BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

EXPLOITING SSD PROPERTIES TO ENHANCE DATA SYSTEMS PERFORMANCE

by

TARIKUL ISLAM PAPON

M.Sc., Bangladesh University of Engineering and Technology, 2019
B.Sc., Bangladesh University of Engineering and Technology, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2025

© 2025 by
TARIKUL ISLAM PAPON
All rights reserved

Approved by

First Reader

Manos Athanassoulis, PhD
Assistant Professor of Computer Science

Second Reader

Vasiliki Kalavri, PhD
Assistant Professor of Computer Science

Third Reader

Renato Mancuso, PhD
Associate Professor of Computer Science

Fourth Reader

George Kollios, PhD
Professor of Computer Science

This is the true joy in life, being used for a purpose recognized by yourself as a mighty one. Being a force of nature instead of a feverish, selfish little clod of ailments and grievances, complaining that the world will not devote itself to making you happy. I am of the opinion that my life belongs to the whole community and as long as I live, it is my privilege to do for it what I can. I want to be thoroughly used up when I die, for the harder I work, the more I live. Life is no brief candle to me. It is a sort of splendid torch which I have got hold of for the moment and I want to make it burn as brightly as possible before handing it on to future generations.

- George Bernard Shaw

Acknowledgments

Although this thesis bears only my name as the author, it is the result of the support, guidance, and encouragement of many individuals. This work would not have been possible without my advisor, collaborators, teachers, friends, and family, each of whom has played a crucial role in shaping my journey.

First and foremost, I would like to express my deepest gratitude to my advisor, Manos Athanassoulis. He rekindled my passion for research, taught me the intricacies of database and storage systems, and helped me develop the skills to manage time and plan research effectively. His constant encouragement, guidance, and mentorship have been invaluable throughout this journey. He inspired me to be a better researcher and a better teacher. He has been the best role model one could ask for.

I would also like to extend my sincere thanks to Renato Mancuso and his excellent group for our collaboration. Their research rigor and meticulous attention to detail were invaluable learning experiences for me. I am truly grateful for the insights I gained from working with them.

I am also thankful to Vasia Kalavri for her input while preparing the rebuttal for the graph work. Her keen insights were instrumental in the success of that publication.

Renato Mancuso and Vasia Kalavri, as members of my PhD committee, alongside George Kollios, have provided invaluable feedback and support. I sincerely thank them for their time and guidance, which have been crucial to my doctoral research.

I am grateful to Salman Toor and Tianru Zhang for our collaboration. Tianru's dedication and work ethic are qualities I deeply admire.

I was fortunate to collaborate with Subhadeep Sarkar during the early stages of my PhD. His research vision, methodological approach, and attention to detail significantly influenced my doctoral work. His feedback and insights have helped shape my research, and many key decisions throughout this journey would have been

impossible without his input. Beyond our collaboration, Subhadeep and Subarna have become some of my closest friends, and I feel truly fortunate to have them in my life. I express my deepest gratitude to both of them and to my next best friend, who is soon to arrive. I wish them all the happiness in the world.

I sincerely thank Abdul Wasay for the opportunity to intern at Intel Labs. My time there broadened my understanding of machine learning and significantly contributed to the later stages of my thesis. I also extend my gratitude to Manoj Syamala for the opportunity to intern at Microsoft Research, where I had an incredible summer learning about column stores.

The members of the DiSC lab have been a significant part of my PhD journey. I am grateful to Zichen Zhu, Dimitris Staratzis, Ju Hyoung Mun, Andy Huynh, Aneesh Raman, Kostas Karatsenidis, Teona Bagashvili, and Anwesha Saha. Their feedback during lab meetings and reading groups has helped refine my thesis, and for that, I am truly grateful. I thank them and wish them all the best in life. Special thanks to Andy Huynh for suggesting one of the research project names. I have also been fortunate to work with Taishan Chen and Shuo Zhang, whose dedication and effort were instrumental in the success of our CAVE work. Mentoring brilliant minds like them has been one of the most rewarding experiences of my PhD journey.

I am forever indebted to all my teachers. I had the privilege of completing my undergraduate in an outstanding institution, the Department of Computer Science and Engineering at Bangladesh University of Engineering and Technology. I was fortunate to learn from inspiring professors. I also had the honor of studying alongside an exceptional group of brilliant minds who are now excelling in their respective fields across the world. A big shout out to the *Mighty 432*: Lipir Buta, Nati, Lamur Buta, Pato, and Saquib. I am grateful to all my friends, whose companionship and intellectual exchanges have enriched my journey.

I owe everything to my parents, Zahurul Islam and Nasima Islam, for their unwavering love and support. I am incredibly fortunate to have them in my life, and I am deeply grateful for their sacrifices and encouragement, especially during the most challenging times. This thesis is dedicated to my father who passed away in 2023. My eldest brother, Zion, has always been a source of inspiration, teaching me to pursue my dreams fearlessly. My elder brother, Leon, instilled in me a passion for science — he has been my teacher and my friend for a long time. I am blessed to have such incredible brothers who have supported me unconditionally.

There are no words to adequately express my gratitude to the love of my life, my wife, Sadia. She has been by my side for as long as I can remember. I would not be on this path if it was not for her. I would not be who I am if it was not for her. She has been my constant through the past 15 years, always pushing me to grow and inspiring me to step beyond my comfort zone. Thank you for enduring me, for caring for me, for making me laugh, and for being my greatest source of happiness.

Lastly, I gratefully acknowledge the generous funding that supported my research. This work was made possible through grants from the National Science Foundation (IIS-1850202 and IIS-2144547), a Facebook Faculty Research Award, a Meta gift, and support from Red Hat. I sincerely thank them for their support.

Tarikul Islam Papon

PhD Student

CS Department

EXPLOITING SSD PROPERTIES TO ENHANCE DATA SYSTEMS PERFORMANCE

TARIKUL ISLAM PAPON

Boston University, Graduate School of Arts and Sciences, 2025

Major Professor: Manos Athanassoulis, PhD
Assistant Professor of Computer Science

ABSTRACT

The rapid growth of data has made modern data systems increasingly storage-intensive, demanding high-performance storage solutions. In response, the shift from traditional Hard Disk Drives (HDDs) to Solid-State Drives (SSDs) has fundamentally transformed data storage, offering significantly higher performance through low-latency access, high internal parallelism, and reduced mechanical overhead. However, despite the widespread adoption of SSDs, many data-intensive systems use outdated assumptions tailored for HDDs. As a result, they fail to adapt to and harness SSD-specific characteristics such as read/write asymmetry (writes are slower than reads) and internal concurrency (SSDs can read or write multiple pages in parallel without hurting latency). Without exploiting internal concurrency, the device can remain vastly underutilized. To fully unlock SSD potential and ensure both optimal performance and device longevity, systems need to be redesigned with SSD-aware strategies.

This thesis bridges the gap between SSD properties and data system optimizations by more faithful modeling that captures key properties of SSDs: asymmetry and concurrency. First, we propose a Parametric I/O Model (PIO) that quantifies SSD properties (read/write asymmetry and concurrency) and provides guidelines for SSD-

aware algorithm design. Next, we develop ACE, an asymmetry/concurrency-aware bufferpool manager, which batches writes to write them in parallel and performs parallel prefetching to utilize the full bandwidth of the underlying SSD device. We then introduce CAVE, an out-of-core graph processing engine that optimally schedules independent graph I/O operations to maximize SSD read parallelism by exploring multiple paths concurrently. We implement parallelized variants of DFS and BFS that exploit storage parallelism to offer the same algorithm guarantees with better average performance. These implementations also serve as the building block of other graph algorithms (weekly connected components, pagerank, random walk). Finally, we present ReStore, the first data-migration policy for multi-tiered architectures that uses storage concurrency and asymmetry when modeling the state of the devices. ReStore dynamically places data across tiers based on the device properties and workload characteristics to optimize performance and cost.

Through extensive benchmarking and experiments on real SSDs, we showcase the benefit of better storage modeling in algorithm and system design to improve performance. Our evaluation shows that ACE bufferpool manager improves runtime performance by up to 32% for transactional workloads, CAVE graph manager is up to 10× faster than state-of-the-art baselines for various traversal algorithms and ReStore tiered data migration achieves up to 2.2× lower runtime and causes up to 10× fewer migrations. The findings in this thesis contribute to the broader field of storage system design, influencing databases, file systems and storage-intensive applications by faithfully leveraging SSD properties for higher performance.

Contents

1 Introduction **1**

1.1 The Data Avalanche Era 1

1.2 The Storage Device Evolution Landscape 2

1.2.1 Fifty Years of Dominance by Hard Disk Drives 3

1.2.2 The Rise of Solid-State Drives 4

1.3 Are HDD-Optimized Systems SSD-Optimized? 5

1.4 SSD-Conscious Data Systems Design 7

1.4.1 Thesis Statement 7

1.4.2 Thesis Contributions 7

1.4.3 Published Papers 8

1.5 Thesis Organization 9

2 SSD Internals and Properties **12**

2.1 Modern Storage Devices 12

2.2 Internal Parallelism of SSDs 14

2.3 Writes in SSDs 15

2.4 Expected Parallelism in Multi-Channel Devices 17

2.5 Empirical Quantification of α and k 19

2.5.1 Benchmarking Setup 19

2.5.2 Optane SSD 20

2.5.3 PCIe SSD 21

2.5.4 Regular SSD 22

2.5.5 Virtual SSD . . . 23
2.5.6 Impact of File System . . . 24
2.5.7 Impact of Access Granularity . . . 26
2.5.8 Characterizing a Storage Device . . . 27
2.6 Summary . . . 28
3 The Parametric I/O Model 29
3.1 Introduction . . . 30
3.2 Parametric I/O Model . . . 31
3.2.1 Performance Analysis . . . 32
3.3 Algorithm Design Guidelines . . . 39
3.4 Discussion . . . 41
3.4.1 Redesigning Algorithms & Operators . . . 41
3.4.2 Automatic Devices Tuning Using PIO . . . 43
3.5 Related Work . . . 43
3.6 Conclusions . . . 44
4 Aymmetry/Concurrency-Aware (ACE) Bufferpool Manager 46
4.1 Introduction . . . 47
4.2 An Augmented Bufferpool Design Space . . . 52
4.2.1 Background on Page Replacement Algorithm . . . 53
4.2.2 Write-back Policy . . . 54
4.2.3 Eviction Policy . . . 55
4.2.4 Read-ahead Policy . . . 56
4.3 ACE Bufferpool Manager . . . 56
4.3.1 Overview of ACE Bufferpool Management . . . 57
4.3.2 Writer . . . 59
4.3.3 Evictor . . . 59

4.3.4 Reader . . . 60
4.3.5 Putting Everything Together . . . 61
4.4 Implementation & Integration . . . 63
4.5 Evaluation . . . 65
4.5.1 Experimental Analysis with Synthetic Data . . . 67
4.5.2 TPC-C Benchmark . . . 75
4.6 Related Work . . . 77
4.7 Conclusions . . . 78

5 CAVE: concurrency-aware graph processing on SSDs 79

5.1 Introduction . . . 80
5.2 Background . . . 85
5.3 Parallelizing Graph Traversal . . . 87
5.3.1 Intra-Subgraph Parallelization . . . 88
5.3.2 Inter-Subgraph Parallelization . . . 89
5.3.3 Discussion . . . 90
5.4 Concurrent Graph Algorithms . . . 91
5.4.1 CAVE Physical Data Layout . . . 91
5.4.2 Building Blocks for Parallelizing . . . 94
5.4.3 Parallel Breadth-First Search . . . 96
5.4.4 Parallel Weakly Connected Components . . . 97
5.4.5 Parallel PageRank . . . 98
5.4.6 Parallel Random Walk . . . 100
5.4.7 Parallel Pseudo Depth-First Search . . . 100
5.5 Implementation . . . 103
5.6 Evaluation . . . 105
5.6.1 Parallel BFS . . . 107

5.6.2 Parallel WCC, PR & RW 112
5.6.3 Parallel pseudo DFS 115
5.7 Related Work 116
5.8 Conclusions 117

6 Restore: RL-Based Data Migration for Multi-Tiered Storage Systems 118

6.1 Introduction 119
6.2 Background & Related Work 125
6.3 Challenges of Page-Level Multi-Tiered Storage 128
6.4 ReStore: RL-based Page Migration 130
6.4.1 A quick primer on Reinforcement Learning 130
6.4.2 RL-based Tiered Storage Management 133
6.5 Emulating Multi-Tiered Storage 142
6.6 Experimental Evaluation 145
6.6.1 Experimental Methodology 146
6.6.2 Experimental Analysis 148
6.7 Conclusions 156

7 Concluding Remarks 158

Curriculum Vitae 182

List of Tables

1.1 Comparison of HDD and SSD trends 3

2.1 Performance comparison of the tested devices. 24

2.2 Empirical *Asymmetry* and *Concurrency*. 28

4.1 Empirical α and k of our SSDs. 66

4.2 Properties of the synthetic workloads 66

4.3 Comparison of buffer miss and logical/physical writes 69

5.1 Dataset Description 106

5.2 Preprocessing Time and Space Comparison 107

6.1 Our proposed approach captures workload features and device properties and quickly adapts to workload drift. 123

6.2 Empirical latency, asymmetry and concurrency of some storage devices for 4KB block size. 128

6.3 Our notation for modeling tiered storage with RL. 132

6.4 Our synthetic workloads. 147

6.5 Standard benchmarks and real-world traces used. 147

6.6 CPU time and memory overhead of each policy. 156

List of Figures

1·1 HDD and SSD internals. (A) HDDs rely on mechanical components such as a rotating platter and a moving read/write head, leading to inherent latency. (B) SSDs are fully electronic, utilizing a shared bus to connect multiple NAND flash chips, enabling high parallelism and low access latency. 4

2·1 (A) *Asymmetry* ($\boldsymbol{\alpha}$) and *Concurrency* ($\boldsymbol{k}$) in recent SSDs; most devices show high $\boldsymbol{\alpha}$ and $\boldsymbol{k}$. (B) Gain increases as more concurrent I/Os are used, and $\boldsymbol{\alpha}$ dictates the gain. 13

2·2 Internal architecture of an SSD. 15

2·3 (A) Not all writes are costly; six pages (A-F) are written in a block. (B) Updates cause invalidation of pages which cannot be overwritten unless the whole block is erased. (C) Periodic garbage collection reclaims the invalidated pages. 16

2·4 (A) The Optane SSD has lower $\boldsymbol{k}$ and $\boldsymbol{\alpha \approx 1}$. (B) Latency does not increase until saturation. 21

2·5 $\boldsymbol{k}$ and $\boldsymbol{\alpha}$ depends on the file system, access type, and block size. (A) The PCIe SSD shows high $\boldsymbol{k}$ with $\boldsymbol{\alpha}$ up to $\mathbf{2.8}$. The benefit of exploiting $\boldsymbol{k}$ can be as high as $\mathbf{40\times}$. (B) The SATA SSD shows lower $\boldsymbol{k}$ than the PCIe SSD with $\boldsymbol{\alpha}$ up to $\mathbf{1.5}$. 22

2·6 (A) The Virtual SSD is software-controlled, exhibiting $\boldsymbol{\alpha} \approx \mathbf{2}$ prior to saturation. (B) The extrapolated performance profile of the Virtual SSD following the trends of other SSDs. 23
2·7 Bypassing the file system allows for **1.7**× (**1.4**×) higher read (write) throughput for PCIe SSD, while the values of both $\boldsymbol{\alpha}$ and $\boldsymbol{k}$ are more stable across the experiments. 25
2·8 (A) The asymmetry is high when I/O size is smaller than native page size; as I/O size increases $\boldsymbol{\alpha}$ decreases. (B) The concurrency decreases for larger I/O size. 27
3·1 Higher asymmetry leads to higher cost since the more expensive write cost is not amortized (through concurrency or otherwise). 33
3·2 Speedup of an *Unbatchable Reads, Batchable Writes* application. The speedup is highest for write-intensive workloads. Furthermore, the speedup depends on the device asymmetry – *the higher the asymmetry, the higher the speedup*. 34
3·3 Speedup of a *Batchable Reads, Unbatchable Writes* application. The speedup is highest for read-intensive workloads. Speedup depends on the device asymmetry, however, the trend is reversed – *the lower the asymmetry, the higher the speedup*. 36
3·4 Speedup of a *Batchable Reads and Writes* application. (A, B) For fewer reads, speedup increases with more conc. write I/Os irrespective of conc. read I/Os. (C, D, E) Read concurrency comes into action as more reads are introduced in the workload. 37
3·5 Speedup increases as more concurrent I/Os are used until the device is saturated. 39

4·1 ACE addresses asymmetry by exploiting concurrency and amortizing writes. 48
4·2 ACE outperforms state-of-the-art due to better device utilization. 50
4·3 Bufferpool design space in terms of the design decisions and various options (RED denotes new components) 52
4·4 Popular eviction policies: (a) Clock Sweep evicts pages based on *usage count*; (b) CFLRU tries to evict *clean* pages first [window size $N/2$ used for brevity]; (c) LRU-WSR keeps a *cold* flag to delay dirty page eviction. 53
4·5 Abstract overview of **ACE** components 57
4·6 ACE page selection policies for $n_w = 3$ and $n_e = 3$. (a) ACE writes three dirty pages (p6, p4, p2) following the LRU order; if prefetching is enabled three pages (p6, p5, p4) are evicted, otherwise one page (p6) is evicted. (b) For LRU-WSR, ACE finds a dirty page with cold flag not set (p3). This page is moved to the front setting its cold flag. The dirty pages with set cold flag (p6, p4, p1) are selected for concurrent writing. 58
4·7 Table structure of the history-based prefetcher 61
4·8 ACE reduces total workload latency for all Clock Sweep, LRU, CFLRU, and LRU-WSR in the PCIe SSD. 67
4·9 ACE causes very small increase in total number of writes 69
4·10 ACE improves runtime for devices with low asymmetry. 70
4·11 Higher writes lead to greater benefits. 71
4·12 ACE is beneficial across a wide range of bufferpool size. 72
4·13 Speedup increases as #I/Os are increased until device gets saturated. . . . 73

4·14 (A) Spectrum of $(\boldsymbol{\alpha}, \boldsymbol{n})$ – as we move towards higher asymmetry, ACE has higher gain. (B) Empirical evidence showing that devices with higher asymmetry has higher gain for ACE. 74
4·15 ACE achieves high speedup for TPC-C mixed transaction, while benefiting the write-intensive transaction the most. 76
4·16 Gain of ACE scales with data size. 77

5·1 (A) Parallellized version of BFS in CAVE takes fewer iterations to converge. (B) CAVE is upto three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph and Mosaic. 83
5·2 Example of Intra/Inter-Subgraph Parallelization. (A) { B, D, E, F } are at the same level of BFS and are processed concurrently by 4 threads. (B) As pseudo-DFS progresses, the stack is split into two subgraphs ({ D, E } and { B, F }), which are processed in parallel by 2 threads. 88
5·3 Architecture of CAVE comprises block-based file structure (right) and a concurrent cache pool (left) 92
5·4 Example of the Parallel Pseudo DFS algorithm, demonstrating the progression of the stack over time. 102
5·5 Lock waiting time remains low as we increase the number of concurrent I/Os. 104
5·6 (A) - (F) Performance graph for BFS on our PCIe SSD. In general, CAVE outperforms the baselines GraphChi, GridGraph and Mosaic for all six datasets. Mosaic performs well for the sparse RN dataset. . 108
5·7 (A, B) Performance of PBFS on Optane SSD and SATA SSD for the FS dataset. CAVE outperforms the baselines. 110

5·8 As we increase the number of concurrent I/Os, the benefit of PBFS increases until the device gets saturated. 110
5·9 (A) CAVE performs well across all datasets for PBFS. Only Mosaic has a better performance for the sparse RN dataset. (B) CPU usage of CAVE remains low showing that its benefit comes from better SSD utilization – CAVE is I/O-bound, not CPU-bound. 111
5·10 (A) Blocked CAVE implementation outperforms other systems for WCC. (B) For dense graphs, GridGraph works well for finding WCC. 112
5·11 (A) CAVE performs well across all datasets for PWCC. (B) CAVE outperforms single-node GraphX deployment. 113
5·12 CAVE achieves lower runtime for PageRank, outperforming GraphChi and GridGraph. 114
5·13 (A) Concurrent I/Os improve performance for CAVE's Random Walk till the device is saturated. (B) CAVE's PDFS can attain the maximum benefit of the device by exploiting its *optimal concurrency*. 115

6·1 From the three different workloads, A is more skewed, and YCSB has more drift. More details can be found in Section 6.6.1. Our approach always outperforms the state of the art by up to 2.2×. 124
6·2 High-level overview of a page-level multi-tiered storage system consisting only of SSDs. 129
6·3 ReStore: A reinforcement learning based policy for data migrations in multi-tiered storage systems. 133
6·4 Membership function representing the categories 'Large' and 'Small', the value of the function stands for how likely a input is 'Large' or 'Small'. 137
6·5 Example of our temparature model. 142

6·6 Architecture of our multi-tiered storage system. 143
6·7 Distribution of page access frequencies in standard benchmarks and real-world workload traces. 148
6·8 Overall runtime (CPU time + data access time) under different tier capacity configuration and three static workloads. In general, ReStore outperforms the other baseline policies. Frequency-based policy like tLFU performs well for this type of skewed workload. Recency-based policy like tLRU struggles to capture the long-term *hf* workload. . . . 149
6·9 Number of page migrations, lighter colors are the number from Tier 2 to Tier 1, darker colors are from Tier 3 to Tier 2. ReStore causes the least number of migrations. 151
6·10 Total runtime under dynamic workloads. ReStore significantly outperforms other baselines with the greatest benefit in high drift scenarios (Workloads C & D). 152
6·11 Performance under varying read/write ratio with asymmetry (top) and without asymmetry (bottom) for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore delivers stable performance. 153
6·12 Normalized latency relative to *Static* of each policy as #pages and #operations increases for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore scales effectively with the data size. 154
6·13 ReStore achieves the lowest normalized latency (relative to *Static*) under standard benchmarks and real-world traces, consistently outperforming all other baselines. 155

List of Abbreviations

ARAM	....	Asymmetric RAM Model
BFS	....	Breadth-First Search
CFLRU	....	Clean-First Least Recently Used
CRF	....	Combined Recency and Frequency
DBMS	....	Database Management System
DFS	....	Depth-First Search
EM	....	External Memory Model
FS	....	Frienster Social Network Dataset
HDD	....	Hard Disk Drive
IOPS	....	Input/Output Operations Per Second
LFU	....	Least Frequently Used
LJ	....	LiveJournal Social Network Dataset
LRU	....	Least Recently Used
LRU-WSR	....	LRU Write Sequence Reordering
PBFS	....	Parallel Breadth-First Search
PDFS	....	Parallel Depth-First Search
PIO	....	Parametric I/O Model
PR	....	Page Rank
PWCC	....	Parallel Weekly Connected Component
RL	....	Reinforcement Learning
RN	....	RoadNet Network of PA Dataset
RW	....	Random Walk
SD	....	Synthetic Dataset
SPDK	....	Intel's Storage Performance Development Kit
SSD	....	Solid-State Drive
TD	....	Temporal Difference
TW	....	Twitter Social Network Dataset
WCC	....	Weekly Connected Component
YT	....	YouTube Social Network Dataset

Chapter 1

Introduction

1.1 The Data Avalanche Era

In this *data avalanche era*, the exponential growth of data has become a defining challenge and opportunity for modern data management systems. One of the most significant challenges emerging from this data revolution is the management of *big data* – an umbrella term encompassing the storage, organization, and analysis of massive-scale data. Organizations across sectors, including finance, healthcare, artificial intelligence, and scientific research, are generating and consuming unprecedented volumes of data. This explosion is driven by the widespread adoption of digital technologies and the popularity of IoT devices. According to market intelligence firm named IDC, the global datasphere (amount of data generated in a year) is expected to reach 175 zettabytes by 2025 [40], with a significant share originating from machine-generated sources such as sensors, automated systems, and interconnected devices. While these advancements have fueled innovation in data analytics and decision-making, they have also introduced immense challenges in data storage and retrieval. Now more than ever, it is crucial to *efficiently* store, manage, and access the vast and continuously expanding volume of digital information.

This digital revolution, also known as the third industrial revolution, has fundamentally reshaped the way data is created, processed, and stored. The decreasing cost of computational resources and storage infrastructure has accelerated the growth of large-scale data processing. However, this surge in data has placed increasing pres-

sure on storage architectures, requiring them to scale efficiently while maintaining low latency and high throughput. Many large-scale transactional systems, real-time analytics platforms, and machine learning pipelines require storage solutions to support high-speed data access to keep up with the rapid pace of data generation and consumption. Addressing these challenges has led to continuous advancements in storage technologies to meet the performance requirements of applications.

1.2 The Storage Device Evolution Landscape

Early Storage Technologies: From Punch Cards to Optical Discs The evolution of data storage has been driven by the demand for higher capacity, faster access, and greater reliability. Among the earliest storage technologies, punch cards emerged in the 19th century as a means of data input and processing for early computing systems. While durable, they were severely limited in storage capacity and access speed. The introduction of magnetic tape in the 1950s revolutionized data storage by offering significantly higher capacities, making it a standard for archival storage due to its cost-effectiveness and longevity. However, its sequential access nature made data retrieval slow and inefficient for random-access workloads.

The 1970s saw the rise of floppy disks, which provided portability and convenience for small-scale data storage. Despite their ease of use, they were constrained by low capacity and vulnerability to physical damage, leading to their eventual obsolescence. Optical discs, such as CDs, DVDs, and later Blu-rays, gained popularity in the 1980s, offering improved durability and portability for media storage. However, their relatively slow write speeds and capacity limitations made them less suitable for large-scale data-intensive applications.

1.2.1 Fifty Years of Dominance by Hard Disk Drives

Hard Disk Drives (HDDs), introduced in the 1970s, became the backbone of digital storage, providing significantly higher capacity and faster random access compared to preceding storage media. HDDs utilize spinning magnetic platters and mechanical read/write heads (Figure 1·1A) to store and retrieve data. For nearly five decades, HDDs remained the primary storage medium for personal computing, enterprise applications, and large-scale data centers. Many systems have been designed according to HDD properties (fast sequential access) because of HDD's widespread prevalence.

Table 1.1: Comparison of HDD and SSD trends

Device	Capacity	Seq. BW	Rand. BW	Time to Read	Cost ($/MB)
HDD 1980	100 MB	1.2 MB/s	0.28 MB/s	1 min	200
HDD 2025	4 TB	125 MB/s	20 MB/s	9 hours	0.00002
SSD 2010	100 GB	700 MB/s	400 MB/s	2.5 mins	0.02
SSD 2025	1 TB	2.5 GB/s	2.2 GB/s	6.5 mins	0.00004

Despite their dominance, HDDs suffer from fundamental performance limitations due to their mechanical nature. Their reliance on moving parts results in long data access times and constrained bandwidth. HDD performance has lagged significantly behind improvements in CPUs and main memory, with access times remaining nearly unchanged since the 1980s. While CPU processing power and memory bandwidth have seen exponential growth, HDD bandwidth has struggled to keep pace with increasing storage capacities. Table 1.1 shows that in 1980, reading an entire disk took about one minute, whereas today, reading a full modern HDD can take several hours. This growing performance gap arises from inherent physical constraints, such as seek time (caused by the movement of the read/write arm) and rotational delay (resulting from the spinning platters). HDDs are nowadays used primarily for archival purposes since they offer large storage capacities at a low cost [164].

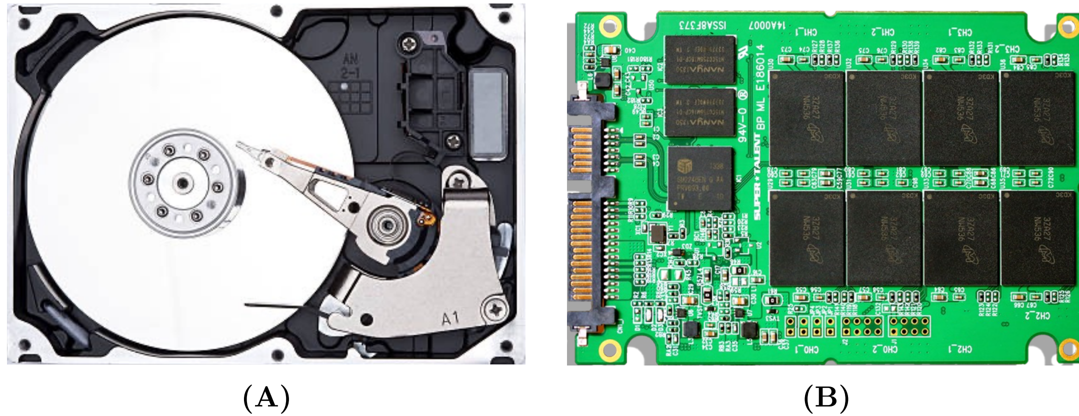


Figure 1·1: HDD and SSD internals. (A) HDDs rely on mechanical components such as a rotating platter and a moving read/write head, leading to inherent latency. (B) SSDs are fully electronic, utilizing a shared bus to connect multiple NAND flash chips, enabling high parallelism and low access latency.

1.2.2 The Rise of Solid-State Drives

The 2000s marked the advent of Solid-State Drives (SSDs), a new storage technology that leverages NAND flash memory instead of mechanical components. SSDs use semiconductor-based storage (Figure 1·1B), enabling orders-of-magnitude faster access times and significantly higher IOPS (Input/Output Operations Per Second). The absence of mechanical delays makes SSDs particularly well-suited for performance-critical applications such as high-speed databases, real-time analytics, and high-performance computing. Their lower power consumption and greater shock resistance further contribute to their widespread adoption. Although SSDs have historically been more expensive per gigabyte than HDDs, the price gap is narrowing rapidly, and projections suggest that SSDs could achieve cost parity with HDDs by 2030 [12]. Another key challenge with SSDs is their finite write endurance, as NAND flash cells degrade with repeated program/erase cycles. However, modern SSDs incorporate various techniques—such as wear leveling, TRIM commands, and over-provisioning—to mitigate these limitations and extend their lifespan.

The introduction of NVMe (Non-Volatile Memory Express) SSDs has further revolutionized storage by replacing the traditional SATA interface with the high-speed PCIe bus. This transition reduces protocol overhead, enables lower latency, and maximizes parallelism, fully exploiting the internal concurrency of NAND flash. As a result, NVMe SSDs have become the dominant choice in modern datacenters and high-performance consumer devices. NAND flash storage has already surpassed HDDs in terms of exabytes shipped annually [48], reinforcing the paradigm shift in storage technology. This trend echoes Jim Gray's famous statement: *"Tape is dead. Disk is Tape. Flash is the new Disk."* The evolution of SSDs marks a fundamental shift in the storage hierarchy, positioning flash memory as the backbone of future storage infrastructures. It is crucial for researchers and practitioner to know how to leverage the full potential of SSD devices by exploiting their fundamental properties.

1.3 Are HDD-Optimized Systems SSD-Optimized?

SSD Properties. The characteristics of SSDs differ significantly from those of HDDs, demanding a fundamental rethinking of storage-intensive data system design. One of the most critical differences is SSD's internal parallelism (we call this property **concurrency**). Modern SSDs consist of multiple NAND flash chips, organized into channels and dies, allowing concurrent read and write operations, achieving much higher bandwidth than HDDs. However, traditional data systems, designed with HDD constraints in mind, often fail to leverage this inherent parallelism. These designs often assume a single queue of I/O requests, serialized in a manner optimized for HDDs, which rely on sequential access patterns to mitigate mechanical seek delays. These optimizations do not translate well to SSDs because (i) random access is no longer expensive for SSDs, (ii) internal parallelism of SSD remain underutilized, and (iii) suboptimal data placement can lead to unnecessary write amplification.

Another defining characteristic of SSDs is **read/write asymmetry**: writes in SSDs can be up to an order of magnitude slower than reads due to the underlying characteristics of NAND flash memory. This asymmetry stems from the internal structure of NAND flash memory, where data must first be erased before new data can be written. Unlike HDDs, where writes simply overwrite existing data, SSDs operate on a block-based erase-before-write mechanism, requiring entire blocks to be erased before new data can be programmed. This erase-before-write mechanism along with an expensive garbage collection operation process introduces latency and amplifies write costs, a phenomenon known as *write amplification*. Furthermore, SSDs have a finite endurance, meaning flash cells degrade after a limited number of program/erase (P/E) cycles. High write amplification exacerbates this issue, reducing the lifespan of SSDs. HDD-optimized systems that generate excessive small, random writes can accelerate SSD wear-out, making it crucial to design storage solutions that minimize unnecessary writes and amortize the high asymmetric write cost.

Despite SSDs becoming the dominant storage medium for performance-critical applications, many data systems remain rooted in HDD-era assumptions. As a result, they fail to harness the full potential of SSD hardware and, in some cases, even degrade SSD performance and longevity. While HDD-centric storage designs focus on reducing seek times and optimizing data locality, **SSD-aware systems must take into account for flash-specific properties such as concurrency and read/write asymmetry**. Failure to adapt storage designs accordingly can lead to performance bottlenecks and inefficient SSD utilization. To bridge this gap, data management systems need redesigning with SSD-specific optimizations like leveraging parallelism and developing write-optimized solutions that align with the unique properties of NAND flash memory. Only then can modern storage systems move beyond HDD-era constraints and fully capitalize on the capabilities of SSD technology.

1.4 SSD-Conscious Data Systems Design

1.4.1 Thesis Statement

HDDs and SSDs have distinct architectures, yet many systems still follow outdated HDD-aware assumptions, ignoring SSD-specific properties like concurrency and read/write asymmetry. SSD-aware systems must incorporate these properties in algorithm and system design, which leads to more faithful storage modeling and, ultimately, to better device utilization.

1.4.2 Thesis Contributions

In an era where vast volumes of data must be efficiently stored and accessed, data-intensive systems rely on SSDs for high-speed retrieval and processing. However, because many systems have not been redesigned to account for SSD properties, they face significant performance bottlenecks. We find that these challenges can be addressed by carefully exploiting SSD properties during system and algorithm design.

This thesis makes the following key technical contributions:

- **The Parametric I/O Model.** We introduce the Parametric I/O Model (PIO), a simple yet expressive framework that captures contemporary SSD behavior by parameterizing two key properties: read/write asymmetry (α) and access concurrency (k). We develop a benchmarking process to quantify these properties and propose some guidelines to drive storage-intensive algorithm design, laying the foundation for SSD-aware system design.

