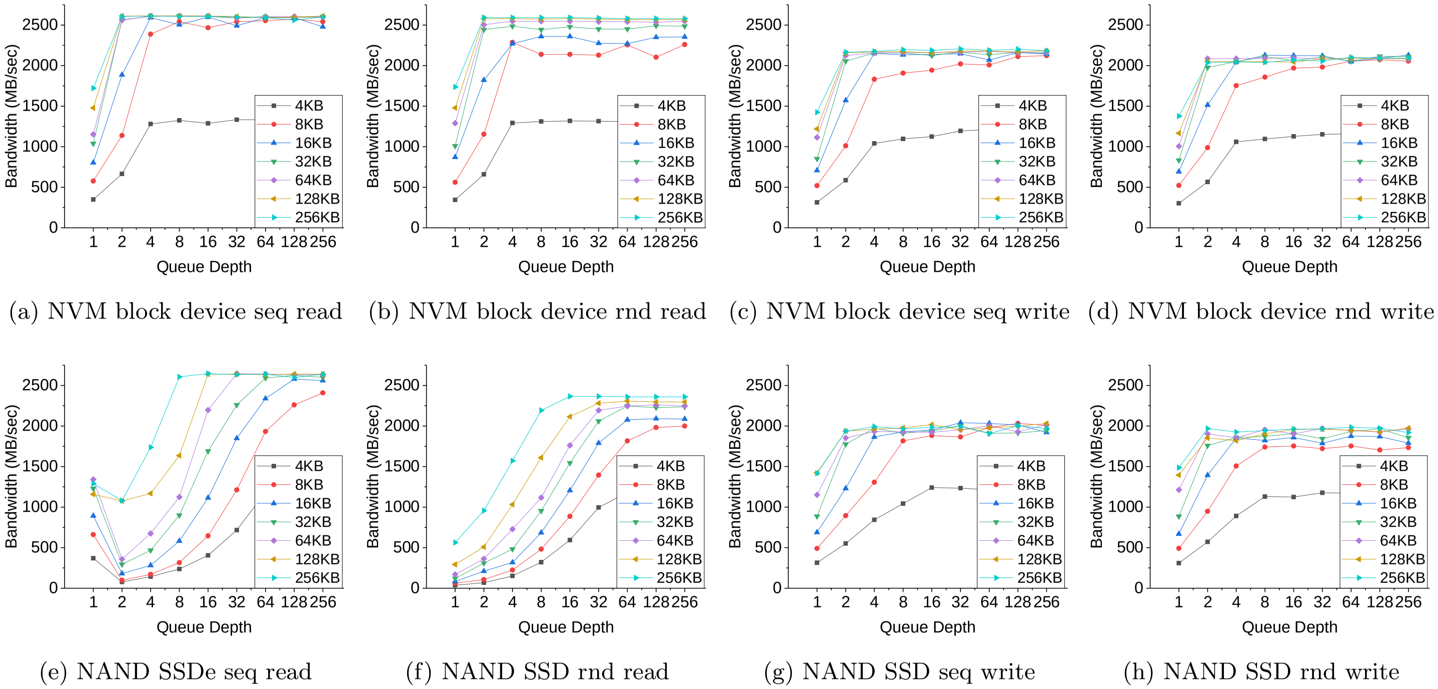
I/O over multiple non-volatile memory media across different channels [44]. In contrast, the NAND SSD accesses a single NAND flash cell to satisfy a 4KB (or larger) I/O. Due to these two improvements, the NVM block device delivers a faster and more compact I/O latency distribution than the NAND SSD.

2.5.2 Advanced Interleaving Technique

To provide high bandwidth, NAND SSD manufacturers normally use an interleaving technique to access pages from multiple channels in parallel. The NVM block device inherits a similar channel-based interleaving technique to that of the NAND SSD, but with advanced implementation [44]. This subsection quantifies the potential benefits of this advanced design inside the NVM block device.

In this experiment, four different I/O workloads, including sequential read, random read, sequential write, and random write, are developed. For each workload, the queue depth is varied from 1 to 256, and their corresponding bandwidth is recorded with different request sizes from 4KB to 256KB.

Figure 2.4 shows the results of this experiment. Compared with the NVM block device's unprecedented performance enhancement, the bandwidth gains delivered by the interleaving technique inside the NAND SSD are limited. Even though the NAND SSD has a maximum bandwidth close to that of the NVM block device (2.5 GB/s for read and 2 GB/s for write), it requires a much greater queue depth than the NVM block device to deliver this performance. As shown in Figures 2.4e and 2.4f, under the 4KB request size, the NAND SSD could provide its maximum sequential and random read bandwidth (1.2 GB/s) only when its queue depth is enlarged to 128. In contrast, the NVM block device delivers the same bandwidth with a queue depth of only 4, which is 32X smaller than the NAND SSD. Similar results are also observed on both sequential and random write operations. With the same queue depth, even with an internal buffer, the NAND SSD still could not provide the same bandwidth improvement as the NVM block device for smaller request sizes such as 4KB and 8KB. Moreover, the interleaving technique used within the NAND SSD is easily affected by other processes. As shown in Figure 2.4e, when the queue depth is decreased to 2, the NAND SSD sequential read bandwidth degrades tremendously and recovered only when the queue depth is continuously increased to be larger than 16. This performance degradation



(a) NVM block device seq read (b) NVM block device rnd read (c) NVM block device seq write (d) NVM block device rnd write

(e) NAND SSDe seq read (f) NAND SSD rnd read (g) NAND SSD seq write (h) NAND SSD rnd write

Figure 2.4: Interleaving technique.

issue resulted from interference between the internal parallelism and the read-ahead mechanisms inside the NAND SSD [120].

Unlike the NAND SSD, which spreads numerous pages (4KB or larger) across multiple NAND flash channels in parallel to deliver higher bandwidth, the interleaving technique inside the NVM block device works in a more efficient manner. Within the NVM block device, the internal controller distributes a single 4KB I/O across multiple independent channels concurrently [44]. This means that the I/O size accessed from a single NVM medium could be smaller than a normal page size. The NVM block device can do this because the NVM is a byte-addressable memory medium and can support access to small data chunks (smaller than 4KB). With the benefit of this advanced interleaving technique and high-performance NVM medium, the NVM block device achieves its maximum bandwidth with a much smaller queue depth than the NAND SSD. As illustrated in Figure 2.4a-d, for smaller request sizes (4KB, 8KB, and 16KB), the NVM block device bandwidth starts to become saturated after the queue depth is increased to 4. For larger request sizes (equal to or larger than 32KB), the NVM block device can reach its maximum bandwidth with a queue depth of only 2. This is possible because the larger request size has partially triggered the NVM block device's internal parallelism, and therefore it requires less queue depth to provide its maximum bandwidth.

In addition, the results obtained here show that the NVM block device provides similar performance gains for different I/O access patterns (either sequential or random) as queue depth increases. This occurs because the NVM has nearly the same access latency for both sequential and random access patterns (Section 2.5.1). Moreover, because the NVM block device directly accesses I/Os from the host to the storage media through a hardware-only path [44], the performance degradation problem that exists inside the NAND SSD for the sequential read access pattern is not observed with the NVM block device.

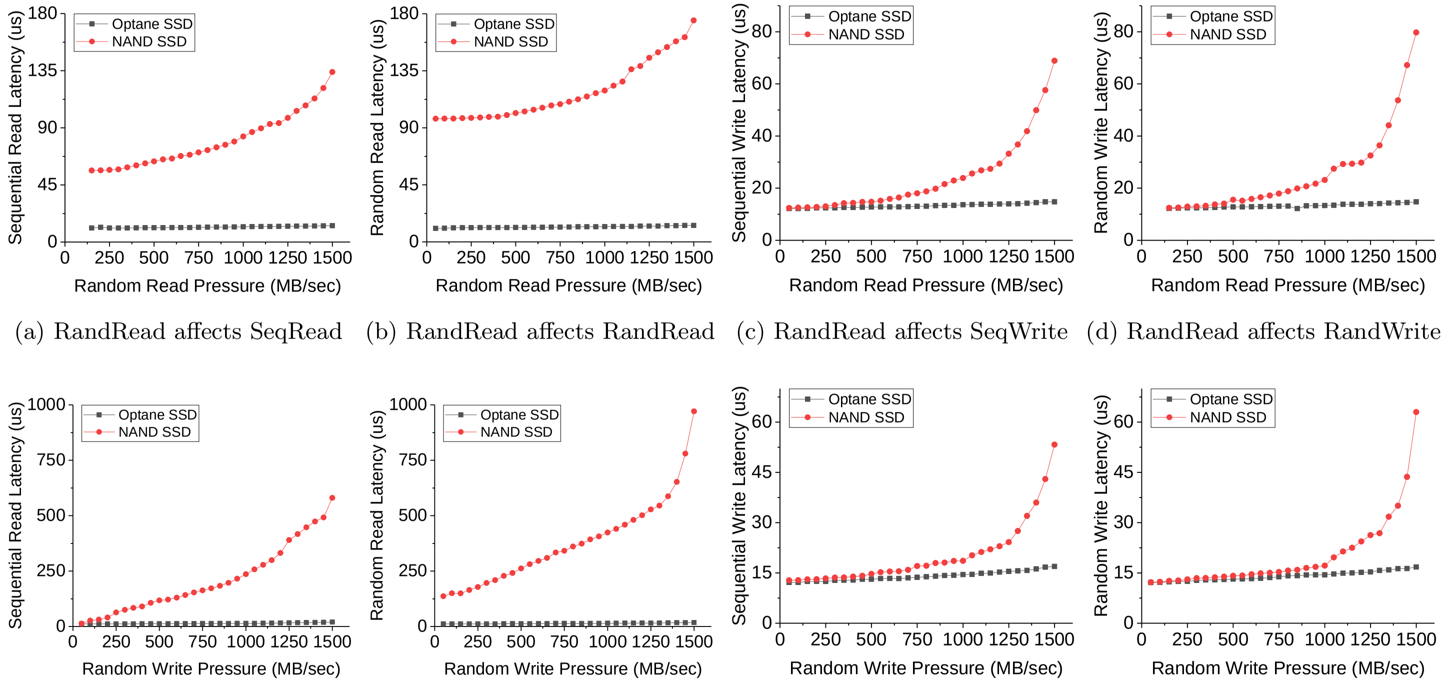
Finally, for both NVM block device and NAND SSD, the maximum bandwidth delivered by 4KB and larger request sizes (8KB or larger) is found to have a 2X difference. This may have occurred because the *clustered page* [108] size in both NVM block device and NAND SSDs is 8KB, where the clustered page size is defined by the number of integrated pages to use the SSD interleaving technique.

2.5.3 Performance Consistency Under Highly Intensive I/O Workload Pressure

The previous subsections have explored the characteristics of the NVM block device using a single-I/O process. However, in real applications, multiple processes (or threads) may submit their I/O requests into storage simultaneously. Under this situation, the performance of a storage device can be impacted significantly due to internal resource competition. To understand the performance consistency issues of the NVM block device under highly intensive I/O workloads, intensive experiments and measurements are carried out and are described in this subsection.

In this experiment, two types of *background* I/O-intensive workloads are generated by FIO, including random read and random write. The random I/O access pattern is used because it triggers more internal operations than sequential patterns and can be expected to reflect the worst performance of storage devices. By varying the data access rate of the background I/O workload in terms of how many data points are read/written from/into the storage device per unit time, the background I/O workload pressure can be controlled. To show how storage performance is affected, when the above background I/O workload is running, a *foreground* process is called to submit a million I/O requests to storage with 4KB request size and a queue depth of 1, and its average I/O access latency is recorded. Figure 2.5 plots the experimental results, where the x-axis represents I/O access pressure. For instance, if the random read pressure is equal to 400 MB/s, then the background I/O workload accesses data from the storage device at a rate of 400 MB/s. The y-axis records the I/O latency in microseconds for four different access patterns.

As discussed in Section 2.5.1, with the benefit of its internal buffer and read-ahead mechanism, under a single-I/O process, the NAND SSD can deliver nearly the same I/O access latency as the NVM block device for sequential read, sequential write, and random write. However, as the background I/O workload pressure is continuously increased, the potential benefits of the NAND SSD's internal optimization mechanisms are greatly impacted. As a result, NAND SSD performance degrades tremendously. As illustrated in Figures 2.5a, 2.5c, and 2.5d, after the background random read pressure is increased to 1500 MB/s, the NAND SSD's I/O access latency for sequential read, sequential write, and random write degrades by 13.25X, 5.78X, and 6.19X respectively



(a) RandRead affects SeqRead (b) RandRead affects RandRead (c) RandRead affects SeqWrite (d) RandRead affects RandWrite

(e) RandWrite affects SeqRead (f) RandWrite affects RandRead (g) RandWrite affects SeqWrite (h) RandWrite affects RandWrite

Figure 2.5: Performance consistency under highly intensive I/O workload.

compared with the results for a single-I/O process. Poorer results are observed under the random write I/O workload. As shown in Figure 2.5e-f, when the background random write pressure is increased to 1500 MB/s, the foreground sequential and random read latency increase to 580 *us* and 971 *us* respectively. Finally, from Figure 2.5a, it can be observed that when the random read I/O pressure is set to 50 MB/s, the prefetching accuracy of the NAND SSD's read-ahead mechanism is significantly affected, and its corresponding sequential read latency increases to 56.32 *us*, amounting to a 5.6X degradation compared with the results for a single-I/O process (10.11 *us*). This NAND SSD performance degradation issue can be attributed to resource competition among its internal optimization mechanisms and the physical nature of the NAND flash memory. For instance, under background random read pressure, as the data access rate is continuously increased, the NAND SSD controller can more easily trigger internal refresh to migrate data into a new location to prevent read disturb errors. As a result, the performance of the foreground process can be tremendously affected. For the background random write I/O workload, because it may result in internal garbage collection, the performance of the foreground process is further degraded.

In contrast with the NAND SSD's inconsistent performance behavior, the experimental results reveal that the NVM block device maintains low I/O access latency even under high-pressure I/O workload. Analyzing the results show that under heavy random read I/O pressure (up to 1500 MB/s), the I/O access latency of the NVM block device on sequential read, random read, sequential write, and random write increase by only 2.2 *us*, 2.39 *us*, 2.62 *us*, and 2.65 *us* respectively compared with results for the single-I/O process. Similar results are also observed for the random write I/O pressure case. When the random write pressure is increased to 1500 MB/s, the I/O latency of the NVM block device for sequential read, random read, sequential write, and random write increase by 8.23 *us*, 6.26 *us*, 4.77 *us*, and 4.6 *us* respectively. It is apparent that the background write pressure has a stronger impact on foreground process performance than the read pressure. This may have occurred because the NVM block device's write operation requires more resources (either hardware or software resources) than the read operation. This consistent performance behavior of the NVM block device can be attributed to the following factors. First, the NVM block device is equipped with a well-optimized internal controller [44], which enables the NVM block device to schedule

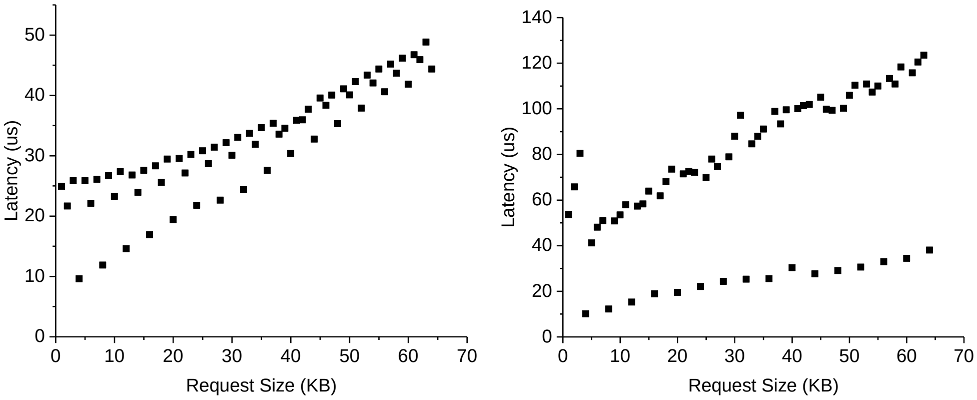
I/O processes more efficiently than the NAND SSD. Second, because I/O requests are performed directly from the host to the storage media by a hardware-only path [44], the extra latency that results from the NAND SSD's internal resource competition under high-pressure I/O workload is removed from the NVM block device. Third, because an update-in-place enabled NVM medium is used, the extra resource consumption resulting from the NAND SSD's internal garbage collection is eliminated from the NVM block device. Due to these factors, the NVM block device can deliver relatively stable performance under increasing background I/O workload pressure.

2.5.4 Performance Influence of Unaligned Request Size

The NAND SSD performs normal read and write operations based on the page unit. To sustain high performance, any requests submitted to the NAND SSD are required to be aligned on a page size. However, in real applications, an I/O request may no longer be bound to a page size. Especially under the sequential write access pattern, a request with unaligned size leads the NAND SSD to trigger more internal operations than necessary, a sequence known as read-modify-write [98], and therefore degrades NAND SSD performance. However, if the NAND flash is replaced by a byte-addressable NVM medium, does the NVM block device still need to perform I/O based on a certain chunk size to avoid performance degradation?

To answer this question, the authors develop an FIO-based I/O micro-benchmark to record the sequential write latency with different request sizes. This benchmark is similar to [108], but with certain modifications. In this experiment, to avoid interference by the host file system, only the raw device performance is recorded. To avoid resource competition between multiple I/O jobs, only one process run the workload with a queue depth of 1. To improve reliability, one million sequential write requests are submitted into storage, and their average I/O access latency is then calculated.

The same benchmark is run on both NVM block device and NAND SSD by varying the request size from 1 to 64KB with granularity 1KB; their corresponding average sequential write latencies are plotted in Figure 2.6. From Figure 2.6b, a dramatic average write latency increment can be observed from the NAND SSD when the request size is not a multiple of 4KB, which is similar to the result presented in [108]. This performance degradation issue resulted from the NAND SSD's extra read, write, and



(a) NVM block device.

(b) NAND SSD.

Figure 2.6: Performance influence of different request sizes.

update mapping table operations during the read-modify-write process (Section 2.2.4). In addition to this, a new phenomenon is observed. For I/Os with a request size smaller than a page size (4KB), their corresponding I/O access latency for the 1KB, 2KB, and 3KB request sizes increase tremendously to 54 *us*, 66 *us*, and 80 *us* respectively. This could have been caused by multiple read-modify-write processes occurring within a single page. For instance, a 1KB request triggers four read-modify-write processes to package four consecutive requests into one page. Furthermore, this small unaligned request size also makes the number of invalid pages within NAND flash memory increase rapidly. As a result, the NAND SSD controller is more likely to invoke garbage collection to recycle these invalid pages.

NVM medium has two notable differences from NAND flash memory. First, because NVM medium supports data update-in-place, the extra read, write, and update mapping table operations caused by read-modify-write are removed. Second, because NVM is a byte-addressable memory medium, it can access small data chunks (smaller than a 4KB page inside the NAND SSD) directly. Because of these advantages, within the NVM block device, the performance degradation resulting from unaligned request size is reduced significantly compared with the NAND SSD results. However, as shown in Figure 2.6a, a performance degradation in the NVM block device can still be observed when the request size is not a multiple of 4KB. There are two ways to explain these NVM block device performance degradation issues. First, although NVM is byte-addressable, because it is integrated into a block storage device, the I/Os submitted to the NVM block device should conform to the block device's functionality - block-based access - to mitigate performance degradation. Second, the NVM block device optimizes its internal I/Os by spreading a single 4KB across multiple memory cells [44]. Any request that is not a multiple of 4KB cannot be well optimized and therefore has longer I/O access latency. In addition, the results for the NVM block device show that there are three latency layers. For the lowest layer, because all I/O requests are aligned on a page size and can be well optimized, they have the lowest access latency. For the middle layer, because the I/O sizes are even values and can be partially optimized, their latency is between best and worst. For the highest layer, because the I/O request sizes are odd numbers that are hard to optimize, the NVM block device delivers its highest access latency.

Finally, unlike the NAND SSD's results that the average sequential write latency increases proportionally to the unaligned request size and always maintains a large gap when the request size is a multiple of 4KB, the performance degradation of the NVM block device that results from the unaligned request size is reduced as request size increases. When the request size is larger than 50KB, the NVM block device delivers almost the same write latency whether or not the request size is bounded on 4KB. The authors believe that this occurs because as the request size increases, more internal parallelisms are invoked within the NVM block device, which hides the single sequential write latency.

2.5.5 Performance Degradation of Write-Driven Garbage Collection

As discussed in Section 2.2.4, the NAND SSD does not allow data update-in-place. When the content within a page is modified, instead of overwriting, the NAND SSD controller writes the updated information into a new page and marks the original page as invalid. As the number of invalid pages continues to increase, the SSD controller invokes internal garbage collection to recycle invalid pages to release more available space. The entire process is known as write-driven garbage collection [99]. Because garbage collection typically triggers extra read, write, and erase operations in the background, it degrades a foreground process's performance once it happens. Moreover, because

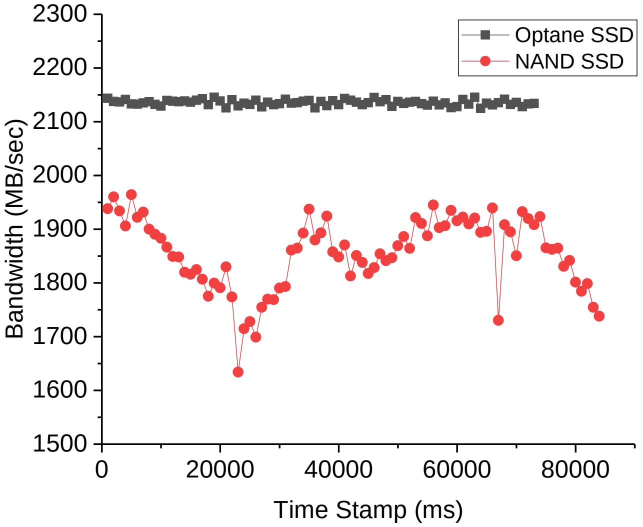


Figure 2.7: Write-driven garbage collection.

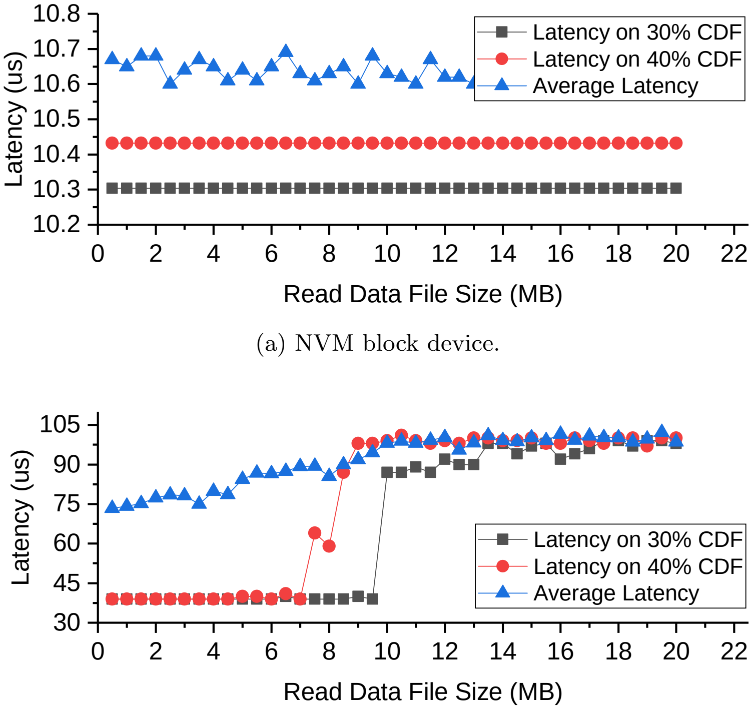
NAND flash memory has a limited number of program/erase cycles, typically 10,000 to 100,000 [100], invoking garbage collection frequently leads a NAND SSD to reach its lifespan early.

However, by replacing the NAND flash memory inside the solid-state drive with the NVM medium, the NVM block device can support data update-in-place. Benefiting from this technology innovation, the NVM block device is believed to have removed write-driven garbage collection and is expected to deliver more consistent performance under the write access pattern than the NAND SSD. This subsection explores the potential of enabling data update-in-place through a custom-designed micro-benchmark. Before the test, both NVM block device and NAND SSDs are filled with random writes, leaving only 150 GB free space. Then a 150GB file is randomly written to fill the remaining available space inside the storage with carefully selected request size and queue depth to enable both NVM block device and NAND SSDs to deliver their maximum bandwidth. In this experiment, the request size and queue depth are set to 16KB and 256 respectively. After pre-processing, both the NVM block device and the NAND SSDs are full. At the end, the previous 150 GB file is overwritten following the exact random write orders, request size, and queue depth used during the pre-processing stage, and the corresponding bandwidth is recorded into a log file. To avoid interference, this log file is preserved on the host memory at the beginning and is flushed into the storage until the overwrite process is terminated. Because the storage device is already filled before the workload is run, the NAND SSD has to trigger garbage collection during overwriting to release space to store the updated information. In addition, because manufacturers usually reserve a specific space inside the NAND SSD (called over-provisioning) to store valid pages during garbage collection, there is enough available space to guarantee that the NAND SSD can complete garbage collection during overwriting. This experiment uses random write because it more easily triggers garbage collection than sequential write.

The micro-benchmark is run on both NVM block device and NAND SSD, and the corresponding bandwidths are plotted in Figure 2.7. As the results show, when overwrite requests are submitted into a full storage device, the random write performance of the NAND SSD is impacted substantially due to background garbage collection. During the entire process, a sharp bandwidth drop is observed for the NAND SSD, and the corresponding performance degradation is 19% (from the maximum 2GB/s to the minimum 1.62GB/s). Although NAND SSD performance can be recovered after garbage collection, its maximum bandwidth can only be maintained within a very short time interval and drops immediately when the next garbage collection starts. On the contrary, because NVM allows data update-in-place, the performance degradation issues that result from write-driven garbage collection are removed from the NVM block device. During the entire process, the NVM block device maintains consistent bandwidth, around 2.1GB/s. Even though the NVM block device could not provide exactly the same bandwidth during overwriting, the bandwidth difference is only 0.5% (from the maximum 2.14GB/s to the minimum 2.13GB/s), which can be attributed to system variation.