- **ACE Bufferpool Manager.** Using insights from PIO, we redesign a critical database component – the bufferpool manager. We propose a new Asymmetry & Concurrency-aware bufferpool management (ACE) that batches writes based on SSD concurrency and performs them in parallel to amortize the asymmetric

write cost. In addition, ACE performs parallel prefetching to exploit the device's read concurrency.

- **CAVE Graph Manager.** We develop CAVE, the first out-of-core graph processing engine that optimally exploits underlying SSD's read parallelism via carefully selecting graph I/Os that can be issued concurrently. CAVE traverses multiple paths and processes multiple nodes and edges concurrently, achieving parallelization at a granular level.

- **ReStore: RL-Based Data Migration for Multi-Tiered Storage.** We propose ReStore, a reinforcement learning-based data migration policy for multi-tiered storage. By leveraging SSD properties and workload patterns, ReStore dynamically optimizes page placement across heterogeneous storage tiers, balancing performance and cost-effectiveness.

These contributions collectively advance SSD-aware system design and provide practical solutions for modern data-intensive applications, including databases, file systems, and big data analytics. By systematically harnessing SSD characteristics, this thesis lays the groundwork for future innovations in high-performance storage-intensive applications.

1.4.3 Published Papers

The research presented in this thesis has formed the foundation for multiple publications in leading international, peer-reviewed venues in the field of databases and data management systems.

1. Tarikul Islam Papon, Manos Athanassoulis. *The Need for a New I/O Model.* Conference on Innovative Data Systems Research **(CIDR)**, 2021.

2. Tarikul Islam Papon, Manos Athanassoulis. *A Parametric I/O Model for Modern Storage Devices.* 17th International Workshop on Data Management on New Hardware (**DAMON**), 2021.

3. Tarikul Islam Papon, Manos Athanassoulis. *ACEing the Bufferpool Management Paradigm for Modern Storage Devices.* 39th IEEE International Conference on Data Engineering (**ICDE**), 2023.

4. Tarikul Islam Papon, Teona Bagashbili, Manos Athanassoulis. *ACE-in-Action: A Smart DBMS Bufferpool for SSDs.* ACM Management of Data (**SIGMOD**), 2025.

5. Tarikul Islam Papon, Taishan Chen, Shuo Zhang, Manos Athanassoulis. *CAVE: Concurrency-Aware Graph Processing on SSDs.* ACM Management of Data (**SIGMOD**), 2024.

6. Tarikul Islam Papon. *Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry.* 40th IEEE International Conference on Data Engineering (**ICDE**) PhD Symposium, 2024.

7. Tianru Zhang, Tarikul Islam Papon, Teona Bagashvili, Manos Athanassoulis, Salman Toor. *Restore: A Reinforcement Learning Approach For Data Migration In Multi-Tiered Storage.* Preparing for publication.

1.5 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides an in-depth background of SSD internals, explaining the origins of read/write asymmetry and internal concurrency. We highlight the importance of the key characteristics of SSDs and present our benchmarking process to quantify read/write asymmetry and concurrency across five different SSD devices.

After reading Chapter 2, Chapters 3 - 6 can be read independently. Chapter 3 presents the Parametric I/O Model, which incorporates both α (asymmetry) and k (concurrency) to better capture SSD behavior. We show the benefits of adding these properties in our model with respect to device utilization and performance. Additionally, we discuss when and how to leverage different forms of concurrency and emphasize the role of asymmetry in performance modeling. Finally, Chapter 3 outlines five key guidelines for designing storage-intensive algorithms that effectively exploit SSD properties.

Chapter 4 shows how to exploit the concurrency property of SSDs from a DBMS bufferpool perspective. We refactor the bufferpool design space by including a write-back policy that consider device-specific properties. The chapter presents ACE, an asymmetry & concurrency-aware bufferpool manager that utilizes the device's concurrency. ACE is designed to be flexible enough to be combined with any existing page replacement policy and prefetching technique. We implement ACE in PostgreSQL, incorporating its default replacement algorithm (Clock Sweep) as well as three additional algorithms (LRU, CFLRU, LRU-WSR) along with their ACE implementations. Our evaluation shows that ACE achieves up to 32.1% lower runtime for a mixed workload with negligible increase in total writes and buffer misses.

Chapter 5 focuses on designing an out-of-core graph processing system optimized for SSDs. We introduce CAVE, the first out-of-core graph processing engine for traversal operations that maximally exploits SSD parallelism by carefully selecting graph I/Os for concurrent execution. By leveraging this parallelism, CAVE traverses multiple paths and processes nodes and edges simultaneously, achieving fine-grained parallelization for improved performance. We build within CAVE parallelized versions of five popular graph algorithms (Breadth-First Search, Depth-First Search, Weakly Connected Components, PageRank, Random Walk) exploiting SSD's read

concurrency. We observe that CAVE achieves up to one order of magnitude speedup compared to the popular out-of-core systems Mosaic and GridGraph, and up to three orders of magnitude speedup in runtime compared to GraphChi.

Chapter 6 presents ReStore, a reinforcement learning-based approach for data migration in multi-tiered storage systems. ReStore adapts to workload patterns such as access frequency, recency and device-specific characteristics like SSD read/write asymmetry, and parallelism. Each storage tier is managed by an RL agent that dynamically updates its parameters using temporal difference learning, allowing it to respond to evolving workloads and system states. Experimental evaluations using industry-grade benchmarks and diverse synthetic workloads demonstrate that ReStore achieves up to 2.2× lower runtime and 10× fewer migrations compared to baseline approaches.

Finally, Chapter 7 concludes the thesis with a summary of findings and directions for future research in SSD-aware data management.

Chapter 2

SSD Internals and Properties

Storage devices have evolved to offer increasingly faster read/write access, through flash-based and other solid-state storage technologies. When compared to classical rotating hard disk drives (HDDs), modern solid-state drives (SSDs) have two key differences: (i) the absence of mechanical parts, and (ii) an inherent difference between the process of reading and writing. The former removes a key performance bottleneck, enabling *internal device parallelism*, whereas the latter manifests as a *read/write performance asymmetry*. In this chapter, we explore SSD internals, focusing on their parallelism and read/write asymmetry, along with the implications of these properties. We first explain the origins of these characteristics, followed by an analytical approximation of the expected parallelism in multi-channel SSDs. We then present our benchmarking procedure [130] to quantify these properties across multiple SSD devices, analyzing the impact of file systems and access granularity.

2.1 Modern Storage Devices

The majority of today's secondary storage devices are solid-state drives (SSDs) while hard disk drives (HDDs) are increasingly used as "archival" storage [164]. NAND flash-based SSDs eliminate the mechanical overhead of HDDs, thus, providing low energy consumption, high chip density, and high random read performance [6, 82, 96, 140]. These benefits can be attributed to the *internal parallelism* of SSDs, which can be exploited to optimize performance [27, 129]. In other words, an SSD needs

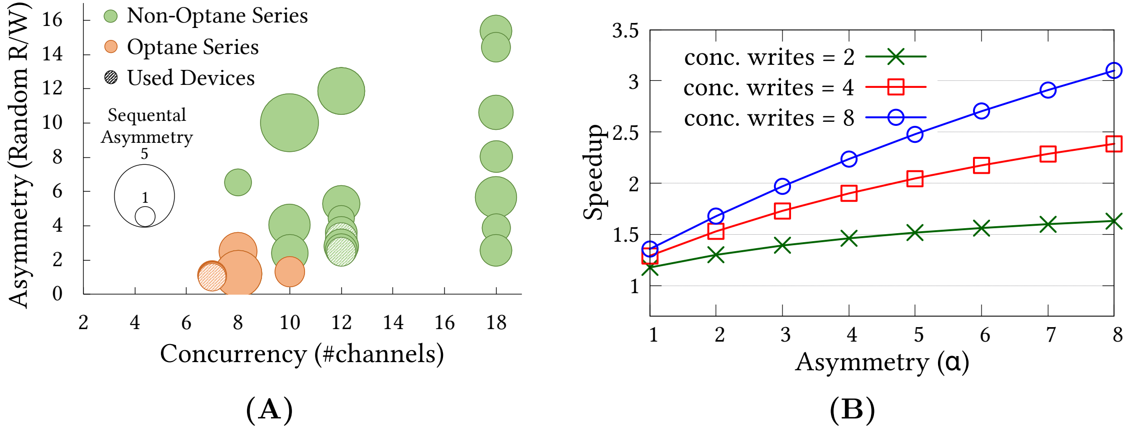


Figure 2·1: (A) *Asymmetry* ($\boldsymbol{\alpha}$) and *Concurrency* ($\boldsymbol{k}$) in recent SSDs; most devices show high $\boldsymbol{\alpha}$ and $\boldsymbol{k}$. (B) Gain increases as more concurrent I/Os are used, and $\boldsymbol{\alpha}$ dictates the gain.

to receive multiple concurrent I/Os (which can be distributed to different components by the flash controller) to saturate its bandwidth. However, due to the physics of NAND flash, writes are generally slower than reads which leads to *read/write asymmetry* [36]. With these two properties, i.e., concurrency (quantified by k) and read/write asymmetry (quantified by α), SSD behavior departs from the one of traditional HDDs. Figure 2·1A shows that most recent SSDs have an asymmetry of more than two ($\alpha > 2$) and concurrency of more than 8 ($k > 8$). Typically, off-the-shelf SSDs use the traditional SATA interface, while, various high-end latest devices use the PCIe interface offering a new improved protocol called Non-Volatile Memory Express (NVMe). NVMe SSDs offer low latency, almost as fast as DRAM while persisting data. There are multiple alternatives in terms of underlying storage technology [25, 33, 63, 150, 158, 176], commonly known as the emerging Non-Volatile Memory (NVM) class. Most NVMs have similar properties (high asymmetry, concurrency, chip density, and low energy consumption) like SSDs [16, 115]. The vision for NVM is to offer a byte-addressable durable memory medium with performance close to main memory's. The most mature technology to date was 3D XPoint [63, 148],

which was commercialized via Intel's Optane series [68] and later discontinued. Optane SSDs generally exhibit lower read/write asymmetry and concurrency than flash-based SSDs [201], which is further corroborated by our experiments in Section 2.5. To summarize, even though there are several variations in terms of different technologies, modern storage devices (SATA/PCIe/Optane SSDs and NVMs) share a few key properties: fast access, high chip density, low energy consumption, and most importantly, *read/write asymmetry* and *access concurrency*.

It is essential to know the concurrency (k) and read/write asymmetry (α) of a device to fully utilize and comprehend its benefit. Figure 2·1B shows the significance of k and α. We consider an application that can exploit the device's write concurrency by batching writes. We simulate the speedup of such an application on devices with different α values (more details in Section 3.2). The figure shows that as we increase the number of concurrent I/Os, the gain of exploiting parallelism increases. However, the gain closely depends on the device asymmetry – the higher the asymmetry, the higher the gain. Without capturing these properties of modern storage devices, we cannot attain their full potential, nor we can tailor the algorithms to the device-at-hand, resulting in subpar performance and suboptimal device utilization.

2.2 Internal Parallelism of SSDs

SSDs exhibit a high degree of **internal parallelism** in their architecture [26, 27, 113]. Figure 2·2 presents the internal architecture of an SSD showing that multiple channels are connected to the flash controller, and each channel consists of a shared bus with multiple chips. Each chip contains multiple dies, each die comprises multiple planes, and finally, each plane constitutes multiple blocks where the pages reside [6]. Hence, to fully utilize the bandwidth of an SSD, we need to submit multiple concurrent I/Os. Ideally, these I/Os will target different parts of the device, and they will be

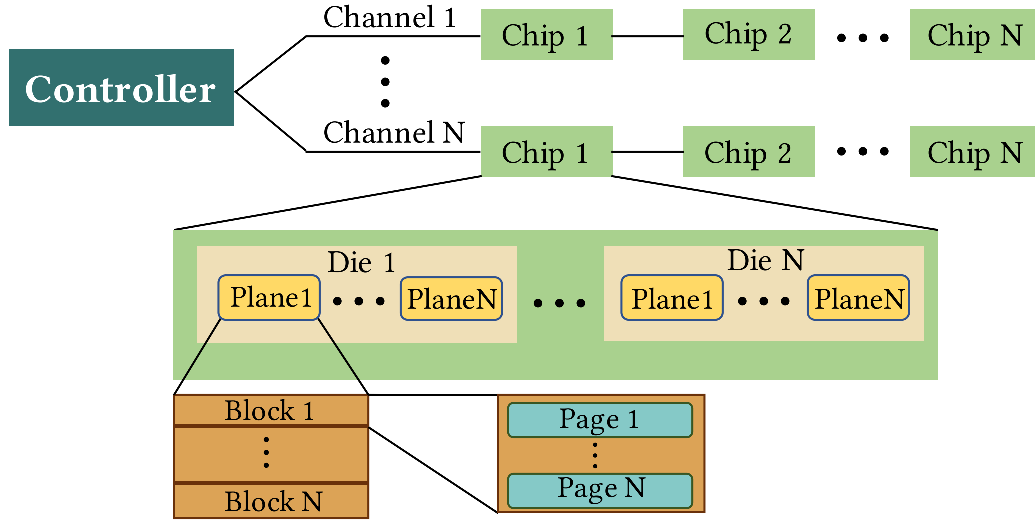


Figure 2·2: Internal architecture of an SSD.

parallelized by the controller [113, 141, 165]. To quantify the supported concurrency, we refer again to Figure 2·1A that shows on the x-axis the number of channels of each device. We use this as a proxy to the supported concurrency which, in most cases, is more than 8. The exact level of concurrency (k) needed to saturate the device depends on the request type (read or write) and on the specifics of the device. Hence, each device has two types of concurrency: (i) read concurrency (k_r) – number of read I/Os the device can perform in parallel, and (ii) write concurrency (k_w) – number of write I/Os the device can perform in parallel.

2.3 Writes in SSDs

Page updates in NAND-based SSDs follow the *erase-before-write* approach. Logical page updates (at the file system level) are always performed as *out-of-place* updates. In order to *physically* update the contents of a flash page, the containing block has to be erased first, which means once a page is written, it cannot be overwritten until that whole block is erased [6]. Figure 2·3 highlights three cases for writing a flash page. When a new write request comes in a free location, it can be performed

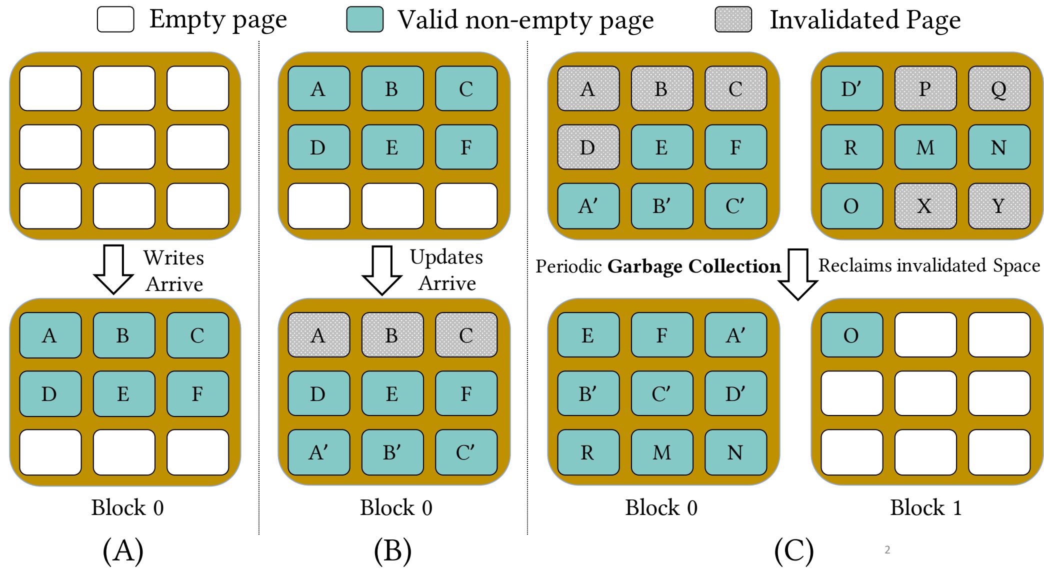


Figure 2·3: (A) Not all writes are costly; six pages (A-F) are written in a block. (B) Updates cause invalidation of pages which cannot be overwritten unless the whole block is erased. (C) Periodic garbage collection reclaims the invalidated pages.

directly which is shown in Figure 2·3A for pages A-F. However, once a page is written, it cannot be overwritten until that whole block is erased. So, when a page *update* arrives, the controller has to invalidate the old page and write the updated page in a new location, as shown in Figure 2·3B for pages A′-C′. As a result, after a number of writes, the SSD may contain several invalidated pages. To reclaim the invalidated space, the flash controller periodically triggers *garbage collection*, which copies the valid pages of a block, writes them in a new block and then erases the previous block, as shown in Figure 2·3C. Note that flash cells generally wear out after a certain number of program/erase cycles, which is a measure of the device's *endurance* [6, 122]. Techniques like wear-leveling, overprovisioning, and bad block skipping are commonly used to mitigate the wear-out effect and expand the SSD lifetime [74, 83, 121]. While the read/write granularity is a flash page (typically with size 512B-32KB), the erasure granularity is an erase block (4MB-64MB). The overhead of maintaining periodic garbage collection along with the extra writes results

in *higher amortized write cost* [22, 36, 52, 123]. The level of asymmetry depends on the specific device and the type of access (sequential/random). Typically, writes can be up to one order of magnitude slower than reads. For example, the advertised random read and write performance of the Intel D5-P4320 SSD is 427K IOPS and 36K IOPS respectively, resulting in an asymmetry of 11.86×. We again refer back to Figure 2·1A which shows the advertised asymmetry of several recent Intel SSDs. The random read/write asymmetry is plotted along the y-axis, while the radius of the circle shows the sequential read/write asymmetry. The light colored circles correspond to devices that we used for benchmarking and experimentation. More details about these devices are presented in Section 2.5. The highest advertised random (sequential) asymmetry among those devices is 15.4× (4.86×). Although some devices advertise low asymmetry (e.g., Optane SSDs), in practice, most devices have an asymmetry of 2× or more. In the rest of the thesis, we use the generic term *read/write asymmetry* (α) or *asymmetry* to refer to both random and sequential read/write asymmetry.

2.4 Expected Parallelism in Multi-Channel Devices

We now present an analytical approximation of the expected parallelism of a multi-channel SSD under uniform I/O distribution. Even if multiple I/O requests are issued, it is not guaranteed that all of the device's channels will be used. It is possible that the I/O requests target blocks are located in the same channel. It is also possible that the scheduler of the I/O requests in the flash controller cannot always disperse the I/Os across different channels. Hence, here, we approximate how many of the channels will be occupied in response to multiple concurrent random I/Os on average.

Consider a device that has n channels and a total of m I/O requests has been issued in these channels. For the sake of generality, we assume a uniform distribution of the I/O requests across the channels. We are interested in the total number of

channels that will be occupied with the I/O requests. To calculate that, we first calculate using recursion, the expected number of channels that will have no I/O request to serve. We first define E_m.

$$E_m = \text{expected number of empty channels after } m \text{ I/Os}$$

By definition, after assigning the first $(m-1)$ I/Os to channels, there will be E_{m-1} empty channels. Now, the m-th I/O must be assigned either to an empty channel or a non-empty channel. The probability of assigning it to an empty channel is $\frac{E_{m-1}}{n}$, because there are E_{m-1} empty channels out of the n channels. In other words, $\frac{E_{m-1}}{n}$ of the time, there will be $(E_{m-1}-1)$ empty channels, while the rest of the time $(1-\frac{E_{m-1}}{n})$, there will be E_{m-1} empty channels. We now express E_m recursively:

$$E_m = \frac{E_{m-1}}{n} \cdot (E_{m-1} - 1) + \left(1 - \frac{E_{m-1}}{n}\right) \cdot E_{m-1} = \frac{n-1}{n} \cdot E_{m-1} \tag{2.1}$$

Since all channels are empty when we receive zero request ($E_0 = n$), the recursion becomes $E_m = n \cdot ((n-1)/n)^m$. So, the fraction of channels that will be empty is $E_m/n = ((n-1)/n)^m = ((1-(1/n))^n)^{\frac{m}{n}}$ which can be approximated by $(1/e)^{\frac{m}{n}}$.

If we allow a device to concurrently serve as many I/O requests as the number of channels ($m \approx n$), then the fraction of idle channels is $(1/e) = 37\%$. As we issue more concurrent I/Os, the fraction of idle channels reaches 0%. We consider an effective concurrency of a device the number of I/Os needed for E_m to approach zero, or $e^{-m/n} \to 0$. We numerically calculate that when $m \approx 3n$, then $E_m = e^{-m/n} = e^{-3} \approx 0.05$. So, when the device controller allows the number of concurrent I/Os to be 3× the number of channels, we utilize $(1 - E_m) = 95\%$ of the device's channels. We use this analysis as a guide to the level of concurrency that can be efficiently supported by modern storage devices, issuing concurrent I/Os between 1× and 3× the number of the device's internal parallelism.

2.5 Empirical Quantification of α and k

We now present a benchmarking methodology to quantify asymmetry (α), read concurrency (k_r) and write concurrency (k_w) of modern storage devices [130]. We experiment on several off-the-shelf and high-end devices. We also analyze the file system's impact on α, k_r and k_w as well as on performance.

2.5.1 Benchmarking Setup

We use a machine with two Intel Xeon Gold 6230 2.1GHz processors each having 20 cores with virtualization enabled and with 27.5MB L3 cache, 384GB of RDIMM main memory. Our experimental server has four storage devices: (i) a 375GB Optane P4800X SSD, (ii) a 1TB PCIe P4510 SSD, (iii) a 240GB SATA S4610 SSD and (iv) a 2TB HDD. We refer to these devices as *Optane SSD*, *PCIe SSD*, *SATA SSD*, and *HDD*, respectively. We also experiment with a *virtualized* storage device from Amazon AWS which is a 1.2TB Provisioned IOPS SSD allowing a maximum of 60000 IOPS. We use both Fio [49] and our custom benchmarking tool to have fine-grained experimental control. Our tool supports synchronous and asynchronous I/Os, and optionally, direct I/O, multithreading, varying queue depth, and varying block size. Our device-level benchmark uses a 29GB file on which we issue I/O requests that differ in request type (read/write), access type (sequential/random), request granularity (4K/8K), and number of threads. We enable direct I/O and use a queue depth of 32. For each experiment, we report the average IOPS, bandwidth, and latency per operation where the numbers are averaged over 10 minutes of execution. For every experiment, the standard deviation was less than 1%. The SSDs were pre-conditioned by writing 3 times prior to the experiments ensuring that the devices have stable performance [44]. Without loss of generality, in the rest of the thesis, we focus primarily on random requests because (i) random accesses have lower performance with higher asymmetry

than sequential, and (ii) in our experiments, we find that the overall trends and observations for sequential accesses are similar to the ones for random accesses.

Computing α & k. We define *concurrency* as the number of concurrent I/Os needed to saturate the device bandwidth. Hence, in these experiments, we increase the number of threads issuing concurrent I/Os for each device to identify the point when the device reaches its maximum bandwidth. It is worth mentioning that for most cases, the increase of the bandwidth is not linear. As a result, we handpick a point where the device bandwidth is close to saturation or where the rate of bandwidth increase drops significantly. In this way, we quantify the concurrency of a device empirically rather than using the specifics of the internal device architecture. As mentioned earlier, SSD devices has two types of concurrency depending on the access pattern: read concurrency (k_r) and write concurrency (k_w). As for *read/write asymmetry* (α), it is calculated by taking the ratio of the maximum read vs. write IOPS (bandwidth) obtained from these experiments.

2.5.2 Optane SSD

Figure 2·4A shows the read and write throughput of our Optane SSD, as we vary the number of threads issuing concurrent I/Os on the x-axis. The dotted lines indicate the point where the device bandwidth reaches (close to) the saturation point for the corresponding access pattern. These figures highlight the significance of utilizing device concurrency. For example, there is a 4× increase in the Optane SSD's read bandwidth when using full concurrency compared to no concurrency. We further observe that the Optane SSD gets saturated quickly indicating that it has lower concurrency ($k \approx 6$). Moreover, the Optane SSD exhibits a very small degree of asymmetry ($\alpha \approx 1.1$). Figure 2·4B shows the latency profile and the impact of concurrency for the Optane SSD. As we increase the number of concurrent I/Os, initially there is almost no increase in latency while the bandwidth increases. When

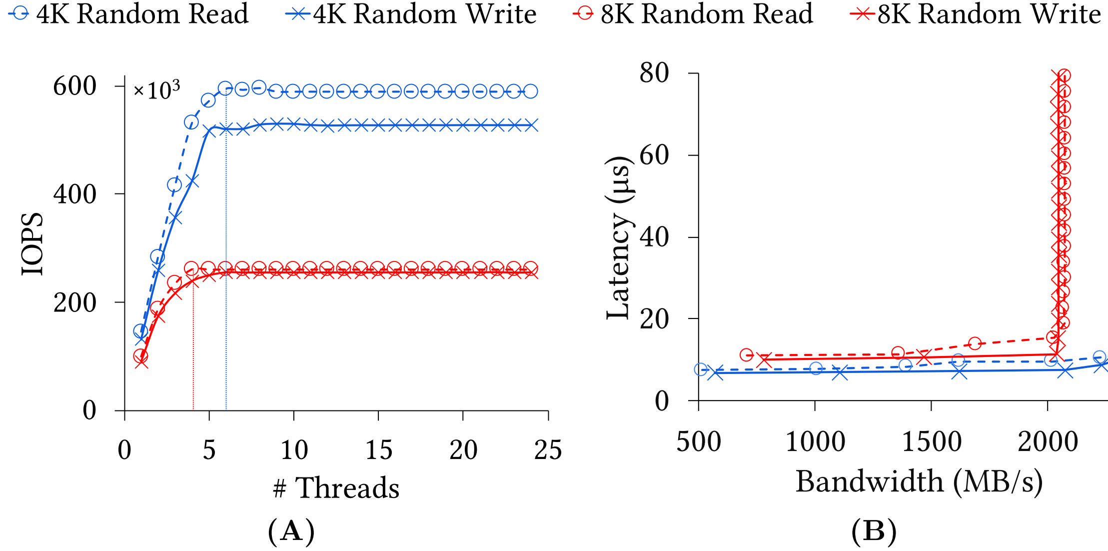
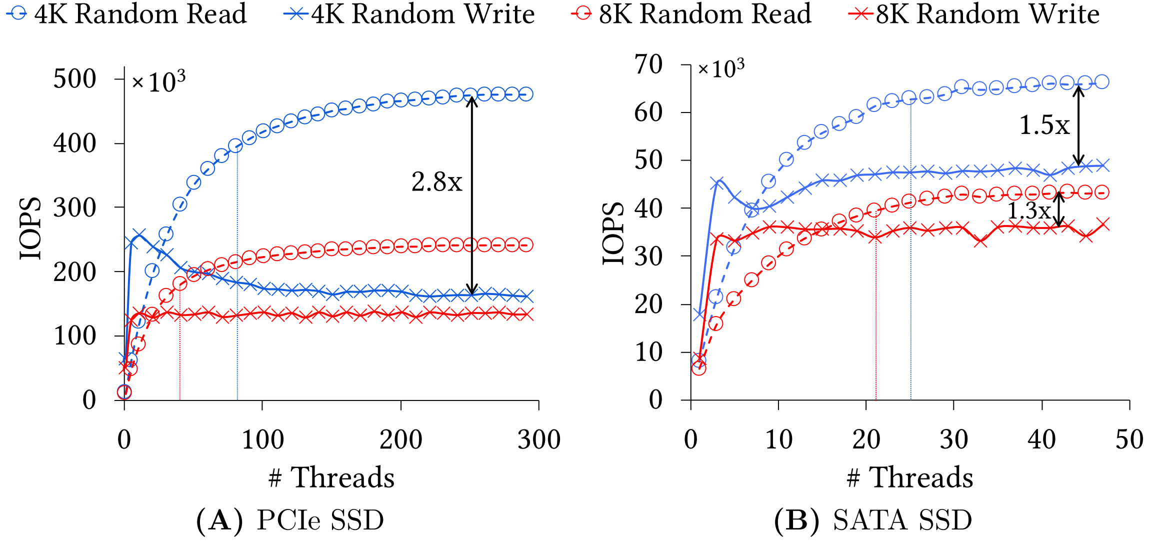


Figure 2·4: (A) The Optane SSD has lower $\boldsymbol{k}$ and $\boldsymbol{\alpha} \approx \mathbf{1}$. (B) Latency does not increase until saturation.

the device bandwidth gets saturated, that latency increases quickly. The *optimal concurrency* (k) of the device is the number of concurrent I/O issued when we have this sudden latency increase.

2.5.3 PCIe SSD

Figure 2·5A presents the read and write throughput of our PCIe SSD, as we vary the number of threads issuing concurrent I/Os on the x-axis. We notice that there is a 40× increase in the PCIe SSD's read bandwidth when using full concurrency compared to no concurrency. Compared to the Optane SSD, PCIe SSD is 3× slower. Figure 2·5A also shows that asymmetry and concurrency also depend on the access granularity. For instance, when using 4KB blocks, $\alpha \approx 2.8$, while for 8KB blocks, $\alpha \approx 1.9$. In addition, the supported concurrency depends on both the block size and the request type (read/write). For instance, the PCIe SSD gets saturated when issuing 80 concurrent 4KB random read I/Os ($\alpha \approx 2.8$, $k_r \approx 80$) whereas for 8KB random writes the device exhibits lower asymmetry and concurrency values ($\alpha \approx$



(A) PCIe SSD

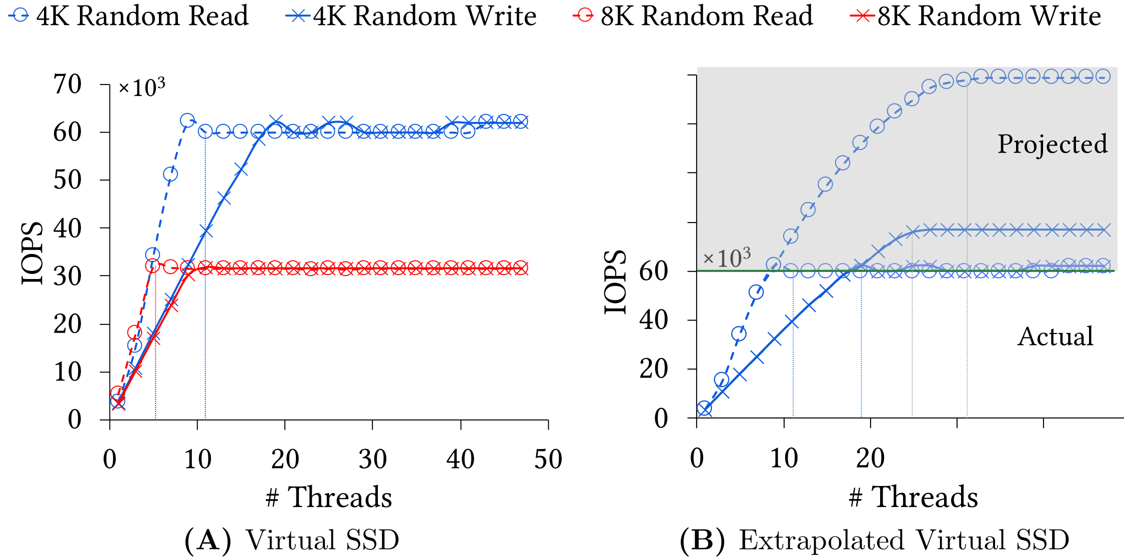
(B) SATA SSD

Figure 2·5: $\boldsymbol{k}$ and $\boldsymbol{\alpha}$ depends on the file system, access type, and block size. (A) The PCIe SSD shows high $\boldsymbol{k}$ with $\boldsymbol{\alpha}$ up to **2.8**. The benefit of exploiting $\boldsymbol{k}$ can be as high as **40**×. (B) The SATA SSD shows lower $\boldsymbol{k}$ than the PCIe SSD with $\boldsymbol{\alpha}$ up to **1.5**.

1.9, $k_w \approx 7$). Finally, we observe that for a low number of concurrent I/Os, writes are actually faster than reads. This is attributed to caching benefits when the controller's memory is enough to capture all the concurrent writes and flush them as a batch. As the number of concurrent I/O increases, garbage collection comes into action, consequently exhibiting somewhat slower, steady-state write performance.

2.5.4 Regular SSD

Figure 2·5B presents the performance profile of our SATA SSD which closely follows the trend of the PCIe SSD. This device shows lower asymmetry and much lower concurrency compared to the PCIe SSD, but higher than that of the Optane SSD. For example, for 4KB random read requests, $\alpha \approx 1.5$ and $k_r \approx 25$. The device gets saturated quickly in the event of concurrent writing; for 8KB random write, we find $\alpha \approx 1.3$ and $k_w \approx 5$.



(A) Virtual SSD

(B) Extrapolated Virtual SSD

Figure 2·6: (A) The Virtual SSD is software-controlled, exhibiting $\boldsymbol{\alpha \approx 2}$ prior to saturation. (B) The extrapolated performance profile of the Virtual SSD following the trends of other SSDs.

2.5.5 Virtual SSD

Figure 2·6A benchmarks the Virtual SSD. For 4K random read requests, there is a 16× throughput improvement when increasing the number of concurrent I/Os from 1 to 11, at which point the device gets saturated. Hence, for a small number of concurrent I/Os, the device remains underutilized and the client does not receive the performance they paid for, resulting in monetary loss. There are two more observations from this graph: (i) since this is a virtual SSD, the device's peak bandwidth is software-capped, yet, by comparing the slope for the 4K read and write performance, we observe that this device has $\alpha \approx 2$, and (ii) the graph is more stable than the graphs of PCIe SSD and SATA SSD i.e., the increase in IOPS is a perfect straight line and there is very little noise in the graph, which can be attributed to the device being virtual, and its performance being managed by the software. Figure 2·6A shows that for 4KB access, the read (write) bandwidth reaches its maximum for 11 (19) concurrent I/Os. Note

Table 2.1: Performance comparison of the tested devices.

Device	Latency (RR)	Latency (RW)
Optane SSD	6.8 μs	7.6 μs
PCIe SSD	12.4 μs	19.6 μs
SATA SSD	100 μs	130 μs
Virtual SSD	180 μs	200 μs
HDD	7000 μs	7300 μs

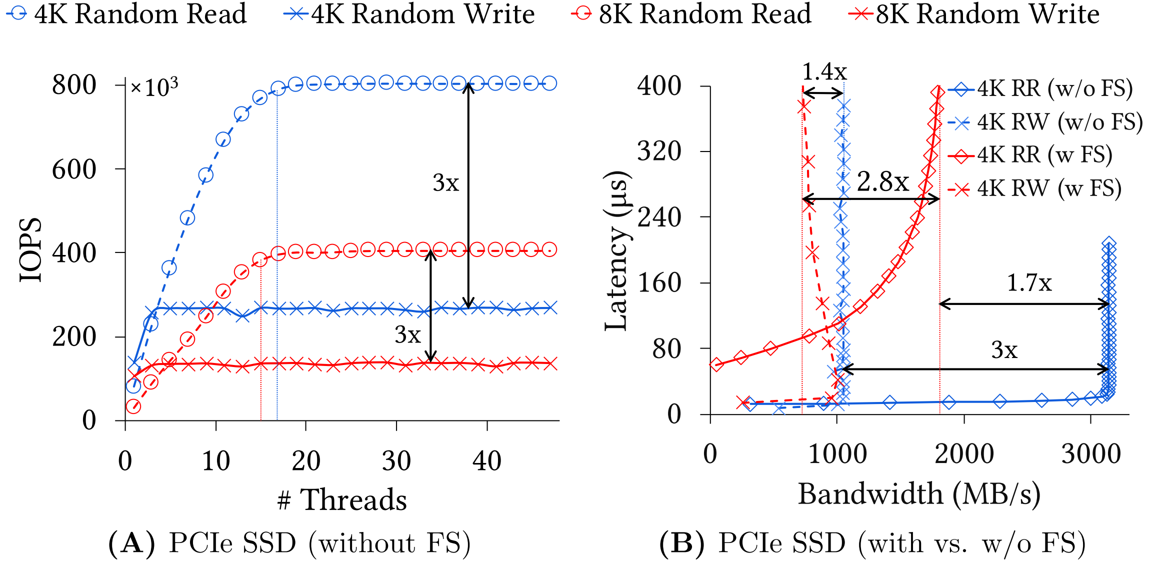
that these values are not the actual concurrency of the device, rather, at this point the maximum allocated IOPS (60K) is achieved and the software layer caps the device bandwidth. We speculate that the actual concurrency of the device is potentially much higher. Figure 2·6B shows an extrapolated profile of the underlying physical device of the Virtual SSD, assuming similar performance patterns to the other tested devices and no software limit in the attainable IOPS.

Optane SSD is the Fastest. Our benchmarking shows that the Optane SSD is the fastest, which is 1.8× faster than the PCIe SSD and the access latency can be as low as $6.8\mu s$. We also find that the PCIe SSD is 8× faster than the SATA SSD and the SATA SSD is 70× faster than our HDD. Table 2.1 lists the access latency of all the devices for 4KB random read/write requests.

HDD. Our experiments show that the HDD has $\alpha = 1$ and $k = 1$ which is expected. We omit the performance graphs since our primary focus is SSDs.

2.5.6 Impact of File System

Bypassing the File System for PCIe SSD. In our previous experiments, we observe that the PCIe SSD has a maximum read and write bandwidth of 1900 MB/s and 700 MB/s for 4KB block size, while the read latency can be as low as 12μs. For such a fast device, the software latency of the file system and the OS kernel can be significant. This device offers raw access to the storage medium by completely bypassing



(A) PCIe SSD (without FS) **(B)** PCIe SSD (with vs. w/o FS)

Figure 2·7: Bypassing the file system allows for **1.7×** (**1.4×**) higher read (write) throughput for PCIe SSD, while the values of both $\boldsymbol{\alpha}$ and $\boldsymbol{k}$ are more stable across the experiments.

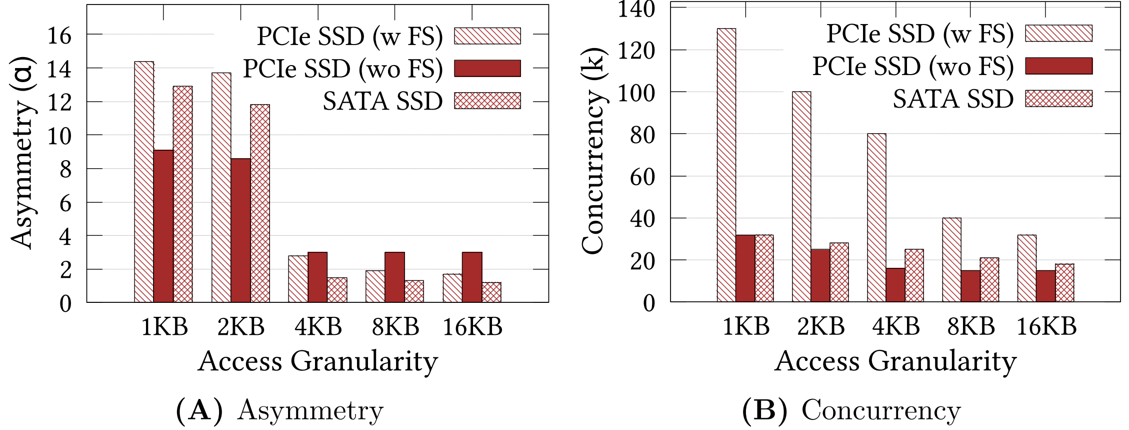
the file system through Intel's Storage Performance Development Kit (SPDK) [175]. SPDK provides a set of libraries for writing high-performance, scalable, user-mode applications. To run an SPDK application (i) the device must not have an active file system, (ii) hugepages must be allocated to facilitate its driver (2GB by default), and (iii) the device must be bound to a Virtual Function IO kernel driver rather than the native kernel drivers to allow direct device access to userspace. Using SPDK and its *perf* tool, we analyze how α and k change, and how performance is affected when we perform raw access to this device. We issue random I/O requests, varying the request type (read/write), the request granularity (4K/8K) and the number of threads.

Figure 2·7A presents the performance profile of the PCIe SSD with raw access. The smoothness of the graph indicates that the device has stable performance. The figure shows that the device has $\alpha = 3$ for either access granularity. Similar to previous experiments, k depends on the request type and access granularity. For example, for 4KB random requests $\alpha \approx 3$, and $k \approx 16$ for reads. Although the concurrency

value is lower, there is a considerable gain in both the maximum read bandwidth (1.7×) and the maximum write bandwidth (1.4×) as compared to Figure 2·5A. Figure 2·7B shows the latency/bandwidth profile of the PCIe SSD with and without the file system, clearly supporting the thesis that for a high-performance device, the software layer becomes a bottleneck. Traditionally, applications interact with storage via the OS using an interrupt-based model. Although the interrupt model has an overhead, historically this overhead was negligible compared to disk latency. However, with the emergence of high-performance storage devices that use faster protocols like NVMe [70, 204] and technologies like 3D XPoint [63, 148], the file system overhead is not negligible any more [85, 203], rather, *the storage bottleneck is shifting from hardware to software*, which is corroborated by our experiments.

2.5.7 Impact of Access Granularity

Our previous experiments hinted that k and α depend on the access granularity. In this experiment, we further vary the access granularity between 1KB and 16KB to see this dependency in more detail. Figure 2·8A and 2·8B shows how α and k vary with respect to I/O size for three different devices. Figure 2·8A shows that for access size smaller than the native page size (4KB), the asymmetry is very high for all the devices. For example, for 1KB I/O size, the SATA SSD and PCIe SSD (with FS) show asymmetry of 13 and 14.4 while for 4KB these values drop down to 1.5 and 2.8. This is because even for a small write (i.e., 1KB or 2KB), the write must be performed at page level. This results in a lot of updates on the SSD block, which consequently trigger excessive garbage collection that in turn significantly slows down the writes. Hence, writes smaller than the native page size should be avoided. Figure 2·8B shows that as I/O size increases, the device bandwidth reaches maximum with fewer I/Os. This is because larger I/O size causes higher data transfer which quickly saturates the device bandwidth.



(A) Asymmetry **(B)** Concurrency

Figure 2·8: (A) The asymmetry is high when I/O size is smaller than native page size; as I/O size increases $\boldsymbol{\alpha}$ decreases. (B) The concurrency decreases for larger I/O size.

2.5.8 Characterizing a Storage Device

To create a succinct representation of each device, we now put together the findings from our experiments with three metrics: asymmetry (α), read concurrency (k_r), and write concurrency (k_w). Table 2.2 shows the values of these metrics for all tested devices for 4KB and 8KB accesses. We note that while for some cases, there is a positive correlation between α and k_r/k_w, the two quantities are not equal. Specifically, we define asymmetry as $\alpha = BW_r^{max}/BW_w^{max}$ where BW_r^{max} and BW_w^{max} correspond to the maximum read and write bandwidth respectively. Further, we can express BW_r^{max} and BW_w^{max} as a function of the number of threads that saturate each operation as follows: $BW_r^{max} = f_r(k_r)$ and $BW_w^{max} = f_w(k_w)$. Following our definition for α, we get $\alpha = f_r(k_r)/f_w(k_w)$. The functions $f_r(\cdot)$ and $f_w(\cdot)$ quantify the attained bandwidth, however, $f_w(\cdot)$ includes the amortized cost of garbage collection. Hence, the two functions are neither identical nor linear, and in general $\alpha \neq k_r/k_w$.

Table 2.2: Empirical *Asymmetry* and *Concurrency*.

Device	4KB			8KB		
	α	k_r	k_w	α	k_r	k_w
Optane SSD	1.1	6	5	1.0	4	4
PCIe SSD (with FS)	2.8	80	8	1.9	40	7
PCIe SSD (w/o FS)	3.0	16	6	3.0	15	4
SATA SSD	1.5	25	9	1.3	21	5
Virtual SSD	2.0	11	19	1.9	6	10

2.6 Summary

We observe from Table 2.2 that all the NAND-based SSDs (PCIe, SATA, and Virtual SSD) exhibit asymmetry and concurrency. The value of α and k varies across devices depending on the internal of the device, access granularity, access pattern and filesystem. Bypassing the file system can unlock higher performance and stabilize the observed asymmetry and concurrency (Table 2.2 and Figure 2·7A). Optane SSDs generally exhibit low asymmetry and concurrency because of their 3D Xpoint technology [201] and our experiments corroborate this. We notice, however, that even exploiting the *low* concurrency in Optane SSD leads to 4× higher throughput. While the performance specifics vary significantly among different devices [19, 208], the vast majority of modern storage devices exhibit a non-trivial degree of both asymmetry and concurrency. Hence, storage-intensive applications can benefit from a more faithful modeling to fully exploit the underlying device and to predict the expected performance benefits.

Chapter 3

The Parametric I/O Model

The performance of storage-resident applications is typically modeled by the number of *disk accesses* performed, inherently assuming *symmetric* read and write performance and the ability to perform *only one I/O at a time*, failing to accurately capture the performance of SSDs. To address this mismatch, we propose a simple yet expressive storage model, termed **Parametric I/O Model** (PIO)[1] [131] that captures contemporary devices by parameterizing *read/write asymmetry* (α) and *access concurrency* (k). PIO enables device-specific decisions at algorithm design time, rather than as an optimization during deployment and testing, thus ensuring optimal algorithm design by taking into account the properties of each device. Further, we show that using carefully quantified values of α and k for each storage device, we can fully exploit the performance it offers, and we lay the groundwork for *asymmetry/concurrency-aware* storage-intensive algorithms. We also highlight that the degree of the performance benefit due to concurrent reads or writes depends on the asymmetry of the underlying device. Finally, we summarize our findings as a set of guidelines for designing storage-intensive algorithms and discuss specific examples for better algorithm and system designs as well as runtime tuning.

[1]The material of this chapter has been the basis for the DAMON 2021 paper "A Parametric I/O Model for Modern Storage Devices" [130] and the CIDR 2021 abstract "The Need for a New I/O Model" [131].

3.1 Introduction

External-Memory Model (EM Model). The performance of data-intensive applications is typically bounded by the time needed to transfer data through the storage and memory hierarchy. Hence, when data resides on slow media, *disk I/O* is the primary bottleneck. As a result, measuring and modeling *disk I/O access* is often used as a proxy to performance. The traditional I/O model (also termed EM model) [5] consists of a two-level memory hierarchy with a fast internal memory of size M (which we simply call *memory*) and a slow external memory (*storage*) of unbounded size; both divided into fixed-size blocks. Any computation requires the corresponding data blocks to be in memory, which is typically orders of magnitude faster than storage. Therefore, the EM model measures cost using only the storage I/O cost (number of storage accesses). As a result, the cost Q of an algorithm in the EM model is defined as the total number of read and write accesses to the storage. In other words, if an algorithm performs Q_r read accesses and Q_w write accesses to the external memory, the cost Q of the algorithm can be defined as $Q = Q_r + Q_w$. Hence, they have very different characteristics than HDD which the EM model does not comprehend. This modeling is accurate when two key assumptions hold: (i) disk reads and writes have similar cost, and (ii) applications can perform one I/O at a time – *none of which is true* for modern storage devices. The mismatch between the EM model and contemporary devices stems from the fact that it was developed for HDDs, which have *symmetric* read-write performance dominated by the disk's mechanical movement. Furthermore, the mechanical parts do not allow HDDs to serve multiple concurrent requests; rather, disk accesses happen serially. In contrast, most flash-based SSDs are composed of electronic components.

Storage Modeling. Without capturing asymmetry and concurrency of SSDs, we cannot attain its full potential, nor we can tailor the algorithms to the characteristics

of the device, resulting in subpar performance and suboptimal device utilization. This raises the need for a new I/O model [131] that can incorporate these device properties. Now, the question we set out to answer is:

How should the I/O model be adapted in light of read/write asymmetry and concurrency?

Parametric I/O Model. We propose a simple yet expressive storage model termed Parametric I/O Model (PIO) [130] that considers *asymmetry* (α) and *concurrency* (k) as parameters. Using the device properties, this richer I/O model can accurately capture modern SSDs. We benchmark different types of state-of-the-art SSD storage devices to quantify their α and k. In addition, we perform an abstract modeling of different types of applications based on our proposed PIO model, and we showcase the impact of utilizing k and modeling α. Our experiments and analysis reveal that *more informed storage modeling leads to better overall application performance.*

Contributions. This chapter makes the following contributions.

- We introduce the **Parametric I/O Model** (PIO) which considers both α and k. We show the benefits of adding these properties in our model with respect to device utilization and performance.
- We discuss *when* and *how* to exploit *which* type of concurrency. We highlight that asymmetry is key to quantifying performance.
- We outline five guidelines that should be considered during storage-intensive algorithm design.

3.2 Parametric I/O Model

In this section, we present the **Parametric I/O Model** (PIO), a new, simple yet expressive model that takes asymmetry and (read and write) concurrency as param-

eters. PIO enables better algorithm design and helps to accurately reason about the performance of storage-intensive algorithms and data structures.

> $\boldsymbol{PIO(M, k_r, k_w, \alpha)}$ *assumes a fast main memory with capacity M, and storage of unbounded capacity that has* ***read/write asymmetry*** *α, and* ***read (write) concurrency*** *k_r (k_w).*

We consider that both memory and storage are divided into fixed-size blocks. Since the model is device-specific, the values of k_r, k_w, and α are either given by the device manufacturer, or by a benchmarking process similar to the one used in Section 2.5. PIO allows us to accurately describe a variety of devices, and reason for their behavior at algorithm-design time. That way, we can make storage-aware optimizations part of the design, as opposed to applying them as an additional *ad hoc* tuning step during deployment. Next, we present how to use PIO to reason about the performance benefits of several fundamental classes of applications.

3.2.1 Performance Analysis

To analyze the performance of a storage-intensive application, we focus on its interaction with the storage device, that is, on the read and write requests it issues. We classify storage-intensive applications into four classes.

- *Unbatchable Reads, Unbatchable Writes.* An application that cannot batch reads nor writes. We include this class only for completeness and to highlight the impact of asymmetry.
- *Unbatchable Reads, Batchable Writes.* An application that batches writes and utilizes the write concurrency of the device (*example*: concurrent eviction of dirty pages from a bufferpool).

- *Batchable Reads, Unbatchable Writes.* An application that batches reads and utilizes the read concurrency (*example*: concurrent traversal of multiple paths in a graph or in a tree index).
- *Batchable Reads and Writes.* An application that batches both reads and writes and utilizes both read and write concurrency (*example*: LSM-tree compaction).

To maintain the generality of the approach, we quantify the performance gain using the frequency of reads (f_r) and writes (f_w), for which $f_r + f_w = 1$. We assume that read requests have unit cost (1) and write requests have cost α, where $\alpha \geq 1$. A device with read concurrency of k_r and write concurrency of k_w can concurrently perform k_r reads and k_w writes.

Unbatchable Reads, Unbatchable Writes.

We first consider the class of applications that cannot batch reads or writes and performs them sequentially. While this class seems to be of no interest as it cannot exploit any concurrency, we include it for completeness and to highlight the importance of asymmetry. Since reads have unit cost and writes have α cost, the normalized cost per operation for this type of application is $f_r + f_w \cdot \alpha$. Figure 3·1 shows this cost as we vary the read/write ratio in the workload. For a device with higher asymmetry, the cost increases as more writes are introduced in the workload. This figure highlights that when there is asymmetry, treating reads and writes equally leads to performance degradation. Rather, the slower writes should ideally be masked either algorithmically, or if possible, via batching.

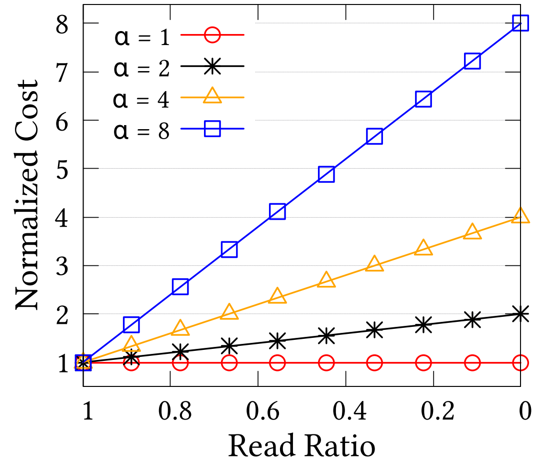
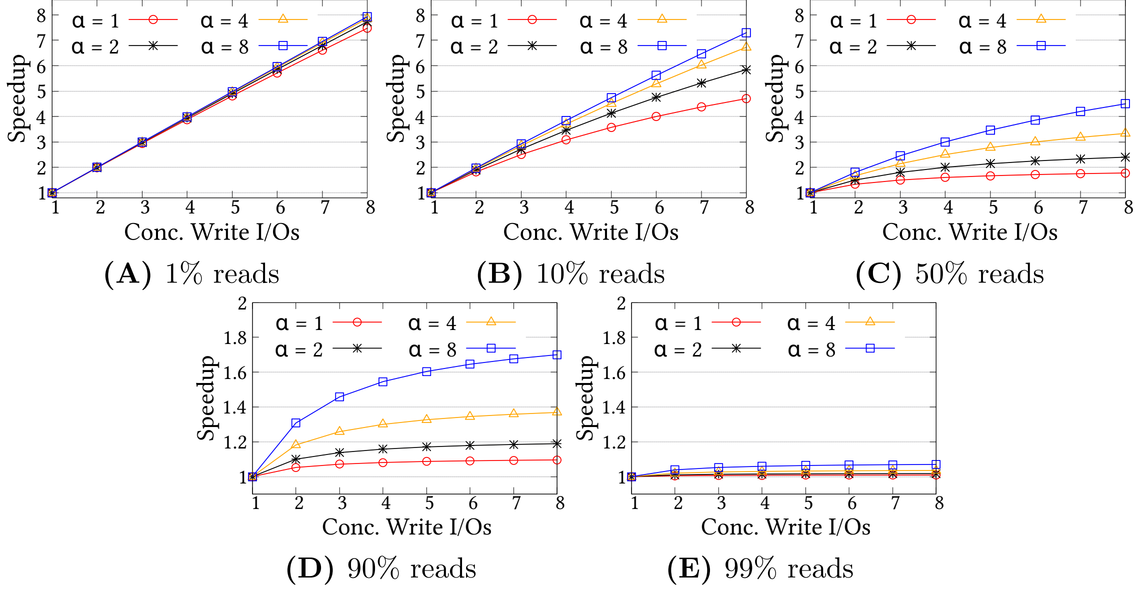


Figure 3·1: Higher asymmetry leads to higher cost since the more expensive write cost is not amortized (through concurrency or otherwise).



(A) 1% reads **(B)** 10% reads **(C)** 50% reads **(D)** 90% reads **(E)** 99% reads

Figure 3·2: Speedup of an *Unbatchable Reads, Batchable Writes* application. The speedup is highest for write-intensive workloads. Furthermore, the speedup depends on the device asymmetry – *the higher the asymmetry, the higher the speedup.*

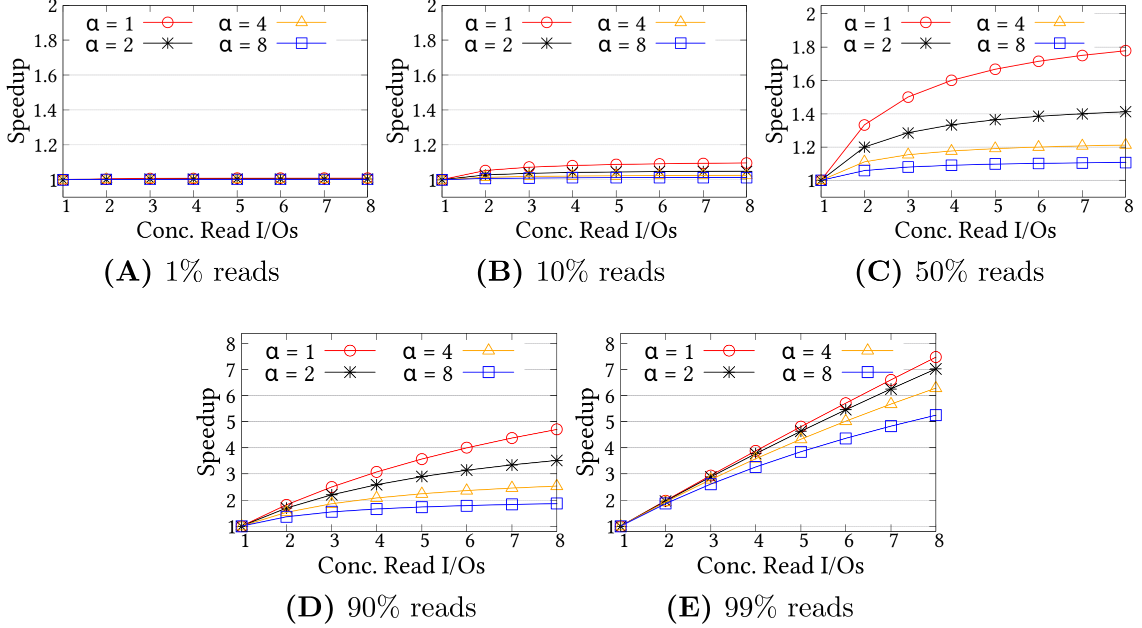
Unbatchable Reads, Batchable Writes. This class of applications exploits the write concurrency of the device to batch write requests. As an example, consider a modified DBMS bufferpool manager that selects several dirty pages and writes them concurrently during an eviction. In this scenario, the application at hand attempts to fully exploit the device's write concurrency via *concurrent flushing* during a page eviction. To achieve this, it submits k_w concurrent writes. Since the device has α asymmetry, the cost of a write following the standard approach of evicting one page at a time, as indicated by the EM model would be $C_W^{EM} = \alpha$. On the other hand, the *amortized* cost per write when we batch k_w writes following PIO is $C_W^{PIO} = \frac{\alpha}{k_w}$. Since reads are not batchable in this application class, each read will have unit cost in both the EM and PIO paradigm, hence, $C_R^{EM} = C_R^{PIO} = 1$. We can now calculate the speedup S_{PIO} of this application based on PIO:

$$S_{PIO} = \frac{f_r \cdot C_R^{EM} + f_w \cdot C_W^{EM}}{f_r \cdot C_R^{PIO} + f_w \cdot C_W^{PIO}} = \frac{f_r + f_w \cdot \alpha}{f_r + f_w \cdot \frac{\alpha}{k_w}} \implies$$

$$S_{PIO} = \frac{k_w \cdot (f_r + f_w \cdot \alpha)}{k_w \cdot f_r + f_w \cdot \alpha} = 1 + \frac{(k_w - 1) \cdot f_w \cdot \alpha}{k_w \cdot f_r + f_w \cdot \alpha}$$

Since $\alpha \geq 1$ and $k_w \geq 1$, then $S_{PIO} \geq 1$ where the maximum value of S_{PIO} can be up to k_w. Figure 3·2 shows the speedup when following PIO for different α and k_w values as we change the read/write ratio. We observe that the speedup increases as more concurrent I/Os are employed, which is expected. Furthermore, we note that the speedup depends on the asymmetry of the device. The *gain is **higher** for a device with **higher** asymmetry.* For example, for $f_r = 0.5$ and when fully exploiting the concurrency of $k_w = 8$ by issuing 8 concurrent I/Os, the speedup for a device with $\alpha = 8$ is 4.5× whereas the speedup for a device with $\alpha = 1$ is 1.78× (Figure 3·2C). Since the application batches writes, the benefit from efficient writing is more pronounced for a write-intensive workload (Figure 3·2A). For a workload that contains *only* reads or writes, the speedup from the PIO paradigm is the same irrespective of α. The key observation is that for batchable writes applications, a higher asymmetry between expensive writes and cheap reads leads to a higher speedup.

Batchable Reads, Unbatchable Writes. The second class of applications models scenarios where reads can be issued concurrently to exploit read concurrency, but not writes. As an example, consider a graph store that traverses multiple paths concurrently, thus can accelerate various algorithms including graph search and shortest path. Essentially, the algorithm can visit multiple nodes in parallel instead of one node at a time, and offer faster search time with the same worst-case guarantees. Another example is concurrent traversal of tree indexes [153]. The application performs k_r reads concurrently, thus, $C_R^{EM} = 1$ and $C_R^{PIO} = \frac{1}{k_r}$. The cost of a write request is $C_W^{EM} = C_W^{PIO} = \alpha$ for both paradigms since the writes are not batched. The speedup of this second class of applications is:



(A) 1% reads **(B)** 10% reads **(C)** 50% reads

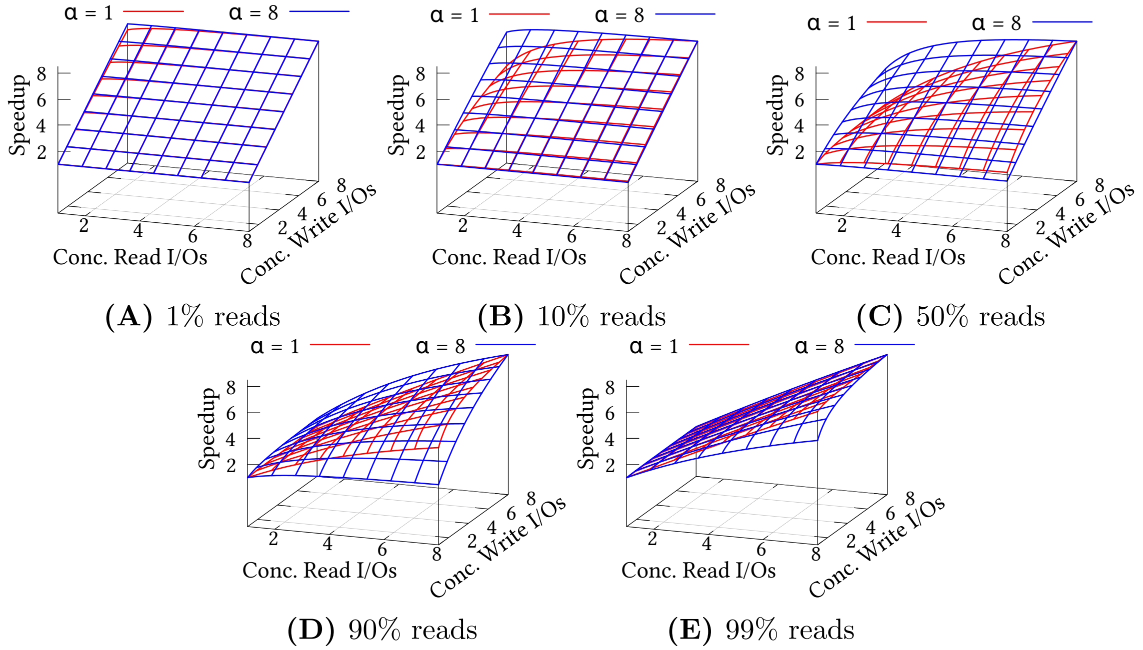
(D) 90% reads **(E)** 99% reads

Figure 3·3: Speedup of a *Batchable Reads, Unbatchable Writes* application. The speedup is highest for read-intensive workloads. Speedup depends on the device asymmetry, however, the trend is reversed – *the lower the asymmetry, the higher the speedup.*

$$S'_{PIO} = \frac{f_r \cdot 1 + f_w \cdot \alpha}{f_r \cdot \frac{1}{k_r} + f_w \cdot \alpha} = \frac{k_r \cdot (f_r + f_w \cdot \alpha)}{f_r + k_r \cdot f_w \cdot \alpha} = 1 + \frac{(k_r - 1) \cdot f_r}{f_r + k_r \cdot f_w \cdot \alpha}$$

Since $\alpha \geq 1$ and $k_r \geq 1$, then $S'_{PIO} \geq 1$, and also $S'_{PIO} \leq k_r$. Figure 3·3 presents the speedup of such an application based on PIO. Like before, the speedup increases as more concurrent I/Os are used. However, this time *the gain is* ***higher*** *for a device with* ***lower*** *asymmetry.* For instance, with $f_r = 0.5$ and $k_r = 8$, the speedup for a device with $\alpha = 1$ is 1.78× and it drops to 1.11× for $\alpha = 8$ (Figure 3·3C). Note that, the overall speedup is lower than the previous application class because, while writes are still more expensive than reads (for $\alpha > 1$), this class of applications can only utilize read concurrency. The speedup is maximized when the workload is read-heavy (Figure 3·3A), showing the benefit of batching reads.

Batchable Reads and Writes. The third class of applications can batch both



(A) 1% reads

(B) 10% reads

(C) 50% reads

(D) 90% reads

(E) 99% reads

Figure 3·4: Speedup of a *Batchable Reads and Writes* application. (A, B) For fewer reads, speedup increases with more conc. write I/Os irrespective of conc. read I/Os. (C, D, E) Read concurrency comes into action as more reads are introduced in the workload.

read and write requests. A notable example of such an application that can concurrently issue many concurrent read and write requests without any interdependency are LSM-trees, and specifically their compaction routine. During a compaction, multiple read/write streams are involved since files from the selected levels are read, sort merged and written to the next level. Since the whole process invokes numerous reads and writes, the compaction scheduler can use PIO to decide the degree of concurrency. Specifically, it can start as many concurrent compactions as needed to fully exploit the read concurrency and the write concurrency. The cost of a classical read is $C_R^{EM} = 1$, and of a classical write is $C_W^{EM} = \alpha$. Since this class of applications can batch both reads and writes, the amortized PIO costs are $C_R^{PIO} = \frac{1}{k_r}$ and $C_W^{PIO} = \frac{\alpha}{k_w}$. The speedup of the third class of applications is:

$$S''_{PIO} = \frac{f_r \cdot 1 + f_w \cdot \alpha}{f_r \cdot \frac{1}{k_r} + f_w \cdot \frac{\alpha}{k_w}} = \frac{k_r \cdot k_w \cdot (f_r + f_w \cdot \alpha)}{k_w \cdot f_r + k_r \cdot f_w \cdot \alpha}$$

The maximum value of S''_{PIO} can be up to k_r or k_w depending on the workload. For fewer reads (Figures 3·4A, 3·4B), the speedup increases as more concurrent write I/Os are issued irrespective of the number of concurrent read I/Os. For workloads with more reads (Figures 3·4C, 3·4D, 3·4E), the impact of concurrent read I/Os becomes more prominent. Similarly to the previous class of applications, the speedup depends on the device asymmetry. Overall, we observe that the impact of utilizing write concurrency is higher than utilizing read concurrency because of the asymmetry.

Impact of Asymmetry (α). The above analysis reveals that the speedup from exploiting concurrency depends on the asymmetry of the device. For an *Unbatchable Reads, Batchable Writes* application, the speedup is higher for a device with higher asymmetry because the higher cost of writes is amortized through batching. On the other hand, for a *Batchable Reads, Unbatchable Writes* application, the speedup is higher for devices with lower asymmetry, because batching actually further reduces the amortized cost of reads. In other words, batching reads exacerbates the impact of read/write asymmetry. To summarize, the degree of the performance gain/loss depends on the asymmetry and the application type, whereas the gain is achieved through exploiting concurrency.

Importance of using the *Optimal k*. The speedup of an application depends on carefully exploiting the actual device concurrency. To demonstrate this, we implement a sample application with *unbatchable reads and batchable writes* (a bufferpool that batches writes of dirty pages and flushes them concurrently during eviction). We run this *concurrency-aware* application on our PCIe SSD that exhibits $k_w = 8$, and we vary the number of concurrently issued write-backs during eviction. Figure 3·5 shows the speedup when comparing with an application that always evicts a single page, for

a workload with $f_r = 0.5$. As we increase the number of concurrent evictions towards the device's $k_w = 8$, the speedup increases as expected, however, after this point, the speedup drops. This is because (i) the device's supported parallelism is 8, hence, there is no benefit from issuing more concurrent writes, and (ii) for higher number of concurrent I/Os, the software overhead of thread management also increases. Hence, the speedup drops after reaching this threshold, indicating that we should refrain from increasing the concurrent requests (especially writes) further than the supported concurrency.

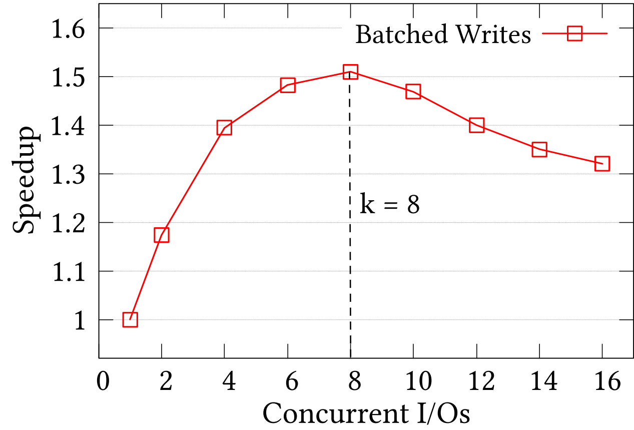


Figure 3·5: Speedup increases as more concurrent I/Os are used until the device is saturated.

3.3 Algorithm Design Guidelines

We now distill five guidelines from our previous discussion that should drive the development of new storage-intensive algorithms.