2.5.6 Read Disturb Issue

This subsection describes the investigation of whether the NVM block device faces the same read disturb issue as the NAND SSD and how much performance degradation the NVM block device experiences during an internal forced refresh. In this experiment, the target page is accessed with high read count on both the NVM block device and NAND SSD, and their corresponding access latencies are recorded. To avoid caching the target



(a) NVM block device.

(b) NAND SSD.

Figure 2.8: Internal buffer size.

page into the storage device's internal buffer, the random read option is used, and the read data size is set to 2X larger than the estimated internal buffer size. The capacity of the internal buffer can be roughly estimated as follows: randomly read a data file repeatedly (3 times) with 4KB request size and record the I/O access latency of each request. When the data file size is smaller than the read buffer size, repeatedly accessing the same data file would allow the internal buffer to cache a portion of the frequently accessed pages. As a result, the cached pages would have much smaller access latency than those that are not cached, and the average read latency of a single 4KB page is relatively small. As the data file size continues to increase and becomes larger than the internal buffer capacity, because the cached pages must be evicted from the internal buffer at each turn and all pages must be read from storage media, all requests have similar read latency, and the average latency becomes high. To see clearly how read latency varies as read data file size is continually increased, the I/O access latency is

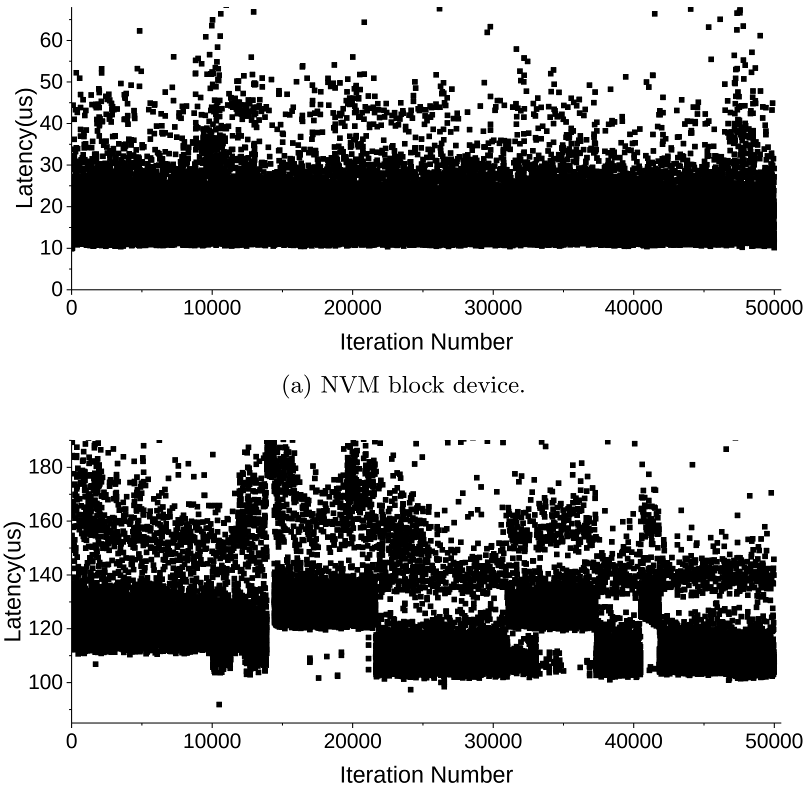
Algorithm 1 Read Disturb Test

```
Input: F, /*target file stored on storage device with size 21MB*/

procedure TESTING PROCEDURE
    for $i \leftarrow 0$ to 50,000 do
        clean host memory
        lseek(F, 0, set) /*read at the beginning of the file*/
        timer start
        read_file(F, 4KB) /*read target page*/
        timer end
        read_file(F, 20MB) /*evict target page from SSD internal buffer by random read an extra 20MB file*/
        record target page read latency into a logfile
    Flush logfile to storage device
```

plotted on the 30%, 40% cumulative density function (CDF) level. The average latency in Figure 2.8. Figure 2.8b shows a sharp read latency increment on the 30%, 40% CDF level, whereas the read data file size increases to 8MB and 10MB respectively. And the average read latency of a single I/O request becomes constant after the data file size exceeds 10MB. Based on these results, the internal buffer size of a NAND SSD can be roughly estimated as 10MB. However, read latency variations are not observed from the NVM block device. This result confirms that the NVM block device directly accesses I/O from the host to the storage media without using firmware optimization techniques such as a internal buffer.

After the estimated read buffer size has been determined, the target page (4KB, the page size has already be explored in Section 2.5.4) is accessed repeatedly on both NVM block device and NAND SSDs to test the read disturb issue. The micro-benchmark is described as Algorithm 1. Some implementation details must be mentioned here: first, to avoid caching the target page in the host, DRAM is clean hosted before the read; second, the target page is read by direct I/O and the read latency recorded into a log file; third, to avoid interference, the log file is preserved in host DRAM and flushed into storage until the program terminated; fourth, after the target page is read, an extra 20MB data are immediately random read to evict the target page from the SSD internal buffer and guarantee that in each iteration, the target page would be read from the storage media; and finally, the number of read cycles on the target page is set to a large value (50,000) to trigger SSD internal forced refresh to prevent the read disturb error.



(a) NVM block device.

(b) NAND SSD.

Figure 2.9: Read disturb issue.

Figure 2.9 shows a plot of the experimental results. The read latency variation from both NVM block device and NAND SSD can be observed. The observed results may arise from internal forced refresh to prevent the read disturb error on both NVM medium and NAND flash memory. In addition, from Figure 2.9b, there are clear latency jumps at iterations number 14700, 32000, and 40000. This may have occurred because the NAND SSD triggers internal garbage collection to recycle invalid pages. As a result, it degrades the foreground read performance further. The NAND SSD's random read

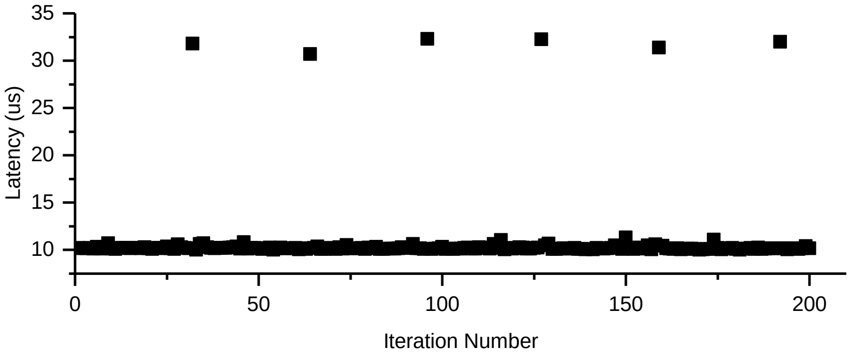


Figure 2.10: NVM block device, only access the target page repeatedly.

latency immediately drops when garbage collection is completed at iterations number 22000, 37500, and 41000. In contrast, because garbage collection is eliminated from the NVM block device, the clear latency jumps and drops that existed for the NAND SSD are not observed.

However, from Figure 2.9, it is hard to determine the precise number of read cycles that triggers the read disturb issue and how much performance would degrade during the SSD internal forced refresh. This difficulty is compounded because, besides accessing the target page repeatedly, it is also necessary to random read an extra 20MB data file in each iteration to evict the target page from the NAND SSD internal buffer, which also causes the read disturb issue and leads to an SSD internal forced refresh. To answer these questions, the target page was simply accessed repeatedly from the NVM block device; the corresponding results are plotted in Figure 2.10. In this experiment, plotting starts after the latest high read latency occurred that is potentially due to the read disturb issue. These results reveal a periodic latency increment at approximately every 30 iterations and a corresponding performance penalty resulting from the internal forced refresh of 20 *us*.

2.5.7 Tail Latency

As data sizes continue to grow, storage performance is becoming critical to many data-intensive applications today. To deliver low and stable response time, modern applications demand that 99.9% of all requests must be answered within a limited time interval [135]. However, due to the impact of garbage collection and other background mechanisms, the NAND SSD typically has a very wide I/O latency distribution, known as tail latency. This performance instability introduces milliseconds of delay and leads to a 1.5X - 2.0X slowdown of storage systems [136]. In this subsection, the tail latency of both NVM block device and NAND SSD are explored under different scenarios (high-pressure I/O workloads, garbage collection, and read disturb). To reflect how background processes impact tail latency, the results in Section 2.5.1 are used as a *baseline*, and it is assumed that only one process submitted random read requests to storage. In this experiment, the random read request is used to examine tail latency. This choice is made because most modern NAND SSDs are equipped with an internal buffer (Section 2.5.1). Using other I/O requests (such as sequential read, sequential write, and random write) cannot reflect the real performance of NAND flash memory.

Table 2.1: Tail latency comparison

Percentile	50	90	99.5	99.9	99.95	99.99
NVM block device (us)						
Baseline	8.6	10.6	13.1	24.7	29.1	44.8
200 MB/s Pressure	8.9	14.5	16.5	26.0	29.3	45.2
400 MB/s Pressure	9.0	22.4	23.7	30.0	31.4	49.3
800 MB/s Pressure	9.3	17.5	19.8	29.6	33.0	54.0
Garbage Collection	9.5	33.5	41.2	45.9	47.9	56.0
Read Disturb	8.6	12.6	13.4	24.9	28.3	45.7
NAND SSD (us)						
Baseline	90	141	159	182	186	297
200 MB/s Pressure	92	148	162	179	189	318
400 MB/s Pressure	97	151	167	190	208	344
800 MB/s Pressure	101	212	253	343	449	832
Garbage Collection	100	2474	2737	2989	3064	3228
Read Disturb	100	285	293	314	408	2089

To understand how background I/O workload impacts storage performance stability, the experiment described in Section 2.5.3 is repeated. A background process is called to submit random read requests to storage with a certain access rate (*200 MB/s*, *400 MB/s*, and *800 MB/s*). Another foreground process is then invoked to submit a million random read requests into storage, and its I/O latency distribution is recorded on the 99th, 99.5th, 99.9th, 99.95th, and 99.99th percentiles, as shown in Table 2.1. To indicate the I/O latency distribution range, the I/O latency on the 50th percentile, called the median latency, is also recorded. The 50th percentile latency can be thought of as representing the average latency of all I/O requests. The results show that as background I/O workload pressure is continuously increased, the NAND SSD's I/O latency becomes distributed over a wider range. For instance, when the background I/O workload pressure is increased to 800 MB/s, the NAND SSD's foreground random read latency ranges from 101 *us* to 832 *us* for the 50th to the 99.99th percentile. In contrast, the NVM block device's foreground random read latency varies only from 9.3 *us* to 54 *us* for the 50th to the 99.99th percentile, or a 16.4X smaller range than that of the NAND SSD.

Garbage collection is another factor that contributes long tail latency to the NAND SSD. To investigate this, the experiment described in Section 2.5.5 is repeated. This experiment overwrite a 150 GB file to storage by a background I/O process, then submitted a foreground random read request to storage and recorded its latency distribution. To minimize the impact of resource competition due to background I/O workload, for the overwrite operation, the request size is decreased to 4KB and the queue depth to 1. As shown in Table 2.1, because of background garbage collection, the NAND SSD's 99.99th percentile latency is increased to 3228 *us*, which is 32X greater than its corresponding median latency. On the contrary, benefiting from the elimination of garbage collection, the NVM block device has only 5.9X difference between the 50th and 99.99th percentile latency.

In addition to the background I/O workload process and garbage collection, data refresh can also intensify internal resource competition in a storage device, resulting in longer tail latency. The experiment described in Section 2.5.6 is therefore repeated. A 20MB data file is randomly read 50,000 times and its I/O latency distribution recorded. As shown in Table 2.1, due to background data refresh and garbage collection (Section 2.5.6), the NAND SSD produces a very wide I/O latency distribution, and its

99.99th percentile latency reaches 2080 *us*, which is 20.9X larger than its median latency. In contrast, the impact of the NVM block device's internal data refresh on foreground performance is negligible. As shown by the results, even though the NVM block device triggers internal data refresh, it delivers almost the same I/O latency distribution as the baseline. This result also indicates that NVM block device's tail latency under the single-I/O process potentially resulted from its internal data refresh to avoid read disturb error.

By summarizing the results from the different scenarios, it can be concluded that the background I/O process, internal data refresh, and garbage collection are the three main factors that contribute long tail latency to the NAND SSD. In contrast, the NVM block achieves a similar response trend among the various scenarios. In other words, the negative impact of internal data refresh and overwriting on the NVM block device's tail latency is reduced, but the background I/O process becomes a main factor that can contribute to long tail latency.

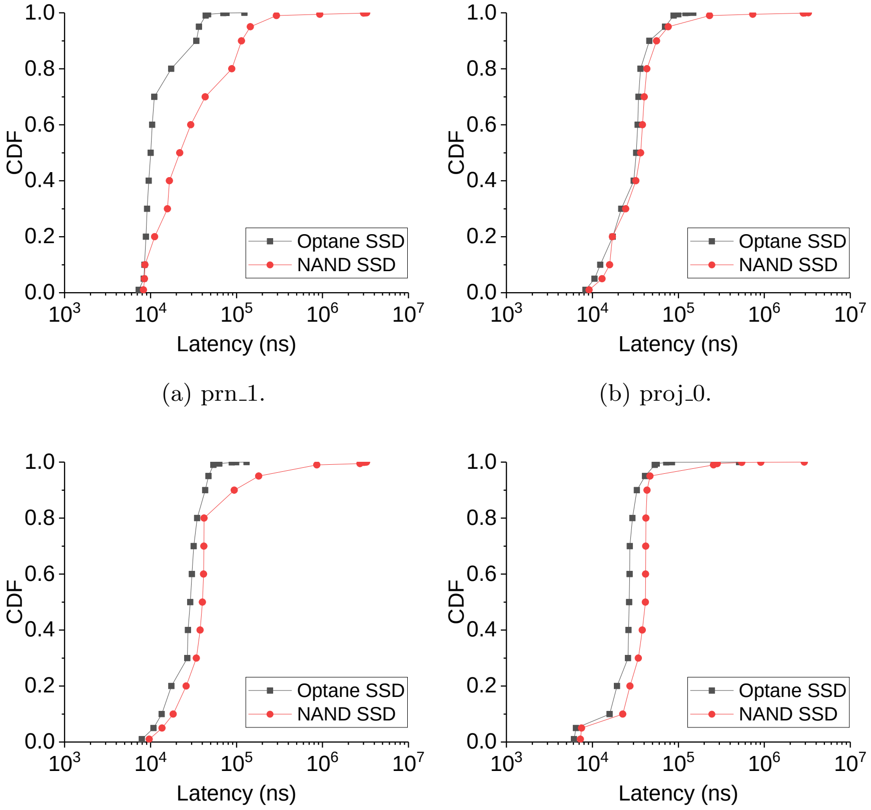
2.5.8 Real Trace Implementation

Table 2.2: MSR trace configuration

	Number of IOs (Millions)		Total request size (GB)	
	Write	Read	Write	Read
prn_1	2.77	8.46	30.78	181.35
proj_0	3.70	0.53	144.27	8.97
usr_2	1.99	8.58	26.47	415.28
web_2	0.04	5.14	0.78	262.82

To explore the impact of the NVM block device on a real storage subsystem, as described in this subsection, an MSR trace [137] is implemented on both the NVM block device and NAND SSD and the I/O latency distribution of each request recorded, as shown in Figure 2.11. The trace characteristics are summarized in Table 2.2.

As Figure 2.11 shows, on the same percentile level, the NVM block device delivers faster I/O access latency than the NAND SSD. Meanwhile, as discussed in Section 2.5.7, due to the elimination of garbage collection and low performance degradation during



(a) prn_1.

(b) proj_0.

(c) usr_2.

(d) web_2.

Figure 2.11: MSR trace implementation.

internal data refresh, the NVM block device produces a more compact I/O latency distribution than the NAND SSD. For instance, for proj_0 trace, the 99.99th percentile latency of the NAND SSD increases to 3256.4 *us*. In contrast, the latency of the NVM block device on the same percentile is only 149.7 *us*, which is 22X lower than the NAND SSD.

2.6 Implications for a Real Database System

Database systems are one of the prime I/O-intensive applications. Their performance is typically limited by laggard storage subsystems. However, by incorporating a new type of non-volatile memory, the NVM block device is expected to provide a new insight for these I/O-intensive applications. This section quantifies the potential benefits of a NVM block device in a relational database management system (RDBMS) using the TPC-H [107], a read-intensive benchmark. Using the TPC-H benchmark, Yang et al. [69] explored the limiting factors that introduce extra I/O cost into database systems, including read amplification issues, high buffer pool cache miss rates, and inefficient use of temporary tables. However, this discussion lacked a detailed analysis of TPC-H queries from the storage level. This chapter mainly focuses on evaluating NVM block device performance on an RDBMS with a detailed explanation. The experimental results in this study have shown that NVM block device delivers up to 6.5X speedup for the TPC-H benchmark.

2.6.1 TPC-H Benchmark and RDBMS

The TPC-H is one of the on-line analytical processing benchmarks that is widely used in both academia and industry. This benchmark program is composed of a number of business-oriented queries and data modifications. The queries and database tables are carefully designed to have broad industry relevance. The performance reported by the TPC-H benchmark reflects multiple capacities of the tested system.

Table 2.3: Database tables' information of TPC-H benchmark

Table name	Abbreviation of each table's name	Table size
SUPPLIER	S	121MB
REGION	R	96KB
PARTSUPP	PS	9.1GB
PART	P	2.1GB
ORDERS	O	14GB
NATION	N	112KB
LINEITEM	L	62GB
CUSTOMER	C	1.8GB

To evaluate the potential of the NVM block device in a real I/O-intensive application, the TPC-H schema is used in this study to derive an RDBMS (MySQL) with a scale factor of 60. After all the database tables had been loaded into storage, the database size increased to 89.1 GB. The size of each table is listed in Table 2.3. The initial setup on MySQL [3] is described below.

1. Database Storage Engine : We use Innodb as the default storage engine of MySQL. Compared with others, such as MyISAM, Innodb provides more functionality, including commit, roll-back, and crash recovery.

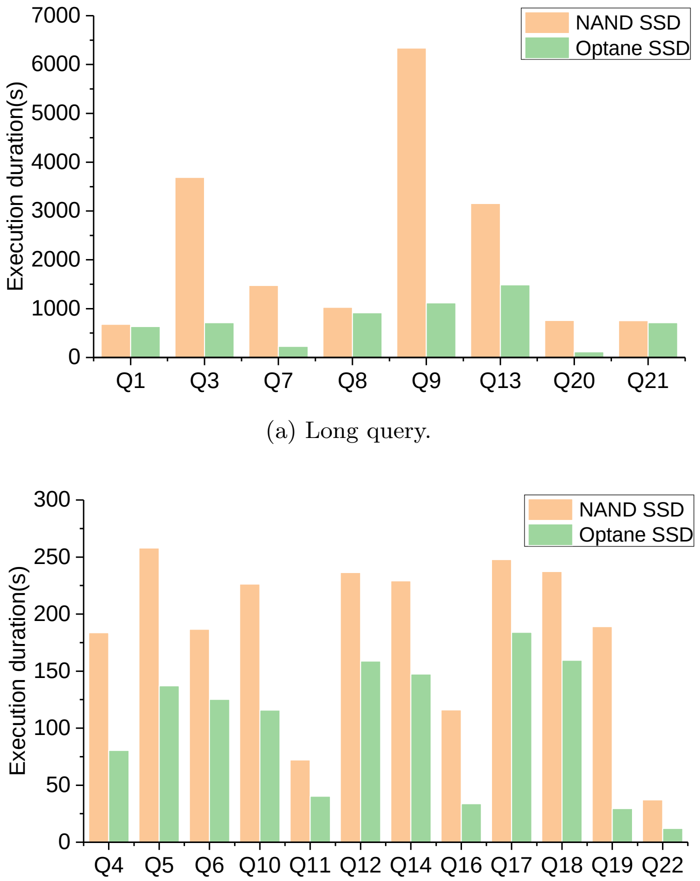
2. Buffer Pool Size: The buffer pool size is set to 4GB for caching database tables and indices in memory.

3. The Number of Innodb I/O Threads: The number of background threads used to perform I/O requests is set to the Innodb default value of 4.

2.6.2 TPC-H Performance Evaluation

In the experiment described in this section, the TPC-H queries are run on both NVM block device and NAND SSD. Their execution durations are shown in Figure 2.12. Here, we do not include Q2 and Q15's results since those two queries failed to complete during testing due to unknown reasons. From these results, the NVM block device achieves more than 3X higher performance than the NAND SSD for queries Q3, Q7, Q9, Q13, Q16, Q19, Q20, and Q22. For the rest of the queries, the NVM block device delivers 1.1X - 3X speedup compared to the NAND SSD. To understand the underlying reasons for these results, the block-level I/O requests of the TPC-H queries are captured by blktrace [138] and analyzed.

The first step is to study the access patterns of each query, including sequential read, random read, sequential write, and random write. We say a page is sequentially accessed, if it is read by database read-ahead mechanism. Otherwise, it is defined as random access. A page is the smallest data unit of MySQL, and the page size is 16KB. As shown in Table 2.4, all TPC-H queries are read-intensive. However, this observation is still not adequate to explain some queries' extremely long execution duration, such as Q9 with 6339s total execution duration in the NAND SSD. To find the reason for this, *the block-level repeat access rate* of each query is tested, where the block-level repeat access rate is defined to be the total number of accessed pages from the block level divided by the number of distinct pages accessed from the block level during querying. Both numbers can be counted from each query's block level trace. For instance, during a query, the block level accessed page sequence is p1, p3, p4, p3, p1, p4, p5, p3. Then the block-level repeat access rate is computed by the number of total accessed pages, 8, divided by the number of distinct accessed pages, 4 (p1, p3, p4, and p5), which equals 2. Table 2.5 presents the results. In the remainder of the chapter, we call the block-level repeat access rate the repeat access rate for simplicity.



(a) Long query.

(b) Short Query.

Figure 2.12: TPC-H queries Result. X-axis represents query name.

Summarizing the results from Tables 2.4 and 2.5 reveals that all TPC-H queries can be separated into three categories: 1) a query is randomly and repeatedly. Here, a query is considered to be randomly read if its random read percentage exceeds 50%.

Table 2.4: TPC-H access pattern. Note, there is no random write for all queries

	Q1	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
seq read	99%	6%	96%	91%	99%	5%	89%	5%	93%	66%
rand read	1%	94%	4%	9%	1%	95%	11%	95%	7%	34%
seq write	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Q12	Q13	Q14	Q16	Q17	Q18	Q19	Q20	Q21	Q22
seq read	99%	4%	96%	20%	24%	99%	13%	8%	99%	49%
rand read	1%	96%	4%	76%	76%	1%	87%	92%	1%	51%
seq write	0%	0%	0%	4%	0%	0%	0%	0%	0%	0%

Table 2.5: TPC-H repeat access rate

	Q1	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
RA rate	1.0	4.6	1.0	1.0	1.0	2.2	1.1	6.2	1.0	1.0
	Q12	Q13	Q14	Q16	Q17	Q18	Q19	Q20	Q21	Q22
RA rate	1.0	16.8	1.0	1.0	1.0	1.0	1.0	1.4	1.0	1.0

In the same way, a query is considered to be repeatedly accessed if its repeat access rate is greater than 1.1. These queries are Q3, Q7, Q9, Q13, and Q20. 2) A query is considered to be randomly, but not repeatedly read, including Q16, Q17, Q19, and Q22. 3) A query is sequentially, but not repeatedly read, including Q1, Q4, Q5, Q6, Q8, Q10, Q11, Q12, Q14, Q18, and Q21. By combining the results in Figure 2.12, it is apparent that for the first category of queries that are randomly and repeatedly accessed, processing them on a NAND SSD requires extremely long execution times on the order of thousands of seconds. There are several explanations for this result. First, the NAND SSD has around 10X lower random read performance than the NVM block device (Section 2.5.1) because randomly accessing the database files involves a huge amount of I/O time. Second, in the experiments performed in this study, four independent threads performed I/O requests concurrently, as discussed in Section 2.5.3, and this highly intensive I/O workload degraded NAND SSD performance tremendously because of the resource competition of its internal optimization firmware. Finally, as described in Section 2.5.6, accessing a given page repeatedly degrades read performance due to the read disturb issue. In addition, because this may also trigger NAND SSD internal garbage collection, foreground read performance is impacted further. For these reasons, executing the queries in category 1 on a NAND SSD results in extremely long I/O time, limiting overall database performance. In contrast, because the NVM block device benefits from a number of advanced technologies, including fast random read speed, absence of garbage collection, and low performance degradation during internal data refresh, processing these same queries on a NVM block device showed more than 3X performance speedup. Especially for Q7, the performance acceleration delivered by NVM block device can be as much as 6.5.

For the second category's queries that are randomly, but not repeatedly read, processing them on the NAND SSD also leads to long execution duration. However, because most pages are accessed only once during querying and because the NAND SSD has a low probability of triggering internal data refresh and garbage collection, the execution times for these queries, on the order of hundreds of seconds, are still acceptable compared with thousands of seconds of execution time for the first query category. In contrast, due to the order of magnitude better performance of the NVM block device on random read access patterns, executing these queries on a NVM block device provides more than 3X performance acceleration. However, Q17 is an exception; its performance is improved only 1.3X by the NVM block device. To find the reason for this, the I/O, 61.2 s, and computation time, 186.22 s, of Q17 are both recorded on the NAND SSD. The results indicated that Q17 is a computation-intensive query. Even though processing Q17 on the NVM block device achieved 5.7X speedup on I/O time, 10.8 s, because the computation time, 183.95 s, is still dominant, the overall performance speedup is limited.