Guideline 1. *Know Thy Device*

When a new device is used, we frequently focus on the raw performance (throughput and latency), however, modern storage devices are complex software-hardware systems that also exhibit read/write asymmetry, as well as read and write concurrency. In Sections 2.5 and 3.2, we see that we need to know and appropriately use both these properties to get the best out of our devices. Hence, before deploying a device it is crucial to benchmark it to quantify its PIO parameters: asymmetry (α), and read (k_r) and write (k_w) concurrency.

Guideline 2. *Exploit Device Concurrency*

When a storage-intensive application has readily available information for multiple read and/or write operations, it should exploit the available parallelism of the device. The modeled concurrency is only an approximation of the actual capabilities of the device (which depend both on the internals of the device and the data access patterns). However, operating at a rate close to the benchmarked concurrency will yield the best device throughput. In addition, algorithms that have been designed with the "one I/O at a time" or "one stream of I/Os at a time" approach should be redesigned. For example, a bufferpool can concurrently write-back multiple dirty pages and concurrently prefetch multiple pages, a graph search can visit multiple nodes in different paths concurrently and cover the graph faster, and an LSM-tree can tune its compaction according to how many concurrent streams can be supported.

Guideline 3. *Use Device Concurrency With Care*

In the effort to fully exploit the bandwidth of a device we might end up pushing the device beyond its limits. In a simple benchmark, read bandwidth simple plateaus for a high number of concurrent I/Os, however, this is not the case for write-intensive or more complex applications. For such scenarios, using the *appropriate* k, i.e., exactly what the device supports, is key to achieve the *optimal* (advertised) performance, while pushing for higher concurrency might have adverse effects (Figure 3·5). Further, real-life applications may have other bottlenecks (locking, synchronization, logging, etc.) that can further diminish the benefits of concurrent accesses.

Guideline 4. *It is suboptimal to treat equally a page read and a page write for a device with asymmetry*

While it may seem natural to consider both a page read and a page write equally in terms of performance because of how main memory and hard disk behave, this

is not the case for modern storage devices. A page read and a page write follow a very different execution path, and building algorithms and data structures that treat them equally in terms of performance leads to suboptimal designs. To address this, when operating on modern storage devices, page writes should either be delayed (hoping that some will be avoided) or they should be amortized through concurrent writing. Overall, performing one write to facilitate one read or vice versa (e.g., when a bufferpool is saturated) is a suboptimal design in the presence of asymmetry.

Guideline 5. *Read/Write Asymmetry Controls Performance*

For applications that have read and write requests, the device asymmetry controls the magnitude of the performance benefits due to utilizing concurrency (Figures 3·2, 3·3, and 3·4). Consider the class of applications with batchable writes. Devices with higher asymmetry benefit more from writing in parallel. On the other hand, for applications with batchable reads, devices with higher asymmetry see smaller benefits, because the asymmetrically high cost of writes cannot be amortized. For example, a bufferpool that is saturated and has to evict a dirty page will exchange a write for a read. If the device exhibits asymmetry, this approach is not *fair*, because one write is more expensive than one read. Hence storage-intensive algorithm design should be both *concurrency* and *asymmetry* aware.

3.4 Discussion

3.4.1 Redesigning Algorithms & Operators

The Parametric I/O model captures the behavior of modern storage devices, and can be used to guide algorithm design for external storage. In this thesis we focus on a set of abstract workloads that represent different types of applications, which can benefit from capturing asymmetry and concurrency. Following the same spirit, this premise

can be applied to various components of data systems. As PIO ensures *better storage utilization* by considering the specific device properties, we envision better algorithm design for almost any component of a system that interacts with storage.

Bufferpool. A component of a database system that can benefit from PIO is the bufferpool. In fact, several bufferpool implementations (e.g., InnoDB, DB2) already exploit write concurrency by batching multiple writes and exploiting background flushing [43, 67]. Using PIO, we can carefully design a concurrency eviction policy that accounts both for the device write concurrency and the device asymmetry to guarantee a balanced workload execution. We will present our proposed asymmetry/concurrency-aware bufferpool in Chapter 4.

Tree and Graph Traversal. In light of PIO, tree and graph traversal algorithms can be redesigned to access multiple nodes concurrently, thus, having a wider search space at the same time. Consider a variation of the depth-first search (DFS) algorithm that can offer DFS guarantees, while at the same time covering a wider search-space by loading concurrently sibling nodes to the degree the underlying concurrency can support it. Conversely, one can construct an algorithm with breadth-first-search (BFS) guarantees that follows a few promising paths as deep as they might get. We will present our concurrency-aware graph manager in Chapter 5.

Query Optimizer Cost Models. Consider a query optimizer that determines which *join* algorithm to employ using cost models [62]. While state-of-the-art optimizers differentiate between sequential random accesses [62], typically, they do not differentiate between reads and writes. For modern storage devices, however, this misses the opportunity to account for the read/write asymmetry. By including asymmetry in the cost-model, we can make more accurate query optimizing decisions. Moreover, the join algorithms themselves can be designed to leverage device concurrency when reading/writing data blocks from/to storage.

LSM-trees Compactions. Finally, a data structure that can also benefit from PIO is the Log-structure merge (LSM) tree's *compaction* process. LSM-trees are widely used as the storage layer of many NoSQL key-value and other data stores [17, 42, 127, 159, 160]. LSM-trees write on disk immutable sorted runs and once a level reaches its threshold, a *compaction* is performed in order to reorganize the data between the saturated level and the next level. An LSM-tree typically initiates multiple compactions at the same time, and each compaction merges a number of sequential streams to a single output. Hence, using PIO, the compaction scheduler can decide how many and which compactions to initiate to better exploit the underlying device.

Overall, incorporating read/write asymmetry (α) and concurrency (k) in algorithm design allows for customizability for different devices which leads to more faithful storage modeling and, ultimately, to better device utilization.

3.4.2 Automatic Devices Tuning Using PIO

Further, the benchmarking presented in Section 2.5 shows that the characteristics of the devices are more accurately captured through careful experimentation, rather than using the advertised performance. Hence, we propose to use our benchmarking methodology to characterize a storage device with respect to their asymmetry (α) and concurrency (k_r and k_w). This is a one-step analysis that allows to carefully tune PIO-aware algorithms or data structures.

3.5 Related Work

Existing Models. External storage has been traditionally modeled as a simple collection of blocks following the simplicity of the design of a hard disk (EM Model [5]). Blelloch et al. [21, 22] proposed the *Asymmetric RAM* (ARAM) model to analyze algorithms for asymmetric read and write costs, targeting asymmetric non-

volatile main memory devices. There are two primary differences between ARAM and PIO. First, ARAM targets main memory that has smaller access granularity. Second, it does not take into account the parallelism of modern storage devices which is central to PIO. The main goal of ARAM is to develop write-efficient main memory algorithms. On the contrary, the goal of PIO is to capture the inherent asymmetry and concurrency of *storage devices*, and study how we can use these in the design process of *storage-intensive algorithms*.

Addressing Read/Write Asymmetry and Concurrency. The read/write asymmetry on storage has been identified as an optimization goal for indexing [15, 30, 31, 99, 102, 192], flash-aware storage engines [124], and other data management operations [60, 137]. In the context of exploiting device parallelism, recent research builds new I/O schedulers for SSDs [113, 141, 165], and designs new data structures [26, 81, 153, 162]. Our work bridges these efforts on addressing asymmetry and concurrency under a unified approach, making both α and k first-class citizens of storage device modeling. PIO lays the groundwork for considering α and k at algorithm-design time, rather than as an optimization during deployment.

3.6 Conclusions

The classical I/O model which was developed considering traditional HDDs, cannot accurately model modern storage devices. Contemporary storage devices are characterized by a *read-write asymmetry* and an access *concurrency*, both of which are essential to fully utilize the device. We propose a simple yet expressive parametric I/O model, termed PIO, that considers the asymmetry (α) between reads and writes, and concurrency (k) that different devices may support to enable better algorithm design. By capturing α and k, device-specific decisions can be tuned at both algorithm design time and during deployment and testing. We illustrate the im-

pact of PIO on different classes of workloads and outline five guidelines that should drive storage-intensive algorithm design. We envision better algorithm design for *any* component of a system that interacts with the storage. Specifically, we envision an *asymmetry/concurrency-aware* bufferpool manager that prefetches reads concurrently and batches dirty evictions to exploit the device parallelism. Further, we envision that algorithms for tree and graph traversal can be redesigned to access multiple nodes concurrently, thus avoiding the classical paradigm of accessing one node at a time, ensuring better device utilization.

Chapter 4

Aymmetry/Concurrency-Aware (ACE) Bufferpool Manager

Modern SSD devices exhibit read/write asymmetry and concurrency. These characteristics have two key implications: (i) proper use of concurrency enables better device utilization, and (ii) treating page reads and writes equally in an asymmetric environment is suboptimal [130, 131]. However, many data-intensive systems have not been thoroughly redesigned to account for these characteristics resulting in device underutilization, which is typically addressed opportunistically by *device-specific tuning* during deployment. The bufferpool management of a Database Management System (DBMS) is tightly connected to the underlying storage device. State-of-the-art approaches treat reads and writes equally, and do not expressly exploit the SSD concurrency, leading to subpar performance.

In this chapter, we propose a new ***A****symmetry & **C**oncurrency-awar**e*** bufferpool management (**ACE**)[1] [132, 133] that batches writes based on device *concurrency* and performs them in parallel to amortize the *asymmetric* write cost. In addition, ACE performs *parallel prefetching* to exploit the device's *read concurrency*. ACE does not modify the existing bufferpool replacement policy, rather, it is a *wrapper* that can be integrated with *any replacement policy*. We implement ACE in PostgreSQL and

[1] The material of this chapter has been the basis for the ICDE 2023 paper "ACEing the Bufferpool Management Paradigm for Modern Storage Devices" [132], the SIGMOD 2025 demonstration paper "ACE-in-Action: A Smart DBMS Bufferpool for SSDs" [133] and the ICDE 2024 PhD Symposium "Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry" [129].

evaluate its benefits using a synthetic benchmark and TPC-C for several popular eviction policies (Clock Sweep, LRU, CFLRU, LRU-WSR). The ACE counterparts of all four policies lead to significant performance improvements, exhibiting up to 32.1% lower runtime for mixed workloads (33.8% for write-intensive TPC-C transactions) with a negligible increase in total disk writes and buffer misses, which shows that incorporating asymmetry and concurrency in algorithm design leads to more faithful storage modeling and, ultimately, to better device utilization.

4.1 Introduction

Bufferpool Manager. The part of a database management system (DBMS) that interacts directly with storage devices is the *bufferpool* which works as the interface between memory and the underlying storage device. The bufferpool keeps in memory a set of pages to minimize the number of (slow) disk accesses. If a requested page is already in the bufferpool, it can be served immediately without accessing the disk. In contrast, if the requested page is not available, it has to be fetched from the disk and placed in the bufferpool. If the bufferpool is already full, another page is first written back to disk (if dirty) and evicted, based on a *page replacement policy* [149]. In this chapter, we show that a bufferpool manager which is carefully tailored to the underlying storage device can significantly improve the system's performance.

The Challenge. In this chapter, we attempt to address two challenges of state-of-the-art bufferpool managers. (A) First, existing bufferpool managers often assume that the underlying devices have no concurrency ($k = 1$). When writing dirty pages to disk, state-of-the-art bufferpool managers write (evict) one page at a time, hence missing the opportunity to exploit the device concurrency. Although durability mechanisms like logging and checkpointing attempt to ameliorate the effect of random I/Os by converting them to sequential I/Os, the physical page writes from the bufferpool

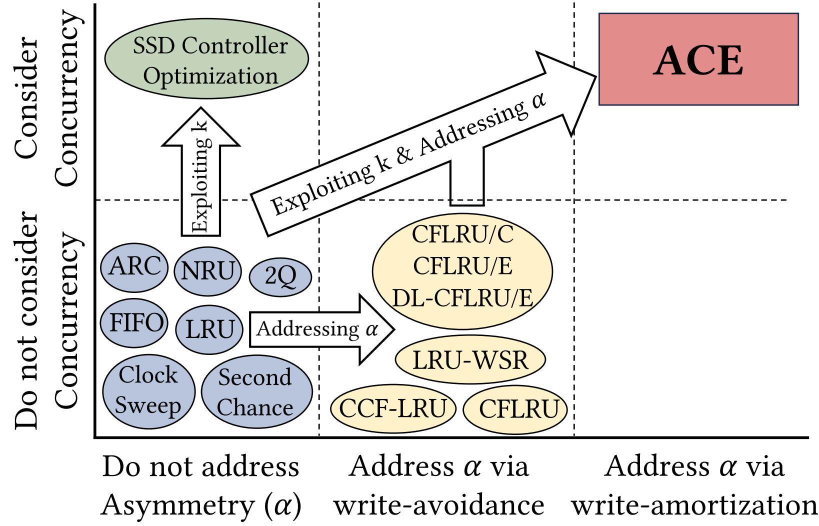


Figure 4·1: ACE addresses asymmetry by exploiting concurrency and amortizing writes.

are performed one I/O at a time, which is our primary focus. Furthermore, some systems employ data prefetching [120, 190], however, they primarily rely on sequential I/O instead of concurrent reading. (B) Second, page replacement policies generally do not consider the device asymmetry (α), instead, they treat read and write requests equally (i.e., they consider $\alpha = 1$). Hence, the *to-be-evicted* page's status (dirty or clean) does not depend on whether the requested page is a read or a write. As a result, it is entirely possible that the bufferpool evicts a dirty page (writes to disk) when the incoming page request is a read, essentially **exchanging a read for a write** irrespective of the device asymmetry, leading to suboptimal performance [130].

Figure 4·1 shows that popular page replacement policies are designed for devices with no asymmetry and concurrency (bottom left, blue). Recently proposed *flash-friendly* policies like CFLRU [139], LRU-WSR [80], and others [100, 207] try to minimize the number of writes by evicting clean pages first, indirectly addressing the asymmetry (bottom middle, yellow). However, these policies also exchange reads and writes interchangeably. There have been some efforts to utilize the device concurrency

via modifying the SSD internals [88, 162, 163] (top left, green). However, these solutions lack general applicability because they require redesigning of the SSD controller and they do not target the DBMS bufferpool. Hence, to the best of our knowledge, no bufferpool manager appropriately considers both asymmetry and concurrency.

A New Bufferpool Design Space. Traditionally, the design space of bufferpool management includes primarily a *page replacement policy* and optionally a *read-ahead policy*. The page replacement policy decides the order that pages are *evicted* and *written back*. If the evicted page is dirty, a write-back is issued for that page. Since traditional systems have a single policy for both eviction and write-back, they essentially make one decision for two separate questions: *which page to evict?* and *which page to write-back?* We separate these two questions by introducing a new *write-back policy*, thus, decoupling write-backs from eviction. We maintain one overall *virtual page ordering* of eviction (which is typically outsourced to the existing *replacement algorithm*), however, we have a different *virtual order for writing-back pages*, which depends on (a) the replacement algorithm, (b) whether the page is dirty, and (c) the support write concurrency of the storage device.

A bufferpool management approach can be described by four design decisions: (i) replacement algorithm, (ii) write-back policy, (iii) eviction policy, and (iv) read-ahead policy.

In this augmented design space, the *replacement algorithm* inform both the *write-back* and the *eviction* policies, however, in a different manner. The *write-back policy* uses the virtual order of pages dictated by the *replacement algorithm* and the degree of write concurrency of the device to write-back *only dirty* pages. The *eviction policy* uses the virtual order of pages dictated by the *replacement algorithm* to evict only clean pages (which may have been just written back or were already clean). The decision of how many pages to evict is decided from the application as a decision

between prioritizing locality (evict only one page) vs. prefetching (evict multiple pages, but use the *read-ahead policy* to populate the free spots).

Design Goals. With this refactored bufferpool design space, we set the following design goals :

- **Exploit concurrency** to ensure proper utilization of the device parallelism.
- **Bridge asymmetry** via write-amortization to ensure that there is no imbalance when the bufferpool is saturated.
- **Ease of adoption**, so that systems can quickly benefit from our design without extensive engineering effort.

Asymmetry & Concurrency-Aware Bufferpool Manager (ACE). To fulfill these goals, we propose ACE [132], a new bufferpool manager that utilizes the underlying device concurrency to bridge the device asymmetry (top right of Fig. 4·1 – colored red). Our approach uses *asymmetry/concurrency-aware* write-back and eviction policies. The write-back policy always writes multiple pages *concurrently* (utilizing the device's write concurrency), hence amortizing the write cost. The eviction policy evicts one or multiple pages at the same time from the bufferpool to enable prefetching. When multiple pages are evicted at once, ACE can *concurrently prefetch* pages to exploit the device's read concurrency. A key advantage of ACE is that it can be integrated with any existing page replacement policy with low engineering effort, while, any prefetching technique can also be integrated, essentially al-

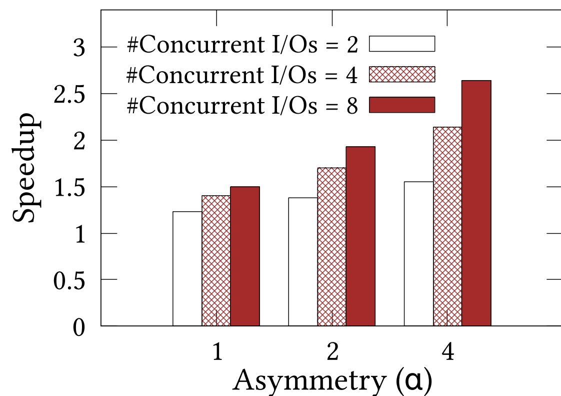


Figure 4·2: ACE outperforms state-of-the-art due to better device utilization.

lowing any existing bufferpool manager to be augmented by our approach. Figure 4·2 shows the *ideal speedup* of such an asymmetry/concurrency-aware bufferpool manager where the baseline system uses LRU. The benefit of ACE is higher for devices with higher asymmetry (up to 2.5×) showing that the asymmetry gap is increasingly important to bridge. We integrate ACE with four page replacement policies and implement them in PostgreSQL to evaluate ACE's efficacy. Our *asymmetry/concurrency-aware* bufferpool manager (i) batches write-back requests (from bufferpool to the disk) in a *storage-aware* manner and (ii) prefetches pages in parallel leading to substantial performance gains with a negligible increase in buffer miss and total writes.

Contributions. This chapter makes the following contributions.

- We refactor the bufferpool design space by including a write-back policy that consider device-specific properties.

- We propose **ACE**, an *asymmetry & concurrency-aware* bufferpool manager that utilizes the device's concurrency. ACE is flexible enough to be combined with any existing page replacement policy and prefetching technique.

- We implement ACE with PostgreSQL's default replacement algorithm (Clock Sweep) and we add three more replacement algorithms (LRU, CFLRU, LRU-WSR) and their ACE counterparts in PostgreSQL.

- We evaluate the efficacy of ACE against these four algorithms in PostgreSQL. ACE achieves up to 32.1% lower runtime for a mixed workload with negligible increase in total writes and buffer misses. For the TPC-C mix, ACE reduces runtime by up to 24.2% while attaining 33.8% lower runtime for the write-heavy transaction.

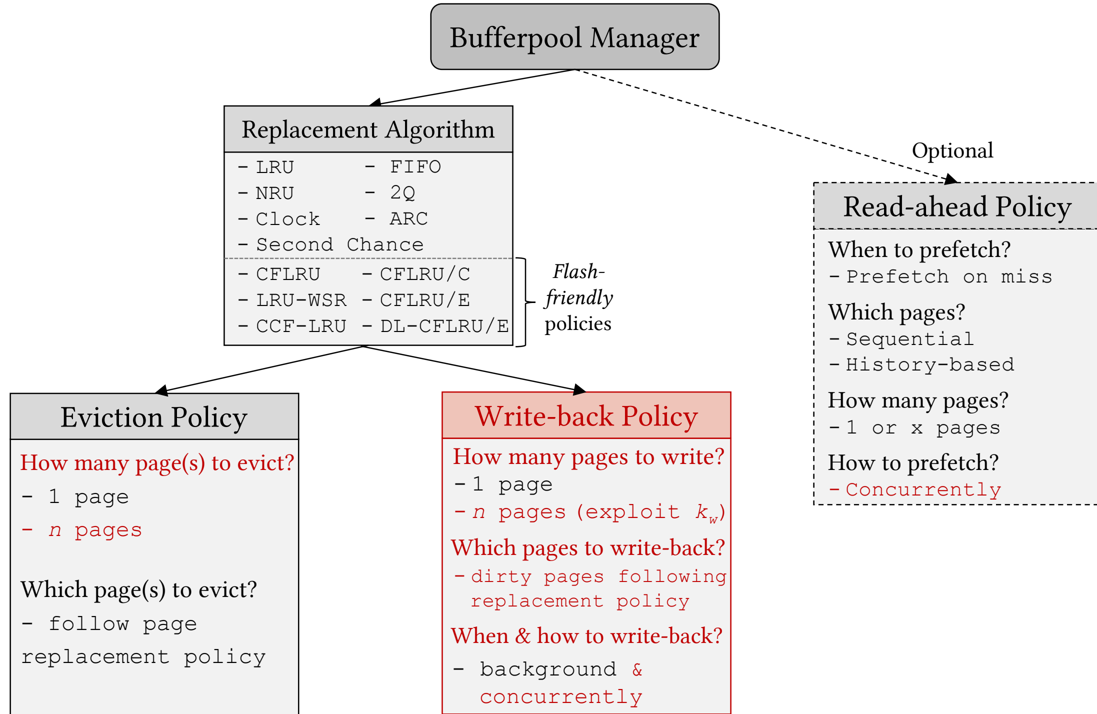


Figure 4·3: Bufferpool design space in terms of the design decisions and various options (RED denotes new components)

4.2 An Augmented Bufferpool Design Space

Traditional bufferpool designs have one main component: the *page replacement policy* (that may lead to a write-back if a dirty page is evicted), which is driven by the replacement algorithm and one optional component: a prefetching module. We depart from this paradigm and separate the write-back decision from the replacement policy. We propose a new bufferpool design paradigm that makes four decisions: (i) **a replacement algorithm** that decides the *to-be-evicted page order* and influences the *to-be-written page order*, (ii) **a write-back policy** that decides when, how many, and which pages to write back, (iii) **an eviction policy** that decides how many and which pages to evict, and (iv) **a read-ahead policy** that decides when, how many, which pages, and how to prefetch. Figure 4·3 presents the proposed augmented bufferpool design space along with the various options for each design decision.

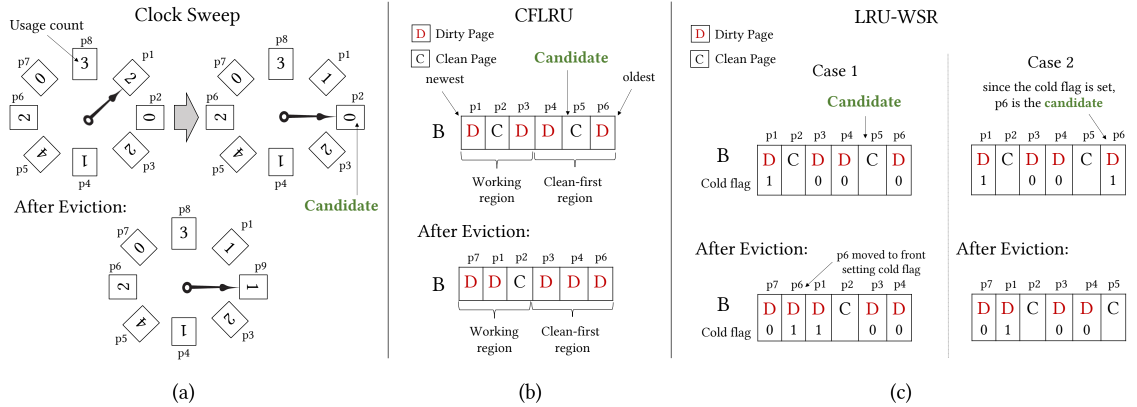


Figure 4·4: Popular eviction policies: (a) Clock Sweep evicts pages based on *usage count*; (b) CFLRU tries to evict *clean* pages first [window size $N/2$ used for brevity]; (c) LRU-WSR keeps a *cold* flag to delay dirty page eviction.

4.2.1 Background on Page Replacement Algorithm

At the core of every bufferpool design is the page replacement algorithm which decides which page needs to be replaced when the bufferpool runs out of space. Note that this decision essentially creates a *virtual order* of the pages to be evicted. The most popular page replacement algorithm is Least Recently Used (LRU) [126] which tries to keep the most recently accessed pages in the bufferpool. Some other popular algorithms are Clock [87], NFU [182], 2Q [77], NRU [47], FIFO [182], ARC [116], and Second Chance [86]. PostgreSQL adopts the Clock Sweep algorithm [146], a variant of NFU. The algorithm maintains the bufferpool pages as a circular list with each page's *usage count*, while the *candidate* pointer rotates clockwise. If the candidate unpinned page's usage count is 0, the page is selected for eviction. Otherwise, its usage count is reduced by 1, and the pointer moves ahead clockwise (Figure 4·4a). This algorithm and all the aforementioned algorithms do not differentiate between reads and writes. In recent literature, when optimizing for SSDs, *flash-friendly* policies reduce device wear by absorbing more writes in memory before evicting a dirty page.

Flash-friendly Policies. Clean-First LRU (CFLRU) maintains the LRU order of the pages and divides the LRU list into two regions: *working region* and *clean-first region* [139]. The working region contains the recently accessed pages, while the clean-first region contains the candidate pages for eviction (Figure 4·4b). To minimize the number of writes, CFLRU evicts clean pages from the clean-first region and when there are no clean pages left in this region, CFLRU evicts dirty pages following LRU. The size of the clean-first region is decided by the *window size* parameter. Although the optimal *window size* depends on the workload, a rule of thumb is that if N is the size of the bufferpool, $\frac{N}{3}$ is a good window size [139].

Another flash-friendly algorithm is LRU with Write Sequence Reordering or LRU-WSR [80]. LRU-WSR delays evicting *cold* dirty pages to reduce the number of writes. Each page in LRU-WSR has a *cold* flag which is cleared every time the page is referenced. If the candidate page for eviction is dirty, it is evicted only if its cold flag is set; otherwise the page is moved to the most recently used position while setting the cold flag and another candidate page is selected based on the LRU order. If the candidate page is clean, it is evicted irrespectively of its cold flag. Figure 4·4c shows two examples of LRU-WSR. Both CFLRU and LRU-WSR outperform LRU for flash devices, because they prioritize evicting clean pages over dirty pages, indirectly addressing the underlying device asymmetry. Other flash-aware policies [100, 207] adopt similar strategies while considering the access frequency and wear-leveling. All these policies prioritize reads over writes to mitigate device wear caused by writes. While they indirectly address asymmetry up to a point, they do not explicitly consider the specific device asymmetry and concurrency.

4.2.2 Write-back Policy

Once a *dirty* page is selected for replacement, the bufferpool has to write the page back to disk. In practice, state-of-the-art systems like PostgreSQL have a separate

dirty page buffer to which dirty pages for replacement are first moved, and flushed in the background. Further, page updates are also written in the write-ahead log (WAL) sequentially to make the database crash-resilient, and dirty pages are written-back during checkpointing as well. However, all these write-backs are performed *one page at a time* following the underlying assumption that storage devices can perform one I/O at a time, and rely on the operating system to re-order or batch disk writes. In this way, the read request of a new (not buffered) page is countered by one write-back when the page for replacement is dirty, *exchanging one write for one read*. Since writes are more expensive than reads in modern storage devices, this is an unfair decision that leads to performance degradation, which we also observe experimentally. To address this, **we augment the write-back policy to write multiple dirty pages at once** following the *virtual order* provided by the page replacement algorithm. Note that we can write-back multiple pages concurrently without a penalty due to the underlying write concurrency of the device (k_w). In order to bridge the asymmetry (α), we parallelize at least α writes to amortize their cost for the single read that caused this write-back, or more if the device concurrency permits (if $k_w > \alpha$).

4.2.3 Eviction Policy

Following the write-back phase, the eviction policy now decides which and how many pages to evict. Traditionally, systems evict always one page which leads to the one read for one write approach. Since following the new write-back policy we have already written back more than one page, the eviction policy may opt to evict more (now clean) pages to make room for additional pages to be prefetched, assuming the prefetcher has high confidence for its predictions. Hence, we include one additional design choice: **the number of pages to be evicted**. The exact pages to be evicted still follow the virtual order imposed by the replacement algorithm. By splitting the page replacement policy into a write-back and an eviction phase, we exploit the

underlying concurrency of the storage device to bridge the asymmetry between reads and writes, using the replacement algorithm as the core decision throughout both phases. This is why our approach benefits any bufferpool page replacement policy.

4.2.4 Read-ahead Policy

The goal of prefetching is to make data available in memory before it is requested, consequently improving the bufferpool hit rate. The prefetching policy determines when to prefetch, how many and which pages to prefetch. The most common prefetching approach is *sequential prefetching* like One Page Lookahead (OPL) and N-Page Lookahead (NPL), where either a single page (OPL) or multiple pages (NPL) beyond the requested page is prefetched [39, 120, 174]. History-based prefetching uses previous access patterns to predict the next pages to be accessed [59, 79, 98]. This type of prefetching improves performance when accesses are localized. Most commercial systems use simple prefetching policies like sequential prefetching [66, 147], especially when the bufferpool has empty slots. However, systems generally prefetch one page at a time instead of issuing multiple parallel I/Os, hence missing the opportunity to exploit the device's read concurrency. Since prefetching is not as effective without good workload knowledge, it is often treated as an optional choice.

4.3 ACE Bufferpool Manager

We now present in detail the proposed *asymmetry & concurrency-aware* bufferpool manager (**ACE**) [132] that addresses the read/write asymmetry via write-amortization. As depicted in Figure 4·5, ACE is comprised of three components: (i) the Evictor, (ii) the Writer, and (iii) the Reader. The **evictor** determines which page(s) to evict, the **writer** writes-back concurrently dirty pages and the **reader** prefetches pages.

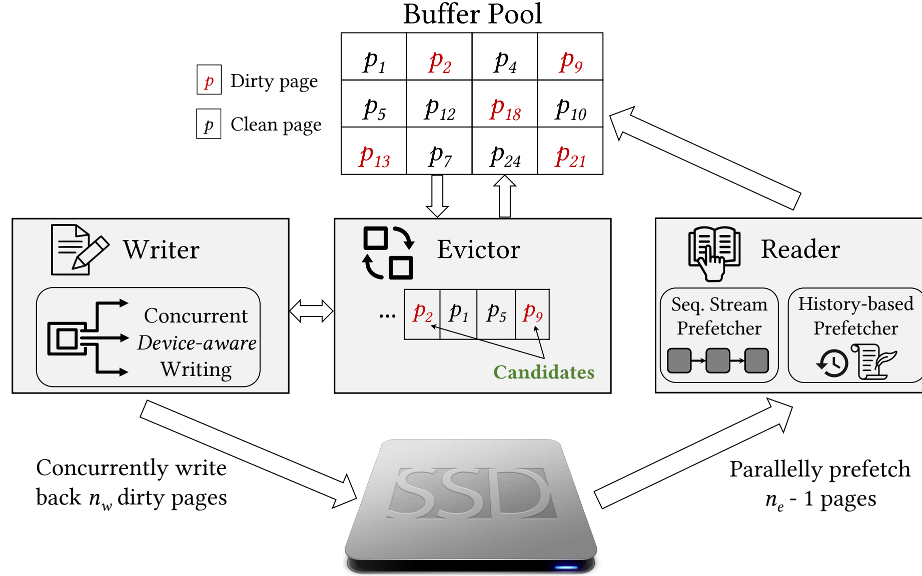


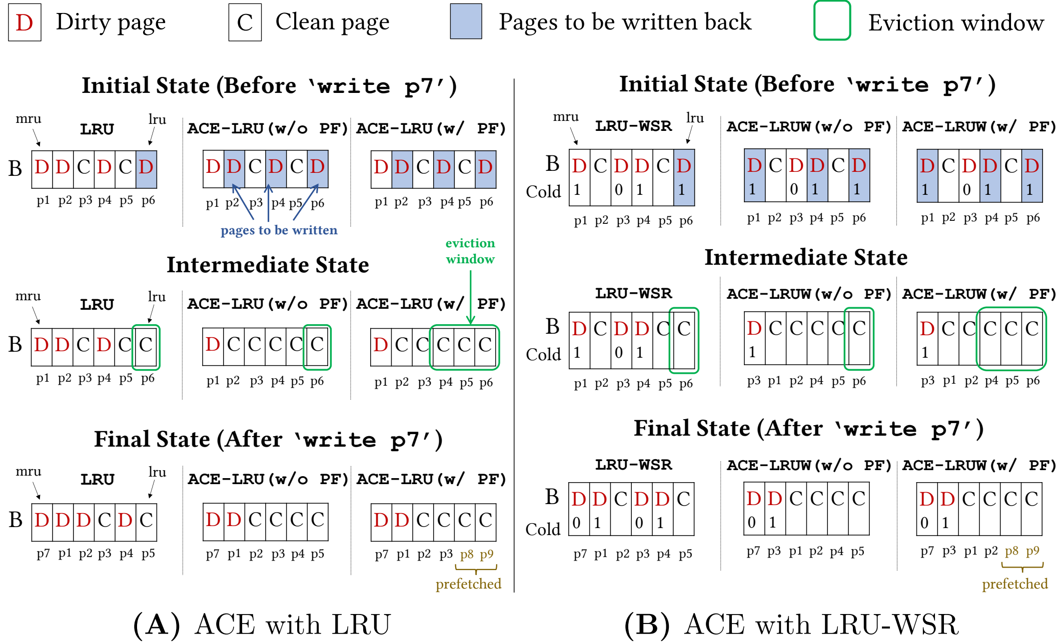
Figure 4·5: Abstract overview of **ACE** components

4.3.1 Overview of ACE Bufferpool Management

When a request for reading or writing a page P is received, we first search through the bufferpool. If P is not found and the bufferpool is full, then (at least) one page has to be evicted. The page replacement algorithm determines the page to be evicted (termed *top page*). If the top page is *clean*, it is evicted and page P is fetched. Up until this part, ACE is identical to any state-of-the-art bufferpool management. However, if the top page is *dirty*, ACE proceeds as follows:

- **ACE without prefetching**: *concurrently write* $\boldsymbol{n_w}$ dirty pages and evict a single page.
- **ACE with prefetching**: *concurrently write* $\boldsymbol{n_w}$ dirty pages, evict $\boldsymbol{n_e}$ pages, and concurrently prefetch $\boldsymbol{n_e - 1}$ pages.

The values n_w and n_e depend on the underlying device concurrency and the potential benefits of prefetching. When prefetching is enabled, ACE evicts n_e pages in order to prefetch $n_e - 1$ pages exploiting the read concurrency of the device. While we



(A) ACE with LRU

(B) ACE with LRU-WSR

Figure 4·6: ACE page selection policies for $n_w = 3$ and $n_e = 3$. (a) ACE writes three dirty pages (p6, p4, p2) following the LRU order; if prefetching is enabled three pages (p6, p5, p4) are evicted, otherwise one page (p6) is evicted. (b) For LRU-WSR, ACE finds a dirty page with cold flag not set (p3). This page is moved to the front setting its cold flag. The dirty pages with set cold flag (p6, p4, p1) are selected for concurrent writing.

anticipate that the prefetching will allow us to have pages that will be accessed by the immediate next requests, the eviction somewhat reduces the locality, so n_e has to carefully balance the read concurrency and the accuracy of the prefetching. We tune ACE to use n_w equal to the optimal write concurrency of the device (k_w). We experimentally tested values for n_e between 1 and k_r, and we empirically set n_e to be also k_w, because evicting k_r pages was hurting locality. Note that for most devices, the read concurrency is significantly higher than the write concurrency ($k_r >> k_w$). Regarding the pages that are selected for write-back and for eviction, both decisions are influenced by the page replacement algorithm. As such, ACE can be combined with any replacement algorithm. In our experiments, we use four popular approaches (LRU,

Clock Sweep, CFLRU, and LRU-WSR) that all benefit from the ACE paradigm. Figure 4·6 shows the effect of incorporating ACE with LRU (Fig. 4·6A) and LRU-WSR (Fig. 4·6B). Note that ACE always writes n_w dirty pages concurrently irrespectively of prefetching. The full ACE algorithm is listed in Algorithm 1.

4.3.2 Writer

The Writer is responsible for concurrently writing-back n_w pages. State-of-the-art systems often write-back pages using a background process, however, these writes are issued one at a time, hence missing out on the opportunity to exploit the parallelism of the underlying storage device. Instead, ACE Writer writes concurrently n_w *dirty* pages. By making sure that $n_w = k_w$, the concurrent writes take place at the same latency as a single write, thus amortizing the cost of k_w writes and fully bridging the read/write asymmetry if $\alpha < k_w$. The pages that are selected for write-back are the next n_w *dirty* pages that the underlying page replacement algorithm would eventually evict. As a result, these carefully batched writes make the subsequent page evictions free (since, with high probability, the following evictions will target clean pages).

4.3.3 Evictor

Following the completion of the write-back process, the Evictor will evict either one or n_e pages. Since at this point, the *top* page is by definition clean (because the Writer has already written it), it will be evicted to read the requested page P. If prefetching is enabled, the Evictor evicts n_e pages in total to allow for an equal number of pages to be prefetched. Note that after the write-back process, there will be at least n_w contiguous clean pages following the order dictated by the page replacement algorithm. Hence, the Evictor can now evict n_e clean pages to create space for the incoming pages. Earlier, we mentioned that we empirically set $n_e = n_w$, however, even if we allow n_e to be greater than n_w, the Evictor will always be able to evict

n_e pages as long as in total the bufferpool has at least n_e clean pages. Essentially, the Writer writes back the n_w first *dirty* pages according to the page replacement algorithm order, and the Evictor evicts the n_e first *clean* pages (after the writing has been performed) according to the page replacement algorithm order. For example, Figure 4·6a shows that ACE with prefetching will write three dirty pages following the LRU order (p6, p4, p2) and evict the last three (now clean) pages (p6, p5, p4) following the LRU order.

4.3.4 Reader

The reader is an optional component whose job is to prefetch pages from disk in case of a buffer miss. Note that for many workloads prefetching does not attain much benefit, hence, commercial systems either do not use any prefetcher, or they use very simple prefetching techniques. The strength of ACE is that any prefetching technique can be employed by the Reader. In fact, we use two prefetchers in our design: a sequential prefetcher and a history-based prefetcher.

Sequential Prefetcher. We use a sequential prefetcher named TaP [98] that uses a table to detect sequential access patterns. The prefetcher uses a *sequential detection module* to determine whether a page miss is part of a sequential stream. Only after detecting a sequential stream, ACE uses the sequential prefetcher, otherwise, ACE uses the history-based prefetcher. When a page miss occurs, the sequential detection module searches the page address in the TaP table. If it is not found in the table, then the address of the next page is inserted in the table, expecting that the current page miss is part of a new sequential stream. In case of a sequential stream, the newly inserted page in the TaP table will be found in the workload soon. Then, the TaP prefetcher starts sequential prefetching, however, ACE does not immediately start prefetching. Instead, if at least 4 sequential page requests are found, then ACE triggers the prefetcher and concurrently reads the next $n_e - 1$ pages along with the

page that caused the buffer miss (P). Old page addresses that are not part of any sequential stream are evicted in a FIFO manner.

History-based Prefetcher. This prefetcher also uses a large table-based structure to store the page access history and predicts the next most probable access [59]. The organization of the table is shown in Figure 4·7. Row i of the table contains the next probable page addresses and their likelihood (in form of a weight) after page i is accessed. For example, given a reference for page 1, the best candidate for prefetching is page 3 since page 3 has a higher weight (9). ACE prefetches only if the weight is more than a certain fetch threshold. The table generation is straightforward: given the current and previous page references, we go to the row in the table indexed by the page number of the previous access. If the `NextPages` vector contains the current page, we increase its corresponding weight in the `Weights` vector. Otherwise, if the `NextPages` vector does not point to the current page and its weight is zero, we place a pointer to the current page in the vector with weight 1. If the weight field is nonzero, we decrease the weight. To prevent growing the `NextPages` vector perpetually, ACE keeps track of the next 3 most probable pages. This table structure takes 0.6% space of the database size.

Index	NextPages	Weights
0	{2, 5, 7}	{10, 4, 2}
1	{10, 3, 18}	{3, 9, 1}
2	{6, 9}	{8, 0}
	⋮	⋮

Figure 4·7: Table structure of the history-based prefetcher

4.3.5 Putting Everything Together

Traditional bufferpool strategies "exchange" one read for one write when evicting a dirty page from a saturated bufferpool. This approach is not optimal when a write is $\alpha\times$ more expensive than a read. However, it is not possible to exchange α reads for one write, since (i) this would grow the bufferpool perpetually, and (ii) unless we batch – thus delay reads, a system receives one request at a time. Hence, ACE tries

Algorithm 1: ACE

```
    Input: P, n_w, n_e is_pf_enabled
1   // P is the accessed page
2   // n_w is the maximum effective write concurrency (n_w = k_w)
3   // n_e is the number of concurrent reads during prefetching
4   // is_pf_enabled determines if prefetching is enabled or not
5   if P in buferpool then
6   |   return P
7   else
8   |   // miss!  need to bring P from disk
9   |   if buferpool not full then
10  |   |   if is_pf_enabled == true then
11  |   |   |   // reads P and prefetches up to n_e - 1 pages from disk (depending on available slots)
12  |   |   |   - prefetch_pages (P, n_e - 1)
13  |   |   else
14  |   |   |   - read P from disk
15  |   |   end if
16  |   else
17  |   |   top_page = replacement_policy.get_one_page_to_evict()
18  |   |   if top_page is clean then
19  |   |   |   // follow classical approach if page is clean
20  |   |   |   - drop top_page from bufferpool
21  |   |   |   - read P from disk
22  |   |   else
23  |   |   |   // top_page is dirty.  concurrently write n_w dirty pages
24  |   |   |   // P_wb is a vector containing the candidate dirty pages
25  |   |   |   - P_wb = populate_pages_to_writeback()
26  |   |   |   - issue ||length(P_wb)|| concurrent writes, ∀p ∈ P_wb
27  |   |   |   - mark ||length(P_wb)|| pages as clean, ∀p ∈ P_wb
28  |   |   |   if is_pf_enabled == true then
29  |   |   |   |   // evict n_e pages
30  |   |   |   |   // pages written and to be evicted can be different
31  |   |   |   |   // P_ev is a vector containing the pages to evict
32  |   |   |   |   P_ev = replacement_policy.get_n_pages_to_evict()
33  |   |   |   |   - drop ||length(P_ev)|| pages from bufferpool, ∀p ∈ P_ev
34  |   |   |   |   // Now, prefetch
35  |   |   |   |   - prefetch_pages (P, n_e - 1)
36  |   |   |   |   - empty P_ev
37  |   |   |   else
38  |   |   |   |   // evict 1 page
39  |   |   |   |   - drop top_page from bufferpool
40  |   |   |   |   - read P from disk
41  |   |   |   end if
42  |   |   |   - empty P_wb
43  |   |   end if
44  |   end if
45  end if

1   Procedure populate_pages_to_writeback()
2   |   // follow the underlying page replacement policy to generate P_wb
3   |   - select next n_w dirty pages based on the underlying page replacement policy
4   |   - return this vector

1   Procedure prefetch_pages(page P, int x)
2   |   if P in Sequential_Table then
3   |   |   // start of a sequential stream!
4   |   |   // read P and the next x pages concurrently
5   |   |   - prefetch_sequential (P)
6   |   else
7   |   |   // use the history based prefetcher
8   |   |   // read P and x pages (selected by prefetcher) concurrently
9   |   |   - prefetch_history (P)
10  |   end if
11  |   /* note that P should be placed in the most recently used position in the bufferpool whereas other
    |      pages should be placed in the least recently used positions                                  */
12  |   - place these x + 1 pages into bufferpool
```

to *amortize* the cost of writes by concurrently issuing $\boldsymbol{n_w}$ writes, where $n_w = k_w$. The complete ACE bufferpool manager's policy is presented in Algorithm 1. If the page to evict is clean, ACE follows the classical approach of simply dropping it from the bufferpool (Line 20). Otherwise, the ACE Writer identifies the pages to be cleaned and writes them concurrently (Lines 25–27). Depending on whether prefetching is enabled, ACE Evictor either evicts multiple (Lines 32–33) or one page (Line 39). The referenced page is placed in the most recently used position while the prefetched pages are placed in the least recently used positions (following the page replacement policy) so that even if the prefetcher's prediction is wrong, the prefetched page can be simply dropped from the bufferpool.

4.4 Implementation & Integration

We now discuss the implementation effort and the integration with a full-blown relational system, PostgreSQL 11.5. We first discuss the bufferpool manager structure and its operation. We then discuss the implementation effort to make PostgreSQL bufferpool asymmetry and concurrency aware.

PostgreSQL Buffer Manager. The PostgreSQL buffer manager consists of a buffer table, buffer descriptors, and a bufferpool. The buffer table and the buffer descriptors hold page metadata and the mapping information between the data pages and bufferpool frames. The bufferpool layer keeps track of file pages (i.e., data, indexes). The bufferpool is organized as an array, where each slot stores one page of a data file, and is referenced by a `buffer_id`. In PostgreSQL, each page is assigned a unique tag (`buffer_tag`), which contains the file mapping and block location and is used when the buffer manager receives a request. During an eviction, the buffer manager uses the *Clock Sweep* page replacement algorithm. Dirty pages are flushed to storage by two background processes: the **background writer** and the **checkpointer**.

Both processes flush dirty pages, however, they have different roles and behaviors. Checkpointer writes a checkpoint record to the WAL file and flushes all the buffered dirty pages during checkpointing. The background writer continues to flush dirty pages in the background to offload the burden of the checkpointer. PostgreSQL uses light-weight locks (of type *content_lock*, *spin_lock* and *io_in_progress_lock*) to protect shared resources and data consistency.

Implementation. To ensure an apples-to-apples comparison we integrate LRU, CFLRU, LRU-WSR, and their ACE counterparts (including the default Clock Sweep) in PostgreSQL. Since all three algorithms are LRU variants, we first implement LRU using an LRU *freelist* queue and then build on it for CFLRU and LRU-WSR. The window size for CFLRU is set to 1/3 of the bufferpool size as suggested by its authors [139]. For LRU-WSR, we added a *cold* bit with each page descriptor which is checked during eviction. The `buffer_tag` of the pages were used to get metadata information of the corresponding page. We follow the default Clock Sweep implementation's locking mechanism to ensure data consistency.

ACE is implemented as a wrapper on top of the underlying page replacement policy. If the candidate page for eviction is dirty, ACE first identifies which n_w pages to write following the replacement policy. We implement a method named `FlushNBuffer()` in `bufmgr.c` which takes the candidate pages as input and flush them out to kernel. Further, we modify the background writer and checkpointer's writing mechanism (which is separate from the bufferpool code) to ensure that they always perform n_w writes concurrently. To do this, we augment several PostgreSQL's low-level writing methods (i.e., `pg_pwrite()`) and their corresponding wrapper functions. After the pages are flushed, the eviction (of one or multiple pages depending on prefetching) is performed as described in Section 4.3.3. Each of the two prefetchers employed by ACE uses a hash table: the sequential prefetcher to implement the TaP

table structure and the history-based one to keep track of the accesses following each page. Throughout the bufferpool implementation, the `buffer_id` and the `buffer_tag` are frequently used to identify each page, the frame it is stored, and which relation is associated with.

4.5 Evaluation

We now show the benefits of the ACE paradigm when applied on four state-of-the-art page replacement policies (LRU, CFLRU, LRU-WSR, and Clock Sweep) using both a synthetic benchmark and the standard TPC-C benchmark.

Experimental Setup. We use a machine with two Intel Xeon Gold 6230 2.1GHz processors each having 20 cores with virtualization enabled and with 384GB of main memory. Our experiments involve three storage devices: (i) a 375GB Optane P4800X SSD, (ii) a 1TB PCIe P4510 SSD, and (iii) a 240GB SATA S4610 SSD. We refer to these devices as *Optane SSD*, *PCIe SSD* and *SATA SSD* respectively. In addition, we use a *virtualized* device from Amazon AWS that has 1.2TB SSD capacity and 60000 provisioned IOPS (high-performance SSD). We refer to this device as *Virtual SSD*, and we attach it to a machine from the *t2.micro* family having 2GB main memory with one virtual CPU. For all four devices, we quantify the asymmetry and concurrency through careful benchmarking [130] (summarized in Table 4.1). Unless otherwise mentioned, we use $n_w = k_w$ of the device in use. In most of our experiments, we employ the PCIe SSD, hence we use $n_w = 8$. Before running the experiments, all devices were pre-conditioned by sequentially writing on the entire device three times to ensure that they have stable performance [44].

Workload. We use four synthetic workloads inspired by prior work [100, 207]. We refer to them as MS (Mixed Skewed), WIS (Write-Intensive Skewed), RIS (Read-Intensive Skewed) and MU (Mixed Uniform). The properties of the workloads are

Table 4.1: Empirical α and k of our SSDs.

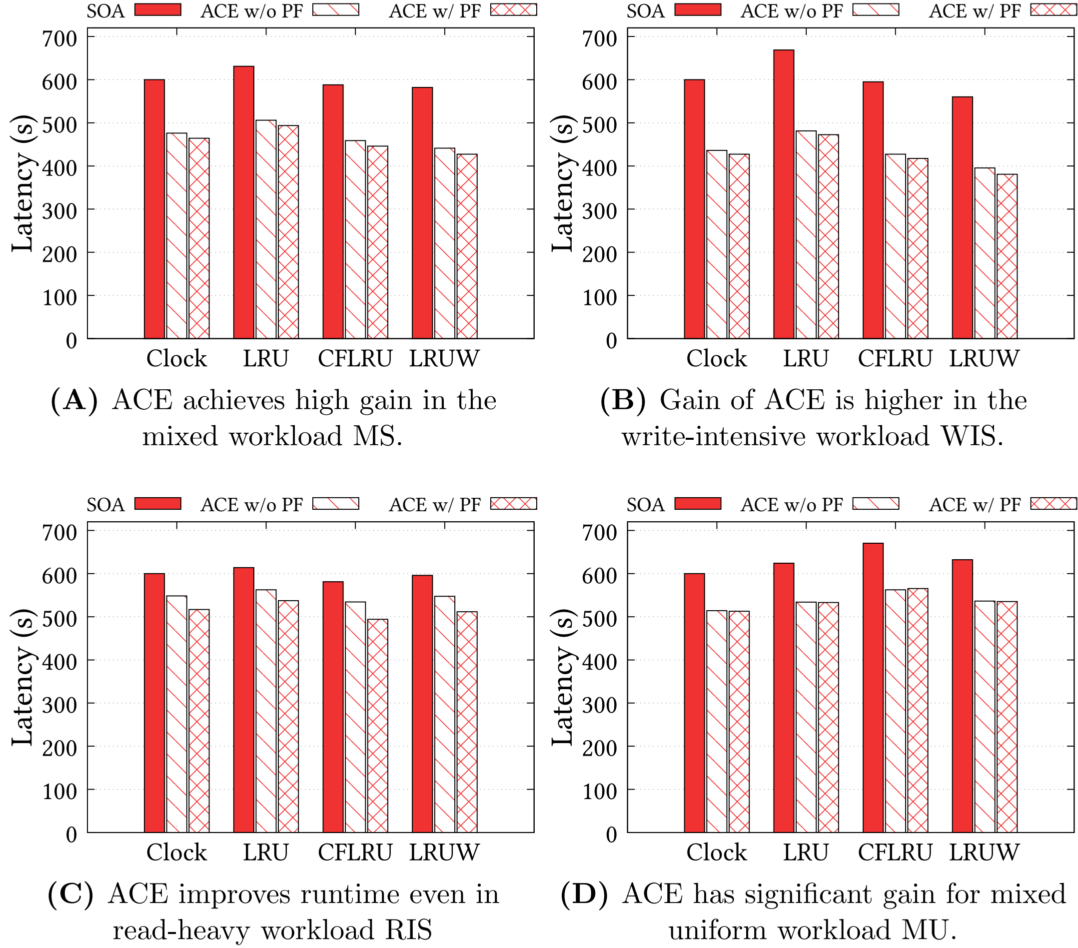
Device	α	k_r	k_w
Optane SSD	1.1	6	5
PCIe SSD	2.8	80	8
SATA SSD	1.5	25	9
Virtual SSD	2.0	11	19

described in Table 4.2. A read/write ratio of 90/10 indicates read (write) operations are 90% (10%) of the total number of operations. A locality 90/10 means that 90% of all the operations are performed on 10% of the pages. We use *pgbench* for our synthetic workloads which is loosely based on TPC-B. We use a scaling factor of 1000 which results in a database size of approximately 15GB. We also show the benefits of our approach with the TPC-C benchmark [186].

Table 4.2: Properties of the synthetic workloads

Workload	Database Size	R/W Ratio	Locality
Mixed Skewed (MS)	15GB	50/50	90/10
Write-Intensive Skewed (WIS)	15GB	10/90	90/10
Read-Intensive Skewed (RIS)	15GB	90/10	90/10
Mixed Uniform (MU)	15GB	50/50	50/50

Experimental Methodology. We run every workload for the default PostgreSQL implementation (Clock Sweep as replacement policy) for 10 minutes and then run the same workload for the other replacement policies and their ACE counterparts. For every experiment, we measure (i) workload latency, (ii) transactions per second, (iii) buffer misses/hits, and (iv) total writes. The experiment results are averaged over 5 iterations and the standard deviation was less than 5%. We generally configure PostgreSQL *shared_buffers* (bufferpool) as 1GB (∼6% of the data size). WAL is



(A) ACE achieves high gain in the mixed workload MS.

(B) Gain of ACE is higher in the write-intensive workload WIS.

(C) ACE improves runtime even in read-heavy workload RIS

(D) ACE has significant gain for mixed uniform workload MU.

Figure 4·8: ACE reduces total workload latency for all Clock Sweep, LRU, CFLRU, and LRU-WSR in the PCIe SSD.

enabled and the WAL file is written in a separate device following common practice.

4.5.1 Experimental Analysis with Synthetic Data

ACE Improves Runtime. Our first experiment shows that ACE bufferpool management (either with or without prefetching) reduces the total workload latency by up to 32.1%. For this set of experiments we use the PCIe SSD that has $k_w = 8$ and $\alpha = 2.8$. Figures 4·8A-D show the workload execution time for the baseline Clock Sweep, LRU, CFLRU, and LRU-WSR along with their ACE counterparts with and without prefetching for the 4 synthetic workloads in PostgreSQL. The runtime of the

ACE policies both with and without prefetching is consistently faster than the baseline. Since ACE policies utilize the device's write parallelism, it writes back pages more aggressively (but hidden due to the device concurrency), resulting in better performance. ACE with prefetching reduces latency by 22.6%, 21.8%, 22.5% and 26.1% for baseline Clock Sweep, LRU, CFLRU and LRU-WSR respectively when running workload MS (Figure 4·8A), while ACE without prefetching reduces latency by 20.6%, 19.7%, 22.0%, and 23.5% respectively. Since the workload is skewed, the prefetching helps avoid some disk access, resulting in slightly better performance. ACE's gain is higher for the write-intensive workload WIS. ACE with prefetching achieves 28.8%, 29.3%, 30.1% and 32.1% lower runtime than baseline Clock Sweep, LRU, CFLRU and LRU-WSR respectively (Figure 4·8B). This is expected, because for a write-intensive workload, the bufferpool needs to write more pages back to the disk, hence the benefits that come from efficient writing are pronounced. In contrast, for the read-intensive skewed workload RIS, ACE has smaller benefit since ACE does not have enough writes to optimize. However, the gain is still significant (Figure 4·8C); ACE achieves 8.1% to 13.9% lower runtime. Now, the benefit of prefetching alone is substantial (up to 5%). Finally, the mixed workload MU causes a small increase in total writes ($\leq 0.1\%$), however, this does not affect the overall trends of performance gains, which range from 14.5% to 15.7% (Figure 4·8D). Since the workload is uniform, the impact of prefetching is insignificant, however, because the prefetched pages are placed last in the virtual order, prefetching does not hurt performance either. In the remainder of our experiments, unless otherwise mentioned, prefetching is enabled.

Performance Gains Do Not Come at a Cost. *We highlight that the overall workload latency improvement observed in these experiments (up to* 32.1%*) does not come at a hidden cost.* Table 4.3 compares the buffer misses, application writes and total physical writes for ACE with prefetching enabled for different workloads. The

Table 4.3: Comparison of buffer miss and logical/physical writes

WL	ACE - Clock			ACE - LRU			ACE - CFLRU			ACE - LRU-WSR		
	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes
MS	-0.001%	0.07%	0.10%	-0.001%	0.06%	0.09%	-0.001%	0.08%	0.12%	-0.001%	0.09%	0.14%
WIS	-0.001%	0.08%	0.12%	-0.001%	0.08%	0.14%	-0.001%	0.08%	0.14%	-0.001%	0.11%	0.17%
RIS	-0.008%	0.06%	0.09%	-0.008%	0.06%	0.09%	-0.008%	0.07%	0.11%	-0.009%	0.12%	0.16%
MU	0.002%	0.09%	0.14%	0.003%	0.10%	0.15%	0.002%	0.09%	0.14%	0.002%	0.10%	0.17%

table shows the percentage difference in buffer miss, logical writes and physical writes between ACE[2] and the baseline values. The maximum increase in buffer misses is 0.003% for ACE-LRU on workload MU, and the maximum increase in total writes is 0.12% for LRU-WSR on workload RIS, thus being negligible. Since ACE writes multiple pages at once, it is possible that after writing a page (but not evicting), another write comes in for the same page, consequently, increasing the total number of logical writes slightly. The table also shows that for skewed workloads, ACE with prefetching attains lower buffer misses (up to 0.009%) compared to the baseline algorithms. In contrast, the number of buffer misses increases slightly for the uniform workload because prefetching can not help much in that case.

Impact on SSD Wear Out. We now analyze the impact of ACE on SSD wear out. Since every cell in a NAND flash can sustain a certain number of erases, the total number of writes primarily contributes to an SSD's wear out. We again refer to Table 4.3 where we list the percentage difference of both logical and physical writes for different workloads with different page replacement policies. We capture the SMART (Self-Monitoring, Analysis and Reporting Technology) [173] attributes to

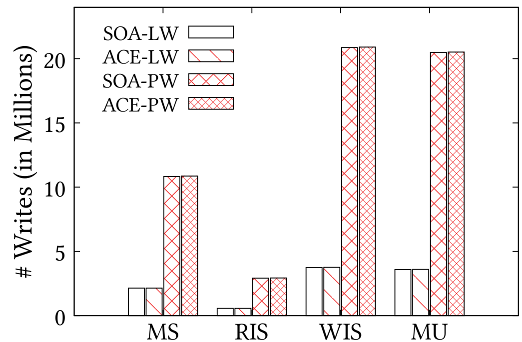
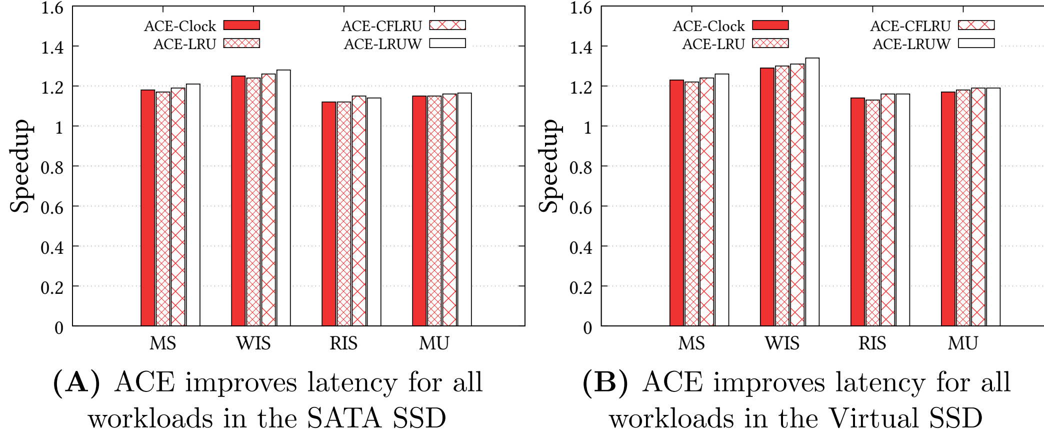


Figure 4·9: ACE causes very small increase in total number of writes

[2] We report the numbers for ACE policies with prefetching enabled as they cause a higher number of writes.



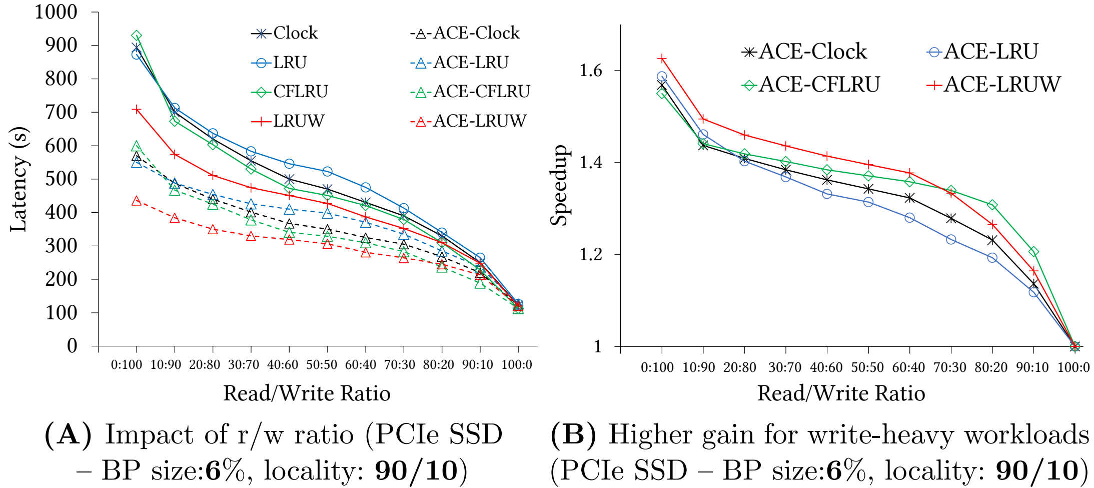
(A) ACE improves latency for all workloads in the SATA SSD

(B) ACE improves latency for all workloads in the Virtual SSD

Figure 4·10: ACE improves runtime for devices with low asymmetry.

collect the number of physical writes on the SSD. The table shows that the maximum increase in physical writes is 0.17% while the maximum increase in logical writes is 0.14%. Figure 4·9 shows the logical and physical writes (LW and PW) on our PCIe SSD as we run our synthetic workloads in PostgreSQL for an extended period (30 mins) with LRU-WSR and ACE-LRU-WSR. Note that, the physical writes are approximately 5-6$\times$ higher than logical writes due to flash's garbage collection and wear-leveling [73]. While the total number of writes (logical and physical) for ACE and its baseline replacement policy remains almost the same, the speedup for ACE can be as high as 1.35$\times$, with a negligible increase in physical writes (0.17%). For a read-heavy workload, ACE's performance benefit is small but still comparable to the negligible increase in physical writes. At any rate, the ACE paradigm is always beneficial regarding the overall workload latency, even with a very small fraction of writes. On the other hand, for a purely read-only workload, there will be no performance benefit (but also no increased write amplification or SSD wear out since there will be no writes).

Better Performance Even in Low Asymmetry Devices. To analyze the impact of ACE for devices with low asymmetry, we run our synthesized workloads on the



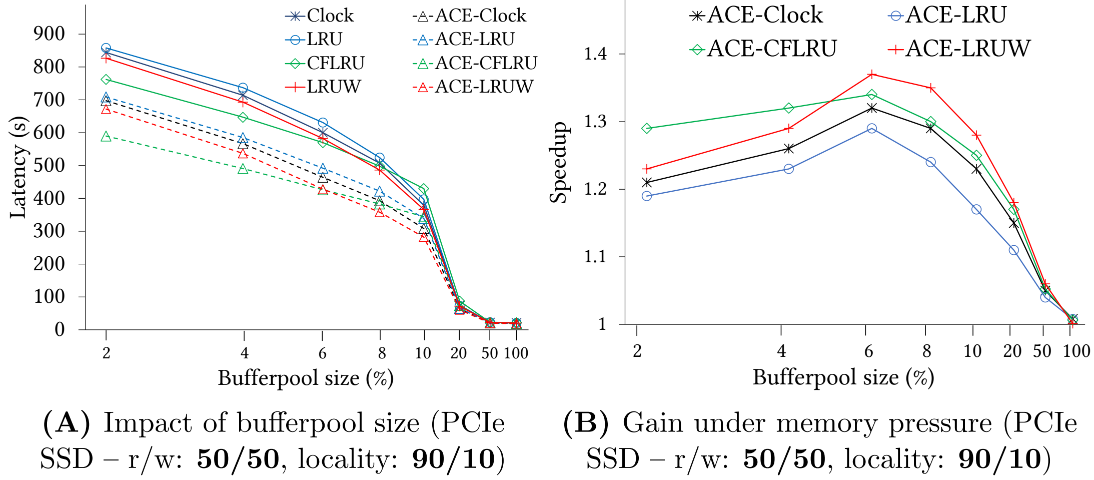
(A) Impact of r/w ratio (PCIe SSD – BP size:**6**%, locality: **90/10**)

(B) Higher gain for write-heavy workloads (PCIe SSD – BP size:**6**%, locality: **90/10**)

Figure 4·11: Higher writes lead to greater benefits.

SATA SSD ($\alpha = 1.5$) and the Virtual SSD ($\alpha = 2.0$). Figures 4·10A and 4·10B show the speedup of ACE (with prefetching) when integrated with the four page replacement algorithms, on the SATA SSD and Virtual SSD, respectively. Both devices are significantly slower than the PCIe SSD, however, the trend of overall performance gain remains. For the regular SSD, the speedup of ACE is $1.12 - 1.28\times$ and for the virtual SSD, it is $1.14 - 1.34\times$. In contrast, the speedup for the PCIe SSD is $1.19 - 1.46\times$ (Figure 4·8). We observe that for a device with higher asymmetry, the benefit of amortizing the *high* write cost is more than that of a device with lower asymmetry. This supports our thesis that the benefits are larger for higher asymmetry, however, even when there is low/no asymmetry, better utilization of the device's internal parallelism still leads to significant gains.

Write-Intensive Workloads Have Higher Gains. A common observation from the above experiments is that a write-intensive workload benefits more from the ACE bufferpool (Figure 4·8B). To verify this, we run an experiment varying the read/write ratio from 0/100 (write-intensive) to 100/0 (read-intensive) where the locality is 90/10. Figures 4·11A and 4·11B illustrate that the performance of ACE improves drastically as we shift to more write-intensive workloads. For example, as we move



(A) Impact of bufferpool size (PCIe SSD – r/w: **50/50**, locality: **90/10**)

(B) Gain under memory pressure (PCIe SSD – r/w: **50/50**, locality: **90/10**)

Figure 4·12: ACE is beneficial across a wide range of bufferpool size.

away from the most write-intensive workload (0/100) to a balanced workload (50/50), the speedup for ACE (with Clock Sweep) drops from 1.57× to 1.34×. As we further move to more read-intensive workloads (towards 100/0), the speedup eventually diminishes. This is because the benefit of ACE stems from efficient concurrent writing, and in a workload with no (fewer) writes, there will be no (less) gain. However, the benefit never fall behind the classical approach since for a read-only workload, ACE behaves exactly the same as a state-of-the-art bufferpool. Figures 4·8B, 4·11A and 4·11B also show that LRU-WSR performs much better than both LRU and CFLRU for a write-intensive workload with high locality. This is because LRU-WSR gives the dirty pages a second chance with the *cold flag*, thus saving a lot of unnecessary disk writes. Figures 4·11A and 4·11B also show that CFLRU performs well in a read-intensive setting because it evicts clean pages first.

Higher Benefits Under Memory Pressure. ACE achieves higher speedup for smaller bufferpool size because a smaller bufferpool causes more evictions, hence more writes. Figures 4·12A and 4·12B show the impact of ACE under memory pressure for a mixed skewed workload like MS in the PCIe SSD. Figure 4·12A shows the actual runtime while Figure 4·12B shows the speedup of the four ACE policies over

the baseline counterparts, as we vary the bufferpool size with respect to the data size. We observe that as the bufferpool size grows beyond 6%, the speedup decreases because larger bufferpool causes fewer evictions (and fewer writes). For a bufferpool size of more than 10%, the latency (and speedup) of all approaches drop drastically because the bufferpool is large enough to hold the working set, resulting in very few disk accesses. On the other side of the spectrum, the speedup for smaller bufferpool sizes (2%, 4%) is slightly less (than that of 6%) because if the bufferpool size is too small, there are too many read I/Os to the disk. Nonetheless, even for a smaller bufferpool, the gain is substantial. For example, for 2% bufferpool size, the speedup for ACE-CFLRU is 1.29×, while for 10%, the speedup is 1.25×. Figure 4·12A also shows that CFLRU performs better than both LRU and LRU-WSR for smaller buffer size. As the buffer size increases, the benefit of CFLRU drops because of its lower hit ratio. On the other hand, LRU-WSR always performs better than LRU and Clock Sweep because of its write-optimization policy. *In all cases, the ACE policies outperform their baseline counterparts.*

Device Concurrency Plays A Crucial Role. We run the mixed-skewed workload MS in the PCIe SSD while varying n_w to capture the impact of the write-back concurrency vs. the device concurrency as shown in Figure 4·13. Each line shows the speedup of ACE over the corresponding baseline algorithm. As concurrency increases, the *speedup* increases in all cases, which is expected. However, after a certain point ($n_w = 8$), the speedup starts decreasing.