Finally, for the third category's queries that are sequentially, but not repeatedly read, processing them on the NVM block device achieves only a 1.1 - 3X performance speedup. There are two explanations for these results. First, because the performance of these queries is mainly limited by the host computation time instead of the I/O latency [69], the I/O time speedup provided by NVM block device is weakened. Second, because only four I/O threads are invoked during the query and the NAND SSD's internal read mechanism could still schedule I/O jobs efficiently, the NAND SSD delivers similar, although slightly lower sequential read performance than the NVM block device. As a

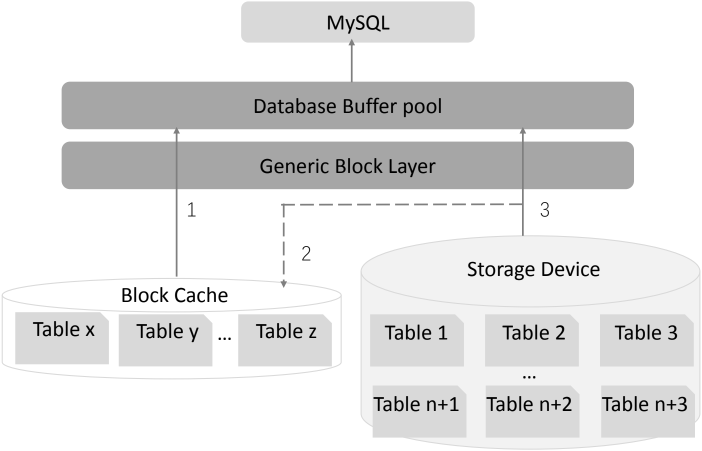


Figure 2.13: Hybrid system with table placement algorithm. Step 1: While processing a database query, the block cache is first consulted. If the target tables have been cached, read them from the block cache. Step 2: Otherwise, block the query, migrate target tables into the block cache, and resume the query. Step 3: If there is no free space on the block cache to place target tables, read target tables from the storage device.

result, the NVM block device achieves limited performance acceleration.

2.7 Investigations of the Hybrid Storage System for a Database Application

Although the NVM block device delivers substantial performance improvement, its price is much higher than that of conventional storage devices. In the real world, only a few NVM block devices may be available, which cannot used to store entire database tables, but many low-price NAND SSDs may be available. In this circumstance, designing a hybrid storage system may become a feasible approach to use existing storage devices efficiently. In this hybrid storage system, the fast storage device, NVM block device, is used as a block cache to buffer highly valuable tables or pages temporarily, and the slow storage device, conventional NAND SSD, is used as a persistent storage to keep all database tables.

Designing a hybrid system is not a new topic. A previous study [93] proposed a table

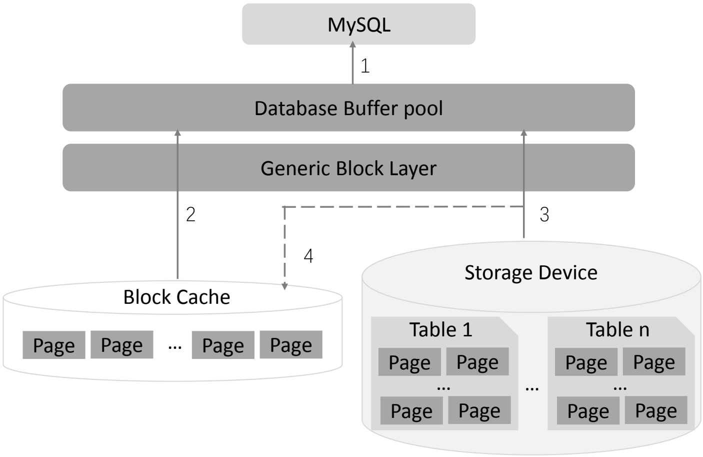


Figure 2.14: Hybrid system with caching algorithm. Step 1: While processing a database query, if the required page has been cached in the host, read it from the database buffer pool. Step 2: Otherwise, read the required page from the block cache. Step 3: If the required page is still not found, fetch it from the storage device. Step 4: Meanwhile, the required page has also been a cache candidate to be buffered into the block cache.

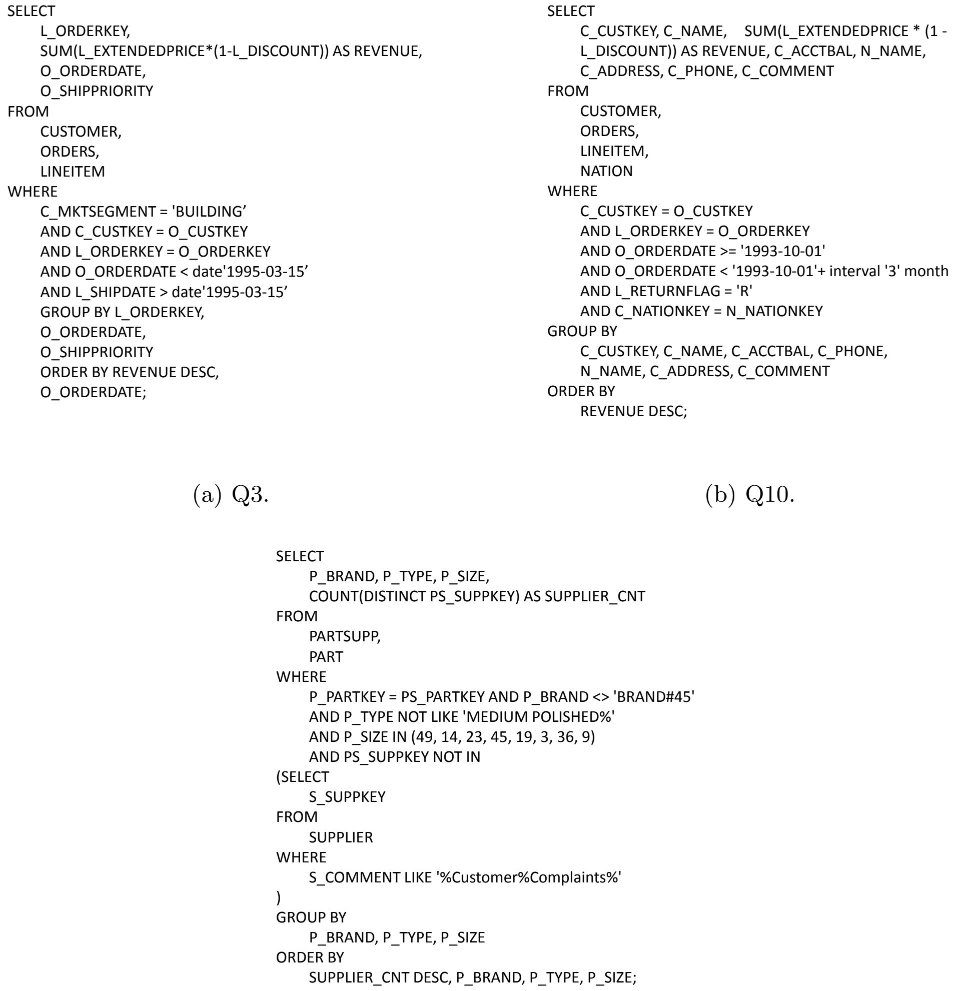
placement algorithm for a hybrid NAND SSD and HDD storage architecture, as shown in Figure 2.13. It assumed that a NAND SSD has much faster random read performance than a HDD, but that both provide the same performance for sequential read. By using information extracted from the database query execution plan tree, the above algorithm tends first to place tables that will be accessed by index into the NAND SSD and then process the query. However, the performance improvement of this algorithm is highly dependent on the prediction accuracy of the query's execution plan tree. For instance, a table that is predicted to be accessed by index may be sequentially scanned in a real application. Canim et al. [91] proposed a temperature aware caching (TAC) algorithm for hybrid NAND SSD and HDD system, its architecture is as presented in Figure 2.14. During database querying, through measuring the temperature of each region, TAC only admits the pages from the hot region. However, since it is possible that a rarely accessed page is from a hot region, TAC suffers miss-caching issues.

Although previous work has some drawbacks, they do provide some insights that prompts the designer to think about hybrid storage architecture. This section describes a performance investigation of a hybrid NVM block device and NAND SSD storage system for a database application with hybrid table replacement algorithm and hybrid caching algorithm, respectively. Proposing a proper algorithm is not the goal of this chapter; instead, the focus is on finding and evaluating the key factors that could impact the performance of this hybrid storage system and thereby provide some suggestions to researchers or engineers to help them to design a high-performance hybrid storage system.

2.7.1 Hybrid System with Table Placement Algorithm

This subsection investigates the performance impacts of table placement algorithm for the hybrid NVM block device and NAND SSD system. We will address the following three questions 1) how does the hybrid table placement algorithm affect the performance of the three types of query that were mentioned in Section 2.6, including sequentially read queries, randomly read queries, and randomly and repeatedly read queries? 2) is it always true that placing a larger table or using more NVM block device space can deliver higher performance for database queries? 3) what are the key factors that could impact a hybrid table placement algorithm's performance for database queries?

To answer these questions, one representative query is chosen from each query type, including Q10, a sequentially read query, Q16, a randomly read query, and Q3, a randomly and repeatedly read query. The clause of each query is shown in Figure 2.15. Moreover, to explore whether the query performance improvement is proportional to the NVM block device's utilization size, the tables required by each query are placed on the NVM block device with different combinations and their corresponding execution durations recorded. Figure 2.16 shows a plot of the experimental results, where the bottom x-axis represents the various table placement scenarios on the NVM block device. For instance, as in Figure 2.16a, "None" means that no table is stored in the NVM block device during the query, "C" represents that only the CUSTOMER table is stored in the NVM block device, "C+O" represents that the tables CUSTOMER and ORDERS are placed in the NVM block device, and "ALL" means that all three required tables, including CUSTOMER, ORDERS, and LINEITEM, are placed on the NVM block device. The top x-axis represents the space required to store the target



```
SELECT
      L_ORDERKEY,
      SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
      O_ORDERDATE,
      O_SHIPPRIORITY
FROM
      CUSTOMER,
      ORDERS,
      LINEITEM
WHERE
      C_MKTSEGMENT = 'BUILDING'
      AND C_CUSTKEY = O_CUSTKEY
      AND L_ORDERKEY = O_ORDERKEY
      AND O_ORDERDATE < date'1995-03-15'
      AND L_SHIPDATE > date'1995-03-15'
      GROUP BY L_ORDERKEY,
      O_ORDERDATE,
      O_SHIPPRIORITY
      ORDER BY REVENUE DESC,
      O_ORDERDATE;
```

(a) Q3.

```
SELECT
      C_CUSTKEY, C_NAME,    SUM(L_EXTENDEDPRICE * (1 -
      L_DISCOUNT)) AS REVENUE, C_ACCTBAL, N_NAME,
      C_ADDRESS, C_PHONE, C_COMMENT
FROM
      CUSTOMER,
      ORDERS,
      LINEITEM,
      NATION
WHERE
      C_CUSTKEY = O_CUSTKEY
      AND L_ORDERKEY = O_ORDERKEY
      AND O_ORDERDATE >= '1993-10-01'
      AND O_ORDERDATE < '1993-10-01'+ interval '3' month
      AND L_RETURNFLAG = 'R'
      AND C_NATIONKEY = N_NATIONKEY
GROUP BY
      C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE,
      N_NAME, C_ADDRESS, C_COMMENT
ORDER BY
      REVENUE DESC;
```

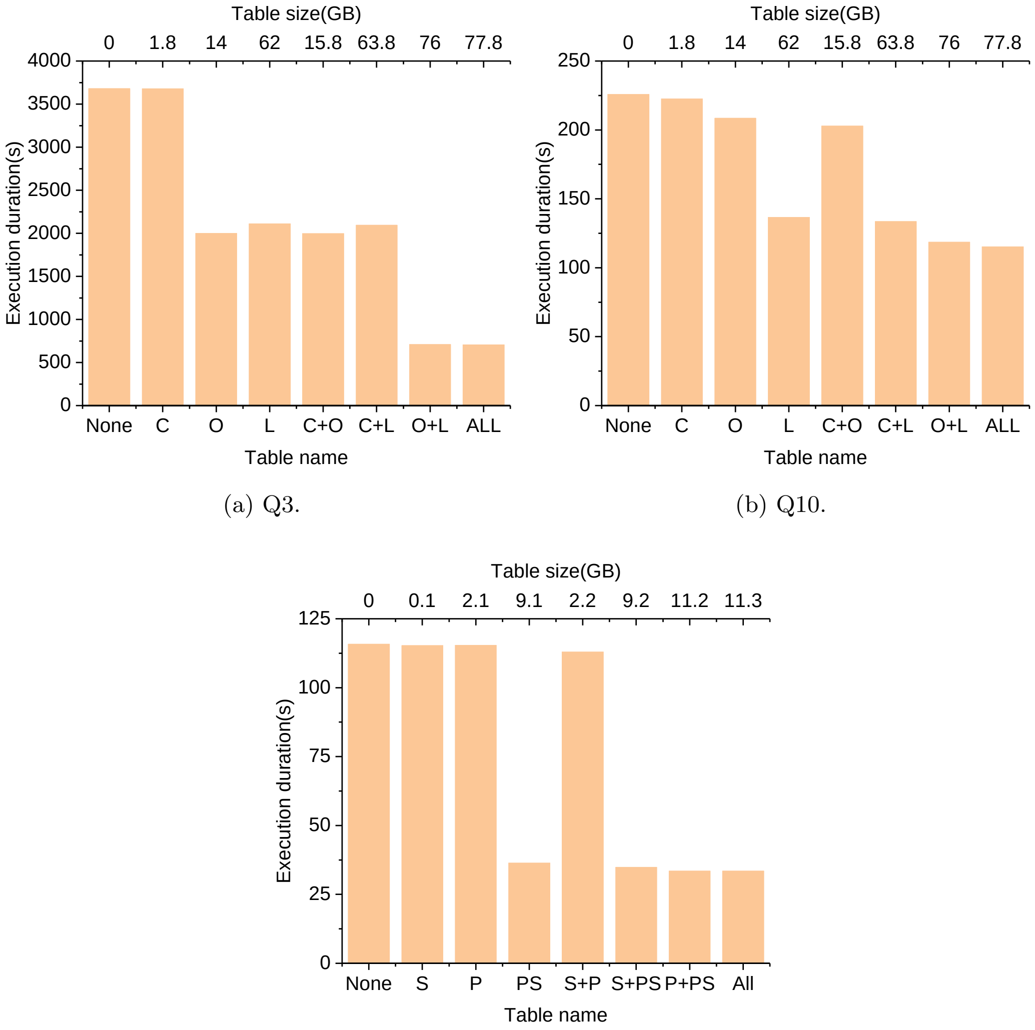
(b) Q10.

```
SELECT
      P_BRAND, P_TYPE, P_SIZE,
      COUNT(DISTINCT PS_SUPPKEY) AS SUPPLIER_CNT
FROM
      PARTSUPP,
      PART
WHERE
      P_PARTKEY = PS_PARTKEY AND P_BRAND <> 'BRAND#45'
      AND P_TYPE NOT LIKE 'MEDIUM POLISHED%'
      AND P_SIZE IN (49, 14, 23, 45, 19, 3, 36, 9)
      AND PS_SUPPKEY NOT IN
(SELECT
      S_SUPPKEY
FROM
      SUPPLIER
WHERE
      S_COMMENT LIKE '%Customer%Complaints%'
)
GROUP BY
      P_BRAND, P_TYPE, P_SIZE
ORDER BY
      SUPPLIER_CNT DESC, P_BRAND, P_TYPE, P_SIZE;
```

(c) Q16.

Figure 2.15: Query clause of Q3, Q10, and Q16.

tables on the NVM block device, in gigabytes. The y-axis records the execution duration of the various table placement combinations. Analyzing the results reveals that when the NVM block device's utilization space is certain, the performance improvement delivered by the hybrid table placement algorithm for each query type may be different from the conclusion drawn in the previous section. In Section 2.6, it is found that when the NVM block device is used as a storage device to store all database tables, the NVM block device delivers the highest performance accelerations for randomly



(a) Q3.

(b) Q10.

(c) Q16.

Figure 2.16: The execution duration for the hybrid table placement algorithm. Note, the abbreviation of each table's name and its corresponding size are listed in Table 2.3.

and repeatedly read queries, a relatively lower speedup for randomly read queries, and the lowest improvement for sequentially read queries. However, for the hybrid table placement algorithm, this conclusion may not always be true. For instance, when the NVM block device utilization size is set to 14 GB, the hybrid system delivers the highest performance acceleration, 3.4X, for randomly read query Q16 (in this scenario, all tables required by Q16 are stored in the NVM block device), a relatively lower speedup, 1.8X, to randomly and repeatedly read query Q3, and the lowest improvement, 1.1X, to sequentially read query Q10. In addition, it is noted that the performance acceleration provided by the hybrid table placement algorithm is not always proportional to the NVM block device's utilization size, especially for randomly and repeatedly read queries. For instance, as shown in Figure2.16a, the execution duration, 2119.38 *s*, of placing a larger table, LINEITEM (62 GB), on NVM block device for Q3 lags even behind the execution duration, 2010.04 *s*, of buffering a smaller table, ORDERS(14 GB). In other words, for the hybrid table placement algorithm, using a larger NVM block device space does not always provide performance gains.

Table 2.6: Characteristics of Q3

	Sequential read%	Random read%	Accessed data size (GB)	Repeat access rate
CUSTOMER	99.6%	0.4%	1.8	1
ORDERS	8.5%	91.5%	145.6	10.4
LINEITEM	2.0%	98.0%	138.5	2.2

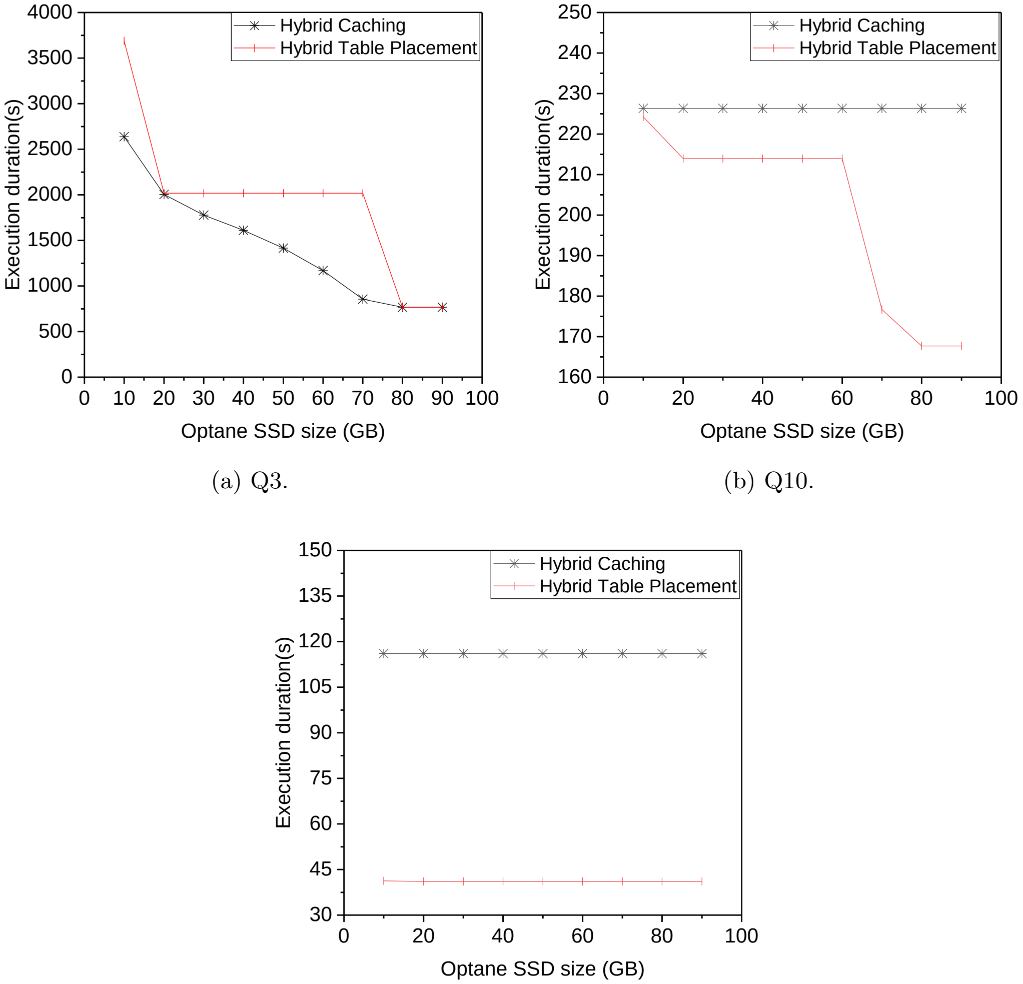
To find the reasons that underlie these results, an effort is made to explore more details for Q3. The sequential and random access percentages, the total accessed data size, and the repeat access rate are recorded for each table required by Q3 in Table 2.6. From the results, it is noted that Q3's performance is mainly limited by two randomly and repeatedly read tables, ORDERS and LINEITEM. During the query, 145 GB and 138.5 GB of data are randomly read from these two tables respectively. Even if one table, ORDER, is placed on the NVM block device, Q3's performance is still limited by randomly reading 138.5 GB of data from the LINEITEM table on the NAND SSD. As a result, when the NVM block device utilization space is set to 14 GB, the hybrid system delivered only 1.8X speedup for Q3, which is much less than its performance acceleration, 3.4X, on a randomly read query, Q10. On the other hand, because Q3 read more data from the ORDERS table, 145 GB, than from the LINEITEM table, 138.5 GB, placing the smaller table, ORDERS, on the NVM block device lead to greater performance improvement than buffering the larger LINEITEM table.

By summarizing these experimental results, it can be concluded that 1) the hybrid table placement algorithm may not always deliver the greatest performance improvement for randomly and repeatedly read queries compared with the other two types of query; 2) for the hybrid table placement algorithm, it is not always true that placing a larger table on SSD or using more NVM block device space can deliver higher performance to a database query; 3) hence, when designing a hybrid table placement algorithm for a database application, the impact of both the access patterns and the repeat access rates of the database tables must be carefully considered to deliver performance improvement.

2.7.2 Hybrid System With Caching Algorithm

We discuss in the previous subsection that for some database queries, such as Q3, the pages within the table would be repeatedly accessed during querying. Thus, admitting those pages that have temporal locality is expected to accelerate a query's performance once they are referenced again. In this subsection, we will study the following two questions: 1) how does the hybrid caching algorithm impact the performance of the three types queries, including sequentially read queries, randomly read queries, and randomly and repeatedly read queries? 2) compared with the hybrid table placement algorithm, does the hybrid caching algorithm always deliver better performance?

To address these questions, we use the same three representative queries as in the previous subsection. These queries include Q3, a randomly and repeatedly read query, Q10, a randomly read query, and Q16, a sequentially read query. The flashcache [89] is built on our system, where, the flashcache is a block cache component which uses fast storage device as a block cache for slow ones. The block cache is managed by an LRU-like caching algorithm. In the hybrid system, the application, MySQL, is running on the top. While I/O requests are sent into the block layer, flashcache remaps I/O into either block cache or storage devices based on its internal caching algorithm. In our experiment, we modify the original LRU-like caching algorithm to LRU, and use NVM block device as a block cache for NAND SSD. We record the execution duration of each representative query under different NVM block device sizes, from 10GB to 90GB, with granularity of 10GB. For comparison, the hybrid table placement algorithms' results for three representative queries are also shown. The experimental results are presented in Figure 2.17. The x-axis represents NVM block device size. The y-axis records the



(a) Q3.

(b) Q10.

(c) Q16.

Figure 2.17: The execution duration for the hybrid caching algorithm.

execution duration. Note here, for the hybrid table placement algorithm, since we know that it is not always true that placing a larger table on SSD or using more NVM block device space can deliver higher performance to a database query. In the experiment, while NVM block device size is fixed, we always record the best performance for different table combinations' results. For instance, for Q3 , while NVM block device size is set to 70GB, the different table combinations that can be placed into NVM block device are, table C, 1.3GB, table O, 14GB, table L, 62GB, tables C+O, 15.8GB, and tables C+L, 63.8GB. Placing table C+O delivers the best performance, 2006.7 sec, compared with others. When NVM block device size is set to 70GB, the execution duration of hybrid table placement algorithm for Q3 is 2006.7sec. Moreover, the table migration time that migrating target table(s) into NVM block device for hybrid table placement algorithm is also included into its overall performance.

As Figure 2.17 illustrated, for the randomly and repeatedly read query, Q3, while NVM block device size is between 10GB and 80GB, the hybrid caching algorithm delivers better performance than hybrid table placement algorithm. This performance gap is continuously increasing as we increase the NVM block device size from 20GB to 70GB. This is because a table is the smallest caching unit for hybrid table placement algorithm. While the available NVM block device space is less a target table's size, since target table cannot be temporally cached, the hybrid table placement algorithm is not able to deliver any performance gains. This performance issue is reflected by a flat line in Figure 2.17a when the NVM block device size is set to 20 to 70GB. In contrast, for the hybrid caching algorithm, since a page is the smallest caching unit, the larger NVM block device size allows more repeatedly accessed pages to be cached, and, as a results, more performance gains are delivered for Q3. As we continuously increase the NVM block device size to 80GB, since all required tables of Q3 can be temporally cached, the hybrid table placement algorithm provides similar performance to the hybrid caching algorithm.

For Q10, a randomly read query, and Q16, a sequentially read query, since all pages are only accessed once during querying, no performance gains are provided by the hybrid caching algorithm. On the contrary, we observe significant performance improvement of hybrid table placement algorithm for Q10 and Q16. There are two reasons to explain the above results. First, NVM block device delivers much higher performance than NAND SSD when processing multiple I/O threads, as discussed in Section 2.5.3. Second, because of read-ahead mechanism, NAND SSD provides similar performance as NVM block device for single thread sequential read operation, as discussed in Section 2.5.1. Thus, the extra table migration time consumed by the hybrid table placement algorithm is relatively small compared to its performance gains. Due to the above two reasons, the hybrid table placement algorithm accelerates both randomly read and sequentially read queries performance significantly. After we increase the NVM block device size to

20GB, since all required tables of Q16 has been temporarily buffered, the hybrid table placement algorithm's performance is not changed anymore as we continued to increase the NVM block device size.

Summarizing our experimental results, it can be concluded that 1) the hybrid caching algorithm does not always outperform the hybrid table placement algorithm, the performance gains provided by two types hybrid algorithms are highly dependent on the query's type, 2) hence, when designing a hybrid system for a specific database query workload the queries' characteristics must be carefully considered to decide which hybrid algorithm should be employed.