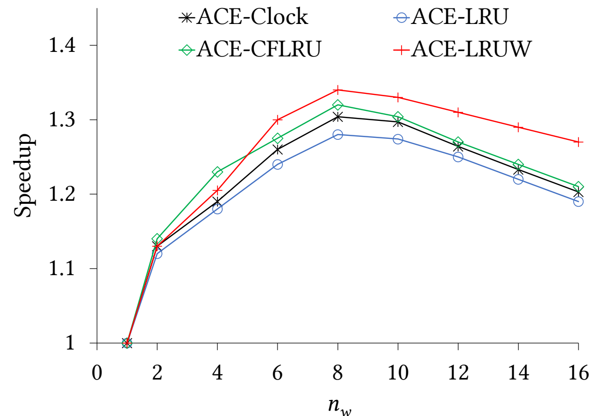
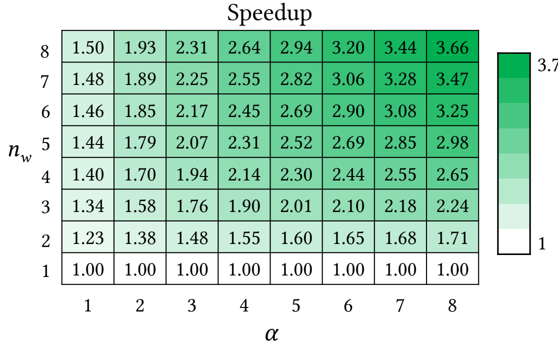
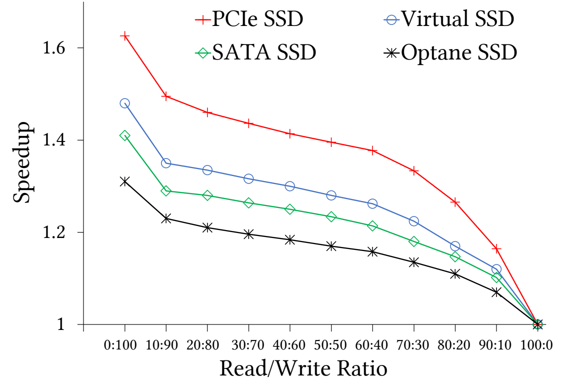


Figure 4·13: Speedup increases as #I/Os are increased until device gets saturated.



(A) Ideal speedup of ACE in the $(\boldsymbol{\alpha}, \boldsymbol{n})$ continuum



(B) Benefit is higher for higher asymmetry (LRU-WSR, BP size: **6**%)

Figure 4·14: (A) Spectrum of $(\boldsymbol{\alpha}, \boldsymbol{n})$ – as we move towards higher asymmetry, ACE has higher gain. (B) Empirical evidence showing that devices with higher asymmetry has higher gain for ACE.

There are several factors contributing to this: (i) as the number of concurrent I/O increases, the overhead of thread management increases, (ii) the ideal device write concurrency is $k_w = 8$ in this experiment, and (iii) going over the ideal concurrency does not yield any benefit since the bandwidth gets saturated and attempting to submit more concurrent I/Os does not further increase write throughput. As a result, the overall speedup starts reducing after reaching this threshold. We highlight that even with a small degree of concurrency ($n_w = 4$ or $n_w = 6$), the speed is substantial ($1.2\times - 1.3\times$).

Performance Gain Increases with Asymmetry. Our last experiment highlights that asymmetry impacts the attained gains of the ACE policies. We implement a bufferpool manager with LRU and ACE-LRU without prefetching, and we test them on an emulated device that has ideal asymmetry varied between $\alpha = 1$ (no asymmetry) and $\alpha = 8$, while the device concurrency is $k_w = 8$. We run the mixed-skewed workload MS and normalize the emulated latency of ACE-LRU with respect to that of the baseline LRU for the respective emulated device. Figure 4·14A shows the ideal speedup of ACE for different values of α and n_w. The continuum of (α, n_w)

shows that as we move towards higher asymmetry, the gain increases and it is highest when both asymmetry and concurrency value are maximized. We run an experiment to show this empirically where we vary the read/write ratio with ACE on top of LRU-WSR in all four of our devices (PCIe SSD, SATA SSD, Virtual SSD, Optane SSD). The n_w values were set according to the device k_w. Figure 4·14B shows that when everything else is the same, the performance gain is higher for devices with higher asymmetry. This is because the benefit of amortizing the asymmetric write cost is higher for a device with a higher write cost. The speedup for the PCIe SSD ($\alpha = 2.8$) is up to 1.63× (write-only workload) while the speedup is limited to 1.48×, 1.41× and 1.33× for the Virtual SSD ($\alpha = 2.0$), SATA SSD ($\alpha = 1.5$), Optane SSD ($\alpha = 1.1$), respectively.

4.5.2 TPC-C Benchmark

We further use the TPC-C benchmark [186] to demonstrate ACE's efficacy. A TPC-C database consists of nine tables (Warehouse, Stock, Item, District, Customer, History, Order, New-Order, and Order-Line), and the data size depends on the number of *Warehouses*. The benchmark consists of five transactions at different frequencies:

- **NewOrder (45%):** This transaction involves both reads and writes (RW) with 1% failure rate due to invalid inputs.
- **Payment (43%):** This transaction has both reads and writes.
- **OrderStatus (4%):** This is a read-only transaction.
- **StockLevel (4%):** This is a read-only transaction.
- **Delivery (4%):** This is a write-heavy transaction

ACE Accelerates TPC-C by 24%. For the first experiment, we run the standard TPC-C benchmark in PostgreSQL for the four page replacement policies and their

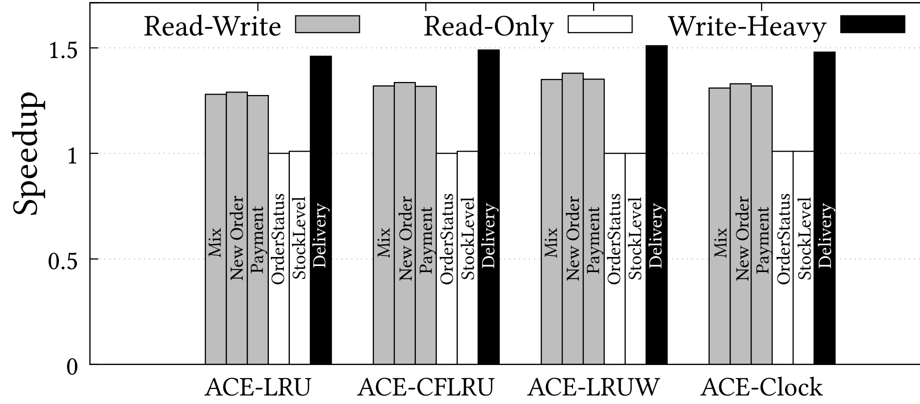


Figure 4·15: ACE achieves high speedup for TPC-C mixed transaction, while benefiting the write-intensive transaction the most.

ACE counterparts. Each baseline run is 10 minutes long and is configured with 500 warehouses, and 20 users and the resulting database size is approximately 50GB. PostgreSQL *shared_buffers* parameter is configured to be 3GB (6%). Figure 4·15 shows the performance gain of ACE for the TPC-C mix, and for five workloads each consisting of one of the TPC-C transactions. ACE achieves significant performance gain when integrated with any page replacement policy. For example, the speedup of ACE for the mixed transaction (combination of all five) is 1.29×, 1.27×, 1.30× and 1.32× when implemented on top of Clock Sweep, LRU, CFLRU, LRU-WSR, respectively. The highest speedup, 1.51×, is obtained when running the *Delivery* transaction which is an update-heavy transaction. As expected, there is no performance gain for the two read-only transactions: *OrderStatus* and *StockLevel*. We observe that the performance results for the TPC-C benchmark corroborate our findings for the synthetic benchmark: (i) ACE offers significant performance benefits when the workload contains even a small fraction of writes, (ii) write-heavy workloads have higher gain, (iii) flash-friendly policies like CFLRU and LRU-WSR outperform other policies. *Overall, ACE reduces the TPC-C mix transaction runtime on modern storage devices by 24% without any significant penalty or other tradeoff.*

ACE Scales with Data Size. Our last experiment shows that the benefits of ACE scale with data size. We again run the TPC-C mix and we increase the number of warehouses, varying the database size from 15GB to 84GB. The bufferpool size is always configured to be 6% of the database size. Figure 4·16 presents the transactions per minute count (tpmC) of this experiment when running with LRU and ACE-LRU (similar trends were observed for the other policies). The figure shows that ACE's benefits remain as we increase the database size. Specifically, the performance gain of ACE for 125 warehouses is 1.33× while for 1000 warehouses it is 1.24× compared to LRU's tpmC. This small decrease is attributed to the overhead of managing a high volume of data. Overall, this experiment shows that ACE policies scale which makes them ideal for large-scale deployments.

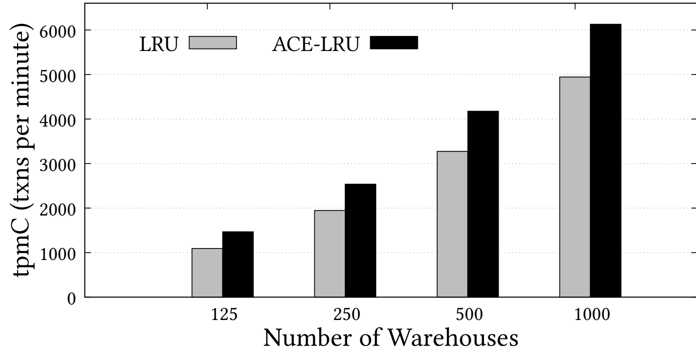


Figure 4·16: Gain of ACE scales with data size.

4.6 Related Work

Bufferpool Management. There have been several efforts that focus on developing efficient page replacement policies [47, 77, 86, 87, 116, 126, 182]. However, they are primarily designed for traditional HDDs, hence, they do not address asymmetry or concurrency. Recent work on bufferpool on top of flash devices prioritizes the eviction of clean pages and reduces page writes to minimize device wear-off [80, 100, 139, 207]. Other flash-friendly policies, like FOR and FOR+ [108] use an operation-aware page weight determination for buffer replacement. All these techniques indirectly address asymmetry, however, they do not exploit the device concurrency. Recent works also

exploit the device parallelism by redesigning the SSD controller [88, 162, 163]. In contrast to these approaches, our goal is to develop a bufferpool that expressly utilizes the device concurrency and consequently addresses asymmetry via write-amortization.

Prefetching. The most popular prefetching technique is sequential prefetching [39, 98, 183] which is adopted by many commercial systems. Stride-based prefetching is also widely studied primarily for processor caches [51, 95]. History-based prefetching techniques attempt to predict future access patterns based on past access patterns by using history-based table [59], Markov predictor [79], and data compression techniques [37, 193]. Any of these prefetching technique(s) can be integrated with ACE.

4.7 Conclusions

Modern solid-state drives are characterized by a *read-write asymmetry* and an access *concurrency*, both of which are essential to fully utilize the device. However, buffer management for DBMS does not explicitly focus on these two properties. In this chapter, we first refactor the bufferpool design space by separating the eviction policy from the write-back policy. We propose ACE, a novel asymmetry/concurrency-aware bufferpool manager paradigm that batches writes based on device concurrency to amortize the asymmetric write cost. Incorporating concurrency into the write-back policy allows us to custom-tailor any bufferpool manager to the device-at-hand, thus utilizing the device's full potential. ACE can be integrated with *any* existing page replacement and prefetching policy with low engineering effort. We implement ACE in PostgreSQL and measure its benefit when integrated with four state-of-the-art page replacement algorithms. ACE improvements performance by up to 32.1% for synthetic workloads and up to 24.2% for the standard TPC-C mixed transaction (33.8% for write-heavy transactions) without any penalty.

Chapter 5

CAVE: concurrency-aware graph processing on SSDs

Large-scale graph analytics has become increasingly common in areas like social networks, physical sciences, transportation networks, and recommendation systems. Since many such practical graphs *do not fit in main memory*, graph analytics performance depends on efficiently utilizing underlying storage devices. These *out-of-core* graph processing systems employ sharding and sub-graph partitioning to optimize for storage while relying on efficient sequential access of traditional hard disks. However, today's storage is increasingly based on SSDs that exhibit *high internal parallelism* and *efficient random accesses*. Yet, state-of-the-art graph processing systems do not *explicitly exploit* those properties, resulting in subpar performance.

In this chapter, we present CAVE[1] [134], the first graph processing engine that optimally exploits underlying SSD-based storage by harnessing the available storage device parallelism via carefully selecting graph I/Os that can be issued concurrently. Thus, CAVE traverses multiple paths and processes multiple nodes and edges concurrently, achieving parallelization at a granular level. We identify two key ways to parallelize graph traversal algorithms based on the graph structure and algorithm: intra and inter-subgraph parallelization. The first identifies subgraphs that contain vertices that can be accessed in parallel, while the latter identifies subgraphs that

[1] The material of this chapter has been the basis for the SIGMOD 2024 paper "CAVE: concurrency-aware graph processing on SSDs" [134] and the ICDE 2024 PhD Symposium "Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry" [129].

can be processed in their entirety in parallel. To showcase the benefit of our approach, we build within CAVE parallelized versions of five popular graph algorithms (Breadth-First Search, Depth-First Search, Weakly Connected Components, PageRank, Random Walk) that exploit the full bandwidth of the underlying device. CAVE uses a blocked file format based on adjacency lists and employs a concurrent cache pool that is essential to the parallelization of graph algorithms. By experimenting with different types of graphs on three SSD devices, we demonstrate that CAVE utilizes the available parallelism, and scales to diverse real-world graph datasets. CAVE achieves up to one order of magnitude speedup compared to the popular out-of-core systems Mosaic and GridGraph, and up to three orders of magnitude speedup in runtime compared to GraphChi.

5.1 Introduction

The Rise of Large Graphs. Graphs are *natural encoders* of interconnected relations that can be leveraged to analyze many real-world applications. With the unprecedented growth of such interconnected data stemming from various applications like machine learning [105], recommendation systems [196], physical sciences [206], and social networks [217], analytics over large graphs is becoming increasingly popular in both academia and industry [4, 76, 103, 112, 142]. Real-world graphs often exhibit a vast scale, frequently encompassing millions, or even billions, of nodes interconnected by several billion edges. The sheer size of these graphs often exceeds the capacity of main memory, posing a significant challenge for efficient processing. Consequently, specialized techniques have emerged to address the need for scalable solutions to handle these massive graphs.

State-of-the-art Graph Management Systems. Many scalable systems have been recently proposed that handle large graphs by *distributed processing* [54, 84,

105, 112, 155, 220], which come with unique challenges such as partitioning, load balancing, cluster management, network overhead, and fault tolerance. On the other hand, *single-node* systems process large graphs in-memory [169, 177, 202, 216] and achieve scalability through increasing memory size and adding more CPUs. This work is orthogonal to the aforementioned approaches, however, it can benefit any system that spills data into storage. For example, our techniques can be applied at the local shard level in distributed graph management systems to enhance performance. *Single-node out-of-core* systems (which we focus on) primarily rely on (i) optimizing data partitioning techniques, (ii) improving memory and disk locality, and (iii) reducing random I/O to utilize fast sequential I/Os [64, 92, 109, 156, 221]. These techniques mainly address *slow random disk access*, which is particularly relevant for traditional hard disk drives (HDDs). However, the storage layer of data-intensive systems today employs solid-state disks (SSDs) and non-volatile memory (NVM) devices that have quite different characteristics than HDDs, which require a careful system redesign to be effectively exploited [129, 130, 131]. As discussed in previous chapters, SSD internals follows a hierarchical structure that creates high *internal parallelism*, which can be leveraged to enhance performance [26, 27, 113, 130, 132, 165].

SSD Parallelism for Graph Processing. Graph traversal operations can utilize SSD concurrency by parallelizing node and edge accesses, effectively distributing the workload across SSD's parallel architecture [16]. This idea takes advantage of the availability of multiple paths that can be explored during graph traversal. However, most out-of-core graph processing systems simply attempt to better utilize underlying storage devices by reducing random (in favor of sequential) I/O. They do not aim to aggressively exploit opportunities for concurrent accesses, thus failing to use the full potential of SSDs. Our goal is to parallelize graph traversal algorithms without changing their core properties in order to fully utilize the underlying SSD concurrency.

We identify two approaches to achieve this goal, each tailored to specific scenarios.

- **Intra-Subgraph Parallelization:** This approach focuses on parallelizing operations within a single subgraph. This approach is effective when the nodes of a subgraph can be processed independently. For example, a parallel version of Breadth-First Search (BFS) can follow this approach since multiple nodes of the same level can be processed independently. The core integrity of the algorithm can be maintained via communication among the processing units, result aggregation and synchronization. This approach harnesses the inherent parallelism present in subgraphs and utilizes modern storage concurrency for faster and more efficient graph traversal.

- **Inter-Subgraph Parallelization:** In contrast to the previous approach, inter-subgraph parallelization involves processing multiple subgraphs concurrently. This method is particularly useful when we can identify that multiple subgraphs can be processed independently. For example, in the pseudo Depth-First Search algorithm [3], the stack used for traversal can be split into smaller stacks and processed in parallel by different threads. Multiple threads can then work on different parts of the graph concurrently, thus traversing multiple branches simultaneously.

In both approaches, the key objective is to maximize the utilization of SSD concurrency, ensuring that multiple operations can be performed in parallel. We integrate both approaches into a prototype graph processing system as discussed next.

Our Approach. We build an SSD-aware graph processing system, named CAVE[2] [134] that is able to harness the *concurrency* of the underlying storage devices via *intra/inter-subgraph parallelization*. Specifically, CAVE provides the necessary infrastructure to parallelize graph traversal algorithms when several independent vertex accesses can be performed in parallel. A prime example is our Parallel Breadth-First Search

[2]**CAVE**: **C**oncurrency-**A**ware Graph (**V**, **E**) system

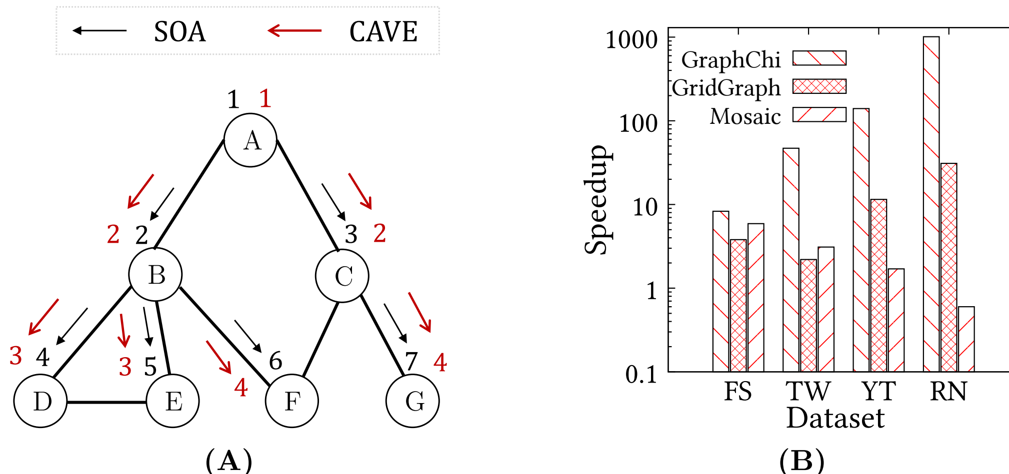


Figure 5·1: (A) Parallellized version of BFS in CAVE takes fewer iterations to converge. (B) CAVE is upto three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph and Mosaic.

(PBFS) implementation that uses *intra-subgraph* parallelization, which is outlined in Figure 5·1A. The algorithm accesses the next *wave* of nodes (as we move on a level-by-level fashion) in parallel since we have already identified the nodes of the next wave while processing the current one. Figure 5·1A is a high-level overview where we consider a device with read concurrency 2. Hence, while vertex D, E, F and G are at the same level, only two of them can be processed in parallel. This leads to a faster response time of the BFS search simply by carefully exploiting the underlying storage concurrency, resulting in faster convergence within fewer iterations. CAVE uses a block-based file format based on adjacency lists, ensuring that graph metadata, vertex information, and edge information are stored in aligned blocks while enabling efficient support for graph traversal and analytical operations by ensuring optimized data retrieval. Furthermore, CAVE employs a concurrent cache pool mechanism that enhances locality and ensures thread safety. Overall, CAVE identifies storage accesses that are independent (thus can be parallelized) based on the task at hand and per-

forms them concurrently based on the device's *optimal concurrency* [130], i.e., the number of I/O requests the device can handle without compromising latency.

To our best knowledge, CAVE is the first graph processing system that is capable of fully exploiting the available parallelism of the underlying flash-based storage leading to significant performance improvements. State-of-the-art graph processing systems focus on the design of graph processing/traversal algorithms and the distribution of the work (e.g., partitioning), but not on the specific characteristics of the underlying hardware and especially storage devices. By building a better understanding of how to efficiently use SSDs, we build a faster graph processing system. Further, one of the key benefits of this approach is that it is applicable in any graph system that spills data on disk, so it can benefit a wide variety of systems. CAVE's architecture is designed to pave the way for developing new parallel graph algorithms that leverage the inherent concurrency of SSDs for both intra/inter-subgraph parallelization. As our first step, we develop in CAVE the parallelized versions of five popular graph algorithms. In addition to Breadth-First Search (BFS), CAVE offers parallelized, SSD-aware versions of Depth-First Search (DFS), Weakly Connected Components (WCC), PageRank (PR), and Random Walk (RW). We compare the performance of CAVE with three popular out-of-core processing systems, GraphChi [92], GridGraph [221] and Mosaic [109], as they are widely recognized for their efficiency in handling large-scale graphs in a single machine. Figure 5·1B shows the speedup of CAVE's PBFS compared to these systems for four datasets (Friendster, Twitter, YouTube, and RoadNet) running on top of our PCIe SSD (details in Section 5.6). We observe that CAVE can be up to three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph and Mosaic.

Contributions. This chapter makes the following contributions.

- We identify the importance of *SSD concurrency* with respect to graph processing.

- We identify two fundamental ways to parallelize graph traversal operations: *intra-subgraph* and *inter-subgraph parallelization*.

- We propose CAVE, the first SSD-aware graph engine that *fully exploits the parallelism of the underlying SSD storage* via concurrent I/O, its novel file structure, and a concurrent cache pool.

- We develop on CAVE the parallelized version of five popular graph algorithms (BFS, DFS, WCC, PageRank, Random Walk) to showcase that CAVE is flexible enough to implement diverse graph traversal algorithms.

- We evaluate CAVE against GraphChi, GridGraph and Mosaic where CAVE achieves up to 984× speedup vs. GraphChi, up to 22× speedup vs. GridGraph and up to 15× speedup vs. Mosaic.

5.2 Background

We now introduce the necessary background for the graph traversal algorithms we use and discuss the opportunities for parallelization.

Breadth-First Search (BFS). BFS is a graph traversal algorithm that starts from a designated starting vertex and then explores all neighboring vertices in a level-by-level manner [35]. It begins by visiting all the immediate neighbors of the starting vertex and then moves on to their neighbors in subsequent levels. By traversing the graph in a level-wise manner, BFS uncovers the shortest paths and analyzes the structural properties of the graph.

Since BFS processes nodes in a level-by-level manner, nodes of the same level can be processed independently (hence concurrently), thus providing an opportunity for parallelizing and, in turn, harnessing the SSD's concurrency.

Depth-First Search (DFS). DFS is a widely-used graph traversal algorithm that

starts from a specified vertex and systematically explores as deep as possible along each branch before backtracking [46]. This approach involves visiting a vertex and then recursively visiting its unvisited neighbors until there are no more unvisited vertices. DFS is particularly useful for identifying cycles, determining connected components, and finding paths between vertices.

While the classical DFS is tricky to parallelize, the pseudo-DFS [3] algorithm offers the opportunity to parallelize by running multiple parallel mini-DFSs. A parallel version of pseudo-DFS can dynamically split and distribute the vertex stack among multiple threads, allowing concurrent exploration of different branches of the graph.

Weakly Connected Components (WCC). In an undirected graph, a connected component refers to a subgraph where every vertex is connected to every other vertex through pathways within the graph. WCC aims to identify and group nodes that are weakly connected [90], meaning they can be reached from each other by traversing the edges regardless of their direction. This algorithm typically involves traversing the graph using techniques like BFS or DFS to identify the connected components.

The previous approaches used to exploit SSD concurrency can be used to parallelize WCC. For example, while using BFS to discover WCCs, each subgraph's connected components can be computed concurrently, and the results from different subgraphs can be merged to determine the weakly connected components.

PageRank (PR). PR is a well-known algorithm to estimate the importance of vertices in graphs, used by Google to rank webpages on the Internet [23]. It works by evaluating the importance of a web page based on the number and quality of links pointing to it. The algorithm assigns a numerical value, known as PR score, to each web page on the Internet and measures the importance of a web page based on its backlinks and the quality of those links. PR employs an iterative process. Initially, all pages are assigned an equal PR score. In each iteration, the scores are updated

based on the scores of linking pages. This process continues until PR scores converge or after a certain number of iterations.

Due to this iterative traversal nature, this algorithm can be parallelized, similar to BFS. Within each subgraph, PR calculations can be performed concurrently by assigning individual nodes to threads. They can independently compute PR values for nodes within their respective subgraphs, leading to efficient parallel execution while preserving the algorithm's core structure. Finally, each subgraph's results should be combined to obtain the overall PR scores.

Random Walk (RW). RW is a probabilistic algorithm in which a walker moves through a network (graph), taking steps based on random choices [104]. It is used to analyze the network structure and understand properties such as connectivity and reachability. RW can be viewed as a Markov Chain, where the probability of transitioning to the next state depends only on the current state.

To accelerate RW, we can divide the graph into manageable subgraphs and simultaneously explore multiple nodes within these subgraphs. This approach accelerates the exploration and allows for parallelization of transition probability calculations, making it suitable for estimating node importance through RWs on vast networks. Further, different subgraphs can be processed in parallel while accounting for crossing into a different subgraph.

5.3 Parallelizing Graph Traversal

Our main objective is to efficiently parallelize graph traversal operations with out-of-core systems while maintaining the core properties of the graph algorithms. In this section, we discuss how to achieve this with **intra-subgraph** and **inter-subgraph** parallelization. We present these two techniques with examples and discuss how they can be seamlessly integrated and leveraged alongside SSD parallelization.

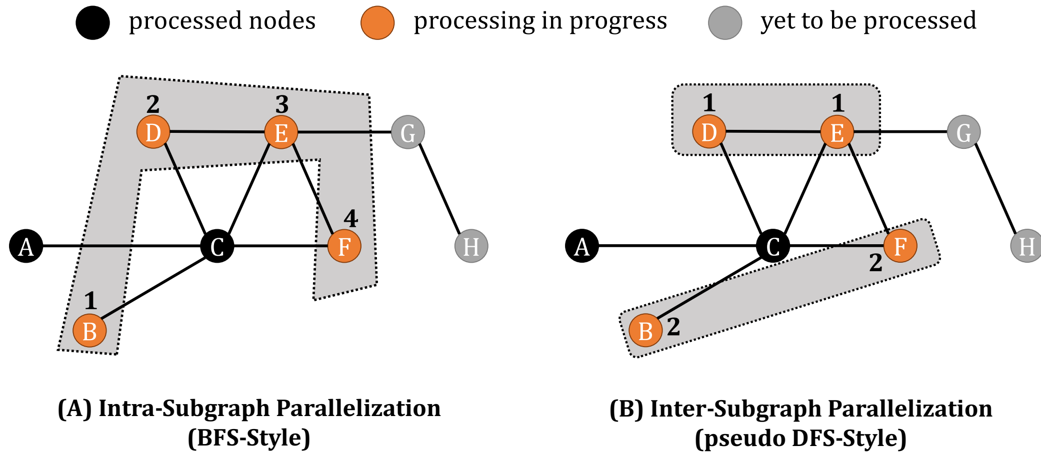


Figure 5·2: Example of Intra/Inter-Subgraph Parallelization. (A) { B, D, E, F } are at the same level of BFS and are processed concurrently by 4 threads. (B) As pseudo-DFS progresses, the stack is split into two subgraphs ({ D, E } and { B, F }), which are processed in parallel by 2 threads.

5.3.1 Intra-Subgraph Parallelization

For this approach, we identify subgraphs, the nodes of which can be processed independently so that we can access them in parallel. This means that the processing of one node does not depend on the result or state of other nodes outside the subgraph. Thus, multiple nodes within the subgraph can be processed concurrently by different computing units (threads), allowing for concurrent I/Os, leading to better device utilization. After processing their respective nodes, the results obtained by each thread are aggregated to produce the final result of the algorithm. This ensures efficient exploitation of the underlying device which can speed up the execution of graph traversal operations by processing multiple graph blocks (vertex and edge) in parallel, resulting in faster convergence.

Example. A prime example of this type of parallelization is a *parallel BFS*. BFS explores the graph level by level, where each level represents a set of equidistant vertices from the source vertex. Since vertices of the same level can be accessed independently

of each other, all vertices within the same level can be processed concurrently, and thus accessed in parallel using multiple threads. A queue maintains the nodes to be visited next, which are ordered on a per-level basis. Each thread dequeues nodes from the shared queue and processes them independently. The edges of each node are accessed from the underlying SSD concurrently. Figure 5·2A illustrates the application of this technique for parallelizing the BFS algorithm. Once nodes A and C have been traversed, nodes B, D, E, and F are all at the same level (a subgraph where nodes are independent), enabling them to be processed concurrently. Other BFS-based algorithms (e.g., PageRank, WCC) can also be parallelized with this approach as a building block.

5.3.2 Inter-Subgraph Parallelization

The subtle difference between Inter-Subgraph and Intra-Subgraph Parallelization is that it identifies subgraphs that can be independently accessed (like two different branches of DFS) and processes them in parallel. That way, multiple subgraphs (or paths) can be traversed concurrently, thus covering the entire graph faster and allowing for faster convergence. The algorithmic correctness and other properties (like the order of accessing nodes) can be ensured by communication and synchronization between the threads processing independent subgraphs. This approach is particularly useful for large-scale graphs that cannot fit entirely in memory or when distributing the computation across multiple threads.

Example. We now use the *pseudo-DFS* [3] as an example. In the classical DFS algorithm, a stack keeps track of the nodes to be explored and maintains the visiting order. In the pseudo-DFS algorithm, a stack can be split into smaller stacks when its size exceeds a predefined threshold, and the smaller stacks are processed in parallel. This allows for multiple threads to work on different subgraphs (paths) concurrently. Figure 5·2B shows an example of this approach. In this example, after traversing

nodes A and C, the stack size grows to four and (assuming this is the threshold) is split in two. The first stack contains nodes D and E, while the second contains B and F. These smaller stacks are processed in parallel, leading to two independent graph traversals with the additional need for communication to avoid crossing from one subgraph (path) to another. Inter-subgraph parallelization also benefits finding Strongly Connected Components (SCCs) or groups of nodes within a graph where each node is accessible from every other node in the same group.

5.3.3 Discussion

Which approach, which data structure? The selection between intra-subgraph and inter-subgraph parallelization, as well as the choice of data structure depends on the algorithm being parallelized. For example, in cases where the algorithm involves BFS-like exploration, intra-subgraph parallelization is the best fit. On the other hand, for algorithms resembling pseudo-DFS or those focused on connectivity exploration, inter-subgraph parallelization can be more effective since it allows different subgraphs to be processed concurrently, facilitating quicker convergence. In both cases, graph traversal is accelerated by overlapping the standard accesses of the original algorithm with several other accesses that would normally be scheduled for later. Thus, a larger subgraph is traversed than the original algorithm without altering its key properties.

Parallelizing Essentials. When parallelizing graph traversal algorithms, we need to guarantee the correctness and the efficiency of the parallel execution. To achieve this, we use *result aggregation*, *synchronization*, and *communication* mechanisms. In algorithms like PageRank, where the goal is to calculate rankings, the individual results obtained from different subgraphs or processing units must be aggregated to calculate the final rankings. Algorithms like DFS require synchronization to prevent race conditions and maintain the same vertex visiting order and, thus, the core guarantees of the algorithm. Further, many algorithms need some form of communication

between the threads working on subgraphs (signaling or message passing) to indicate convergence. Minimizing such communication and synchronization overhead is a key challenge to avoid bottlenecks.

5.4 Concurrent Graph Algorithms

The core idea of our approach is to implement parallel graph algorithms that take advantage of concurrency at the storage level. Our system, CAVE, identifies and parallelizes independent I/Os, similar to how out-of-order processors parallelize load and store commands that are not dependent on each other. This enables parallel graph data processing, allowing multiple nodes to be accessed simultaneously, thus significantly reducing the number of iterations required. We carefully tune CAVE to employ the *optimal concurrency* [130] for the underlying storage devices to guarantee maximum benefit. To do this, we issue in parallel as many *independent I/O operations* as the storage device supports without hurting latency. As a result, graph algorithms in CAVE have faster convergence and more efficient data accesses. Overall, the work presented in this chapter contributes to the system-level understanding of how to build efficient graph processing systems that maximize the utilization of the underlying SSD. To demonstrate the benefits of our approach, we parallelize five of the most common graph traversal algorithms: BFS, WCC, PageRank, Random Walk, and DFS. In this section, we first provide a quick overview of the physical data layout CAVE and then discuss the details of the parallel versions of these algorithms.

5.4.1 CAVE Physical Data Layout

CAVE uses a memory-mapped binary file format, with three main parts: the metadata block, the vertex block, and the edge block – right part of Figure 5·3. They are stored using 4KB aligned blocks to support direct reading and writing from/to the SSDs. All blocks are cached in memory by a cache pool described in detail in Section 5.5.