2.8 Conclusions

In this study, through intensive experiments and measurements, it is found that: 1) by using a similar internal architecture, NVM block device exhibits the same performance trend as NAND SSD: either larger request size or lager queue depth can increase NVM block device's performance due its internal parallelism. On the other hand, benefits from employing the high performance non-volatile memory media, NVM block device delivers substantial performance improvement compared with conventional NAND flash based SSD. This performance advantage allows NVM block device to provide a bridge between slower storage devices and high-performance DRAM; 2) even though NVM is a byte-addressable memory, because it is integrated into a block-level storage device, any request submitted to NVM block device must be aligned with a page size to avoid performance degradation; 3) because NVM is byte-addressable and supports data update-in-place, some of the common problems that typically exist within the conventional NAND SSD, including read-modify-write and write-driven garbage collection, have been removed from NVM block device. Thus, designing a hybrid storage system using NVM block device as a write buffer for NAND SSD to improve NAND SSD's performance and lifespan could become a potential research direction in the future; 4) Read disturb issue is one of the key factors that contributes to the long tail latency in the NVM block device. For modern applications which have strict requirements on I/O latency distribution, how to mitigate NVM block device's performance variation due to its read disturb issue is a potentially significant research direction; 5) when designing the hybrid NVM block device and NAND SSD storage system for a database application, the characteristics of database query workload are key factors that must be considered carefully to decide which hybrid algorithm should be used to deliver better performance: either hybrid table placement algorithm or hybrid caching algorithm. In general, the present research provides a new insight into the NVM block device and studies its implications for a real database system. It is hoped that these experimental results can inspire researchers in OS design and system optimization and help them rethink their storage hierarchy designs in their further research.

Chapter 3

HeuristicDB: A Hybrid Storage Database System Using a Non-Volatile Memory Block Device

3.1 Introduction

A relational database management system (RDBMS) is a type of database based on the relation model. Compared with a NoSQL database, an RDBMS has several advantages, such as being ACID-compliant [65] for transactions, using a standardized SQL language, and supporting complex query functions. Because of these advantages, RDBMS technology is prevalent and will co-exist with NoSQL databases for a long time. However, RDBMS performance is usually limited by laggard storage devices [66, 67, 68]. Chapter 2 shows that if an NVM block device is used as persistent storage for RDBMS, I/O time is no longer a bottleneck in many cases. However, this is an expensive solution. Because of the high cost of NVM block devices, it might be impossible to provide an adequate number to preserve the entire database in real time. In this circumstance, building a hybrid storage database system that uses an NVM block device as a block cache for conventional storage devices becomes a cost-efficient way to improve RDBMS performance.

Directly employing such conventional replacement caching algorithms as the LRU, MRU, LFU, and ARC [75] in a hybrid database system may not be suitable because these algorithms are designed for the first level database buffer pool caching. In a database system, because all required pages must be read into the buffer pool for processing, admitting and buffering the accessed pages does not incur extra cost. On the contrary, in a hybrid storage architecture, as a page is read from the storage device into the database buffer pool, it is a candidate for admission into the second level block cache device. Admitting a page that will not be accessed in the near future not only pollutes the limited block cache space, but also consumes unnecessary power for data migration. This leads conventional replacement algorithms designed for the first level database buffer pool caching to be able to deliver only suboptimal solutions for hybrid storage systems.

In addition, multiple factors limit previous hybrid storage algorithms [91, 139, 92, 93] to provide optimal performance for NVM block device. First, the NVM block device exhibits various performance-related properties [26, 25, 53], such as much higher read-write bandwidth, the elimination of write-driven garbage collection [51, 52] compared to NAND solid-state drives (SSDs). Direct use of prior hybrid storage schemes [91, 139, 92, 93] that use a NAND SSD as a block cache for the hard-disk drive (HDD) and only admit random requests into NAND SSD cannot yield the best performance for NVM block devices. Second, ignoring the unique characteristics of database block requests made earlier systems [95, 91, 139, 92, 93] deliver limited performance improvement and involved migration of many non-reusable pages from storage devices to the block cache. We provide a more detailed discussion of earlier studies and their limitations in the next section. Because NVM block devices still suffer from the problem of endurance [140, 141, 142, 143], frequently admitting pages that are then not reused causes the NVM block device to reach its lifespan early without delivering any performance acceleration. Finally, without an efficient demotion mechanism for cached pages, prior studies suffered severe performance degradation every time the database workload access pattern changed.

The limitations of prior studies motivated the authors to develop a new hybrid storage management system, called HeuristicDB, which uses an NVM block device as an extension of the database buffer pool. By considering the intrinsic performance behavior of the NVM block device and the characteristics of database block requests, a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority are proposed to guide HeuristicDB in managing database block requests more efficiently. To support the system implementation of the proposed rules, four programs, including *a query profile queue*, *an eviction and demotion (EV) program*, *a table access pattern detector*, and *a page placement controller* have been developed. The table access pattern detector prioritizes each table or index required by a query by monitoring its access behavior. The page placement controller is used to redirect each database block request either to the NVM block device or to storage based on both the reusability of the accessed page and the priority of the table or index to which the page belonged. Finally, to maintain high performance under changing database query access patterns, a low-overhead query profile queue and an EV program are developed and used to adaptively adjust the priorities of both the database tables and indices and the cached pages using the profiles of past and newly submitted queries. In the rest of the chapter, for consistency, the NVM block device inside the system is referred to as the NVM buffer pool.

HeuristicDB is empirically assessed under different circumstances using an online analytical processing (OLAP) benchmark, TPC-H [107], and an transactional processing (OLTP) benchmark, TPC-E [144]. Both trace-based examination and system implementation are conducted to test the cache hit ratio, data migration size between storage and the NVM buffer pool, execution time, and transaction rate of HeuristicDB and six other baseline algorithms. The experimental results show that HeuristicDB provides up to 75% higher performance and migrates 18X fewer data between storage and the NVM block device than existing systems.

The remainder of this chapter is organized as follows. Section 3.2 provides background information about NVM block device and discusses the limitations of past research in this area. Section 3.3 explains explains the system design principles and presents a set of heuristic rules that associate database (block) requests with the appropriate service quality. Section 3.4.2 gives an overview of HeuristicDB. Section 3.5 discusses detailed system implementation. A trace based evaluation and a real system performance evaluation are conducted at Section 3.6 and Section 3.7. Section 5.1 draws conclusions.

3.2 Backgrounds and Related Work

This section presents background information on the NVM block device and the hybrid storage database system.

3.2.1 NVM block device

With DRAM-like performance and disk-like persistency and capacity, non-volatile memory such as phase-change memory (PCM) [28, 29, 30, 31], conductive bridging RAM (CBRAM) [32, 33, 34], and 3D Xpoint [25, 26, 27] is becoming critical importance for various data-intensive applications. The NVM serves as either main memory through a memory bus [145], known as an NVM DIMM [146], or persistent storage through a PCIe interface [147], known as an NVM block device [140]. The NVM block device has several advantages. First, compared with conventional NAND SSDs, the NVM block device delivers orders of magnitude better performance on both read and write operations [26]. Second, because NVM supports in-place update, there is no need for garbage collection [51, 52, 26] which usually causes significant performance degradation for the NAND SSD. Third, the NVM block device inherits the PCIe interface developed for the NAND SSD and can be directly controlled by all existing systems without requiring any hardware updates. In contrast, the NVM DIMM works only when a specific processor is provided [53, 54, 55]. Considering all these advantages, this work has focused on using an NVM block device in a hybrid storage database system.

3.2.2 Related Work & Limitation

Many studies have investigated hybrid storage architectures in the past [85, 86, 87, 88, 89, 90]. Designing a hybrid storage system with an appropriate replacement algorithm to accelerate database applications is an important research topic. Zhou et al. [95] presented a multiple-queue (MQ) algorithm, which maintained multiple LRU queues to keep cached pages with different priorities to manage the second-level cache. A cached page was demoted once it had not been accessed after expiredTime. However, an inappropriate setting of expiredTime can significantly influence MQ performance.

Canim et al. [91] developed a temperature-aware caching (TAC) algorithm that used a NAND SSD as an extension of the database buffer pool. During query processing, only randomly accessed pages from hot regions are admitted. To avoid a poor cache hit ratio when the database access pattern changes, TAC demotes the priorities of cached pages after a certain number of accesses. However, its performance is affected by multiple tunable parameters, such as the region size, the band size, and the number of reset accesses. In addition, because pages within the same region have different access frequencies, a rarely accessed page inside a hot region may be mistakenly cached into the NAND SSD, resulting in performance degradation. Do et al. [139] proposed a lazy cleaning (LC) method for dealing with pages exited from the database buffer pool. The LC method manages the block cache device using the LRU-2 [148] replacement algorithm. However, because the LRU-2 algorithm's performance is highly correlated with reuse distance [95], the LC method delivers even worse performance than TAC in most cases.

Developing an appropriate table placement algorithm is another approach to hybrid database system design. Canim et al. [92] proposed a replacement advisor for a database engine. By profiling database workloads, the tables or indices that generate the most random I/Os are placed in high-performance storage to yield the maximum performance gains from the NAND SSD. However, because replacement decisions are made based on the granularity of an entire database table or index, once the size of the target becomes larger than the capacity of the block cache device, the hybrid storage system cannot deliver any more performance gain. Similarly, Luo et al. [93] introduced a heterogeneity-aware data management policy. By extracting information from the database query execution plan tree, tables that are accessed by the indices are placed in the NAND SSD in advance. This design suffers from the same issue as discussed above. In addition, because the information provided by the query execution plan tree cannot always accurately predict the access pattern of database tables during a query, the heterogeneity-aware data management system encounters misplacement issues.

Several hybrid storage solutions have been released. Srinivasan et al. developed a hybrid kernel module called Flashcache [89], where the high-end storage device is employed as an extension of the host DRAM and managed by a FIFO/LRU-like replacement policy. The TeraData Virtual Storage System [90], a data placement solution, continuously monitors the temperature of the accessed data and migrates hot data to a high-performance storage device. However, such products are designed for general applications, and thus bring about a limited acceleration in performance for database systems.

Recently, multiple approaches for DRAM-NVM-NAND SSD hybrid systems have been proposed [149, 150]. In these systems, the NVM serves as a main memory, and the database application can directly access the data inside it without copying it into the DRAM. Those schemes were evaluated using NVM emulators. However, a recent study [55] has shown that the direct access performance of a real NVM device drops significantly as the thread count increases. As a result, the prior DRAM-NVM-NAND SSD hybrid system could encounter a performance degradation for multi-threads applications such as RDBMS. In contrast, our work employs an NVM block device as an extension of the database buffer pool, and the data within it must be copied to the DRAM for producing query results to deliver scalable performance. Thus, our work is more suitable for multi-threaded database applications.

This study has shown that by considering the characteristics of database block requests, the proposed HeuristicDB system can overcome the limitations of earlier approaches.

3.3 Design Principles & Heuristic Rules

This section first considers the characteristics of database block requests. This provides a high-level overview of how a database query is executed from the perspective of the storage layer. Based on an understanding of these characteristics, the design principles of HeuristicDB can then be explained, and a set of heuristic rules that associate database (block) requests with the appropriate quality of service can then be presented.

3.3.1 Characteristics of Database Block Requests

To determine caching priorities, this subsection examines the characteristics of database block requests from the following two aspects.

File Type: Three major file types are considered, including regular table, index, and page. A *table* is a collection of data elements. It consumes most of database storage capacity. An *index* is a data structure [151, 152], usually a B+tree, that is developed to retrieve data quickly without accessing all data elements inside a table based on a subpart of the data called a key [65]. Each item inside the index contains a key and that key's block address in storage. A *page* is the smallest unit of the storage engine that is accessible inside a table or index. The page size is a multiple of 4KB.

Access Behavior: This refers to the access pattern of a table or index during query processing. After a query has been optimized [151, 153, 154] and sent into the execution engine, the access pattern of each required table or index is determined. The possibilities include: (1) *sequential access*: when no index is used for a query, the database engine sequentially scans all data elements inside the corresponding table to fetch the matching data; and (2) *random access*: when an index is provided, the database engine locates the position to search and then retrieves the matching data elements from the table using the address inside the corresponding index. Both index and table are accessed randomly; and (3) *repeated access*: when multiple operators access a table or index during a query, some pages inside that specific table or index are accessed repeatedly [69]. This case can usually be observed during execution of complex OLAP queries [26, 69]. In addition, a table or index can also be repeatedly accessed while multiple queries are running concurrently and require the same table or index.

3.3.2 System Design Principles

The objective of this study is to leverage the NVM block device as an extension of the database buffer pool with an appropriate management strategy to boost database application performance while migrating less data between storage and the NVM buffer pool. By considering the characteristics of database block requests and the intrinsic performance behaviors of the NVM block device, the goal is approached from the following three directions.

Guarantee reusability: For the hybrid storage system, a performance gain can be delivered only when a hit occurs on the NVM buffer pool. Hence, the reusability of an accessed page is the main factor to consider for page admission. In this work, the tables and indices required by a query or queries are prioritized based on their reusability. Only pages from a high-priority table or index are admitted into the NVM buffer pool. A table or index is defined as reused either when some pages inside that table or index are re-accessed during query execution or when that table or index is also required by subsequent queries.

Adaptive demotion or eviction is required: How long a page stays in the NVM buffer pool is another key factor affecting the performance of a hybrid storage system. For a database application, the pages inside a table or index have a chance to be accessed only when that specific table or index is required by a query. Because the tables or indices required by the previous query may no longer be required by the subsequent query, and the set of hot pages inside the same table or index varies as the queries change. Keeping (hot) pages required by the previous query but are rarely accessed by the new incoming query in the NVM buffer pool reduces the performance of a hybrid storage database system. Hence, it is reasonable to adaptively adjust the priorities of database tables, indices, and cached pages before running a new query.

Performance benefit for caching a page in the NVM buffer pool: The NVM block device provides unprecedented performance enhancement on both read and write operations and eliminates the write-driven garbage collection compared with conventional storage devices such as NAND SSD and HDD. Hence, it is preferable to cache all block requests, including sequential read, random read, sequential write, and random write, with high reuse probability in the NVM buffer pool to deliver performance gains.

3.3.3 Heuristic Rules for a Query

Based on the characteristics of database block requests and the design principles discussed above, a set of heuristic rules is presented that associate database (block) requests with the appropriate quality of service for the purpose of caching priority. This subsection considers priority assignments during the execution of a single query. Priority assignments for concurrent queries will be discussed in the next subsection.

Sequential Requests

When a table is accessed sequentially, because all pages inside that specific table are accessed once, caching that mass of accessed pages delivers zero performance benefit. Instead, it pollutes the limited space in the NVM buffer pool and causes unnecessary data migration between storage devices and the NVM buffer pool.

RULE 1: *A page belonging to a sequentially accessed table is not qualified for admission into the NVM buffer pool if it is read from storage.*

Random Requests

When a table or index is randomly accessed, only matching pages belonging to the corresponding table or index are fetched from storage. Because the number of these matched pages usually is extremely small, the cache pollution issues caused by admitting those matched pages are limited. On the contrary, those pages can deliver performance benefits once they are reused in subsequent queries.

RULE 2: *Pages from a randomly accessed table or index are qualified for admission into the NVM buffer pool.*

Repeated Requests

When multiple operators access a table or index, some pages inside that table or index will be accessed more than once during query execution [26, 69]. Identifying and preserving those pages that are accessed more than once in the NVM buffer pool guarantees performance gains.

RULE 3: *If a table or index is repeatedly accessed, it is preferable to cache the pages inside that table or index.*

To maximize performance gains and avoid non-beneficial page replacement, the pages to be cached from a repeatedly accessed table or index are discriminated based on their access frequency.

RULE 4: *During query execution, page replacement happens unless the temperature of the accessed page is higher than the coldest one inside the NVM buffer pool.*

This design guarantees that the pages in the NVM buffer pool are the most profitable ones during a query. Moreover, it avoids unnecessary and frequent page replacements that consume substantial power and shorten the lifespan of the NVM block device.

Write Requests

During query execution, a block write request(s) is generated when a temporary table is created or an UPDATE/INSERT operation is required.

A temporary table is a special type of table for storing intermediate data sets generated during query execution. The pages within a temporary table will be read later to produce the final query results. The database engine deletes a temporary table automatically once the query is completed. Considering its reusability, in the proposed design, a temporary table is placed in the NVM buffer when it is created.

UPDATE and INSERT are the two most frequently used operations in OLTP workloads. They are used to renew existing content and add new data elements from or into a specified table or index. Considering the unprecedented write performance and the elimination of write-driven garbage collection of the NVM block device, in the proposed design, both updated and inserted pages are written into the NVM buffer pool first and are flushed into the storage device asynchronously later.

Based on this understanding, the rule for database write requests can be presented.

RULE 5: *All write requests generated during a query are placed in the NVM buffer pool and flushed into storage asynchronously later if needed.*

New Query Requests

The tables or indices required and the access behavior of each table or index vary as queries change. To admit incoming hot pages to deliver performance acceleration, demotion or eviction of cached pages is required when a new query is submitted to the database server. The specific action depends on two factors: (1) the performance gains that have already been delivered from cached pages in the past; and (2) the existence of hot tables or indices that are required by both the previous and the newly submitted queries.

RULE 6:

1. When a new query is submitted, an eviction of cached pages is required if no performance benefit has been provided in the past from cached pages and if one or more hot tables or indices are required by both the previous and the newly submitted queries. Otherwise, the cached pages are demoted.

2. If an eviction is preferred, pages inside each hot table or index are allowed to admitted into the NVM buffer pool during execution of the new query, regardless of their access behavior.

The goal of eviction is to admit pages from one or more high-reusability tables or indices by evicting pages that do not provide a performance gain inside the NVM buffer pool. Hence, in the proposed system, eviction can be triggered only when two conditions are satisfied: (1) no performance benefit has been provided by cached pages in the past; and (2) there exist one or more hot tables or indices that provide clues about tables or indices with high reusability.

3.3.4 Heuristic Rules for Concurrent Queries

The scenario of concurrent queries happens while a new query is submitted when other query(s) is running. Then, those queries are co-running for a while. From the perspective of the storage layer, the block level characteristics of concurrent queries are similar as the single query's, both of them are accessing the required tables or indices by sending block requests to storage. Hence, the priority assignment rule for concurrent queries is much like those used in a single query execution.

RULE 7:

1. When a query is submitted while one or more other queries are still running, a priority adjustment of the cached pages is required. The detailed criteria are the same as in Rule 6.

2. The access behavior of a table or index may change after a new query is submitted and is running concurrently with others. For instance, a table may be sequentially and randomly accessed by two separate queries running independently. This means that the table may be repeatedly accessed while those two queries are running concurrently. Hence, an access behavior re-verification is required for all tables and indices required by concurrent queries.

3. When multiple queries run concurrently, the quality of service for database block requests follows Rules 1, 2, 3, 4, and 5.

3.4 HeuristicDB Overview

In this section, we provide an overview of HeuristicDB. To support the implementation of the heuristic rules in a real system, four programs, include *a query profile queue*, *an eviction/demotion (ED) program*, *an access pattern detector*, and *a page placement controller*, are developed inside HeuristicDB. Where, the query profile queue and the ED program are used to adjust the priorities of cached pages and detect the hot table(s)/index(es) while a query is submitted into a database server by following Rule 6 and 7. The access pattern detector probes the access behavior of required table(s)/index(es) in the execution of a query(s). Then, using those detected information, the page placement controller redirects database block requests into either NVM buffer pool or storage following Rule 1,2,3,4,5.

3.4.1 Metadata Management

In the system, several hash tables are created in the memory, as shown in the left part of Figure 3.1, to facilitate page placement during execution of a query(s).

A cache lookup table: It is used to help the lookup of cached pages. Each item inside this cache lookup table is defined as ⟨LBA, NVMLBA⟩. Where LBA is the logical block address of a cached page on the storage. It is used as a key. The value, NVMLBA, is the logical block address of the cached page on the NVM buffer pool.

Counting table: the counting table is included in the proposed system to avoid recounting the access frequency of a hot page once it has been evicted from the NVM buffer pool. It records the access frequency of all pages accessed during query processing. Each item in the table is defined as ⟨LBA, freq⟩.

Status map: this is used to indicate the access behavior of each table or index within a database. An item within this hashmap is defined as ⟨T, Status⟩, where T is a table or index name. During query processing, the status of each table or index is determined as one and only one of the following: *unknown*, *sequential*, *random*, *repeated*, and *hot*.

Profile table: This table is used to facilitate system lookup of items belonging to an accessed page. Each item within this profile table is defined as ⟨T, V⟩, where T is a

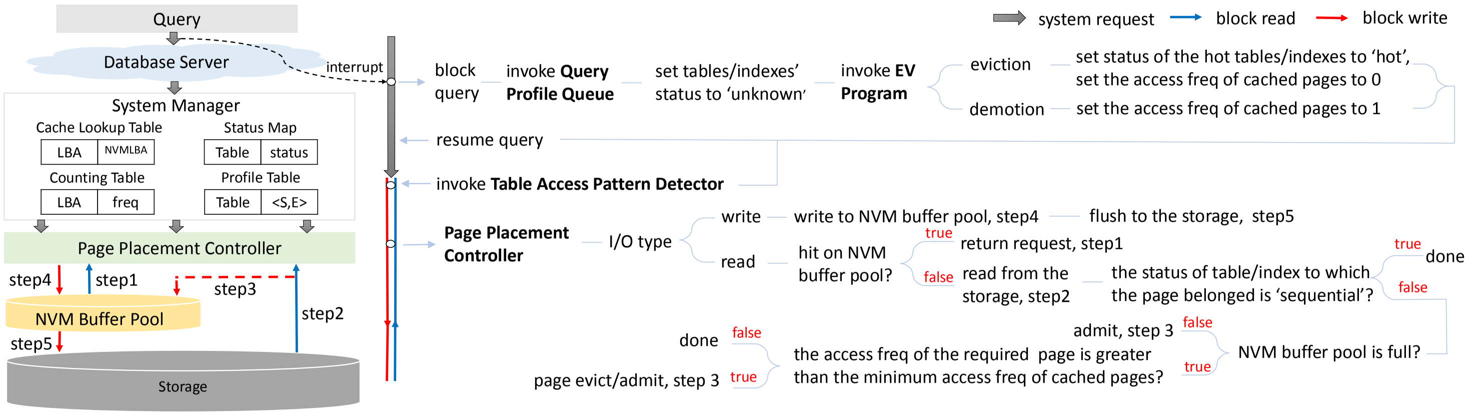

system request
block read
block write
Query
Database Server
interrupt
System Manager
Cache Lookup Table
LBA
NVMLBA
Status Map
Table
status
Counting Table
LBA
freq
Profile Table
Table
<S,E>
Page Placement Controller
step4
step1
step3
step2
NVM Buffer Pool
step5
Storage
block query
invoke Query Profile Queue
set tables/indexes' status to 'unknown'
invoke EV Program
eviction
set status of the hot tables/indexes to 'hot', set the access freq of cached pages to 0
demotion
set the access freq of cached pages to 1
resume query
invoke Table Access Pattern Detector
Page Placement Controller
I/O type
write
write to NVM buffer pool, step4
flush to the storage, step5
read
hit on NVM buffer pool?
true
return request, step1
false
read from the storage, step2
the status of table/index to which the page belonged is 'sequential'?
true
done
false
NVM buffer pool is full?
false
admit, step 3
true
the access freq of the required page is greater than the minimum access freq of cached pages?
false
done
true
page evict/admit, step 3

Figure 3.1: HeuristicDB overview. The left part shows the internal architecture of HeuristicDB, and the right part explains its working processes.

table or index name and V contains two tuples, ⟨startLBA, endLBA⟩. These two tuples represent the starting and ending logical block addresses of the corresponding table or index in storage. Note that when content is removed from or added to a table or index because of a database DELETE/INSERT operation, the address of that content should also be updated consequently.

3.4.2 System Overview

The left part of Fig. 3.1 shows the architecture of HeuristicDB. In the system, a database application is running on top. The NVM block device is used as an extension of the database buffer pool. All database tables and indices are preserved permanently in storage.

We present the working processes of HeuristicDB in the right part of Figure 3.1. Every time while a query is submitted into a database server, an interrupt signal is generated and is sent to HeuristicDB to invoke its *query profile queue* and *EV program*. Through evaluating the performance benefit that has been delivered from the cached pages and detecting the existence of a hot table(s)/index(es) which is required by both the past and the newly submitted queries, multiple further operations are triggered. If no performance benefit is delivered in the past from the cached pages and there exits a hot table(es)/index(es), then all cached pages are set as eviction candidate by resetting their access frequency to zero. Meanwhile, the status of the table(s)/index(es) required by those two consecutive queries is set to *hot* and those of the remaining tables/index(es) required by the newly submitted query are set to *unknown*. Otherwise, a demotion operation is required to reset the access frequency of all cached pages to one. Meanwhile the status of all tables/indexes required by the newly submitted query is set to *unknown*. In our system, we say a performance benefit is delivered if the number of cache hit since the last query has run is greater than zero. Once the query profile queue and the EV program is terminated, query processing is resumed. At the same time, the *table access pattern detector* is triggered to run asynchronously to detect the access behavior of each required table/index from the block layer. If the status of a required table/index is *unknown*, the detector continuously monitors it access behavior and set its status to any one of the follows three status, include *random*, *sequential*, and *repeated*, once made a decision. After all table's status are set, the access pattern detector is suspended.

In HeuristicDB, *the page placement controller* is assigned to redirect database block requests to their proper destination by following heuristic rules. When a database block request is generated, its corresponding access frequency is updated in *the counting table* first to reflect its temperature. For a block write request, considering the performance benefit of the NVM buffer pool, it is preferably written to the block cache, as step 4 in Fig. 3.1, and then lazily updated to the lower-performance storage device asynchronously later, as step 5. For a block read request, the NVM buffer pool is queried first. If the required page has been cached, the page placement controller returns the request immediately, as step 1. Otherwise, a cache miss has occurred, and the required page must be fetched from storage, as step 2. Meanwhile, by combining the collected information, the page is assessed to determine whether it is worth caching in the NVM buffer pool. If the NVM buffer pool is not full, the page can be admitted only if the status of the table or index to which this page belonged is not *sequential.* Otherwise, in addition to the above condition, the access frequency of the required page must be higher than the minimum access frequency for cached pages for it to be admitted into the NVM buffer pool, as in step 3.

3.5 System Implementations

In this section, we present the detailed system implementations of our proposed four programs within the HeuristicDB.

3.5.1 Query Profile Queue

The query profile queue is developed to analyze the profiles of past and newly submitted queries and to send either an eviction or a demotion request to the EV program according to Rules 6 and 7.