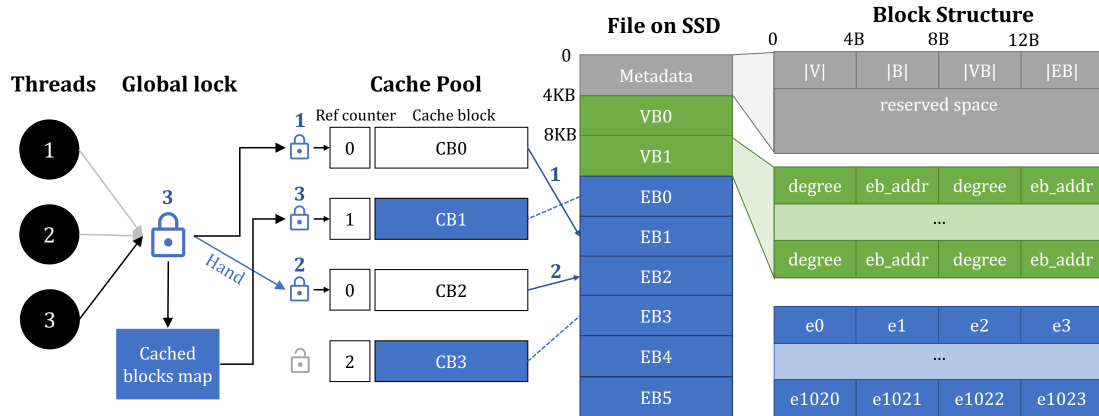


Figure 5·3: Architecture of CAVE comprises block-based file structure (right) and a concurrent cache pool (left)

Metadata Block. The metadata block serves as a repository for essential graph information such as the number of vertices, the total number of blocks, edge blocks, and vertex blocks, each of which is stored as a 32-bit integer. The remaining space is reserved for future utilization, allowing for additional usage-specific information to be incorporated when necessary.

Vertex Block. Each vertex block, sized at 4KB, stores information about up to 512 vertices. Within each vertex, 8 bytes are allocated, encompassing two 32-bit unsigned integers: $degree$, eb_addr (edge block index and offset). The low 10-bit of eb_addr represents the offset eb_offset inside of an edge block, which just fits its capacity of 1024. The high 22-bit states the index of the edge block eb_idx. The reading process can start by calculating the appropriate address $((eb_idx \cdot 4\text{KB}) + eb_offset)$.

Edge Block. To optimize storage and retrieval, we utilize a compact representation of edges. Each edge is represented by a 4-byte integer denoting the index of the ending vertex. Hence, each edge block can store up to 1024 edges (adding up to 4KB). The edges of vertices with a degree less than 1024 are contained within a single edge block (note that in many datasets, most nodes have indeed a degree of less than 1024). This ensures efficient single read I/O access, while the starting index inside the block

(eb_offset) can vary. However, vertices with a degree over 1024 will occupy multiple edge blocks. In this case, the first block always has an $eb_offset = 0$ to simplify the packing and subsequent reading process. The number of edge blocks per vertex is given by its degree divided by 1024.

Bin-Packing Edges. Previous practices often stored all edges in a single extensive list, resulting in inefficient I/O operations when accessing edges of small vertices spanning multiple blocks. To mitigate this issue and optimize I/O and cache utilization, we approached it as a *bin-packing problem*. Edges of small vertices are stored within a single container (block), while larger vertices span multiple consecutive blocks. We employ an offline first-fit strategy, determining the appropriate block to insert new vertex neighbors and ensuring efficient packing and retrieval of edge data.

Handling Updates. While CAVE currently does not support graph updates, we consider this as part of our future work. Currently, we have a very compact representation of vertex and edges. To handle updates on vertex/edge values (e.g., edge weights, vertex payloads), we need to modify our file architecture to accommodate these values. For instance, for each edge, we store a vertex ID using 4 bytes, which would need to be increased to account for additional edge information like weights. To exploit the *write concurrency* of the underlying device, updates can be batched in a memory buffer and applied to the corresponding blocks concurrently (using the appropriate degree of concurrency when writing). Further, this style of updating (and deleting) can use storage-resident update components similar to the log-structured merge (LSM) design [127, 159]. In this case, updates (and deletes) will be buffered in memory and later organized on disk before being eventually merged with the base data [17]. That way, the update mechanism can exploit both the good sequential performance and, when merging with the based data, the device concurrency.

5.4.2 Building Blocks for Parallelizing

***ProcessQueue* function.** In the context of BFS, WCC, PR, and RW algorithms, the parallelization process is structured as an iterative procedure. Each iteration involves processing a list of vertices (known as the $frontier$), accessing the neighbors of each vertex, updating vertex values, and determining which vertices should be visited in the next iteration, which are stored in the *next* queue. This iterative process can be naturally parallelized by having multiple threads working on individual vertices of the $frontier$ (intra-subgraph parallelization). We achieve this using a *ProcessQueue* function, which takes the $frontier$, a user-defined *process* function and the device's read concurrency k_r as parameters. The *process* function specifies the actions the algorithm should perform for each vertex and its neighbors. The *ProcessQueue* function parallelizes at the vertex level based on the k_r value where each thread is responsible for processing a vertex and executes a *getEdge* operation to retrieve the edge block from the cache pool. Since each edge block stores neighbors of multiple vertices, it is possible that an edge block swapped out from the cache will need to be read again from the disk, especially when the cache size is limited.

***ProcessQueueBlock* function.** To avoid multiple accesses of the same edge blocks, we provide a new variation that processes data at the granularity of edge blocks to benefit from caching. Initially, all edge blocks associated with vertices in the $frontier$ are found. Next, each thread is assigned to work on one of the edge blocks. That block, in turn, may contain (i) the edges of a single vertex where the execution will be the same as before, or (ii) the edges of multiple vertices where the processing of those vertices will now be completed with a single I/O. By simultaneously processing all vertices connected to a specific block, the approach ensures that each edge block is only read once in each iteration. While this strategy involves some overhead in terms of preprocessing the

Algorithm 2: Parallelization Building Blocks

```
 1: function ProcessQueue(frontier, Func Process, k_r)
 2:     next ← ∅
 3:     // Process vertices in parallel with max k_r threads
 4:     for v_1 in frontier do
 5:         // Read neighbors from the cache pool
 6:         neighbors ← GetEdges(v_1)
 7:         // Process v_1 with its neighbors, get next_v queue
 8:         next_v ← Process(v_1, neighbors)
 9:         // Merge next_v to next
10:         mtx.lock()
11:         next.insert(next_v)
12:         mtx.unlock()
13:     end for
14:     return next
15: end function
16:
17: function ProcessQueueBlock(frontier, Func Process, k_r)
18:     block_set ← HashSet()
19:     for v_1 in frontier do
20:         block_idx ← GetBlockIdx(v_1)
21:         block_set.Insert(block_idx)
22:         block[block_idx].Insert(v_1)
23:     end for
24:     // next queue of whole frontier
25:     next ← ∅
26:     // Process blocks in parallel with max k_r threads
27:     for block_idx in block_set do
28:         // next queue of this block
29:         next_b ← ∅
30:         block_data ← GetBlock(block_idx)
31:         // For each v_1 associated with this edge block
32:         for v_1 in block[block_idx] do
33:             // Get v_1 neighbors from this block locally
34:             neighbors ← ReadFromBlock(block_data, v_1)
35:             next_v ← Process(v_1, neighbors)
36:             // Merge next_v in next_b
37:             next_b.insert(next_v)
38:         end for
39:         mtx.lock()
40:         next.Insert(next_b)
41:         mtx.unlock()
42:     end for
43:     return next
44: end function
```

$frontier$, it offers the advantage of being minimally impacted by the size of the cache. Further, the edge block retrieval is performed concurrently, which contributes to its superior runtime. The two building-block algorithms are outlined in Algorithm 2.

Algorithm 3: Parallel Breadth-first Search

```
1: function BFSProcess(v_1, neighbors)
2:     next_v ← ∅
3:     for v_2 in neighbors do
4:         if visited[v_2].CAS(False, True) then
5:             // Add v_2 to the next queue of v_1
6:             next_v.INSERT(v_2)
7:         end if
8:     end for
9:     return next_v
10: end function
11:
12: function PBFS(v_s, k_r)
13:     frontier ← {v_s}
14:     vertices_count ← 0
15:     while frontier.SIZE > 0 do
16:         next ← ProcessQueue(frontier, BFSprocess, k_r)
17:         // Or call ProcessQueueBlock()
18:         vertices_count ← vertices_count + frontier.SIZE
19:         frontier ← next
20:     end while
21:     return vertices_count
22: end function
```

5.4.3 Parallel Breadth-First Search

We develop a parallel BFS (PBFS for short) algorithm using two queues: the $frontier$ queue, which contains the indices of vertices in the current level, and the $next$ queue, which stores the indices of the neighbors of vertices in the $frontier$ queue, which correspond to the vertices in the next level. To leverage parallelism, each vertex in the $frontier$ queue is assigned to a separate thread so that multiple I/Os can be issued in parallel as shown in Figure 5·1A. The complete algorithm is listed in Algorithm 3. For each vertex in $frontier$, as $BFSprocess$ defines, $ProcessQueue$ will assign threads

to vertices. Each thread accesses the assigned vertex, retrieves the indices of its neighbors, checks and flags the index of every neighbor as *visited*, inserts it to $next_v$ queue of this vertex, and merges $next_v$ of all vertices to the final *next* protected by a global lock *mtx* to prevent data races and ensure thread safety. The PBFS level of concurrency is controlled by the number of threads, which we tune according to the *optimal concurrency* of the SSD. Once all the vertices in the $frontier$ queue have been processed, the contents of the *next* queue are copied back to the $frontier$ queue, and the *next* queue is cleared. This process is repeated until the $frontier$ queue becomes empty, signifying the completion of the BFS traversal. We also developed a *blocked* variant of the $frontier$ processing that uses the *ProcessQueueBlock* function. As discussed in Section 5.4.2, this approach discovers the edge blocks of the $frontier$ vertices and allocates threads to edge blocks, parallelizing at the edge block level while ensuring that each edge block is read only once during an iteration. This results in two benefits: (i) overall runtime improvement since edge blocks are not read multiple times, and (ii) performance does not depend on cache pool size.

5.4.4 Parallel Weakly Connected Components

Computing WCC entails repeatedly searching from each vertex in the graph. Since we utilize the adjacency list format, the most efficient approach to compute WCC involves repeatedly applying the search algorithm starting from each vertex. During the search process, a visited vertex is marked as *true* and subsequently avoided in subsequent iterations. We parallelize WCC by performing multiple concurrent searches using PBFS due to its low overhead and well-established efficiency. Algorithm 4 lists the algorithm for PWCC.

Algorithm 4: Parallel Weakly Connected Component

```
1:  function PWCC(k_r)
2:      wcc_count ← 0
3:      for i ← 0 to N − 1 do
4:          // If not flagged
5:          if visited[i] = False then
6:              // Call BFS to flag all vertices in this WCC
7:              PBFS(i, k_r)
8:              wcc_count ← wcc_count + 1
9:          end if
10:     end for
11:     return wcc_count
12: end function
```

5.4.5 Parallel PageRank

We consider the topology approach for PR, which involves updating the PR values (pr) of all vertices based on the values of their neighbors from the previous iteration (Algorithm 5). Since all vertices need to be processed in each iteration, the $frontier$ queue always contains the entire list of vertices, and there is no need for a $next$ queue. Initially, the $frontier$ queue includes all vertices, from vertex 0 to vertex $N-1$. In every iteration, the $ProcessQueue$ is called with the desired concurrency to parallelize each step of the algorithm. For the blocked implementation, the $ProcessQueueBlock$ function is called. The initial PageRank values, $pr[i]$ and $pr_{next}[i]$, are assigned as the inverses of the degrees of their respective vertices v_i. It is worth noting that in the original PageRank algorithm, the initial PageRank value for each vertex is set to 1, and its neighbors are assigned values of $\frac{pr[i]}{deg[i]}$. To optimize the computation, we perform this division in advance so it does not need to be repeatedly calculated by the neighbors in each iteration.

Algorithm 5: Parallel PageRank

1: **function** PRPROCESS(v_1, $neighbors$)
2: $pr_{next}[v_1] \leftarrow 0$
3: **for** v_2 in $neighbors$ **do**
4: *// Sum up last pr value of neighbors*
5: $pr_{next}[v_1] \leftarrow pr_{next}[v_1] + pr[v_2]$
6: **end for**
7: *// Add damping factor and divide by its degree*
8: $pr_{next}[v_1] \leftarrow \frac{(1-d)+d \cdot pr_{next}[v_1]}{\text{GETDEGREE}(v_1)}$
9: **return** $\emptyset$
10: **end function**
11:
12: **function** PARALLELPAGERANK($iterations$, k_r)
13: $frontier \leftarrow \{0, 1, \ldots, N-1\}$
14: **for** $i \leftarrow 0$ to $N-1$ **do**
15: $pr[i] \leftarrow \frac{1}{\text{GETDEGREE}(i)}$
16: $pr_{next}[i] \leftarrow pr[i]$
17: **end for**
18: **while** iterations ¿ 0 **do**
19: PROCESSQUEUE($frontier$, $PRprocess$, k_r)
20: $pr \leftarrow pr_{next}$
21: $iterations \leftarrow iterations - 1$
22: **end while**
23: *// Prepare return value*
24: **for** $i \leftarrow 0$ to $N-1$ **do**
25: $pr[i] \leftarrow pr[i] \cdot \text{GETDEGREE}(i)$
26: **end for**
27: **return** pr
28: **end function**

5.4.6 Parallel Random Walk

A single random walk is inherently a serial process and does not significantly benefit from data concurrency. However, an effective strategy is to run multiple random walks concurrently, which not only improves the precision of the results but also reduces the overall running time. Initially, k vertices are randomly chosen from the whole vertex set and put in the $frontier$ queue. In each iteration, the $RWprocess$ function randomly selects one of the neighbors for each vertex in $frontier$ as the successor in the next iteration. Algorithm 6 outlines the complete algorithm.

Algorithm 6: Parallel Random Walk

```
1:  function RWPROCESS(v_1, neighbors)
2:      // Randomly selects a neighbor
3:      v_2 ← RANDOMSELECT(neighbors)
4:      return {v_2}
5:  end function
6:
7:  function PARALLELRANDOMWALK(K, iterations, k_r)
8:      frontier ← ∅
9:      visited_count ← 0
10:     for i ← 0 to K − 1 do
11:         // Initialize starting vertices randomly
12:         frontier[i] ← RANDOM(0, N − 1)
13:     end for
14:     while iterations ¿ 0 do
15:         next ← PROCESSQUEUE(frontier, RWprocess, k_r)
16:         frontier ← next
17:         visited_count ← visited_count + K
18:         iterations ← iterations − 1
19:     end while
20:     return visited_count
21: end function
```

5.4.7 Parallel Pseudo Depth-First Search

While DFS is inherently a serialized algorithm, it is possible to enhance its performance by introducing parallelism through a technique known as *unordered* or

Algorithm 7: Parallel Pseudo Depth-first Search

```
1: function DFSTask(stack, k_r)
2:     max_stack_count ← k_r
3:     while stack.SIZE() > 0 do
4:         // Get and pop vertex at the stack top
5:         v_1 ← stack.TOP()
6:         stack.POP()
7:         visited_count ← visited_count + 1
8:         neighbors ← GetEdges(v_1)
9:         // Push all unvisited neighbors on stack
10:        for v_2 in neighbors do
11:            if visited[v_2].CAS(False, True) then
12:                stack.PUSH(v_2)
13:            end if
14:        end for
15:        // Check if the stack size is larger than threshold
16:        while stack.SIZE() > max_stack_size do
17:            if stack_count < max_stack_count then
18:                stack_count ← stack_count + 1
19:                // Split the stack and generate new task
20:                new_stack, stack ← stack.SPLIT()
21:                ThreadPool.PUSH(DFStask, new_stack)
22:            end if
23:        end while
24:    end while
25:    stack_count ← stack_count − 1
26: end function
27:
28: function ParallelPseudoDFS(v_s, k_r)
29:    init_stack ← {v_s}
30:    visited[v_s] ← True
31:    stack_count ← 1
32:    visited_count ← 0
33:    // Push the initial task in the thread pool
34:    ThreadPool.PUSH(DFStask(init_stack, k_r))
35:    // Wait for all tasks to be finished
36:    ThreadPool.WaitAll()
37:    return visited_count
38: end function
```

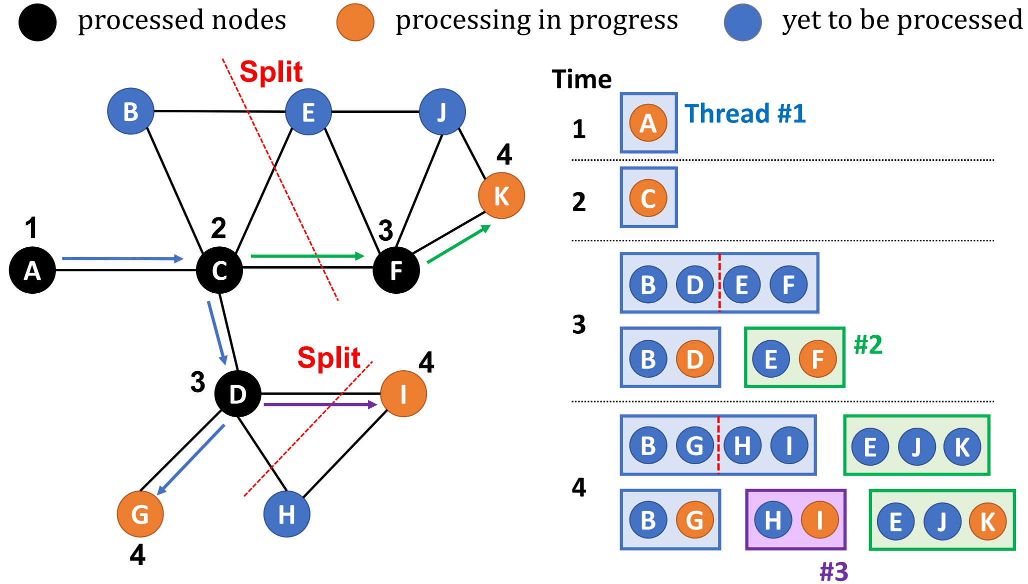


Figure 5·4: Example of the Parallel Pseudo DFS algorithm, demonstrating the progression of the stack over time.

pseudo-DFS [3]. We take inspiration from this idea, and we incorporate a mechanism to monitor the size of the vertex stack for each thread in our implementation (Algorithm 7). In the beginning, only one stack is active with the starting vertex v_s. We create a new $DFStask$ with this stack in the thread pool. The $DFStask$ continuously pops the stack, reads its neighbors, and pushes them into the stack as a normal DFS does. After visiting the neighbors of a vertex, we check if the size of the stack exceeds a predefined threshold. If it does, the stack is evenly divided into two smaller stacks, and one of these stacks is assigned to a new thread for further exploration. Figure 5·4 illustrates the algorithm, with the right side of the figure depicting the timeline status of the stack and its splitting. The graph is a snapshot after *time 4*, where three threads are working in parallel to process the nodes. This approach allows each thread to independently perform DFS on its allocated stack and split it when necessary. By dynamically splitting the stacks in this manner, we achieve increased concurrency during the DFS traversal. The choice of the threshold value

determines the trade-off between concurrency and thread creation overhead. Setting a smaller threshold allows for higher concurrency but may result in a larger number of threads created. On the other hand, a larger threshold reduces the number of thread creations but may limit the degree of parallelism. The selection of an appropriate threshold is crucial to strike a balance between concurrency and overhead.

5.5 Implementation

In this section, we present implementation details of CAVE.

Concurrent Cache Pool. To prevent redundant disk reads, CAVE has a cache pool that stores recently used edge blocks in main memory and employs a clock eviction policy. This caching mechanism becomes crucial because an edge block can contain information for multiple small vertices. It is designed to support concurrent access from multiple threads and enables concurrent I/O operations. As shown in the left-hand-side of Figure 5·3, it comprises three key components: a global lock, a list of slots to store cached blocks, and a cached block map that tracks the mapping of cached block IDs to their positions in the list. Each cached slot within the pool has its lock and a reference counter, while the global lock ensures that only one thread can manipulate the clock hand and modify the map of cached blocks at a given time, preventing potential conflicts.

When a thread requires a specific block, it first checks the cached block map to determine if it is already cached. If the block is found, the thread attempts to acquire the associated lock. Upon successful acquisition, the thread retrieves the content from the block, releases both the global lock and the block lock, and proceeds with the required operations. However, when the desired block is not found in the cache, the thread searches for an available or evicted cached block by moving the clock hand and decrementing the reference counter. Once a suitable cached block is identified,

the thread acquires the lock associated with the cache block, releases the global lock to allow other threads to enter the cache pool, and initiates the process of loading the data block from the SSD. With the global lock released, multiple threads can access the cache pool concurrently and initiate their own I/O operations. After reading the block into the cache slot, the lock is released, making the block available for subsequent use by other threads.

Note that the global lock is kept for a small duration: either until the cached page is accessed in memory, or until a block for eviction/loading is identified and locked. After this, the global lock is released and more threads can enter the cache pool. Our experiments show that in I/O-bound scenarios this short-lived critical section does not create a bottleneck. Figure 5·5 shows the percentage of time that is spent due to the global lock when running parallel BFS in the Friendster dataset (details in §6). The total lock waiting time remains low even with a high number of concurrent I/Os (e.g., around 1.3% of running time for 64 concurrent I/Os), which shows that the global lock does not create a bottleneck. With its concurrent access support and efficient management of cached blocks, this caching mechanism significantly improves overall performance by minimizing unnecessary storage accesses.

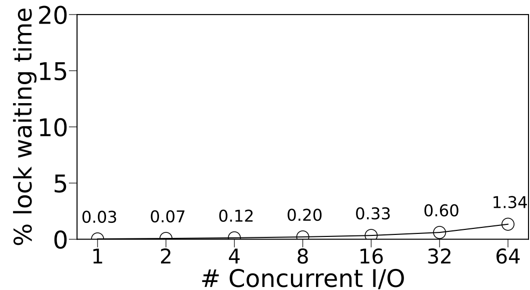


Figure 5·5: Lock waiting time remains low as we increase the number of concurrent I/Os.

Codebase. We develop CAVE using `C++17`, and we leverage its native support for concurrent execution through the `std::thread` functionality. We incorporate the lightweight `BS::thread_pool` library [168] that enhances portability and minimizes overhead. We also use the library parallel hashmap [145] for the cache pool and ensure high accuracy in our runtime measurements with `chrono:: high_resolution_clock`.

I/O Interface. To have full control over the device, we perform direct I/O using the `O_DIRECT` flag so that data is transferred directly from the storage device to main memory, bypassing the system cache. Our blocked file structure ensures that each access is aligned. This alignment is further guaranteed by using the `aligned_alloc()` function whenever new blocks are allocated. For concurrent I/O, we use `pread` and `pwrite` in conjunction with the `BS::thread_pool` library, and we are compatible with both Linux and Windows (leveraging *Overlapped I/O* for the latter).

Data Files. We developed a custom parser to convert common graph data into our binary file structure. It accepts standard adjacent list and edge list files in plain text format as input, parses them, and converts them to our binary file structure.

5.6 Evaluation

We now present the experimental evaluation of CAVE for the five algorithms and compare it with three storage-optimized graph processing systems GraphChi [92], Mosaic [109] and GridGraph [221] for multiple datasets and devices.

Experimental Setup. Our experimental server includes two Intel Xeon Gold 6230 CPUs, each with 20 cores with virtualization, and with 384GB of main memory. We experiment with three storage devices: (i) an *Optane SSD* (375GB P4800X), (ii) an *PCIe SSD* (1TB PCIe P4510), and (iii) a *SATA SSD* (240GB SATA S4610). For all three devices, we quantify the read concurrency (k_r) through careful benchmarking (6 for Optane SSD, 60 for PCIe SSD, and 25 for SATA SSD). Unless otherwise mentioned, we match the number of concurrent I/Os to k_r of the corresponding device for optimal device utilization [130]. All devices were pre-conditioned by sequentially writing on the entire device three times before running the experiments to ensure stable performance [44]. All experimental results are averaged over three iterations, and the standard deviation was less than 1%.

Dataset. We use five datasets of different sizes and types from the Stanford Large Network Dataset Collection [97] and LDBC Graph Analytics Benchmark [69]: Friendster Social Network (FS), Twitter Social Network (TW), RoadNet Network of PA (RN), LiveJournal Social Network (LJ) and YouTube Social Network (YT). FS is the largest dataset among these, with 65M nodes and 32GB size. We also experiment with a synthetic dataset (SD) which is generated following the Barabási–Albert model [8]. We configured the graph with 50 million vertices, each connected to 25 neighbors, resulting in a total of 1.25 billion edges. Note that the RN graph is very sparse while the SD graph is extremely dense. The key properties of the datasets are presented in Table 5.1.

Table 5.1: Dataset Description

Dataset	Description	#Nodes	#Edges	Diameter	Size
FS	Friendster Social Network	65M	1.8B	32	32 GB
TW	Twitter Social Network	53M	2B	18	28 GB
RN	RoadNet Network of PA	1M	1.5M	786	47 MB
LJ	LiveJournal Social Network	5M	69M	16	1 GB
YT	YouTube Social Network	1.1M	3M	20	39 MB
SD	Synthetic data	50M	1.25B	6	20 GB

Preprocessing time and space requirement. Table 5.2 presents the preprocessing time and space requirement of all systems for the FS and TW dataset. CAVE exhibits the lowest preprocessing time and a reduced space requirement compared to GridGraph and Mosaic. For example, Mosaic's preprocessing time and space requirement is 9× and 2× that of CAVE for the FS dataset, respectively. While GridGraph has a similar preprocessing time to CAVE, its space requirement is 6× that of CAVE. This efficiency stems from CAVE's compact file architecture and simple design, contrasting with the more demanding preprocessing requirements of systems like GridGraph and Mosaic.

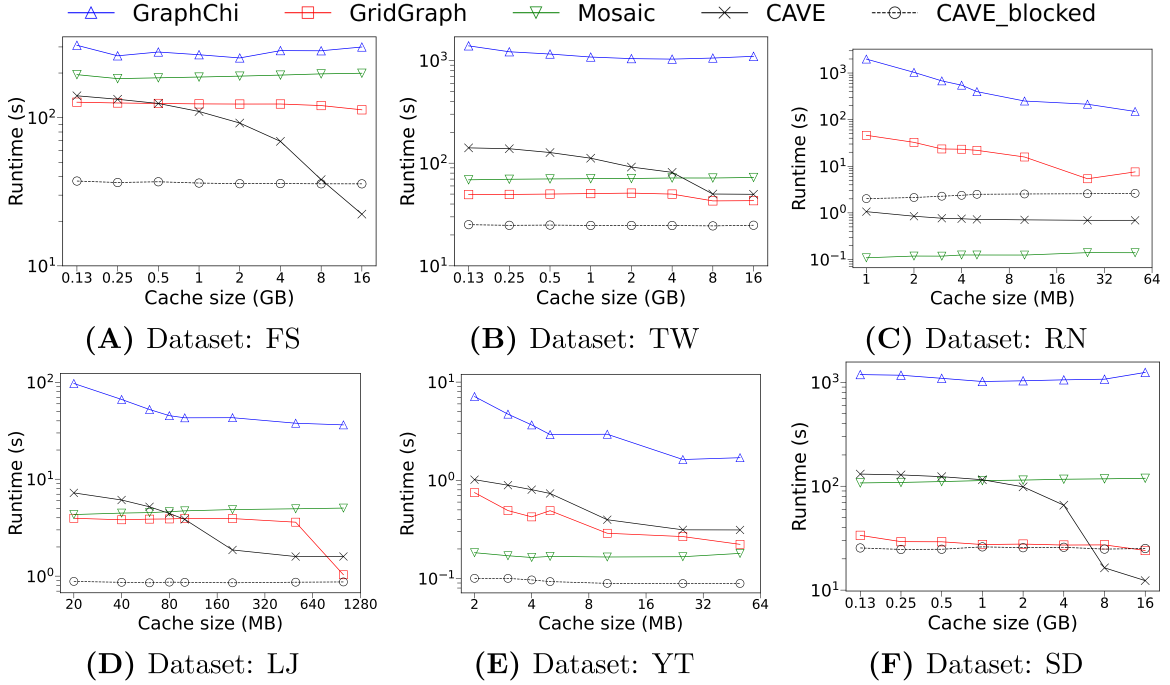
Table 5.2: Preprocessing Time and Space Comparison

System	Preprocessing Time (s)		Data File Size (GB)	
	Dataset: FS	Dataset: TW	Dataset: FS	Dataset: TW
GraphChi	819 s	784 s	8.3 GB	8.4 GB
GridGraph	55 s	86 s	84 GB	75 GB
Mosaic	469 s	370 s	27 GB	17 GB
CAVE	52 s	49 s	14 GB	13 GB

5.6.1 Parallel BFS

CAVE Outperforms all baselines. In our first set of experiments, we evaluate the performance of CAVE, GraphChi, GridGraph and Mosaic as we vary the cache size for all six datasets. Figures 5·6(A) - (F) show the performance of the four systems for BFS when the underlying device is the PCIe SSD. We compare using both the blocked and non-blocked variants of the frontier processing to see their effect on different graphs. Since the datasets have different sizes, the cache value is set accordingly. The results show that CAVE (both blocked and non-blocked) significantly outperforms GraphChi for any cache ratio and any dataset. Notably, when the cache ratio is low, CAVE outperforms GraphChi with a significantly higher speedup due to its better utilization of SSD concurrency.

For example, Figure 5·6(A) presents a performance comparison of the four systems for the Friendster dataset (65M nodes, 32GB size). The figure shows that the non-blocked implementation benefits from a higher cache size while the blocked implementation remains unaffected by the cache size. This is due to the design techniques of the blocked implementation, which ensure that all edge blocks are read only once during an iteration. We observe that GridGraph and Mosaic are faster than GraphChi, but both CAVE implementations outperform them. Specifically, the blocked version provides up to 3.4× and 5.6× speedup across various cache sizes against GridGraph and Mosaic, respectively. The non-blocked variant delivers comparable performance



(A) Dataset: FS **(B)** Dataset: TW **(C)** Dataset: RN

(D) Dataset: LJ **(E)** Dataset: YT **(F)** Dataset: SD

Figure 5·6: (A) - (F) Performance graph for BFS on our PCIe SSD. In general, CAVE outperforms the baselines GraphChi, GridGraph and Mosaic for all six datasets. Mosaic performs well for the sparse RN dataset.

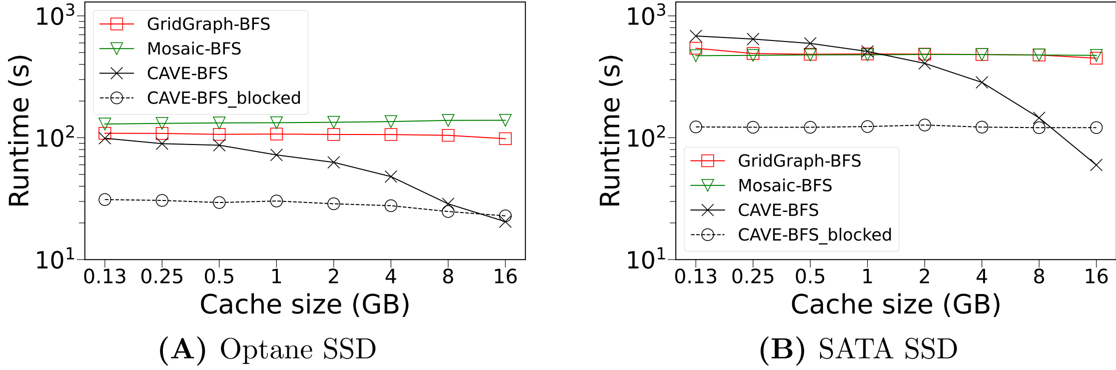
to GridGraph for smaller cache sizes and up to 5.1× speedup for larger cache sizes while always outperforming Mosaic by a higher margin. CAVE's blocked implementation outperforms the other three systems for the TW, LJ and YT datasets as shown in Figures 5·6(B), (D) and (E).

CAVE is Well-suited for Sparse Graphs. In Figure 5·6(C), we see that in a sparse graph with an unusually high diameter (RN dataset), CAVE performs better than both GraphChi and GridGraph, but falls short of Mosaic. The non-blocked implementation of CAVE works well for this graph because, in sparse graphs with lower average degrees where edge blocks can be associated with multiple vertices, all required edge blocks for an iteration can fit in the cache pool without swapping. With a sufficiently large cache, the non-blocked variant benefits from multiple threads work-

ing on cache data. The figure shows that the blocked implementation (black dashed line) outperforms both GraphChi and GridGraph while the non-blocked implementation (black solid line) can be up to 3.8× faster than the blocked one. While CAVE consistently outperforms Mosaic for other datasets, Mosaic outperforms CAVE for this dataset. This is because Mosaic uses *Hilbert-ordered tiles* as its graph representation, which allows to skip lots of empty/unneeded tiles for sparse graphs.

CAVE Provides Good Performance for Dense Graphs. Our synthetic SD dataset is an unusually dense graph (the diameter is only 6 with 50M nodes and 1.25B edges). In Figure 5·6(F), we observe that CAVE outperforms GraphChi and Mosaic. However, CAVE-blocked provides marginal benefit compared to GridGraph for the SD dataset. The reason behind this is that GridGraph is designed for dense graphs because its data structures and algorithms are optimized to efficiently handle the high connectivity and dense nature of such graphs. GridGraph achieves this by utilizing a grid structure to partition and manage the graph's data. This approach allows it to optimize memory usage and reduce access times for dense graph structures. However, CAVE still provides up to 25.4% faster runtime compared to GridGraph for dense graphs.

Similar Performance Benefit Across All Devices. We now compare CAVE against Mosaic and GridGraph in the Optane SSD and SATA SSD as we vary the cache size for BFS on the FS dataset. Figures 5·7(A) and (B) show that the performance trend of these systems remains similar to the PCIe SSD (Figure 5·6(A)). CAVE consistently outperforms both GridGraph and Mosaic. We also observe that all systems running on the Optane SSD have an overall lower runtime than the PCIe SSD because of Optane SSD's faster read performance. In contrast, all systems running on the SATA SSD have a higher runtime than the PCIe SSD due to the slowness of the SATA SSD.



(A) Optane SSD **(B)** SATA SSD

Figure 5·7: (A, B) Performance of PBFS on Optane SSD and SATA SSD for the FS dataset. CAVE outperforms the baselines.