A queue data structure, as shown in Fig. 3.2, forms part of the proposed system. Each item in the queue contains two main types of variables that are used for query profiling: (1) a table linked list, which is used to record the tables or indices required by the corresponding query; and (2) a performance variable, which is called hits and

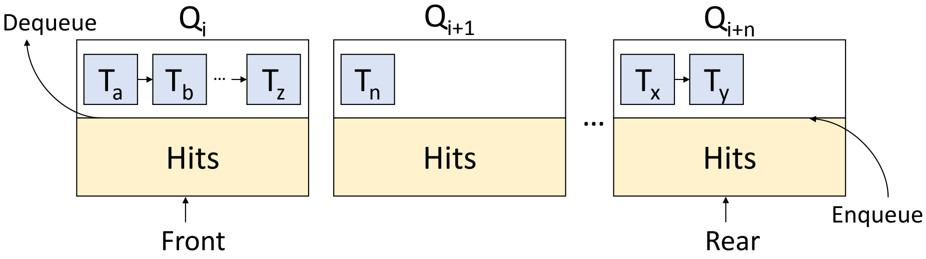


Figure 3.2: Query profile queue.

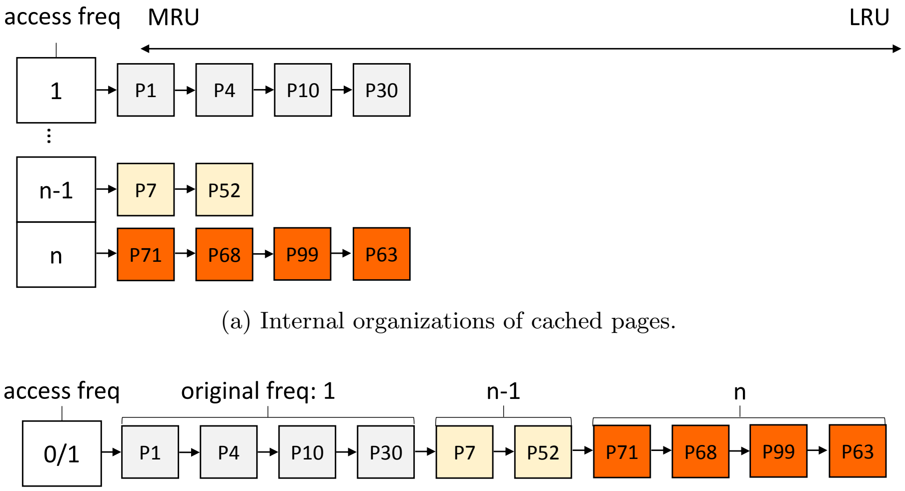
is used to record the number of hits since the query has run. It is increased by one whenever a hit happens during query execution.

Every time that a query is submitted to the database server, a query profile block is inserted at the rear of the queue. The initial value of the performance variable, hits, is set to zero. Then the profile information of the previous query is extracted from one block ahead of the rear of the queue. If the value of the performance variable inside that block is zero and there exists a table or index that is required by both the previous and the newly submitted query, an eviction request is generated, and the status of the specific table or index required by both queries is set to *hot*. Otherwise, a demotion request is generated. Because only information about the previous and newly submitted queries is needed, in the proposed system, the size of the profile queue is set to two. In addition, because multiple queries can be running concurrently for a certain time, the number of cache hits during that time is counted in the profile block of the last submitted query.

3.5.2 Eviction and Demotion Program

The EV program is triggered after an eviction or a demotion request is received. Before explaining its working mechanism, this subsection provides a detailed discussion of how cached pages are organized inside the NVM buffer pool.

In HeuristicDB, accessed pages are prioritized based on their access frequency. To efficiently distinguish the hotness of each page and evict the coldest one from the NVM buffer pool when a replacement is required, a least frequently used (LFU) data structure is used. In the proposed system, all cached pages are managed by a hash table, as shown in Fig. 3.3a. Each item within this hash table is defined as ⟨access freq, ⟨linked list⟩⟩. The access freq is used as a key to record the access count. Pages with the same access count are assigned to the same linked list. When a page admission is needed, the target page is always inserted in the head, the most recently used (MRU) place, of the corresponding linked list. When a page replacement is required, the coldest cached page is evicted from the back, the least recently used (LRU) place, of the lowest linked list that is not empty.



(a) Internal organizations of cached pages.

(b) Access freq equals to 0 for eviction and 1 for demotion.

Figure 3.3: Internal organizations of cached pages and EV operations demonstration.

When fulfilling an eviction request, HeuristicDB does not actually evict all cached pages; instead, their access frequency is set to zero. Hence, after eviction, those pages are still in the NVM buffer pool, but have the lowest priority. The detailed operations are described below. A new item with an access frequency of zero is created first if it does not already exist in the LFU data structure. At the beginning, the value (the linked list) inside this item is empty. All items in the LFU data structure are then scanned from the lowest to the highest access frequency in sequence. If the linked list inside an item is not empty, that linked list is first removed and then re-inserted at the back of the number zero linked list. These operations are repeated until all items inside the hash table have been scanned. After the eviction, as shown in Fig. 3.3b, the most frequently used pages stay in the least recently used place of the number zero linked list. This design enables HeuristicDB to replace more easily pages that are accessed frequently in the past, but are rarely used in new submitted queries.

For a demotion request, the EV program performs similar operations as for an eviction request. The only difference is that it inserts all cached pages into the number one linked list, as shown in Fig. 3.3b. Note that after an eviction or demotion has been completed, the pages in the NVM buffer pool are synchronized to *the counting table*. The counting table then contains LBA for cached pages only, and their corresponding access frequency is zero if eviction is performed, otherwise it is one.

3.5.3 Access Pattern Detector

To determine the access behavior of each table or index required by a query, an access pattern detector is developed. The detection program consisted of two main stages. In the first stage, the detector is only asked to detect whether a table or index is accessed sequentially or randomly. The detector continues to monitor while the first stage is terminated. If a page inside a sequentially or randomly accessed table or index is found to be reused, the detector resets its status to *repeated* immediately. The proposed system concludes that a page is reused if its access frequency shown in the counting table is greater than one. A detailed discussion of how the table access pattern detector works in the first stage is presented below.

The proposed system decides whether a given table or index is accessed sequentially or randomly by computing its *Sequential Ratio*. The Sequential Ratio is defined as the number of pages accessed sequentially divided by the total number of pages accessed from that table or index during query execution. The system sets the status of a table or index to *sequential* if its Sequential Ratio exceeds 90%, otherwise to *random*. If a file is accessed sequentially, this means that most pages inside that file are also accessed sequentially. Therefore, instead of deciding after an entire table or index had been accessed, in the present study, only the first 1% of pages accessed are monitored to determine the access behavior of a table or index. This is done to minimize the extra overhead imposed by this detection program. For instance, for a table or index containing 10,000 pages, only the first 100 pages accessed must be monitored to compute its Sequential Ratio by dividing the number of sequentially accessed pages by 100.

Hence, the overhead incurred by this monitoring process is negligible. The number of sequentially accessed pages in each table or index is reset to zero every time that a new query is submitted to the database engine and is increased by one if a page is read using the database *read-ahead mechanism* [155] during query processing. Hence, the overhead incurred by this monitoring process is negligible.

3.5.4 Page Placement Controller

The page placement controller is developed in HeuristicDB to redirect database block requests to the appropriate destination. Three data structures, including *the counting table*, *the LFU data structure*, and *the profile table*, work cooperatively with the page placement controller to execute a query. The counting table records the access frequency of all pages accessed during query processing, as described in subsection 3.4.1. This design avoids recounting the access frequency of a hot page once it has been evicted from the NVM buffer pool. In addition, to distinguish the hottest degree and facilitate page eviction when a replacement is required, all pages that stay in the NVM buffer pool are organized into an LFU data structure, as shown in Fig. 3.3a. Finally, the profile table is used to check the status of the table or index to which the accessed page belonged, as described in subsection 3.4.1.

During query execution, when a block request is generated, the page placement controller first updates its access frequency inside the counting table. For a block write request, the page placement controller then checks space availability in the NVM buffer pool for page allocation. If there is no space left, the coldest cached page is evicted. The target page is then written into the NVM buffer pool and flushed into the storage device asynchronously later.

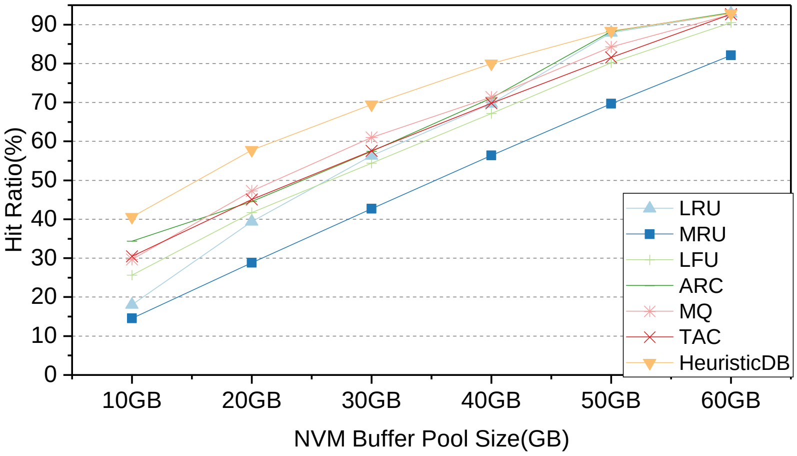
For a block read request, if the required page has already been cached, let us suppose in the i-th linked list in Fig. 3.3a, the controller first removes it from that linked list and then re-inserts the page in front of the i+1-st linked list. Otherwise, a cache miss happens, and the required page is read from storage. An extra evaluation for page admission is also required. The detailed criteria for page admission have been discussed in subsection 3.4.2. To clarify further, the lowest access frequency among the cached pages can be acquired by scanning items inside the LFU data structure in sequence until an item with a non-empty value (the linked list) is encountered. The key inside that item then represents the lowest access frequency among cached pages. When a page replacement is required, the least recently used page in the lowest linked list is first evicted from the NVM buffer pool to release extra space. The page placement controller then inserts the new incoming page into the most recently used place in the corresponding linked list.

3.6 Trace Based Evaluation

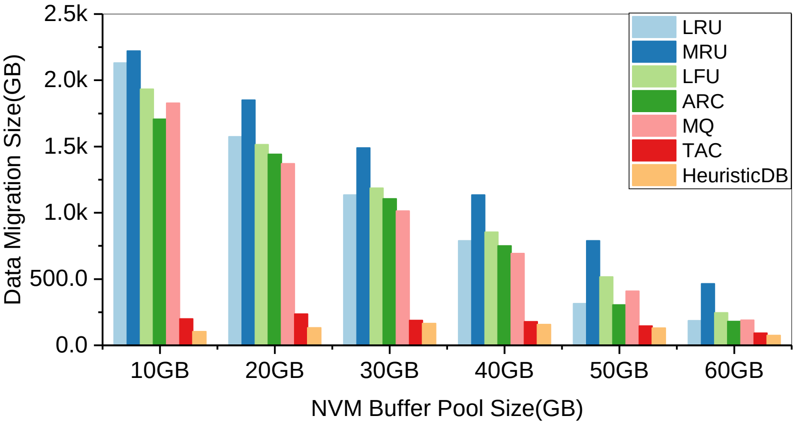
This section describes a trace-based evaluation that is conducted on the proposed system. Six baseline replacement algorithms, including the least recently used (LRU), the most recently used (MRU), the least frequently used (LFU), the adaptive replacement cache (ARC) [75], MQ [95], and TAC [91], are also tested for comparison. The block traces of the database workload are collected by blktrace [138]. These traces recorded the LBA, the read-write type, and the block size.

Two database benchmarks, TPC-H [107] and TPC-E [144], are used in the experiments. TPC-H is an OLAP benchmark consisting of eight tables with corresponding indices and a variety of ad-hoc queries. In the experiment, the Sacle Factor (SF) is set to 60, and after all data had been loaded into storage, the TPC-H database size had increased to 90GB. TPC-E is an OLTP benchmark consisting of 33 tables with corresponding indices and diverse concurrent transactions. For this experiment, a 7000-customer database is chosen, and after all data had been loaded into storage, the TPC-E database size had increased to 97GB.

Two metrics are used to evaluate the performance of HeuristicDB and the other six baseline replacement algorithms: (1) the cache hit ratio, which reflects the performance gain that each caching algorithm delivers; and (2) the data migration size between storage and the NVM buffer pool. Because migrating data from storage requires energy, the higher this value, the more energy is consumed. In addition, considering the endurance problem of the NVM block device [140, 141, 142, 143], frequently migrating data into the NVM buffer pool shortens its lifespan. For these reasons, data migration size is considered as a key performance evaluation factor. The evaluations are conducted under three database workloads, including a complex OLAP query workload, a TPC-H workload, and a TPC-E workload. It is assumed that the NVM buffer pool is empty before each evaluation. Its size is set to values from 10 to 60 GB with a granularity of 10 GB. A performance gain analysis is provided in the last subsection.



(a) Complex OLAP workload hit ratio.



(b) Complex OLAP workload migration size.

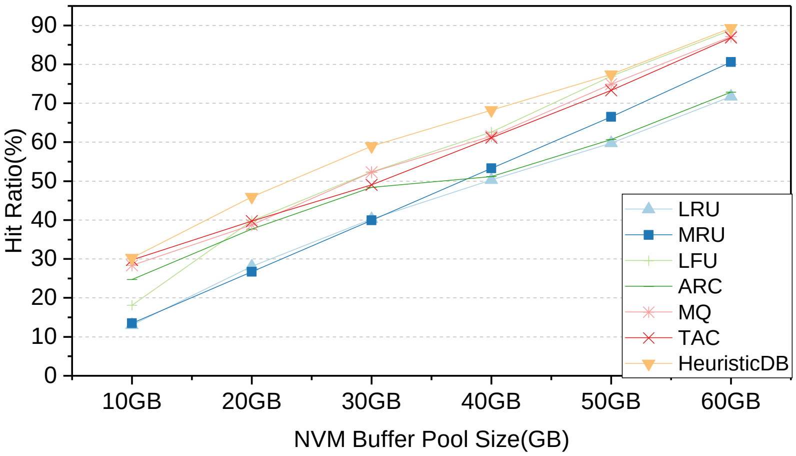
Figure 3.4: Experimental results of trace based evaluation - Complex OLAP Workload.

3.6.1 Complex OLAP Workload

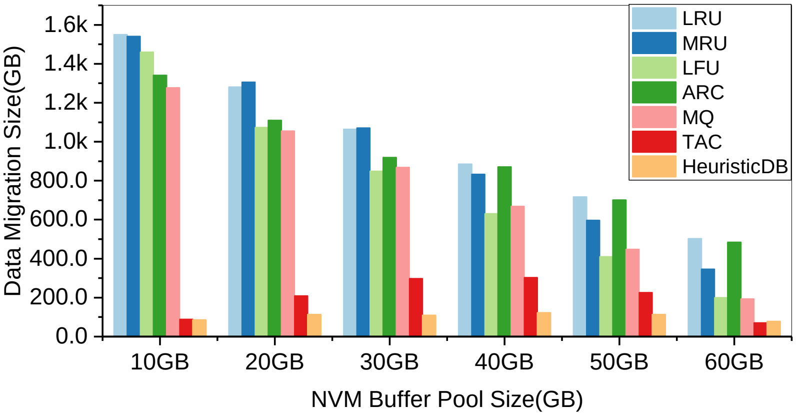
OLAP aims to enable enterprises to make better decisions by analyzing data elements inside collected databases [156]. Processing high-complexity OLAP queries usually requires massive block operations from storage. As a result, I/O time becomes a main bottleneck in the database system [26, 69]. This subsection describes an evaluation of how HeuristicDB and the other six baselines work for a complex OLAP workload.

The experimental workload is constructed based on five types of complex TPC-H queries where performance is limited by I/O time [26]. These are Queries 3, 7, 9, 13, and 20. To avoid cases where a replacement algorithm works well only for a specific type of query that consumes most of the block operations in the workload and is therefore producing a misleading conclusion, in the experiment, the number of each type of query is inversely proportional to its execution time. In addition, to avoid a query's results being cached because it has been run before, in the experiment, different query conditions are used for the same type of query. The query conditions are randomly drawn from a uniform distribution in the range given by the TPC-H specification [107]. Using these principles, 20 complex OLAP queries are generated in random order and issued by the database system. Fig. 3.8a lists the generated query sequence.

Each replacement algorithm is run separately on the same trace. Fig. 3.4 shows plots of the experimental results. As Fig. 3.4a shows, HeuristicDB outperforms all other baselines on the cache hit ratio. Most significantly, it maintains about a 10% higher cache hit ratio relative to the second-best replacement algorithm, MQ, in a large cache size range from 20GB to 40GB. In contrast, TAC, one of the main competitors of HeuristicDB, delivers an even worse cache ratio than ARC when the NVM buffer pool size is set to 10GB. HeuristicDB also works well on data migration size. As shown in Fig. 3.4b, HeuristicDB migrates only half as much data as TAC with NVM buffer pool sizes equal to 10GB and 20GB. The performance advantage of HeuristicDB disappears gradually after the NVM buffer pool size rose to 50GB. This occurs because a larger NVM buffer pool leads to more hot pages being cached, and the performance difference for each replacement algorithm is reduced.



(a) TPC-H workload hit ratio.



(b) TPC-H workload migration size.

Figure 3.5: Experimental results of trace based evaluation - TPC-H Workload.

3.6.2 TPC-H Workload

This subsection describes a performance evaluation of HeuristicDB on a TPC-H query sequence. The experimental results are plotted in Figs. 3.5a and 3.5b. As shown in Fig. 3.5a, HeuristicDB outperforms all other baselines and delivers a cache hit ratio about 7% higher than the second-best replacement algorithms, TAC and LFU, when the NVM buffer pool size is set to 20GB, 30GB, and 40GB. In addition, there are two more interesting observations to be made. First, for the TPC-H workload, MQ could not maintain its second-highest cache hit ratio as it did for the complex OLAP workload. Instead, TAC and LFU shares that distinction. Second, ARC delivers a highly unstable performance improvement on cache hit ratio. For instance, when the NVM buffer pool size is less than 40GB, ARC delivers a cache hit ratio up to 11% higher than MRU. However, as the NVM buffer pool size continues to rise to 50GB and 60GB, ARC delivers an 8% lower cache hit ratio than MRU. In contrast, HeuristicDB maintains the highest cache hit ratio for all NVM buffer pool sizes.

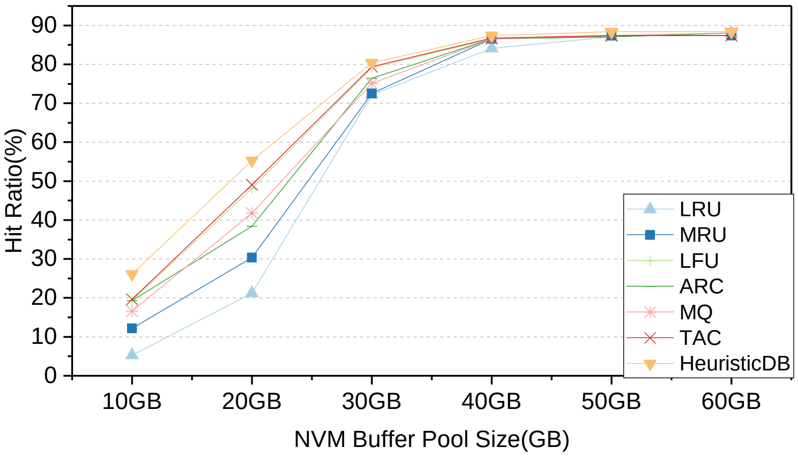
Finally, HeuristicDB achieved a significant reduction in data migration size. As shown in Fig. 3.5b, when the NVM buffer pool size was equal to 30GB and 40GB, HeuristicDB migrated 3X less data between storage and the NVM buffer pool than TAC.

3.6.3 TPC-E Workload

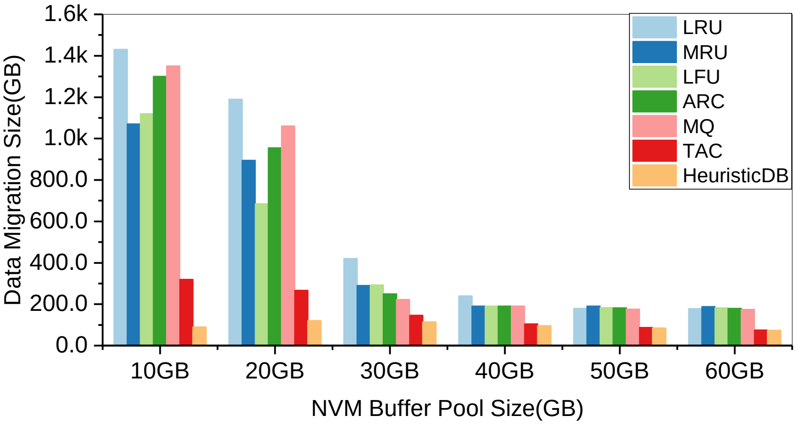
To validate the effectiveness of HeuristicDB on the OLTP workload, a five-hour block trace of the TPC-E benchmark is collected and used for performance evaluation. The experimental results are plotted in Figs. 3.6a and 3.6b. From these results, when the NVM buffer pool size is set to 10GB and 20GB, HeuristicDB provides a 6% higher cache hit ratio than the second-best replacement algorithm, LFU, and migrates 3.6X less data than TAC. As the NVM buffer pool size continues to increase, the performance gap between HeuristicDB and other baselines on both cache hit ratio and database migration size decrease. This occurs because most of the block operations inside the TPC-E benchmark are generated for accessing the 28GB trade table. When the NVM buffer pool size is equal to or greater than 30GB, because most of the frequently used pages had already been cached in the NVM buffer pool, HeuristicDB and the other baselines produce similar performance on both cache hit ratio and data migration.

3.6.4 Performance Gain Analysis

This section describes a performance evaluation of HeuristicDB and other baselines using the block traces of three types of database workloads. From the results, HeuristicDB delivers a significantly enhanced cache hit ratio and incurs much less data migration size than the other alternatives. Most significantly, the proposed system maintains



(a) TPC-E workload hit ratio.



(b) TPC-E workload migration size.

Figure 3.6: Experimental results of trace based evaluation - TPC-E Workload.

the best performance for all tested database workloads. In contrast, the performance improvement provided by the baseline algorithms varies greatly as database workloads changed or even with different NVM buffer pool sizes. For instance, MQ delivers the second-best performance on cache hit ratio for the complex OLAP workload, but delivers an even lower cache hit ratio than LFU after the workload is changed to TPC-E. The performance advantages of HeuristicDB can be attributed to two design principles: (1) by considering the characteristics of each database block request, the tables and indices required by the query are prioritized based on their reusability, and only the hottest pages from a high-priority table or index are allowed to be cached. This enables HeuristicDB to deliver maximum performance gains with less data migration in query execution; and (2) the adaptive eviction and demotion policy enables HeuristicDB to evict or demoted cached pages and admit new incoming hot pages more efficiently. Because of these designs, HeuristicDB outperforms all other baselines under the tested database workloads.

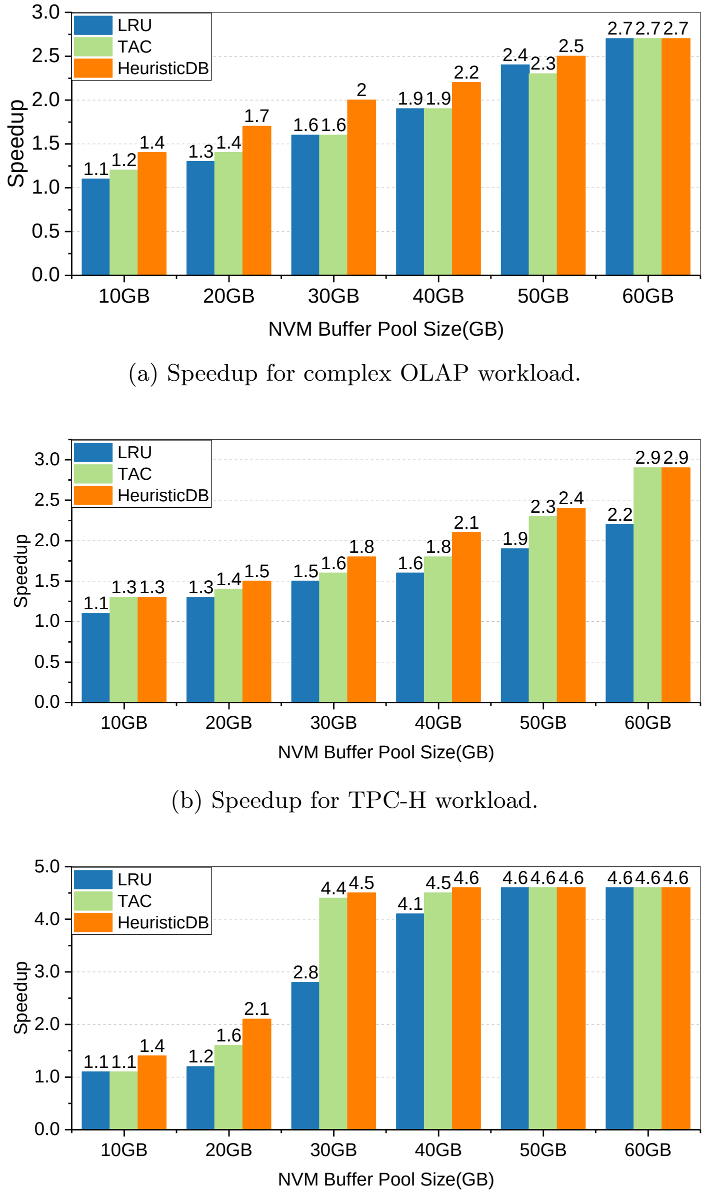
3.7 Performance Evaluation on a Real System

This section describes the implementation of a HeuristicDB prototype on an empirical system for performance evaluation. For comparison, the authors also implemented LRU, which is the most widely used replacement algorithm, and TAC, which is a hybrid storage management algorithm for database application. In the test, the same workloads are used as in the previous section. Two metrics, including execution time and transaction rate, are used to evaluate the performance of HeuristicDB and the other two baseline algorithms.

3.7.1 Implementation Details

LRU, TAC, and HeuristicDB are implemented based on Flashcache [89], a block cache component for the Linux kernel. During query processing, the database server runs on top of the system. When block requests are sent to the storage layer, Flashcache automatically maps them either to the NVM buffer pool or to storage devices based on its internal replacement algorithm.

The experiments are conducted on a Dell Precision Tower 3620 workstation equipped with a four-Intel i7-6700 3.4 GHz CPU and 16 GB of memory. The Ubuntu 18.04 LTS operating system is installed on the workstation. Two types of storage devices are used, an Intel 168 GB DC P4800X Optane SSD [46] and two Intel 2TB DC P3700 NAND SSDs [131]. The Optane SSD acts as an NVM buffer pool, and the NAND SSDs are used as persistent storage. To avoid interference, the operating system and database are stored on two separate NAND SSDs. MySQL 5.7 [157] with the InnoDB storage engine [158] is running on the system. The page size of all tablespaces is set equal to 16KB, and the database buffer pool is set to 4 GB. The number of InnoDB I/O threads



(a) Speedup for complex OLAP workload.

(b) Speedup for TPC-H workload.

(c) Speedup for TPC-E workload.

Figure 3.7: The speedup of HeuristicDB and other baselines to database workloads over the case without NVM buffer pool. The execution time(in sec) and transaction rate(TpsE) for complex OLAP workload, TPC-H workload, and TPC-E workload of the case without NVM buffer pool are 34993.22sec, 20507.77sec, and 59.3 TpsE, respectively.

is set to its default value of four. In the experiment, after each test run, the MySQL database application is restarted to clean up both main memory and the NVM buffer pool.

3.7.2 Experimental Results

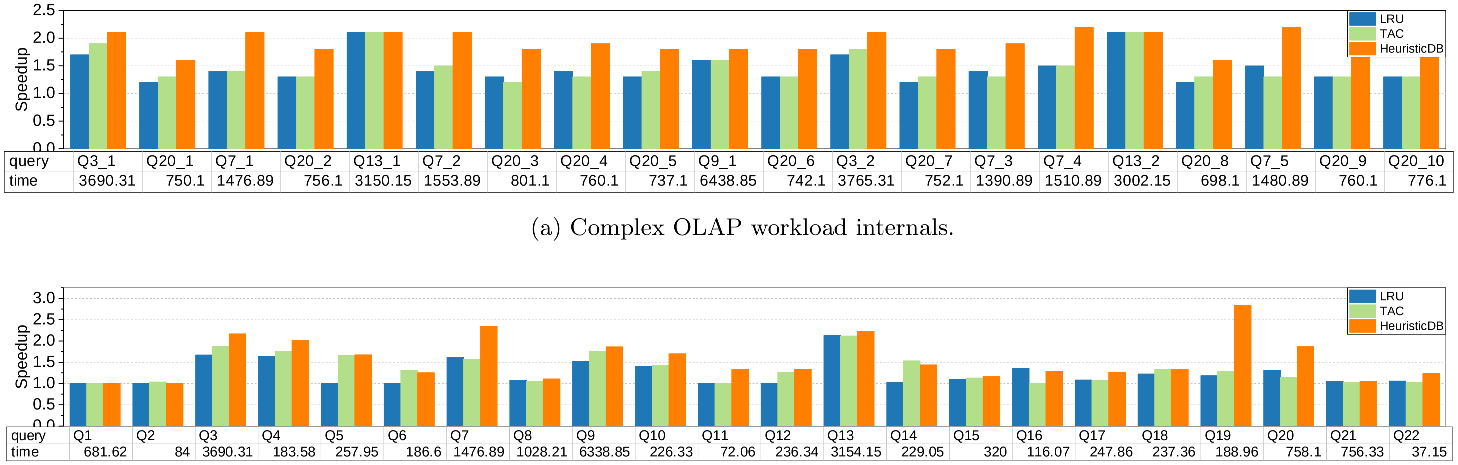
Fig. 3.7 illustrates the performance speedup of the three replacement algorithms for the case without the NVM buffer pool. For each type of workload, different performance metrics are used. For the OLAP workloads, including complex OLAP and TPC-H, the total execution time to complete the workload is measured. For the OLTP workload, TPC-E, the number of transactions executed per second (TpsE) is measured. From the results, HeuristicDB delivers the highest performance accelerations compared with the other two baselines. This observation matches the results described in Section 3.6.

As shown in Fig. 3.7a, for the complex OLAP workload, HeuristicDB improves database performance over LRU and TAC by up to 25% when the NVM buffer pool size is set to 30GB. In addition, HeuristicDB maintains the best performance for all NVM buffer pool sizes. In contrast, TAC shows unstable improvement in performance. For instance, when the NVM buffer pool is set to 50GB, TAC delivers even worse performance than LRU. For the TPC-H workload, as shown in Fig. 3.7b, for an NVM buffer pool size of 40GB, HeuristicDB provides 17% and 31% higher performance than TAC and LRU respectively.

HeuristicDB also works well for the OLTP workload. As shown in Fig. 3.7c, HeuristicDB maintains the best performance for the TPC-E benchmark. It enhances the transaction rate over LRU and TAC by 75% and 31% respectively when the NVM buffer pool is set to 20GB. However, because more hot pages are cached with continued increases in NVM buffer pool size, the performance advantages of the proposed system relative to other baseline replacement algorithms gradually disappears.

3.7.3 Internals on the OLAP Workload

Compared with OLTP, which accesses the database using an index and retrieves only a tiny portion of data to execute a transaction, an OLAP query has higher complexity and usually requires many more block operations. Therefore, in most cases, I/O time



query	Q3_1	Q20_1	Q7_1	Q20_2	Q13_1	Q7_2	Q20_3	Q20_4	Q20_5	Q9_1	Q20_6	Q3_2	Q20_7	Q7_3	Q7_4	Q13_2	Q20_8	Q7_5	Q20_9	Q20_10
time	3690.31	750.1	1476.89	756.1	3150.15	1553.89	801.1	760.1	737.1	6438.85	742.1	3765.31	752.1	1390.89	1510.89	3002.15	698.1	1480.89	760.1	776.1

(a) Complex OLAP workload internals.

query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
time	681.62	84	3690.31	183.58	257.95	186.6	1476.89	1028.21	6338.85	226.33	72.06	236.34	3154.15	229.05	320	116.07	247.86	237.36	188.96	758.1	756.33	37.15

(b) TPC-H workload internals.

Figure 3.8: The speedup of different replacement algorithm to each query inside the OLAP workload. The NVM buffer pool is set to 30GB. We record the execution time(in seconds) of each query of the case without NVM buffer pool buffer below each figure.

is the main bottleneck for OLAP queries [68, 66, 67].

This subsection describes in detail the internals of the OLAP workload and examines how different placement algorithms benefit the individual query inside an OLAP workload. The speedup achieved by HeuristicDB and the other two baselines on each individual query is plotted inside both the complex OLAP and the TPC-H workloads, as shown in Fig. 3.8. In the experiment, the NVM buffer pool is set to 30GB. In addition, to show query complexity clearly, the execution time of each query for the case without the NVM buffer pool is recorded below the figure. From the results, HeuristicDB delivers the highest performance acceleration for almost all queries inside both the complex OLAP and the TPC-H workloads. This is the case because HeuristicDB adaptively adjusts the priorities of cache pages and tracks hot tables and indices continuously when a query is submitted. In addition, it discriminates tables, indices, and pages based on their reuse probability. These designs enabled HeuristicDB to preserve the most profitable pages in the NVM buffer pool during each single query to deliver the highest performance acceleration. In contrast, the performance improvement of TAC on each single OLAP query is unstable. For instance, as shown in Fig. 3.8a, TAC experiences performance degradation and even delivers lower speedup than LRU for queries Q20_3, Q20_4, Q7_3, and Q7_5. A similar result can be observed for the TPC-H workload, as shown in Fig. 3.8b, for queries Q7, Q8, Q13, Q16, Q20, Q21, and Q22.

3.8 Conclusion

This chapter presents HeuristicDB, which uses an NVM block device as an extension of the database buffer pool. By considering the intrinsic performance behavior of the NVM block device and the characteristics of database block requests, seven rules that associate database (block) requests with the appropriate quality of service are proposed. To support the system implementation of these proposed rules, four programs, including the query profile queue, the EV program, the access pattern detector, and the page placement controller, are then developed inside HeuristicDB. Compared with earlier studies, HeuristicDB delivers the highest performance and migrates much less data between storage and the NVM buffer pool for all tested database workloads.

To evaluate the effectiveness of this proposed design, intensive experiments are conducted using both OLAP and OLTP benchmarks. The results of a trace-based experiment show that HeuristicDB provides up to 29% higher cache hit ratio and 18X less data migration size than competing baselines. Subsequently, the HeuristicDB, LRU, and TAC algorithms are implemented on Flashcache [89], and MySQL is run on a hybrid system with different block cache sizes. The results show that HeuristicDB improves database performance by up to 75% over baselines in real system testing.

Chapter 4

ExpCache: Online-Learning based Cache Replacement Policy for Non-Volatile Memory

4.1 Introduction

With the rapidly increased volume of newly generated data, the demand for high-performance memory media becomes more critical than before. People pursue a high access bandwidth and expect a large density and non-volatility property on memory devices. As one of the emerging memory technologies, non-volatile memory (NVM) provides both the competitive performance as DRAM meanwhile and the property of non-volatility as storage devices. Because all data reside in NVM is persistent, there is no need to write updated pages (i.e., dirty pages) back to storage for preventing data loss while power failure, which significantly increases the system performance. According to those advantages, NVM is expected to replace DRAM to serve as the main memory in the near future. Figure 4.1 illustrates this architecture that we have adopted through the chapter.

In order to fully take the performance advantages of NVM, designing a proper cache algorithm to manage the data elements within NVM becomes important. The cache algorithms have been intensively investigated over time. However, those algorithms were

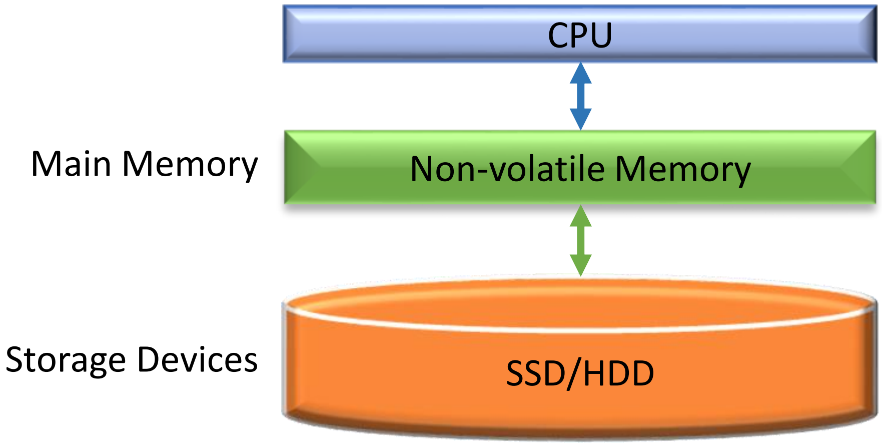


Figure 4.1: NVM-based System Architecture.

developed to make a replacement decision based on certain pre-defined rules (e.g., LRU algorithm always evicts the least recently used page when a replacement is required) instead of the real access patterns of a workload. Since these pre-defined rules are not adapted to the dynamically changed workload access patterns, these cache algorithms suffer significant performance degradation. This explains why conventional cache algorithms [75, 76, 77, 78] only work well on a specific set of workloads.

Machine learning (ML) algorithms have been well studied in the past decade and have been introduced in various areas for performance improvement [79, 80, 81]. Researches show that machine learning techniques perform well, especially on data tracking and making predictions [159, 160]. Motivated by this, it has a large potential to employ machine learning techniques in cache algorithm designs for adapting to dynamically changed workloads to improve the system performance. Two machine learning based cache algorithms, including LeCaR [96] and Cacheus [161], were proposed recently. They can perform well and beat some traditional cache algorithms. However, they failed to explain how to choose parameters in machine learning algorithms theoretically. Therefore, they perform mostly well only with small cache sizes. More detailed discussions about the existing cache algorithms are provided in Section 4.2.3.

All those schemes mentioned above were originally designed for DRAM-based systems. So, they are unlikely to deliver excellent performance improvement for NVM-based systems due to the following reasons. First, those policies do not consider the non-volatility of NVM and thus ignore the influence of write operations in-memory systems. Second, they do not work well on establishing the proper model for tracking dynamically changed workloads adaptively and accurately. Finally, previous machine learning-based caching policies lack the theoretical proof to select proper parameters for ML algorithms and indicate the lower or higher bounds of the algorithms. Thus, it is hard to convince people that ML-based algorithms can improve cache performance in general. Due to the above limitations, designing a new cache policy for managing NVM becomes necessary. Although H-ARC [76] as the most relevant study is an improvement of the ARC algorithm [75] designed for NVM, it has similar problems as mentioned above such as lacking tracking dynamically changed workloads and lacking the prevention of the eviction of dirty pages for NVM.

In this chapter, we propose a new online learning-based cache replacement algorithm, called ExpCache, for NVM-based systems to overcome the shortcomings of prior studies. The goals of ExpCache are improving both read and write hit ratios of NVM meanwhile eliminate the number of dirty pages written back to the storage. In the system, the real cache (i.e., NVM main memory) is divided into a read cache and a write cache for retaining different types of requests. Two experts, including an LRU policy and an LFU policy, are employed to manage the data elements within each of the real caches to balance both recency and frequency factors. In addition, two ghost caches are built to store the evicted pages from the real caches and emulate the LRU and LFU policies independently. The time-dependent *online Expert* algorithm continuously monitors the hit ratio inside those ghost caches and uses them as an indicator of workload access patterns. While a replacement is required, ExpCache selects a proper eviction policy based on its tracked access pattern. Finally, using the selected eviction policy, ExpCache sets evicting a page from the read cache having a higher priority while a certain condition is met to reduce the number of dirty pages written back to the storage.

The organization of this chapter is as follows. Section 4.2 introduces the background of NVM, previous caching policies and online learning algorithm. Section 4.3 describes the proposed ExpCache algorithm. The experimental results are shown in Section 4.4.

Finally, a conclusion is drawn in Section 5.1.

4.2 Background and Related Work

4.2.1 Non-volatile Memory

By blending the high performance characteristics of DRAM and the non-volatile property of conventional storage devices, non-volatile memory (NVM) such as 3D Xpoint [25, 26, 27], phase-change memory (PCM) [28, 29, 30, 31], and conductive bridging RAM (CBRAM) [32, 33, 34] has been becoming critical importance for modern systems. Compared with DRAM, NVM not only delivers similar read-write bandwidth but also has a much higher density. Most significantly, all data resided in NVM is persistent. Thus, there is no need to write updated dirty pages back to storage to prevent data loss when power is off. Compared with conventional storage devices connected to an external block-addressable PCIe or SATA interface, NVM communicates with processors more efficiently using a byte-addressable memory bus. In addition, by employing high-performance memory cells, NVM presents much higher read-write bandwidth than conventional storage devices. Because of those performance advantages, NVM is expected to replace DRAM to serve as the main memory in the near future.

4.2.2 Online Expert Selection

Online expert selection problem is extensively studied in the online learning community over the past two decades [162, 163, 164]. The basic setting is: At each time step t, we have n experts giving predictions on the considered problem. Before observing the loss vector $\ell_t \in \mathbb{R}^{n \times 1}$ associated with the experts' prediction at time step t, we select one expert to try to have as small cumulative cost as possible. As we can see, it is the missing information on ℓ_t while making a decision on the expert selection that makes the problem difficult. The loss vectors ℓ_t, $t = 1, 2, \ldots, T$ are from a bounded set and can be generated deterministically, stochastically, or adversarially.

One standard metric to measure the effectiveness of the online expert selection algorithm is called regret $\mathcal{R}_T$, which is defined as:

$$\mathcal{R}_T = \sum_{t=1}^{T} \mathbb{E}[\ell_t^\top x_t] - \min_{u \in \mathcal{S}} \sum_{t=1}^{T} \ell_t^\top u \tag{4.1}$$

where both $x_t \in \mathcal{S}$ and $u \in \mathcal{S}$ represent the selection result by the algorithm and comparator, respectively. The vector set $\mathcal{S}$ denotes the set with only one non-zero element equal to 1. The expectation $\mathbb{E}$ is taken over the expert selection probability distribution of x_t. The intuition behind the metric of Eq. (4.1) is that it measures the difference between the algorithm cumulative cost and the one with the best-fixed decision in hindsight.

Prior work such as [165, 166, 167] can upper bound the regret by the order of $O(\sqrt{T})$, which is asymptotically optimal [168]. Such sub-linear upper bound also indicates that the performance of the proposed algorithm converges to the best fixed decision as time horizon $T \to \infty$. However, when the underlying environment that generates the loss sequences is changing, the regret in Eq. (4.1) is no longer appropriate. This is because the single fixed optimal comparator is not the good baseline, since its cumulative loss cab be high as well.

Alternatively, a new metric called adaptive regret is proposed in [169] to capture the changing environment, and is defined as the maximum regret over any contiguous time interval as follows:

$$\mathcal{R}_a = \max_{s+1-r \leq \tau} \Big\{ \sum_{t=r}^{s} \ell_t^\top x_t - \min_{u \in \mathcal{S}} \sum_{t=r}^{s} \ell_t^\top u \Big\} \tag{4.2}$$

The algorithm proposed in [170] can upper bound the adaptive regret by the order of $O(\sqrt{\tau \log(\tau)})$, which is sub-linear for any compared time interval and generalizes the result of the regret defined in Eq. (4.1).

In this chapter, we extend the algorithm proposed in [170] to design a new cache replacement policy that can adapt to the dynamically changed caching environment. The proposed scheme can provide a theoretical performance guarantee in terms of adaptive regret and achieve better experimental results than prior caching policy works.

4.2.3 Related Work & Limitations

Past cache algorithms make a replacement decision based on certain rules instead of adapting to the real workload access patterns. As a result, they are hard to obtain good performance as workloads or cache sizes changed [94, 75, 95, 77]. With utilizing online learning algorithms, Vietri et al. presented a DRAM-based cache algorithm called LeCaR [96]. In the system, LeCaR manages cached data elements using two experts (i.e., a LRU policy and a LFU policy) and selects the proper one as an eviction policy when a replacement is required. LeCaR encounters two limitations. First, it must pre-define some tuning parameters to set its reinforcement online learning algorithm. Second, because LeCaR losses either the recency or the frequency information of each evicted page, it is failed to track the access patterns of a workload accurately and adaptively in some cases. Rodriguez et al. [161] extended LeCaR work by replacing the pre-defined tuning parameters within LeCaR to dynamically changed ones and providing two more expert candidates. The newly developed policy is named as CACHEUS. Because CACHEUS inherits the design principle as LeCaR developed, it suffers the second issue of LeCaR as mentioned above. Moreover, to utilize CACHEUS algorithm, system administrator must select two experts algorithms from the expert pool. An improper selection affects the performance of CACHEUS significantly.

H-ARC is the most relevant study in this area proposed by Fan et al. [76]. It introduced an improvement of ARC, called H-ARC, for NVM-based memory systems to improve both cache read and write hits. H-ARC splits the real cache into four small caches, including recency clean cache, frequency clean cache, recency dirty cache, and frequency dirty cache. Each of those small caches owns a ghost cache to store the evicted pages from it. While a replacement is required, H-ARC compares the size of each cache with their desired size and evicts a page from the one whose size is larger than the desired. However, H-ARC suffers from several limitations. First, dividing the real cache into too many smaller fragments breaks collected recency and frequency information. As a result, H-ARC cannot track workloads' characteristics correctly and produces a low cache hit ratio in some circumstances. Second, there is no specific mechanism within H-ARC to prevent evicting the dirty pages to storage. Thus, H-ARC usually produces high write counts. For the systems that use NAND SSD as storage, frequently evicting dirty pages back into the storage degrades system performance tremulously due to NAND

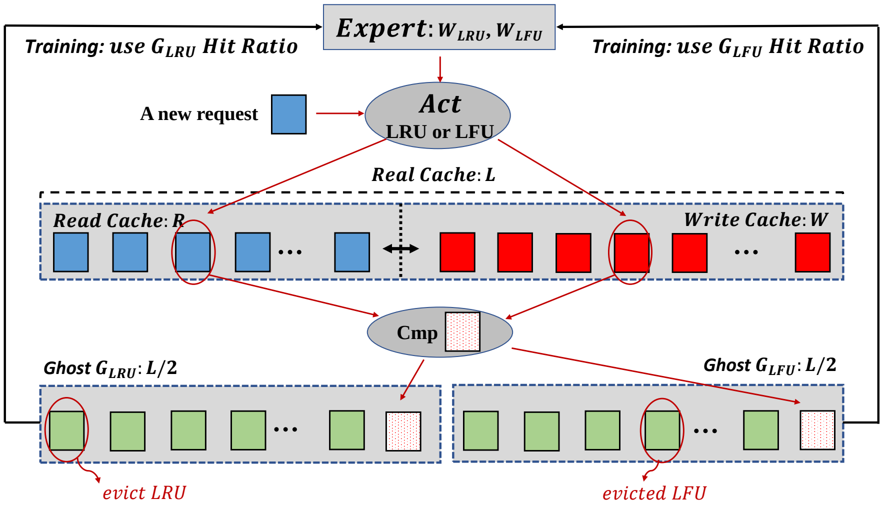


Figure 4.2: ExpCache Architecture.

SSD garbage collection issue [51, 52].

The other types of caching policies such as CFLRU [171], LRU-WSR [172], AD-LRU [173], and DPW-LRU [174] were developed for NAND flash-based devices. Those policies reduce the number of dirty pages evicted to NAND flash memory and leave the cache hit ratio in second place. In addition, they only consider the recency factor and overlook the frequency factor.

4.3 ExpCache Algorithm

In this section, we provide a detailed description of the ExpCache algorithm. We introduce the design principle and the internal architecture of ExpCache, and its working processes at first. Two key components of ExpCache, including the time-dependent online Expert and read-write discriminated eviction policy, are discussed then.

4.3.1 ExpCache Design Principle and Internal Architecture

Two fundamental factors (i.e., recency and frequency) are widely used in cache policies as discussed in Section 4.2.3. This is because the patterns of most workloads follow either recency or frequency manner in one period of time. Thus, ExpCache is proposed by assuming that at every time step, there has the best strategy among two fundamental experts, recency expert (i.e., LRU policy) and frequency expert (i.e., LFU policy). The goal of the ExpCache is to learn the caching environment, such as dynamically changed workloads and hit ratios, and then select a proper eviction scheme from the experts for the system. In order to map the *online expert algorithm* into the proposed policy properly, meanwhile consider the non-volatile property of NVM, the architecture of the ExpCache is carefully designed.

As shown in Figure 4.2, in the system, we split the real cache into two small caches, including a read cache denoted as R and a write cache denoted as W, for retaining different types of requests. Pages within the read cache are clean (i.e., have never been modified). Once a page is modified (i.e., the page becomes dirty), it will be moved into the write cache. In order to perform a replacement efficiently using any one of the two experts, the pages within each small cache are managed by both LRU and LFU-based structures.

Two ghost caches denoted as G_{LRU} and G_{LFU}, are built to simulate the behaviors of LRU policy and LFU policy independently. Each ghost cache records the metadata (i.e., page identifier) of the most recent evictions from the real cache. When a victim is evicted from the real cache, it is inserted into both G_{LRU} and G_{LFU}. The elements inside the G_{LRU} are strictly organized in an LRU order based on their timestamps inherited from the real cache. This can be done by using a heap data structure. Here, the timestamp of a page is the access number of the last access to that page. The time complexity of adding or deleting an element into or from the G_{LRU} is $log_2 N$. Where, N is the number elements within G_{LRU}. Since N is a constant and is usually much smaller than the number of requests within a workload, the overhead on operating G_{LRU} is negligible. Similarly, the elements inside G_{LFU} are strictly organized in an LFU order based on their corresponding access frequency and timestamp inherited from the real cache. A more detailed analysis of the space and computation overheads of ExpCache is discussed in Section 4.4.5.

The real caches and the ghost caches support the following operations.

UpdateDataStructure(r): is to update LFU and LRU based data structures in different caches in ExpCache. The detailed update procedures are:

- For LRU policy: move the target page, r, into the most recently used (MRU) place within the LRU data structure;

- For LFU policy: supposing the access frequency of the target page is *i*, then after a hit happens, we move that page into the MRU place of the LRU chain whose access frequency equals to *i+1*.

Delete(r): deletes the target page, *r*, from the corresponding cache. Note here, deleting a page from R or W means removing the page from both LRU and LFU data structure/policy.

Add(r, freq, timestamp): is to insert an element, *r*, in to a cache based on the frequency *freq* and/or the recency *timestamp*. The detailed procedures depend on what type the cache is:

- For G$_{LRU}$: we add the target page, *r*, into the proper position of G$_{LRU}$ based on its *timestamp*.

- For G$_{LFU}$: we add the target page into the proper position of G$_{LFU}$ based on its *freq* and *timestamp*.

- For R and W: For LRU policy, we add the target page into the most recently used (MRU) place; For LFU policy, we add the target page into the MRU place of the LRU chain whose access frequency equals to *freq*.

Finally, ExpCache maintains the probabilities (i.e., weights) of selecting one of two experts being the eviction scheme. The weight of each expert, denoted as w_{LRU} and w_{LFU}, is associated with the hit ratio inside its corresponding ghost cache. In order to track the access patterns of a workload adaptively and accurately, ExpCache updates w_{LRU} and w_{LFU} every time when a miss occurs in the real cache using *online expert algorithm*. With using this *online expert algorithm*, ExpCache guarantees the lower bound of the performance gains (we provide more detailed discussions in Section 4.3.3).

The size of each cache refers to the number of pages stored in it. If we assume the real cache size is L, then the sum of R and W never exceeds L. Two ghost caches, G_{LRU} and G_{LFU}, have the same size. And the size of each of them is never larger than *L/2*.

4.3.2 ExpCache Working Processes

Algorithm 2 presents the working processes of the ExpCache algorithm. When a read or a write request (r) incoming, either a real cache hit or a real cache miss happens. In the following paragraphs, we introduce the processes of a miss and a hit, respectively.

Real Cache Hit

When a cache hit occurs as shown at Line 4 of Algorithm 2, the following procedures happen: 1) If a cache hit for a read request r occurs in R, ExpCache updates the data structures inside R. 2) Similarly, if a cache hit for a write request r occurs in R, this page becomes dirty. Then, we increase its corresponding access frequency by one, record the increased value, and then move that page from R to W. 3) If a cache hit for either a read or a write request occurs in W, this page remains in W. ExpCache only updates the data structures inside W.

Real Cache Miss

If a real cache miss occurs (at Line 15 of Algorithm 2), the following operations are executed sequentially:

1. Ghost caches checking: If the required page hits in G_{LRU}, we delete this page from G_{LRU} first and then increase the number of ghost LRU hits, G_{LRU_Hits}, by one. Similarly, if the r hits in G_{LFU}, we record its access frequency into variable *tfreq* first, and then delete it from G_{LFU} and increase the number of ghost LFU hits, G_{LFU_Hits}, by one. Note that the variable *tfreq* is initialized with one at Line 16.

2. Online expert weights updating: Using the collected information, ExpCache updates the recency weight, w_{LRU}, and the frequency weight, w_{LFU}, by calling *UpdateExpert* algorithm (see Subsection 4.3.3).

3. Real cache capacity checking: In order to admit incoming requests into the real cache, ExpCache checks space availability first. If the sum of R and W is equal to L, we need to evict a page from the real cache to release a free space. ExpCache randomly selects a eviction policy, either LRU or LFU, based on the probability distribution of w_{LRU} and w_{LFU}. Once the replacement decision is made, ExpCache evicts the page

Algorithm 2 : ExpCache(LRU, LFU)

```
Input: request $r$, request type $rwtype$
accessCount++;
if ($r$ in R or W) then
    //Real cache hit
    if ($r$ in R) then
        if ($rwtype$ == Read) then
            R.UpdateDataStructure($r$)
        else
            cfreq = the access freq of $r$ in $R + 1$
            R.Delete($r$)
            W.Add($r$, cfreq, null)
    else
        W.UpdateDataStructure($r$)
else
    //Real cache miss
    tfreq = 1
    //Ghost caches checking
    if ($r$ in $G_{LRU}$) then
        $G_{LRU}$.Delete($r$)
        $G_{LRU\_Hits}$++
    if ($r$ in $G_{LFU}$) then
        tfreq = the access freq of $r$ in $G_{LFU}$
        $G_{LFU}$.Delete($r$)
        $G_{LFU\_Hits}$++
    //Online expert weights updating
    UpdateExpert(accessCount, $G_{LRU\_Hits}$, $G_{LFU\_Hits}$)
    //Real cache capacity checking
    if (R.size + W.size = L) then
        action = (LRU, LFU) / prob($w_{LRU}$, $w_{LFU}$)
        if (action == LRU) then
            victim = EvictVictim(LRU)
        else
            victim = EvictVictim(LFU)
        //adding evicted page into ghost caches
        vfreq = the access freq of the victim in the real cache
        vtimestamp = the timestamp of the victim in the real cache
        if ($G_{LRU}$.size $\geq$ L/2) then
            $G_{LRU}$.Delete(LRU($G_{LRU}$))
        $G_{LRU}$.Add(victim, null, vtimestamp)
        if ($G_{LFU}$.size $\geq$ L/2) then
            $G_{LFU}$.Delete(LFU($G_{LFU}$))
        $G_{LFU}$.Add(victim, vfreq, vtimestamp)
    //Page admission
    if ($rt$ == Read) then
        R.Add($r$, tfreq, null)
    else
        W.Add($r$, tfreq, null)
```

from a real cache by calling *EvictVictim* algorithm (see Subsection 4.3.4).

4. Adding the evicted page into ghost caches: After the real cache eviction is completed, ExpCache adds the victim into both G_{LRU} and G_{LFU}. The space availability checking for ghost caches is conducted at first. For G_{LRU}, if it is full, we delete the least recently used (LRU) page first and then adds the victim. Similarly, for G_{LFU}, if it is full, we delete the least frequently used (LFU) page and then add the victim.

5. Page admission: After all the above operations are completed, ExpCache adds the incoming request into the read cache R if it is a read request. Otherwise, ExpCache adds the request into write cache *W*. For adding the incoming page into a real cache, for LRU policy, ExpCache adds the page into the MRU place of LRU. For LFU policy, ExpCache adds its into the MRU place of the LRU chain whose access frequency equals to *freq*.

4.3.3 Online Expert Weights Update - Time Dependent Expert

This subsection introduces the weight update system by using a time dependent expert as shown in Algorithm 3. The reason of using time-dependent update is that the hyper-parameters such as η and α do not dependent on the time-horizon T (i.e., the number of total requests inside a workload), which is usually unknown in the caching problems.

In Algorithm 3, the loss function is $1 - \mathrm{G}_{Hits}/\mathrm{accessCount}$ related to the goal of maximizing the hit ratio in the real cache. The main steps are at Line 11, which are composed of exponential weight distribution and a fixed-share update.

The exponential weight update ensures that the regret in Eq. (4.1) is upper bounded sub-linearly. Since the caching environment is usually drifting over time, LRU and LFU might be good for the workloads at different periods. The adaptive regret in Eq. (4.2) is desirable in this case to capture the changes of the optimal choice. The fixed-share step is the key to guarantee the theoretical performance of adaptive regret. Thus, it makes sure that the weights associated with each expert are lower bounded, which enables the quick response of weight shift. Note that the update in Algorithm 3 is done in the LRU and LFU ghost caches, since we cannot observe both LRU and LFU performance in real cache simultaneously. In other words, we assume that the performance in the ghost cache is a good indicator of the performance in real cache. Given that the adaptive regret in Eq. (4.2) is upper bounded by $O(\sqrt{\tau\log(\tau)})$ in ghost cache. Thus, with the

Algorithm 3 : UpdateExpert(accessCount, G_{LRU_Hits}, G_{LFU_Hits})

```
Input: $accessCount$, $G_{LRU\_Hits}$, $G_{LFU\_Hits}$
$\ell_{LRU}$ = 1 - $G_{LRU\_Hits}$ / accessCount
$\ell_{LFU}$ = 1 - $G_{LFU\_Hits}$ / accessCount
$\alpha$ = 1 / accessCount
if (accessCount $\leq$ 3) then
    $\eta = \sqrt{\ln(2 * 3)/accessCount}$
    $\eta_{pre} = \eta$
else
    $\eta = \sqrt{\ln(2 * accessCount)/accessCount}$
for ($i$ in LRU, LFU) do
    $v_i = \frac{w_i^{\frac{\eta}{\eta_{pre}}} \exp(-\eta\ell_i)}{\sum_{j=LRU}^{LFU} w_j^{\frac{\eta}{\eta_{pre}}} \exp(-\eta\ell_j)}$    (4.3a)
    $w_i = \frac{\alpha}{2} + (1-\alpha)v_i$    (4.3b)
$\eta_{pre} = \eta$
$w_{LRU} = w_{LRU} / (w_{LRU} + w_{LFU})$
$w_{LRU} = 1 - w_{LFU}$
```

above assumption, the performance of tracking best expert is guaranteed in the real cache setup.

4.3.4 Read-write Discriminated Eviction Policy

Since ExpCache maintains two caches (i.e., R and W), when an eviction is required, ExpCache needs to select a victim from either R or W based on the input eviction policy. In order to reduce the number of dirty pages writing back to storage meanwhile remaining the recency and frequency of workloads, ExpCache sets evicting a page from R having a high priority while a certain condition is met as shown in Algorithm 4. The

Algorithm 4 : EvictVictim(eviction policy)

```
Input: eviction policy: LRU or LFU. Note, T[p]: the timestamp of page p. F[p]: the access frequency of page p. TF: tolerance factor.
Output: victim.
if (eviction policy == LRU) then
    if (R.Empty() or W.Empty()) then
        victim = R.Empty() ? LRU(W) :LRU(R)
        R.Empty() ? W.Delete(LRU(W)) : R.Delete(LRU(R))
    else
        if (T[LRU(R)] < T[LRU(W)] or T[LRU(R)] - T[LRU(W)] ≤ L*TF) then
            victim = LRU(R)
            R.Delete(LRU(R))
        else
            victim = LRU(W)
            W.Delete(LRU(W))
else
    if (R.Empty() or W.Empty()) then
        victim = R.Empty() ? LFU(W) : LFU(R)
        R.Empty() ? W.Delete(LFU(W)) : R.Delete(LFU(R))
    else
        if (F[LFU(R)] != F[LFU(W)]) then
            victim = F[LFU(R)] < F[LFU(W)] ? LFU(R) : LRU(W)
            F[LFU(R)] < F[LFU(W)] ? R.Delete(LFU(R)) : W.Delete(LRU(W))
        else
            if (T[LFU(R)] ≤ T[LFU(W)] or T[LFU(R)] - T[LFU(W)] ≤ L*TF) then
                victim = LFU(R)
                R.Delete(LFU(R))
            else
                victim = LFU(W)
                W.Delete(LFU(W))
return victim
```

input of Algorithm 4 is an eviction policy. The output is the victim page ID. This ID is returned to Algorithm2 for inserting it to G_{LRU} and G_{LFU}.

If LRU is selected, ExpCache checks the size of R and W first. If one of them is empty, ExpCache evicts the LRU page from the non-empty cache. If none of them is empty, ExpCache compares the recency information between the LRU pages in R (i.e., *T[LRU(R)]*) and W (i.e., *T[LRU(W)]*). If *T[LRU(R)]* is smaller than *T[LRU(W)]* or the access gap between *T[LRU(R)]* and *T[LRU(W)]* is smaller than *ToleranceFactor*L*, ExpCache evicts the LRU page from R. Otherwise, ExpCache evicts the LRU page from W. *ToleranceFactor(TF)* is a tuning parameter with a range of 0% to 100%. In the proposed scheme, we set *TF* to an empirical value of 10%. The performance influence of other *TF* values are investigated in Section 4.4.4.

Similarly, if LFU is selected, ExpCache checks the size of R and W first. If one of them is empty, ExpCache evicts an LFU page from the non-empty cache. If none of them is empty, ExpCache always evicts the page whose access frequency is smaller. If the two eviction candidates have the same access frequency, ExpCache then compares their recency information using the same criteria as LRU eviction policy employed.

4.4 System Evaluation

In this section, we evaluate the effectiveness of our purposed design. Four baseline caching polices, including LRU, LFU, LeCaR [96], and H-ARC [76], are used for the comparison. Two types of workloads, including MSR [137] and FIU [175] as shown in Table 4.1, are used for testing those cache polices. In the experiment, we configure the page size to be 4KB. A custom-built simulator is developed to emulate the behaviors of each cache algorithm. It processes the requests inside each trace file in time series manner.

We use the same metrics from H-ARC [76] to evaluate the effectiveness of the tested cache algorithms, **Hit Ratio** and **Write Count**. Hit Ratio equals the sum of read hits and write hits divided by the the number total requests. The value of Hit Ratio sits between 0 and 1. Write Count indicates the number of dirty pages evicted from NVM to storage. In the experiment we define the NVM cache size as the ratio between the number of pages in NVM and the number of unique pages within each workload (i.e.,

Table 4.1: Characteristics of experimental workloads

Trace Name	Total Requests	Unique Pages	Read Write Ratio
MSR: hm_0	8,872,768	703,128	1:2.08
MSR: rsrch_0	3,253,639	107,333	1:7.92
MSR: src1_2	13,953,873	500,529	1:4.97
MSR: src2_0	2,957,747	223,913	1:7.14
MSR: ts_0	4,181,323	276,748	1:2.85
MSR: usr_0	12,764,783	645,348	1:0.38
FIU: online	5,700,499	196,811	1:2.83
FIU: webmail	22,089,973	751,480	1:3.88
FIU: webmailonline	14,294,158	549,174	1:3.59
FIU: webusers	8,111,037	167,938	1:5.86

cache size/workload size). We set the cache size to 0.5%, 1%, 5%, 10%, and 20%.

4.4.1 Results Analysis: Hit Ratio

Figure 4.3 shows the cache hit ratio comparison between ExpCache and other baselines using formula: *ExpCache hit ratio - baseline hit ratio*. From the results, ExpCache outperforms all baselines in the most cases and exhibits two advantages:

1. **Higher performance**: in general, ExpCache delivers much higher performance than other baselines as shown in Figure 4.3. For instance, for *rsrch_0* with the cache size of 20%, ExpCache delivers up to 0.21 higher cache hit ratio than other baselines. Similar observations can be found in *webmail*, *webmailonline*, and *webusers* with the cache size of 20%.

Moreover, for some workloads that single-factor based caching policies (e.g., LFU) can perform well, ExpCache is capable of delivering similar performance as them. For example, for *online* with the cache size of 5%, and *webusers* with the cache size of 10%, ExpCache can achieve similar performance as LFU while LeCaR and H-ARC provide a 0.02 lower cache hit ratio than LFU.

2. **More stable performance**: the performance of prior caching policies varies a lot as workload or cache size is changed. For instance, as shown in Figure 4.3, for *rsrch_0*

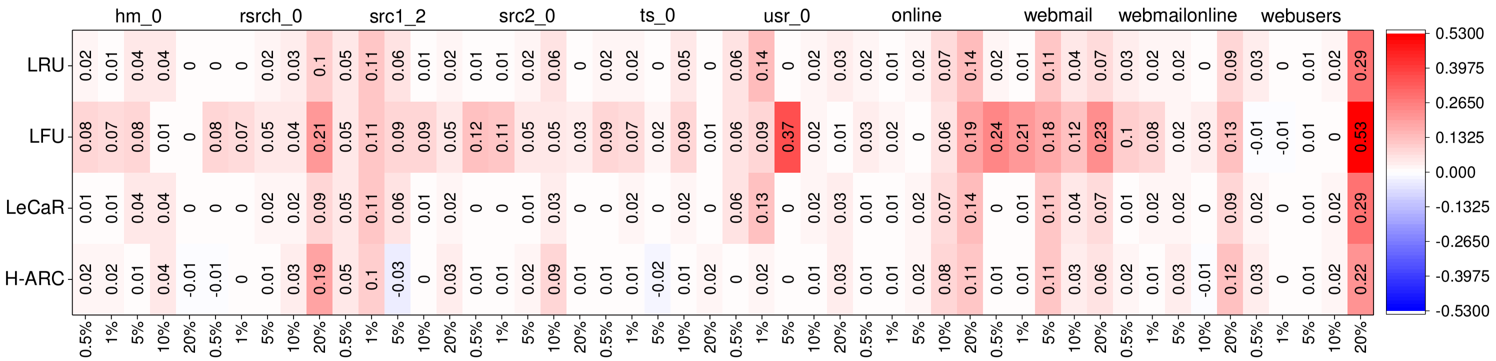


Figure 4.3: Hit Ratio comparison between ExpCache and other baselines. The values inside each cell are computed using formula: *ExpCache hit ratio - baseline hit ratio*. The red color indicates ExpCache outperforms the baseline. The blue color indicates ExpCache has worse performance than the baseline. The white color indicates there is no significant performance difference.

with the cache size of 20%, LeCaR delivers 0.1 higher cache hit ratio than H-ARC. However, for *usr_0* with the cache size of 1%, LeCaR provides even 0.11 lower cache hit ratio than that of H-ARC. In contrast, ExpCache has a more stable performance as workloads and cache sizes are changed as indicated in Figure 4.3. Moreover, although ExpCache has lower cache hit ratio in few cases, the performance gap between ExpCache and the best one for those cases is quite small. For example, for *src1_2* with the cache size of 5%, the ExpCache only has 0.03 lower cache hit ratio compared with the best algorithm, H-ARC.

We attribute the high performance of ExpCache to two factors, including employing the online expert algorithm for tracking workload characteristics and using a new design cache architecture for keeping both recency and frequency information of each accessed page. On the contrary, because only considering the one eviction factor but overlooking the other or although considering both recency and frequency eviction factors but using an improper cache architecture, the tested baseline caching policies provides worse performance than ExpCache in most cases.

4.4.2 Results Analysis: Write Count

Figure 4.4 quantifies the improvement of ExpCache relative to other baselines based on the metric write count using formula: *baseline write count / ExpCache write count*. From the results, ExpCache has smaller write count than other baselines in most of cases. Especially, for *webusers* with the cache size of 20%, ExpCache delivers more than 8X less write count than all baselines.

Another interesting observation is that for most workloads, including rsrch_0, src1_2, online, webmail, webmailonline, and webusers, as we continuously increase the cache size, the write count difference between ExpCache and other baselines is increased. This is because ExpCache sets a high priority to evicting a victim from R when the recency difference between the eviction candidates inside W and R is smaller than *ToleranceFactor*L*. Since *ToleranceFactor* is a constant, the larger the cache size L is, it is harder to evict a dirty page from NVM. As a result, the larger cache sizes produce a smaller write count.

We attribute the performance advantage of ExpCache on write count to two factors. First, ExpCache has higher hit ratios than others, which means the number of

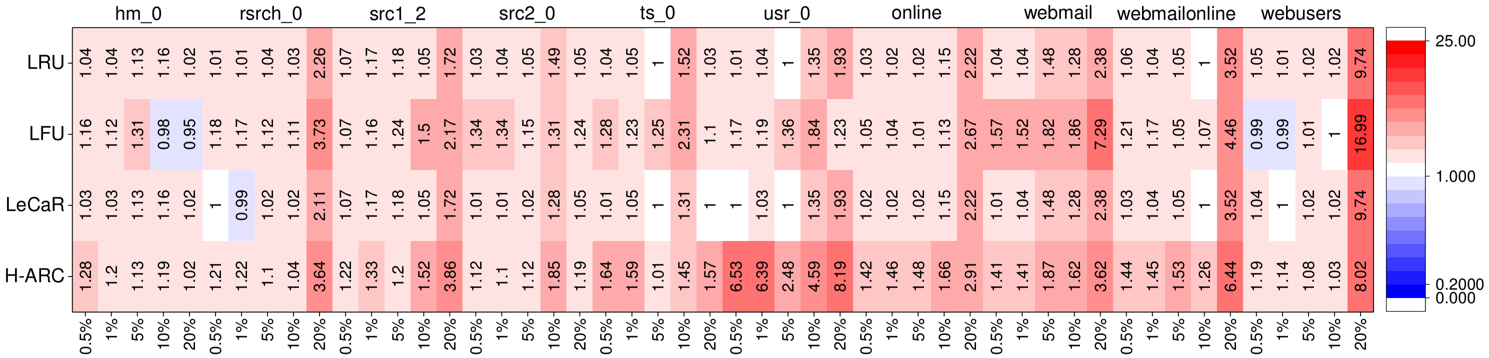


Figure 4.4: Write Count comparison between ExpCache and other baselines. The value inside each cell records the write count difference by formula: *baseline write count / ExpCache write count.* The red color indicates ExpCache has less write count other the baseline. The blue color indicates ExpCache has higher write count than the baseline. The white color indicates there is no significant performance difference.

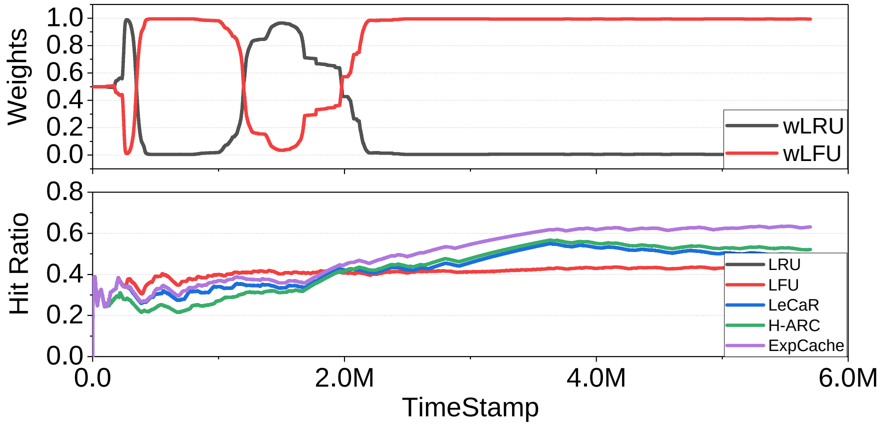
replacements/eviction for both clean pages (i.e., pages in read cache) and dirty pages (i.e., pages in write cache) for ExpCache is smaller. Second, ExpCache sets evicting the clean pages to a high priority while a replacement is required. Thus, the dirty pages have a higher possibility to be preserved in NVM to decrease the write count.

4.4.3 Results Analysis: Phase Change of ExpCache

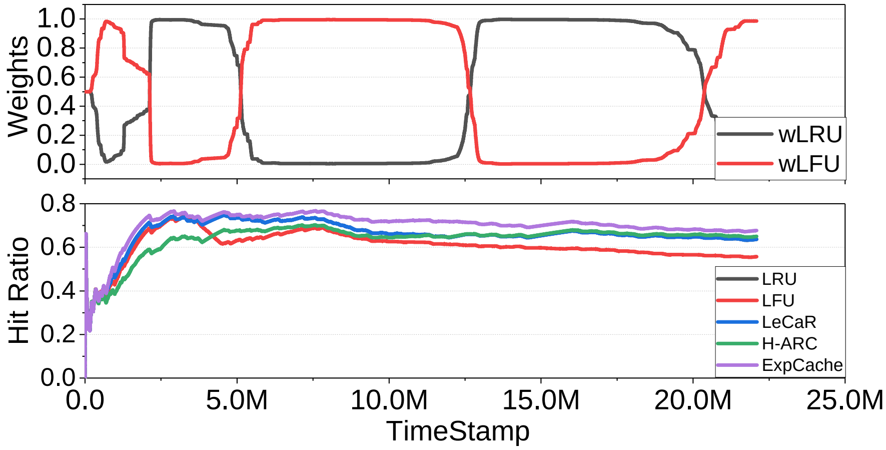
In this subsection, we study the phase changes of ExpCache over using *webmail* and *online* workloads. We also plot the cache hit ratios of other cache policies for comparison. As Figure 4.5 shows, the LRU expert weight, (w_{LRU}), and LFU expert weight, (w_{LFU}), of ExpCache continuously vary to track workload characteristics. Using those frequently updated weights, ExpCache selects a proper replacement policy while eviction is required. As a result, ExpCache delivers the highest performance most time compared to other baselines.

The other advantage of ExpCache is its capability to correct an improper replacement decision using an online expert algorithm. As Figure 4.5a shows, ExpCache only provides the second-highest cache hit ratio between the times 200,000 to 2,000,000. This is because the online expert algorithm needs a certain time to warm up its learning pattern. In such duration, ExpCache may make some improper decisions and is therefore producing a low performance. However, once ExpCache is well trained, its performance is boomed. As Figure 4.5a shows, after timestamp 2,000,000, ExpCache achieves the highest performance compared to others and delivers 0.11 higher cache hit ratio compared to the second-best policy H-ARC.

Note that the variation of w_{LRU} and w_{LFU} of ExpCache algorithm represents a trace preference based on the pages within the cache. Because the cached pages are managed by both LRU and LFU policies and those two polices affect each other. Both the cached pages and their corresponding organization within ExpCache algorithm are different from those managed by pure LRU or LFU policy. Thus, the variation w_{LRU} and w_{LFU} are uncorrelated the cache hit ratio of the pure LRU and LFU policies. This explains why the LFU cache hit ratio drops to the lowest one after timestamp 2,000,000 in Figure 4.5a, but the w_{LRU} of ExpCache is still high and approaches to one.



(a) online workload with cache size 20%. Note, LRU and LeCaR cache hit ratio are overlapped.

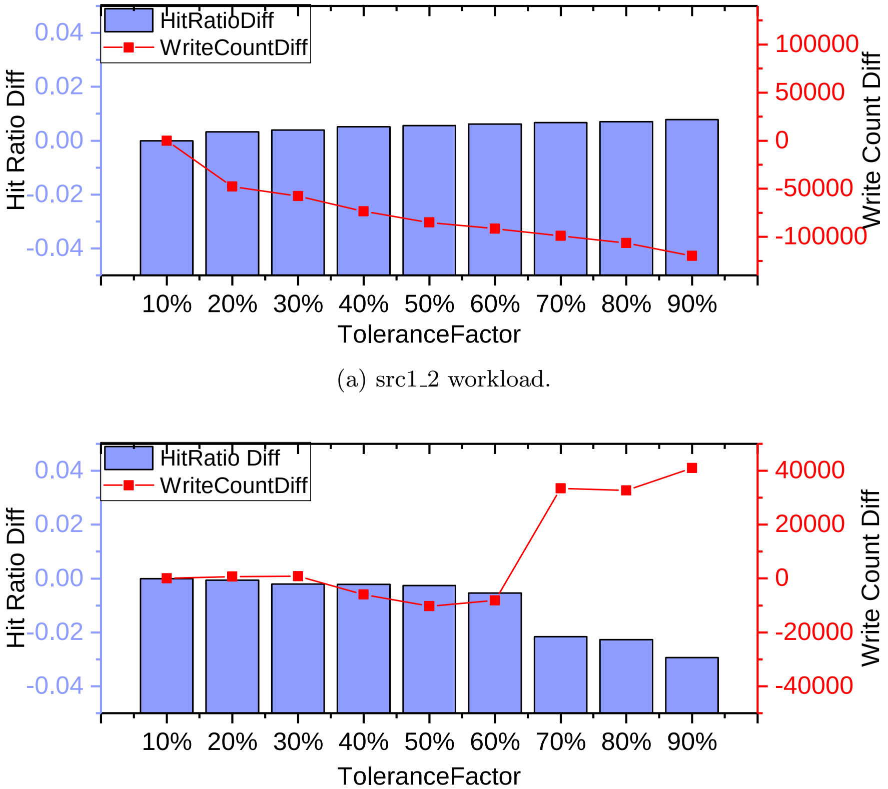


(b) webmail workload with cache size 10%. Note, LRU and LeCaR cache hit ratio are overlapped.

Figure 4.5: Phase change of ExpCache and different caching policies over time. The upper one shows the variations of two weights in ExpCache. The lower one shows the variations of hit ratios for different caching policies.

4.4.4 Results Analysis: Impacts of the ToleranceFactor

In this subsection, we explore how different values of *ToleranceFactor* affect the performance of ExpCache using rsrch_0 and src1_2 workloads. The cache size sets to 5%. The value of *ToleranceFactor* varies from 10% to 90% with an incremental step of 10%.



(a) src1_2 workload.

(b) rsrch_0 workload.

Figure 4.6: ExpCache performance variation as varying the value of *ToleranceFactor*. (10% as the reference to compute the difference.

Both the variations of cache hit ratio and write count are recorded in Figure 4.6 using the results that *ToleranceFactor* equals to 10% as the baseline (i.e. the performance of ExpCache on certain *ToleranceFactor* - the performance ExpCache on *ToleranceFactor* equals 10%).

As shown in Figure 4.6, the impact of different *ToleranceFactor* values on the ExpCache performance is uncertain and depends on workload characteristics. For instance, for src1_2, as illustrated in Figure 4.6a, when continuously increasing *ToleranceFactor*, the hit ratio of ExpCache remains similar. On the other hand, the larger *ToleranceFactor* eliminates the number of write count to the storage for src1_2. This is because the higher *ToleranceFactor* leads ExpCache to have higher chances to evict a page from the read cache, *R*, instead of the write cache, *W*, while a replacement is required. As a result, the dirty pages writing back to the storage are declined.

However, the larger *ToleranceFactor* does not always bring in positive results to ExpCache. As shown in Figure 4.6b, for rsrch_0, as we increase *ToleranceFactor*, the cache hit ratio of ExpCache drops 0.03. In addition, after *ToleranceFactor* exceeds 50%, as continuously raising its value, the write count increases a lot. This happens because the high *ToleranceFactor* may break the collected recency and frequency information, resulting in the degradation of the overall cache hit ratio. In addition, the low cache hit ratio means that more replacements are required, which increases the chance to write a dirty page back to the storage. Considering the above negative influence of large *ToleranceFactor* on some workloads, in the real system, we suggest to set *ToleranceFactor* to a small value (e.g. 10%).

4.4.5 Space and Computation Overheads Analysis

ExpCache maintains 2L pieces of metadata where L is cache size. It uses L units to track items reside in the real cache and L additional units to track items in the ghost. This is roughly the same as state-of-art algorithms such as LeCaR and H-ARC which each maintains additional L pieces of metadata to track items in the history. For a 100GB NVM, suppose a block size is 4KB, then the memory space required by ExpCache for maintaining its metadata is 100GB/4KB*2*(4bytes/blockID + 4bytes/block_freq + 4bytes/block_timestamps)≈0.6GB which is negligible compared to the NVM capacity. In the real time, since G_{LRU} and G_{LFU} may contains the same block IDs and their metadata can be merge for saving space, the memory overhead we shown on the above is upper bound of ExpCache algorithm. The computation overhead of ExpCache comes from updating its ghost caches and is bound by $log_2(L/2)$. Overall, considering the significant performance improvement of the ExpCache on both cache hit ratio and write count, the above small overhead is acceptable.

4.5 Conclusion

In this work, we propose an ExpCache algorithm for computer systems that use NVM as the main memory. By considering recency and frequency factors and the non-volatility characteristics of the NVM device, we split the whole cache into two small caches (i.e., a write cache and a read cache) for preserving different types of requests. Each cache is managed by both LRU and LFU policies for balancing recency and frequency factors. By employing the online Expert algorithm, ExpCache adaptively selects a proper replacement algorithm for managing the cached pages based on workloads' characteristics. Intensive experiments are conducted to evaluate the effectiveness of our proposed design. The experimental results show that ExpCache delivers significant performance improvement on both cache hit ratio and write count compared with other baselines.

Chapter 5

Conclusion and Future Work

Coming into 21st century, data volume increases exponentially. Through analyzing the collected big data, many types of information can be extracted for decision making, market analysis, and making prediction. While big data is prominent, the rapid data growth poses many challenges to modern storage systems. One fundamental problem for large scale data storage is how to store and retrieve data efficiently. In this thesis, we approach the above challenge by developing new types of storage systems using emerging storage and memory devices.

5.1 Conclusion

In Chapter 2, we carry out an in-depth performance evaluation of the NVM block device. We propose multiple custom-designed micro-benchmarks for exploring the intrinsic characteristics of NVM block device, including basic I/O performance, internal parallelism, performance consistency issue, byte addressability, the elimination of write-driven garbage collection, the read disturb issue, and the tail latency problem. Besides, we comprehensively study the implications of NVM block device to modern database application. We find NVM block device provides asymmetric performance accelerations to different types of queries. And the potential reasons behind those results are also analyzed. Finally, we implement the existing hybrid storage database schema on the NVM block device and NAND SSD hybrid storage architecture. Several key factors that can impact the hybrid database system's performance are pointed out.

In Chapter 3, we investigate prior researches of hybrid storage database system. We find past works failed to produce high performance for dynamically changed workloads and migrated many unnecessary data blocks between storage and block cache devices which consumes extra power. To resolve the limitations of prior works, we present a new hybrid storage database system, named as HeuristicDB, that uses NVM block device as an extension of database buffer pool. By considering the unique performance behaviors of NVM block device and the characteristics of database block requests, we propose a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority to guide HeuristicDB manage database block requests. To support system implementation of the proposed rules, we develop four assistant programs, including a query profile queue, an eviction and demotion program, a table access pattern detector, and a page placement controller. The experimental results show HeuristicDB delivers up to 75% higher performance than prior works for both OLAP and OLTP workloads.

In Chapter 4, we study existing cache replacement polices and find ignoring the write operations led prior cache algorithms delivers sub-optimal performance for NVM-based system. In addition, we notice prior works suffer a sever performance degradation while the access pattern of a workload mismatches with their internal pre-defined replacement rules. To handle the above limitations, we propose a new online-learning based cache replacement policy for NVM-based system, named as ExpCache. In the system, two candidates, including a recency dominated replacement policy and a frequency dominated replacement, locate in the selection pool. With employing the online Expert algorithm, ExpCache adaptively selects a proper replacement algorithm for managing the cached pages based on workloads' access pattern. The experimental results show that ExpCache delivers significant performance improvement on both cache hit ratio and write count compared with other baselines.

5.2 Future Work

With the rapidly growth of large scale data, the demand for high performance storage system becomes more critical than before. As many types of emerging storage devices were released recently, including NVM, NVM block device, large capacity SSD, and

SMR drives, it is time to re-think the storage system design with considering the characteristics of those emerging storage technologies. Based on the works covered by this thesis, we discuss the following potential future works.

5.2.1 NVM Oriented Database System

In order to deliver high performance meanwhile maintain data persistency, conventional disk-oriented database system is built on the hybrid DRAM and persistent storage architecture. In the system, the DRAM is used as a global cache to buffer the most frequently accessed data to deliver high performance. And a database application periodically writes the data reside in the DRAM back into persistent storage (e.g. NAND SSD, HDD) to avoid data loss while power is off. This kind of the design has several disadvantages. First, while a cache miss occurs on the DRAM, a database application must fetch the required data block from the laggard storage, which degrades database performance dramatically. Second, in order to keep data consistency, a database application needs to record all data operations into binary log files and flush them into storage later, which brings extra overhead. Finally, while a recovery is needed after system failure, a database application replays all operations recorded in the binary log files. The entire recovery process is time consuming.

As the first commercial NVM (i.e., Optane DIMM) is released recently, it is the time to consider the NVM-oriented database system [176, 177, 178] design. In thus system, the NVM is used as a persistent memory to replace both DRAM and persistent storage devices within the conventional storage architecture. Since all data resides in the NVM is persistent, there is no need to maintain the binary log files anymore. In addition, because NVM delivers unprecedented performance improvement compared with conventional storage devices, the NVM-oriented database system is expected to provides much higher performance than disk-oriented database system. However, designing an efficient data structure that fits the unique properties of the NVM-oriented database system is more challenge and would be one potential research direction in the future.

5.2.2 Database on Warm Storage With Emerging Storage Devices

The trend of modern database systems is to evolve towards warm storage (i.e., computation and storage decoupling) for high accessibility. Some recent examples are Amazon Aura and Azure SQL Database [23, 24]. This model has multiple advantages. First, the computation node and storage node scale independently, which allows system administrators to scale either computation or storage node on demand. Second, it allows more efficient hardware usage. Finally, compared with traditional database system, the database on warm storage has faster recovery time since the data is always consistent.

While database on warm storage is promising, it is not without its challenges. Since database server communicates with storage node through a network, database on warm storage usually has higher access latency than conventional database systems that access data locally. In order to improve its performance, adding a cache layer becomes to one potential direction in the future. With considering the data consistency issue between the cache layer and storage node, both NVM and NVM block device are the good candidates to be used as such cache device. In addition, designing an efficient cache schema to manage the data within thus cache device is an other research direction.

5.2.3 Hierarchical Storage System Using NVM

As we have discussed in Chapter 4, NVM-based system has multiple advantages compared with DRAM-based system, including larger capacity, competitive performance, and there is no need to write updated pages back to the storage. However, a recent research [55] shown NVM suffers a severe performance degradation as the number of thread increases. Designing a hierarchical storage system that uses DRAM as a cache for NVM becomes to one potential approach to mitigate the performance degradation issues of NVM. In thus system, the DRAM caches the hottest data and delivers high performance as thread count increasing, the NVM acts as a large capacity second level cache to hold the warm data, and the persistent storage preserves all data elements. However, designing an efficient algorithm to manage the data flow within thus hierarchical storage architecture is more complex and could be the future work.

5.3 Summary

In summary, in this thesis, we mainly focus on the high performance storage system design using emerging storage technologies, including NVM and NVM block device. The NVM block device performance characterization, the hybrid storage database system, and the online learning based cache replacement policy for NVM are studied and proposed in this thesis. The experimental results show that the proposed schema enhanced modern storage system significantly.

References

[1] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. 2016.

[2] Oracle Database. `https://www.oracle.com/database/`.

[3] MySQL. `https://www.mysql.com/`.

[4] IBM Db2. `https://www.ibm.com/analytics/db2`.

[5] MariaDB. `https://mariadb.org/`.

[6] PostgreSQL. `https://www.postgresql.org/`.

[7] SQLite. `https://www.sqlite.org/index.html`.

[8] LevelDB. `https://github.com/google/leveldb`.

[9] RocksDB. `http://rocksdb.org/`.

[10] Memcached. `https://memcached.org/`.

[11] Redis. `https://redis.io/`.

[12] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 159–171, 2019.

[13] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[14] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[15] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled disks. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 105–120, 2017.

[16] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[19] Google Cloud. `https://cloud.google.com/storage`.

[20] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 380–383. IEEE, 2010.

[21] Wenying Zeng, Yuelong Zhao, Kairi Ou, and Wei Song. Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 1044–1048, 2009.

[22] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240, 2010.

[23] Amazon Aurora. `https://aws.amazon.com/rds/aurora/?aurora-whats-new.sort-by=item.additionalFields.postDateTime&aurora-whats-new.sort-order=desc`.

[24] Azure SQL Database. `https://azure.microsoft.com/en-us/products/azure-sql/database/`.

[25] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[26] Jinfeng Yang, Bingzhe Li, and David J Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(1):1–28, 2020.

[27] Kyungjune Son, Kyungjun Cho, Subin Kim, Gapyeol Park, Kyunghwan Song, and Journ Kim. Modeling and signal integrity analysis of 3d xpoint memory cells and interconnections with memory size variations during read operation. In *2018 IEEE Symposium on Electromagnetic Compatibility, Signal Integrity and Power Integrity (EMC, SI & PI)*, pages 223–227. IEEE, 2018.

[28] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[29] Stefan Lai. Current status of the phase change memory and its future. In *IEEE International Electron Devices Meeting 2003*, pages 10–1. IEEE, 2003.

[30] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 24–33, 2009.

[31] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, 28(2):223–262, 2010.

[32] Michael Kund, Gerhard Beitel, C-U Pinnow, Thomas Rohr, Jorg Schumann, Ralf Symanczyk, K Ufert, and Gerhard Muller. Conductive bridging ram (cbram): An emerging non-volatile memory technology scalable to sub 20nm. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 754–757. IEEE, 2005.

[33] Ralf Symanczyk, Jan Keller, Michael Kund, Gerhard Muller, Bernhard Ruf, Paul-Henri Albarede, Serge Bournat, Laurent Bouteille, Alexander Duch, et al. Conductive bridging memory development from single cells to 2mbit memory arrays. In *2007 Non-Volatile Memory Technology Symposium*, pages 71–75. IEEE, 2007.

[34] John R Jameson, Philippe Blanchard, John Dinh, Nathan Gonzales, Vasudevan Gopalakrishnan, Berenice Guichet, Shane Hollmer, Sue Hsu, Gideon Intrater, Deepak Kamalanathan, et al. Conductive bridging ram (cbram): then, now, and tomorrow. *ECS Transactions*, 75(5):41, 2016.

[35] Roger Wood, Mason Williams, Aleksandar Kavcic, and Jim Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics*, 45(2):917–923, 2009.

[36] Garth Gibson and Milo Polte. Directions for shingled-write and twodimensional magnetic recording system architectures: Synergies with solid-state disks. *Parallel Data Lab, Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-PDL-09-014*, 2009.

[37] Mark H Kryder, Edward C Gage, Terry W McDaniel, William A Challener, Robert E Rottmayer, Ganping Ju, Yiao-Tee Hsia, and M Fatih Erden. Heat assisted magnetic recording. *Proceedings of the IEEE*, 96(11):1810–1835, 2008.

[38] Robert E Rottmayer, Sharat Batra, Dorothea Buechel, William A Challener, Julius Hohlfeld, Yukiko Kubota, Lei Li, Bin Lu, Christophe Mihalcea, Keith Mountfield, et al. Heat-assisted magnetic recording. *IEEE Transactions on Magnetics*, 42(10):2417–2421, 2006.

[39] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories.* Springer Science & Business Media, 2010.

[40] Jiyoung Kim, Augustin J Hong, Sung Min Kim, Emil B Song, Jeung Hun Park, Jeonghee Han, Siyoung Choi, Deahyun Jang, Joo-Tae Moon, and Kang L Wang. Novel vertical-stacked-array-transistor (vsat) for ultra-high-density and cost-effective nand flash memory devices and ssd (solid state drive). In *2009 Symposium on VLSI Technology*, pages 186–187. IEEE, 2009.

[41] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.

[42] Ken Takeuchi. Novel co-design of nand flash memory and nand flash controller circuits for sub-30 nm low-power high-speed solid-state drives (ssd). *IEEE Journal of solid-state circuits*, 44(4):1227–1234, 2009.

[43] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107, 2011.

[44] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[46] Product Brief: Intel Optane SSD DC P4800X Series. `https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html`, 2018. [Online].

[47] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[48] Matthew J Koop, Wei Huang, Karthik Gopalakrishnan, and Dhabaleswar K Panda. Performance analysis and evaluation of pcie 2.0 and quad-data rate infiniband. In *2008 16th IEEE Symposium on High Performance Interconnects*, pages 85–92. IEEE, 2008.

[49] J Anderson, K Bauer, A Borga, H Boterenbrood, H Chen, K Chen, G Drake, M Dönszelmann, D Francis, D Guest, et al. Felix: a pcie based high-throughput approach for interfacing front-end and trigger electronics in the atlas upgrade framework. *Journal of Instrumentation*, 11(12):C12023, 2016.

[50] Masakazu Kawamoto. Hdd interface technologies. *Fujitsu scientific and technical journal*, 42(1):78–92, 2006.

[51] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 57, 2008.

[52] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[53] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[54] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39(2):29–36, 2019.

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.

[56] Edgar F Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[57] Edgar F Codd. Relational database: A practical foundation for productivity. In *Readings in Artificial Intelligence and Databases*, pages 60–68. Elsevier, 1989.

[58] Paolo Atzeni and Valeria De Antonellis. *Relational database theory.* Benjamin-Cummings Publishing Co., Inc., 1993.

[59] Paris C Kanellakis. Elements of relational database theory. In *Formal models and semantics*, pages 1073–1156. Elsevier, 1990.

[60] Jan L Harrington. *Relational database design and implementation.* Morgan Kaufmann, 2016.

[61] Raymond Reiter. Towards a logical reconstruction of relational database theory. In *Readings in Artificial Intelligence and Databases*, pages 301–327. Elsevier, 1989.

[62] Redshift. `https://aws.amazon.com/redshift/?nc=sn&loc=0&whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc`.

[63] BigQuery. `https://cloud.google.com/bigquery`.

[64] Snowflake. `https://www.snowflake.com/`.

[65] Charles Andrew Bell and Sven Sandberg. *Expert MySQL*, volume 3. Springer, 2012.

[66] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016.

[67] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[68] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[69] Jinfeng Yang and David J Lilja. Reducing relational database performance bottlenecks using 3d xpoint storage technology. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1804–1808. IEEE, 2018.

[70] Intel Corporation. Intel Optane DC Persistent Memory. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`.

[71] Ziqi Fan, Alireza Haghdoost, David HC Du, and Doug Voigt. Non-volatile memory based cache replacement policy.

[72] Ziqi Fan, Alireza Haghdoost, David HC Du, and Doug Voigt. I/o-cache: A non-volatile memory based buffer cache policy to improve storage performance. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 102–111. IEEE, 2015.

[73] Z Fan, F Wu, D Park, J Diehl, D Voigt, and Hibachi Du D H. A cooperative hybrid cache with nvram and dram for storage arrays. In *Proc. the 33rd IEEE Symposium on Mass Storage Systems and Technologies*, 2017.

[74] Ziqi Fan, Fenggang Wu, Jim Diehl, David HC Du, and Doug Voigt. Cdbb: an nvram-based burst buffer coordination system for parallel file systems. In *Proceedings of the High Performance Computing Symposium*, pages 1–12, 2018.

[75] Nimrod Megiddo and Dharmendra S Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[76] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[77] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[78] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.

[79] Bingzhe Li, Chunhua Deng, Jinfeng Yang, David Lilja, Bo Yuan, and David Du. Haml-ssd: A hardware accelerated hotness-aware machine learning based ssd management. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[80] Wonkyung Kang and Sungjoo Yoo. q-value prediction for reinforcement learning assisted garbage collection to reduce long tail latency in ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2240–2253, 2019.

[81] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[82] Anthony McGregor, Mark Hall, Perry Lorier, and James Brunskill. Flow clustering using machine learning techniques. In *International workshop on passive and active network measurement*, pages 205–214. Springer, 2004.

[83] Mario Bkassiny, Yang Li, and Sudharman K Jayaweera. A survey on machine-learning techniques in cognitive radios. *IEEE Communications Surveys & Tutorials*, 15(3):1136–1159, 2012.

[84] Duc T Pham and Ashraf A Afify. Machine-learning techniques and their applications in manufacturing. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 219(5):395–412, 2005.

[85] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[86] Reza Salkhordeh, Mostafa Hadizadeh, and Hossein Asadi. An efficient hybrid i/o caching architecture using heterogeneous ssds. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1238–1250, 2018.

[87] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *FAST*, volume 5, pages 9–9, 2005.

[88] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[89] Paul Saab Mohan Srinivasan. Flashcache. `https://github.com/facebookarchive/flashcache`, 2010.

[90] Teradata virtual storage system. `http://assets.teradata.com/resourceCenter/downloads/Brochures/EB5944.pdf?processed=1`, 2009.

[91] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. Ssd bufferpool extensions for database systems. *Proceedings of the VLDB Endowment*, 3(1-2):1435–1446, 2010.

[92] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. An object placement advisor for db2 using solid state storage. *Proceedings of the VLDB Endowment*, 2(2):1318–1329, 2009.

[93] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proceedings of the VLDB Endowment*, 5(10):1076–1087, 2012.

[94] Theodore Johnson, Dennis Shasha, et al. 2q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.

[95] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

[96] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[97] Jinfeng Yang, Bingzhe Li, and David J Lilja. Heuristicdb: a hybrid storage database system using a non-volatile memory block device. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[98] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 57, 2008.

[99] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 279–289. ACM, 2009.

[100] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[101] Mark Moshayedi and Patrick Wilkison. Enterprise ssds. *Queue*, 6(4):32–39, 2008.

[102] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[103] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM*

SIGMOD International Conference on Management of data, pages 863–870. ACM, 2009.

[104] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.

[105] Hiroyuki Nakamoto, Daisuke Yamazaki, Takuji Yamamoto, Hajime Kurata, Satoshi Yamada, Kenji Mukaida, Tsuzumi Ninomiya, Takashi Ohkawa, Shoichi Masui, and Kunihiko Gotoh. A passive uhf rf identification cmos tag ic using ferroelectric ram in 0.35-um technology. *IEEE Journal of Solid-State Circuits*, 42(1):101–110, 2007.

[106] World's Most Responsive Data Center SSD. `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-brief.pdf`, 2018. [Online].

[107] TPC Benchmark H. `http://www.tpc.org/tpch/`, 2001.

[108] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware i/o management for solid state disks (ssds). *IEEE Transactions on Computers*, 61(5):636–649, 2012.

[109] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[110] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2013.

[111] Seon-yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based ssds. *IEEE Computer Architecture Letters*, 9(1):9–12, 2010.

[112] Seongwook Jin, Jaehong Kim, Jaegeuk Kim, Jaehyuk Huh, and Seungryoul Maeng. Sector log: fine-grained storage management for solid state drives. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 360–367. ACM, 2011.

[113] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[114] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock ftl: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):40, 2010.

[115] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170. ACM, 2006.

[116] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[117] Holloway H Frost, Charles J Camp, Timothy J Fisher, James A Fuxa, and Lance W Shelton. Efficient reduction of read disturb errors in nand flash memory, October 19 2010. US Patent 7,818,525.

[118] Prashant J Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K Qureshi. Reducing read latency of phase change memory via early read and turbo read. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 309–319. IEEE, 2015.

[119] Simone Lavizzari, Daniele Ielmini, Deepak Sharma, and ANDREA LEONARDO Lacaita. Transient effects of delay, switching and recovery in phase change memory

(pcm) devices. In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*, pages 1–4. IEEE, 2008.

[120] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277. IEEE, 2011.

[121] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa linux symposium*, volume 2001, 2001.

[122] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *USENIX Annual Technical Conference, General Track*, pages 89–104, 2005.

[123] Yaffs. `https://yaffs.net/`, 2012.

[124] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[125] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[126] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.

[127] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, volume 13, 2013.

[128] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[129] Jiachen Zhang, Peng Li, Bo Liu, Trent G Marbach, Xiaoguang Liu, and Gang Wang. Performance analysis of 3d xpoint ssds in virtualized and non-virtualized

environments. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–10. IEEE, 2018.

[130] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

[131] Intel SSD DC P3700 Series. `https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p3700-series.html`, 2014. [Online].

[132] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304. ACM, 2009.

[133] Flexible I/O Tester. `https://linux.die.net/man/1/fio`, 2010. [Online].

[134] H Strass. An introduction to nvme, 2016.

[135] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, volume 11, pages 163–176, 2011.

[136] Mingzhe Hao, Gokul Soundararajan, Deepak R Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *FAST*, pages 263–276, 2016.

[137] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[138] Alan D Brunelle. Block i/o layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 2006.

[139] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. Turbocharging dbms buffer pool using ssds. In

Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pages 1113–1124. ACM, 2011.

[140] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

[141] Jiaxin Ou, Jiwu Shu, Youyou Lu, Letian Yi, and Wei Wang. Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 787–796. IEEE, 2014.

[142] Hiroki Fujii, Kousuke Miyaji, Koh Johguchi, Kazuhide Higuchi, Chao Sun, and Ken Takeuchi. x11 performance increase, x6. 9 endurance enhancement, 93% energy reduction of 3d tsv-integrated hybrid reram/mlc nand ssds by data fragmentation suppression. In *2012 symposium on VLSI circuits (VLSIC)*, pages 134–135. IEEE, 2012.

[143] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of {NAND} flash-based storage systems using dynamic program and erase scaling. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 61–74, 2014.

[144] Tpc benchmark e. `http://www.tpc.org/tpce/`, 2007.

[145] Persistent memory development kit. `https://https://github.com/pmem/pmdk`, 2019. [Online].

[146] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based {KV} stores with matrix container in {NVM}. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 17–31, 2020.

[147] Qian Zhao and Hao Chen. Pci express interface, March 2 2010. US Patent 7,673,092.

[148] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[149] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*, 2019.

[150] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[151] Tapio Lahdenmaki and Mike Leach. *Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al.* John Wiley & Sons, 2005.

[152] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. In *Proceedings 11th Australasian Database Conference. ADC 2000 (Cat. No. PR00528)*, pages 65–73. IEEE, 2000.

[153] Michael L Brundage and Andrew E Kimball. Query optimizer system and method, December 5 2006. US Patent 7,146,352.

[154] Curtis Neal Boger, John Francis Edwards, Randy Lynn Egan, and Michael S Faunce. Metadata manager for database query optimizer, February 7 2006. US Patent 6,996,556.

[155] Optimizing innodb disk i/o. `https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-diskio.html`, 2020.

[156] Surajit Chaudhuri and Umeshwar Dayal. Data warehousing and olap for decision support. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 507–508, 1997.

[157] Mysql 5.7. `https://dev.mysql.com/doc/refman/5.7/en/`, 2019.

[158] Ryan Bannon, Alvin Chin, Faryaaz Kassam, Andrew Roszko, and Ric Holt. Innodb concrete architecture. *University of Waterloo*, 2002.

[159] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[160] Jerome T Connor, R Douglas Martin, and Les E Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

[161] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with {CACHEUS}. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 341–354, 2021.

[162] Mark Herbster and Manfred K Warmuth. Tracking the best expert. *Machine learning*, 32(2):151–178, 1998.

[163] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research*, 8:2755–2790, 2007.

[164] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[165] Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

[166] Vladimir Vovk. A game of prediction with expert advice. *Journal of Computer and System Sciences*, 56(2):153–173, 1998.

[167] Nicolo Cesa-Bianchi, Yishay Mansour, and Gilles Stoltz. Improved second-order bounds for prediction with expert advice. *Machine Learning*, 66(2):321–352, 2007.

[168] Nicolo Cesa-Bianchi, Yoav Freund, David Haussler, David P Helmbold, Robert E Schapire, and Manfred K Warmuth. How to use expert advice. *Journal of the ACM (JACM)*, 44(3):427–485, 1997.

[169] Elad Hazan and Comandur Seshadhri. Efficient learning algorithms for changing environments. In *Proceedings of the 26th annual international conference on machine learning*, pages 393–400, 2009.

[170] Nicolo Cesa-Bianchi, Pierre Gaillard, Gábor Lugosi, and Gilles Stoltz. A new look at shifting regret. *arXiv preprint arXiv:1202.3323*, 2012.

[171] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[172] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. Lru-wsr: integration of lru and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics*, 54(3):1215–1223, 2008.

[173] Peiquan Jin, Yi Ou, Theo Härder, and Zhi Li. Ad-lru: An efficient buffer replacement algorithm for flash-based databases. *Data & Knowledge Engineering*, 72:83–102, 2012.

[174] Youwei Yuan, Jintao Zhang, Guangjie Han, Gangyong Jia, Lamei Yan, and Wanqing Li. Dpw-lru: An efficient buffer management policy based on dynamic page weight for flash memory in cyber-physical systems. *IEEE Access*, 7:58810–58821, 2019.

[175] SNIA FIU trace. `http://iotta.snia.org/traces/390`.

[176] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[177] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[178] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.