CAVE Utilizes Concurrent I/Os. To analyze how the concurrent I/O affects the performance when using various devices and datasets for the algorithms, we now vary the number of concurrent I/Os in the blocked PBFS implementation. Since GraphChi, GridGraph or Mosaic do not have the support of varying concurrent I/Os, we do not include them in this experiment. Figure 5·8 shows CAVE's performance graph for the FS dataset for all three of our devices. The figure shows that as we increase the number of concurrent I/Os, PBFS's runtime decreases until the device becomes saturated. For example, the SATA SSD gets saturated when using 16-32 concurrent I/Os (red line), which is consistent with the device's optimal concurrency value (25). However, if we issue more concurrent I/Os, performance starts to degrade because of the thread management overhead while the device is already saturated. The PCIe SSD curve is moved towards higher concurrency, as expected, and Optane SSD has a flatter curve, which is consistent with prior work [130, 132].

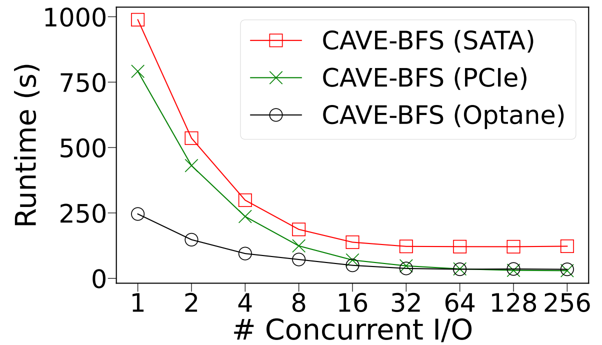
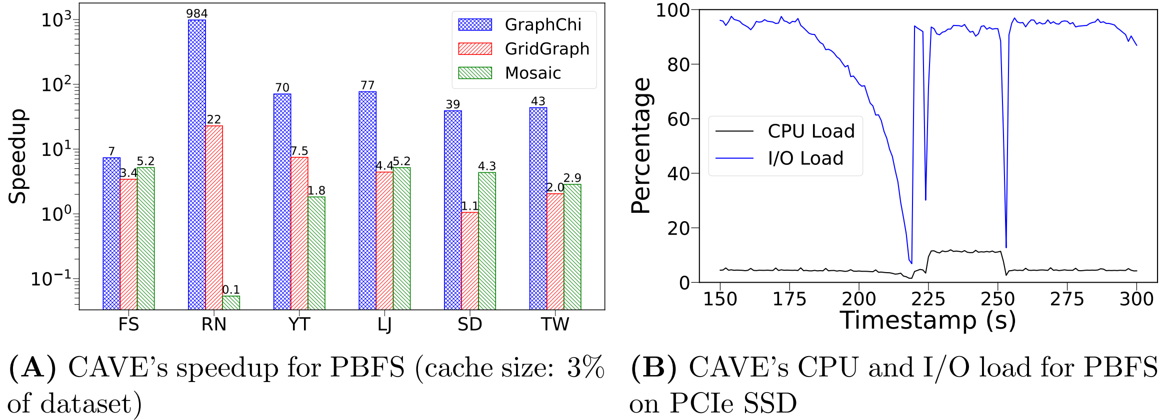


Figure 5·8: As we increase the number of concurrent I/Os, the benefit of PBFS increases until the device gets saturated.



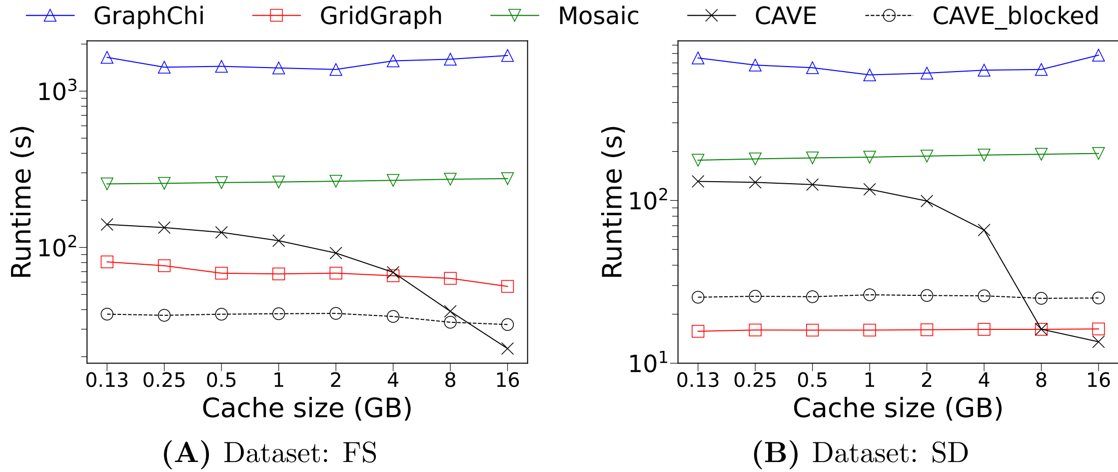
(A) CAVE's speedup for PBFS (cache size: 3% of dataset)

(B) CAVE's CPU and I/O load for PBFS on PCIe SSD

Figure 5·9: (A) CAVE performs well across all datasets for PBFS. Only Mosaic has a better performance for the sparse RN dataset. (B) CPU usage of CAVE remains low showing that its benefit comes from better SSD utilization – CAVE is I/O-bound, not CPU-bound.

CAVE Excels Across Different Datasets. Figure 5·9(A) shows CAVE's speedup vs. GraphChi, GridGraph, and Mosaic for all five datasets with cache size 3% of the dataset when running on the PCIe SSD. The speedup of CAVE compared to GraphChi, GridGraph, and Mosaic ranges from $7-984\times$, $1.1-22\times$, and $0.1-5.2\times$, respectively. The unusually high speedup compared to GraphChi for the RN dataset is attributed to the high diameter of the graph, where GraphChi needs an very large number of iterations to converge. GridGraph also takes a high number of iterations to converge which shows that CAVE can handle graphs with high diameters better than both GraphChi and GridGraph. Although Mosaic performs the best for this sparse dataset, CAVE outperforms Mosaic for the other datasets. For dense graphs like SD, GridGraph performs well because of its grid structure to partition and manage dense graphs, however, CAVE still outperforms GridGraph.

CAVE's Benefits Come From Better Storage Utilization. Our profiling shows that CAVE is I/O-bound (rather than CPU-bound), thus, the performance benefits come from better utilization of the underlying storage devices. We capture a snapshot of the CPU load and disk bandwidth for PBFS on the FS dataset in Figure 5·9(B).



(A) Dataset: FS

(B) Dataset: SD

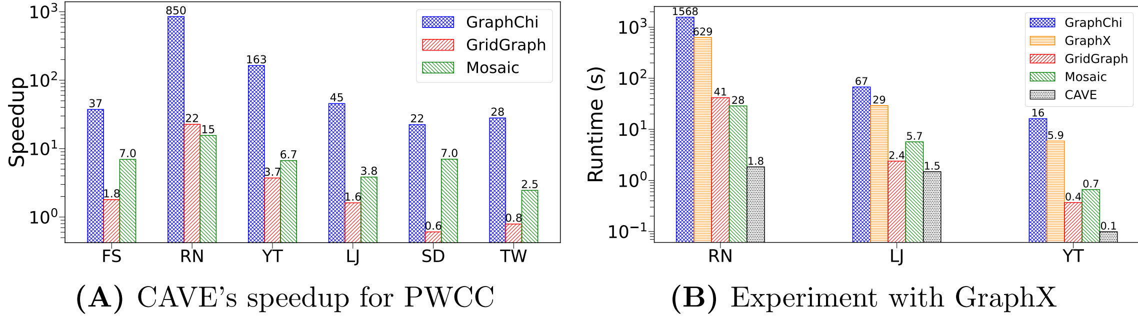
Figure 5·10: (A) Blocked CAVE implementation outperforms other systems for WCC. (B) For dense graphs, GridGraph works well for finding WCC.

The figure shows that CAVE consistently maintains low CPU utilization, generally below 5%, while the disk bandwidth remains around 100%. To calculate bandwidth utilization, we divide the observed bandwidth by the maximum device bandwidth achieved when using the same number of concurrent threads to issue I/Os.

5.6.2 Parallel WCC, PR & RW

Parallel Weakly Connected Components

Figures 5·10(A) and (B) present the performance graph of two CAVE implementations, GraphChi, GridGraph, and Mosaic, for weekly connected components as we vary cache size running on the PCIe SSD device for FS and SD datasets. Similarly to the PBFS experiments, the runtime of the blocked implementation does not depend on the cache size. Figure 5·10(A) shows that CAVE's blocked implementation achieves up to 44×, 2.2×, and 8.6× speedup compared to GraphChi, GridGraph and Mosaic for the FS dataset. However, for the SD dataset in Figure 5·10(B), GridGraph achieves 1.6× better runtime than CAVE. As mentioned earlier, this is because the

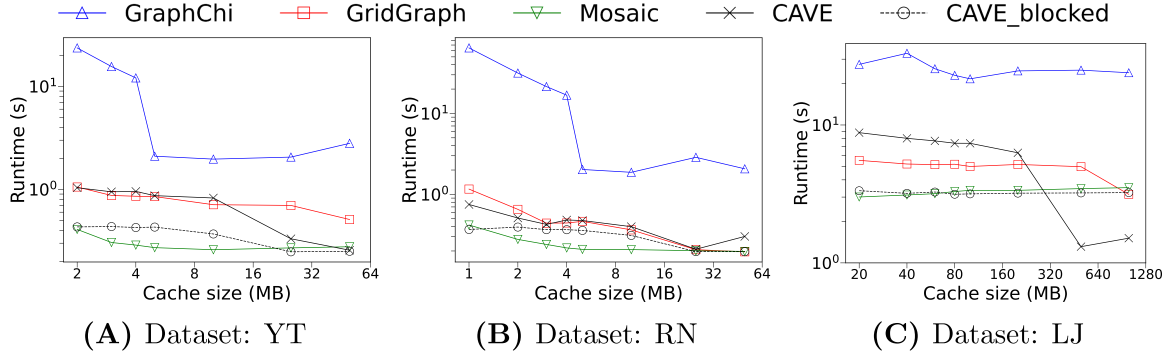


(A) CAVE's speedup for PWCC **(B)** Experiment with GraphX

Figure 5·11: (A) CAVE performs well across all datasets for PWCC. (B) CAVE outperforms single-node GraphX deployment.

SD dataset is seriously dense which works in favor of GridGraph. For both datasets, the non-blocked implementation has a higher runtime when the cache size is small. However, the performance improves as the cache size increases, providing up to 2.5× and 1.2× speedup compared to GridGraph for the FS and SD datasets for higher cache sizes. Figure 5·11(A) presents a summary result of WCC for all five datasets on the PCIe SSD (3% cache size for each dataset) which shows CAVE can achieve $22-850\times$, $0.6-22\times$ and $2.5-15\times$ speedup compared to GraphChi, GridGraph and Mosaic respectively.

Comparison against GraphX. For completeness, we experiment with GraphX [55] that implements graph-parallel computation by distributing the computation in multiple compute nodes using a different abstraction while requiring all data in memory. Since CAVE is a single-node out-of-core system, a direct comparison against a distributed system should be taken with a grain of salt. We experiment with a single-node deployment of GraphX on our server. Figure 5·11(B) presents the runtime of GraphChi, GraphX, GridGraph, Mosaic, and CAVE for three datasets when running WCC. We observe that CAVE can be up to two orders of magnitude faster than GraphX. Specifically, CAVE achieves 350×, 19× and 59× speedup compared to GraphX for RN, LJ and YT datasets. It is worth highlighting that CAVE achieves



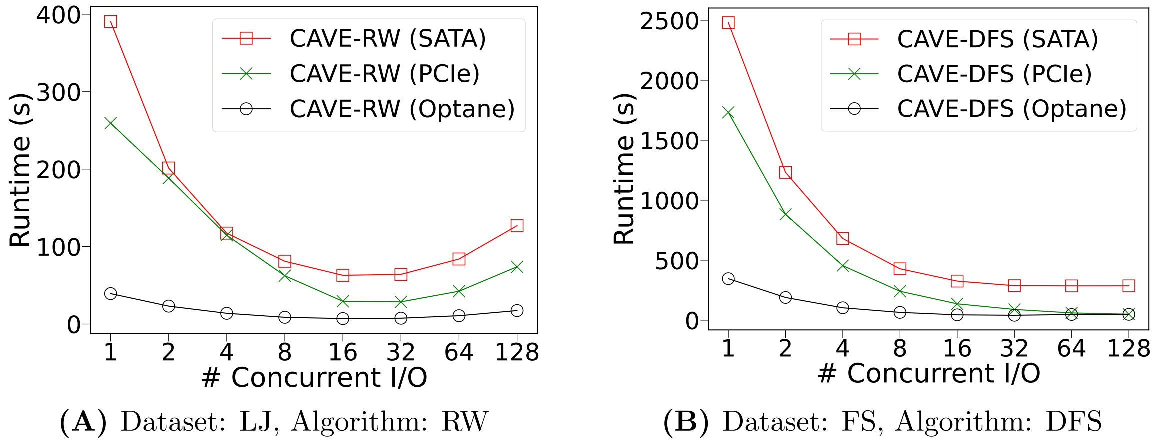
(A) Dataset: YT **(B)** Dataset: RN **(C)** Dataset: LJ

Figure 5·12: CAVE achieves lower runtime for PageRank, outperforming GraphChi and GridGraph.

this superior performance despite GraphX having all data in memory whereas CAVE only uses 3% memory of the dataset size. We recognize that GraphX is designed for distributed environments, however, the potential costs associated with distributed computing, both monetary expenses and the overheads of managing a distributed computing infrastructure can be significant.

Parallel PageRank

Figures 5·12(A) - (C) illustrates the performance graph for YT, RN and LJ datasets in the PCIe SSD. The blocked implementation of CAVE achieves lower runtime than both GraphChi and GridGraph while achieving comparable performance to Mosaic. The figure shows that CAVE achieves up to 98× and 3.3× speedup compared to GraphChi and GridGraph, respectively, for the YT dataset. For the RN dataset shown in Figure 5·12(B), CAVE's speedup is up to 170× (3.0×), while for the LJ dataset shown in Figure 5·12(C), CAVE achieves 8.2× (1.6×) speedup compared to GraphChi (GridGraph). Similarly to our previous experiments, the performance of the non-blocked implementation of CAVE improves as the cache size increases.



(A) Dataset: LJ, Algorithm: RW

(B) Dataset: FS, Algorithm: DFS

Figure 5·13: (A) Concurrent I/Os improve performance for CAVE's Random Walk till the device is saturated. (B) CAVE's PDFS can attain the maximum benefit of the device by exploiting its *optimal concurrency*.

Parallel Random Walk

Figure 5·13(A) shows the representative performance graph of CAVE as we vary the number of concurrent I/O for the LJ dataset across our three SSDs. The figure shows that as we increase the number of concurrent I/Os the runtime decreases across all devices. As expected, we also observe that the Optane SSD is faster than the PCIe SSD and the PCIe SSD is faster than the SATA SSD. As the number of concurrent I/Os reaches the *optimal concurrency* of each device, its runtime is minimized. However, beyond this point, the runtime starts to increase as the device is saturated, and additional parallelism increases the thread management overhead.

5.6.3 Parallel pseudo DFS

We now focus on the impact of I/O concurrency on the PDFS algorithm. We generate a list of target keys at random and run the algorithm to search each key multiple times in a depth-first manner. Note that GraphChi, GridGraph, Mosaic, and most graph processing systems do not support DFS.

CAVE's PDFS Exploits Device Concurrency. Figure 5·13(B) shows the performance of CAVE's PDFS as we vary the number of concurrent I/Os for the FS dataset across three devices. We observe that for all devices, as we increase the number of concurrent I/Os, we have improved runtime until the device performance plateaus. The figure shows that CAVE's PDFS achieves 7.7×, 12.6×, 7.6× speedup on the Optane SSD, SATA SSD, and PCIe SSD. We can also reason about each device's concurrency values from the graph as the runtime flattens when the device bandwidth is saturated. The figures also show that the fastest device (Optane SSD) has a much lower runtime than the slowest device (SATA SSD). We observe a similar trend for other datasets. Overall, these experiments show that CAVE can perform parallel pseudo-DFS while leveraging the underlying SSD concurrency.

5.7 Related Work

Many scalable graph processing systems like PowerLyra [29], GraphX [55], GBase [84], TurboGraph++ [91], PowerGraph [54], Chaos [155], GraphLab [105], Pregel [112], Gemini [220], VC-Tune [222] can process large graphs in a distributed manner which requires finding optimal partitioning, load-balancing, fault tolerance, and managing the communication overhead. There are some single-node shared-memory systems like Ligra [169], GraphMat [177], GRACE [202], Polymer [211], CGraph [216] that process graphs in memory, and as expected, these systems are highly CPU bound. There are several popular out-of-core processing systems including GraphChi [92], TurboGraph [64], Graphene [101], Mosaic [109], X-Stream [156], GridGraph [221], Graspan [194], RStream [195], and FlashGraph [218]. These systems attempt to minimize random disk access while relying on sequential I/O, extensive preprocessing, and optimal data placement. GraphSSD [114] is a graph-aware SSD framework where the SSD controller is made aware of the graph data structures stored on the SSD. In

contrast to these approaches, our goal is to develop a general approach for parallelizing graph traversal algorithms and develop the necessary infrastructure to exploit the underlying *SSD concurrency*.

5.8 Conclusions

Modern SSDs are characterized by their access *concurrency*, which needs to be carefully harnessed to attain maximum benefit from the device. Graph processing systems are a natural candidate for exploiting this property, yet, most systems do not consider it, resulting in device underutilization. We propose CAVE, a concurrency-aware graph processing system designed to leverage the underlying SSD concurrency. CAVE parallelizes independent I/Os through its concurrent cache pool design, supported by its file structure, enabling the implementation of storage-aware parallel graph algorithms. We develop the parallelized versions of five popular graph algorithms in CAVE and compare their performance with three out-of-core systems, GraphChi, GridGraph, and Mosaic, for multiple datasets and devices. Our evaluation reveals that CAVE achieves up to three orders of magnitude higher speedup than GraphChi and up to one order of magnitude higher speedup than GridGraph and Mosaic.

Chapter 6

Restore: RL-Based Data Migration for Multi-Tiered Storage Systems

With the development of storage technologies, a wide variety of storage devices with differing performance characteristics and cost profiles have emerged. As a result, data systems are increasingly adopting multi-tiered storage solutions, where fast (small) devices are placed in higher tiers, while lower tiers consist of slow (large) devices. A primary challenge in multi-tiered storage systems is data placement, as data must be dynamically stored and migrated across different storage tiers to optimize overall performance. Effective data migration policies should be able to adapt to workload variations while also considering the unique characteristics of underlying devices (such as PCIe/SATA SSD, or HDD), notably their read/write asymmetry and parallelism.

In this chapter, we introduce ReStore[1] [215], a reinforcement learning (RL) approach for data migration in multi-tiered storage systems. ReStore leverages RL to capture both workload patterns and device-specific characteristics, including access frequency and recency, as well as device read/write asymmetry and parallelism. Each storage tier uses a different device and is associated with an RL agent that dynamically updates its parameter using temporal difference learning, ensuring continuous adaptability to changing workloads and system states. We experimentally show that ReStore achieves up to 2.2× lower runtime and up to 10× fewer migrations using

[1] The material of this chapter has been the basis for a paper under preparation entitled "Restore: A Reinforcement Learning Approach For Data Migration In Multi-Tiered Storage" [215].

industry-grade benchmarks, like TPC-C/E and YCSB, real-life traces, like Google Thesios, and a wide variety of synthetic workloads.

6.1 Introduction

Rise of Big Data. Data generation is growing at an unprecedented rate due to sources like social media, digital transactions, sensors, and the expanding network of Internet of Things (IoT) devices [28, 185]. As a result, a key challenge relates to data storage, retrieval, and maintenance, leading to the need for new scalable storage architectures that balance performance with cost.

Tiered Storage Architecture for Big Data Management. To address these needs, cost-effective storage solutions have been developed that scale (i) horizontally, resulting in distributed storage (file) systems (DFS) such as GFS [53], Cassandra [94], HDFS [170], Ceph [198]; and (ii) vertically, resulting in the development of tiered storage systems like HP AutoRAID [200], IBM Storage Tank [34, 118]. Data-intensive applications are increasingly employing *tiered-storage systems* that offer a unified interface to a collection of devices organized in *tiers*, each with varying levels of performance, capacity, and cost similar to the classical memory hierarchy [10]. Generally, the top tier's devices are faster and smaller, where the lower tier devices are slower and larger. However, unlike the classical memory hierarchy, applications can read from any tier directly without always having to push to a higher level [65]. An effective tiered storage management strategy is essential to maximize the resource utilization and performance while balancing the infrastructure costs [212]. It achieves this by dynamically prioritizing *important* data for placement on faster, more expensive storage devices while relegating less critical data to slower, low-cost devices. We now ask the question: *how do modern storage device properties affect the performance of a tiered storage system?*

Storage Devices and Their Properties. Since SSDs are now the fastest and dominant storage device, multiple (or all) tiers of a tiered storage system typically employ SSDs, hence, we focus on how to better exploit these devices. SSDs exhibit a high degree of internal parallelism (*concurrency*) that can be harnessed to increase performance [130, 134]. On the other hand, for flash-based SSDs, the cost of reading is generally lower than the (amortized) cost of writing, leading to an SSD read/write asymmetry [36]. These two properties of SSDs are crucial: (i) proper use of SSD concurrency enables better device utilization, and (ii) special care of expensive writes can optimize overall workload execution [131, 132]. Hence, proper performance modeling of SSDs that captures (and exploits) device properties can help tiered storage systems improve resource utilization and performance.

Challenges of Tiered Storage Management. The management of large datasets in a multi-tiered environment is a complex, dynamic problem that requires flexible solutions. The best strategy depends on data type, access patterns, storage device, and long- and short-term availability of the data. A recurring challenge is the financial cost associated with the data stores. Data access patterns typically evolve over time, and keeping the data on the fastest storage devices is economically challenging as faster tiers are generally expensive and small. On the contrary, pushing all the data on the slow tiers (slow SSD, HDD, or even an object store) degrades the overall performance of the applications. Therefore, it is essential to have an effective data migration policy in multi-tiered storage that governs when and how data is moved between tiers. Most common migration policies are recency-based, frequency-based, or a hybrid of these two approaches, and all face three major challenges.

Challenge 1: Optimal Data Placement. The data placement strategy or migration strategy (across tiers) is *the most crucial design decision* for any multi-tiered storage system. Ideally, we want the *hot* data in the faster tiers while the *cold* data

should be in the slower tiers. In case of dynamic workloads, this is not straightforward as data hotness can vary drastically over time. Systems that rely on simple rule-based strategies (e.g., LRU-style or LFU-style policies) for data placement cannot adapt to workload drift, leading to suboptimal performance. Further, these rule-based strategies along with classical caching policies (e.g., clock, FIFO, LRU) are designed primarily for a simpler problem setting (e.g., two tiers) and they do not consider storage device characteristics.

Challenge 2: Capturing Storage Device Properties. SSD concurrency and asymmetry have not been exploited in tiered storage systems. For example, there is a 40× increase in the read bandwidth of the 1TB PCIe Intel P4510 SSD when using full concurrency compared to no concurrency and 4KB writes are 3× slower than 4KB reads [130]. Without capturing these storage properties accurately, the devices can remain vastly underutilized.

Challenge 3: Finer Migration Granularity. Existing data migration policies for tiered storage are designed for data files/objects. However, many applications (i.e., DBMS, KV-store) running on tiered storage operate on pages, creating the need for new page migration policies for tiered storage systems. To work on such finer granularity, data migration policies must be *lightweight* while capturing both workload and device properties.

Overall, existing tiered storage systems rely on predefined rules (e.g., recency-based, frequency-based) for data files/object placement and migration, which do not work well for page-level migration and do not adapt well to changing workload patterns and access frequencies. Further, *rule-based approaches do not interact with the system and, thus, do not take into account device properties, rather, rely on basic heuristics that lead to suboptimal data placement.*

Reinforcement Learning to the Rescue. Reinforcement Learning (RL) uses in-

telligent *agents* to make decisions by *interacting* with the environment and receiving feedback in the form of rewards or penalties aiming to maximize a cumulative reward. The RL formulation typically represents the state of the modeled system, and, thus, is well-suited for learning how to migrate data in tiered storage systems, as it can adapt to changing workloads and an evolving device state to optimize data placement. Further, an RL-based data migration approach can incorporate new storage technologies as they emerge, providing a future-proof solution.

Design Goals. In developing an RL-based page migration policy for multi-tiered storage devices, we set the following design goals:

- **Model Heterogeneous Tier Properties:** Use a multi-agent approach to address diverse tier properties such as capacity, read/write asymmetry, and concurrency. By associating one agent per tier, we can capture these system characteristics.
- **Dynamic Tier Adaptability:** Support seamless addition or removal of tiers with one RL agent per tier for modularity.
- **Lightweight Agent Design:** Ensure lightweight RL processes to handle frequent, fine-grained data (page) placement decisions with minimal overhead. By having minimal parameters per agent, we can ensure scalability and real-time operation.
- **Approximate Value Function:** Prioritize relative value estimates over accurate complex calculations. Many RL approaches accurately calculate the value function at the expense of computational cost. Since in this use case, relative values are sufficient to make a decision, we can use simpler approximations.

ReStore: An RL-Based Tiered Storage Migration Policy. To address the above challenges, we propose ReStore, an RL-based multi-tiered storage migration policy, which can be integrated into any multi-tiered storage system that allows for **page-level** accesses. As part of the RL formulation, we define state variables that

Table 6.1: Our proposed approach captures workload features and device properties and quickly adapts to workload drift.

features	**EXD**	**LRFU**	**tLRU**	**tLFU**	**Our Approach**
access frequency	✓	✓	×	✓	✓
access recency	✓	✓	✓	×	✓
dynamic allocation	✓	✓	✓	✓	✓
workload drift	∼	∼	×	×	✓
device properties	×	×	×	×	✓

capture the workload features (recency and frequency) and device properties (concurrency and asymmetry), and use a reward function tied to system runtime. The policy is then optimized by maximizing the accumulated rewards through value functions that quantify the quality of the current system state, which in turn minimizes workload execution cost. To ensure adaptability, we employ Temporal Difference learning to update the value functions dynamically. A prime benefit of ReStore is that its RL-based data migration gets feedback from the system through interaction and adapts accordingly. Table 6.1 contrasts current migration policies for multi-tiered storage. State-of-the-art policies like EXD [50] and LRFU [119] that rely on a utility function to capture workload features, ignore device properties, and struggle with workload drift. tLRU and tLFU that adapt the LRU and LFU caching policies for migration capture access statistics, but are slow to adapt to drift and do not consider device properties. Compared to state-of-the-art approaches, ReStore is the only policy that considers device properties and responds well to dynamic workloads.

We compare ReStore with different rule-based policies, highlighting that ReStore causes significantly fewer data migrations and maximizes resource utilization and runtime performance. This is because ReStore takes into account storage device properties like concurrency and asymmetry to ensure optimal device utilization, while

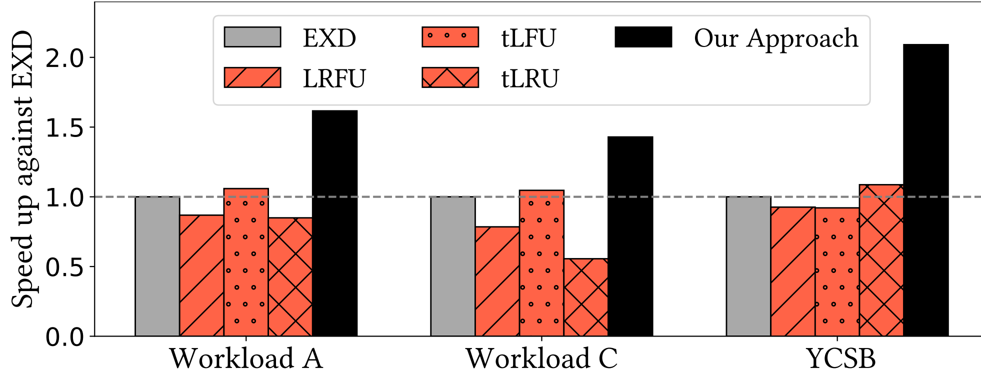


Figure 6·1: From the three different workloads, A is more skewed, and YCSB has more drift. More details can be found in Section 6.6.1. Our approach always outperforms the state of the art by up to 2.2×.

being computationally lightweight. Figure 6·1 compares ReStore with other baseline migration policies in case of workload drift (all details for the experimental setup and the workloads are in Section 6.6.1). The figure shows that ReStore achieves up to 2.2× speedup compared to other baselines for changing workloads, showcasing ReStore's flexibility to adapt to dynamic workloads.

Contributions. This chapter makes the following contributions.

- We identify the key challenges of multi-tiered storage management: optimal data placement, page-level migration, dynamic access pattern and capturing storage device properties.

- We propose ReStore [215], a multi-tiered storage management approach that uses *reinforcement learning* for *page-level migration*. By interacting with its environment and underlying storage, ReStore accurately captures the system state to make migration decisions based on data access patterns. ReStore models the system based on (i) the *temperature* of each page, which captures the recency and frequency of page accesses, and (ii) SSD properties (concurrency and asymmetry).

- We build a multi-tiered storage emulation framework that accurately simulates various storage devices while capturing the operating principles of tiered storage.

We deploy ReStore in the framework to experiment with a variety of configurations.

- We extensively evaluate the efficacy of ReStore against state-of-the-art approaches, and show that it can speed up execution by up to 2.2× while having up to 10× fewer page migrations.

6.2 Background & Related Work

Multi-Tiered Storage Systems. Data migration has long been integral to storage systems [1, 2], especially in multi-tiered architectures, to optimize system performance, scalability, and reliability while balancing resource utilization. For example, AutoRAID [199] automatically migrates frequently accessed data to faster tiers and demotes less accessed data to slower tiers. Many policies have been proposed for data migration in tiered storage at coarser granularity.

Data Migration at File/Object Level. The process of moving data objects or files from one storage layer to another is typically based on predefined policies. For instance, LMsT [181] introduces multiple constraints and rules to support applications with dynamic workloads and varying service level agreements. EDT-DTM (Extent-Based Dynamic Tiering Management) [61] leverages a *resource consumption model* to enable cost-effective tiering. Data migration cost models have been used as constraints [161], resulting in algorithms including ABD, $\text{OPT}^{\text{bounded}}$, SPACE, TIME, and LMIN. Similarly, the bandwidth-to-space ratio can be utilized to manage migration based on storage and access costs [178]. Hotness or temperature-based migration criteria are also widely used. MTM [151], for example, employs the exponential moving average (EMA) of hotness indication to guide migration, while another work focuses on an extended data temperature value using new metadata and user-defined variables [41]. ADLAM [210] also leverages extent temperature in their adaptive look ahead data migration model. Siberia [45] is main-memory database engine that keeps cold data

in storage devices. All these policies work at the file/object level and/or for two tiers (memory and storage), while our objective is to perform page-level migration in a storage multi-tiered setup. These policies cannot support such finer migration granularity due to the high complexity nature of the problem setup. Since no such policies for tiered storage exist formally in the literature, we take inspiration from in-memory caching techniques which work on page-level granularity.

Page-level Caching and Eviction Policies. Caching is a well-studied problem relevant to many areas of computer science [144]. Broadly, cache eviction policies can be categorized into four classes: recency-based, frequency-based, hybrid, and function-based approaches. *(A) Recency-based policies* prioritize data based on how recently it has been accessed like the classical 5-minute rule [58] and their recent adaptations [13, 56, 57]. The most well-known example is Least Recently Used (LRU) [125], which evicts (migrates to bottom tier) the least recently accessed data to make room for newer requests. Variations like 2Q [78], MQ [219], and CLOCK-Pro [71] either try to improve the adaptability or reduce overhead. *(B) Frequency-based policies* focus on the number of times data is accessed. Least Frequently Used (LFU) evicts data that has been accessed the least. Variants like LFU with Dynamic Aging (LFUDA) [14] tackle the issue of aging by penalizing older data, helping avoid the accumulation of stale data. Both recency and frequency based approaches struggle when workload patterns change. *(C) Hybrid policies* combine recency and frequency to capture both short-term and long-term access trends. ARC (Adaptive Replacement Cache) [117], LIRS (Low Inter-reference Recency Set) [72] are popular examples of this class that dynamically adjust between LRU and LFU depending on the workload. *(D) Function-based policies* use models, heuristics, or algorithms to decide when and where to move data between tiers. For example, EXD (Exponential Decay) [50] determines data migration based on an exponential decay score that weighs recency and frequency; An-

other example is LRFU [119], which controls replacements using a Combined Recency and Frequency (CRF) value based on a parameterized exponential function.

ML-based Data Migration. In recent years, machine learning (ML) techniques like logistic regression, neural networks, genetic algorithms, and random forests have been used for web caching [9, 24, 191]. In a tiered setup, the migration function is typically learned using supervised learning where the system train predictive models on historical workload data to predict future access patterns and optimize data migration. Approaches such as Support Vector Machines (SVM) [166], decision trees (CART) [32], XGBoost [65] and popular neural networks, including MLP, LSTM, and CNNs [152, 167, 184], have been applied. Unsupervised learning methods, including clustering [11, 128, 166] and evolutionary algorithms like Particle Swarm Optimization (PSO) [7, 209], have also been explored to group similar access patterns together, guiding migration decisions based on inferred workloads. Both approaches are capable of learning in dynamic environments, however, they come with heavy overhead. Reinforcement learning (RL) and other control theory have also been used for data migration [89, 106, 107, 172, 197]. These approaches employ deep RL which creates a large overhead and can not applied for *page-level data migration* in a tiered setting. Meanwhile, beyond CHROME [107] that take concurrency into consideration, none of the above mentioned methods capture the storage device properties. In contrast to these above-mentioned approaches, our proposed RL-based approach ReStore is (i) lightweight, (ii) capable of page-level data migration, (iii) adaptive to varying workloads, and (iv) aware of device-specific properties.

SSD Asymmetry and Concurrency. Table 6.2 lists the access latency, asymmetry (α), read concurrency (k_r) and write concurrency (k_w) of some representative devices where we see that both α and k vary across devices, access granularity, and access pattern, while all NAND-based SSDs exhibit both asymmetry and concurrency.

Table 6.2: Empirical latency, asymmetry and concurrency of some storage devices for 4KB block size.

Device	Read Latency (μs)	Write Latency (μs)	$\boldsymbol{\alpha}$	$\boldsymbol{k_r}$	$\boldsymbol{k_w}$
Intel Optane P4800X SSD	6.8	7.6	1.1	6	5
Intel PCIe P4510 SSD	12.4	19.6	2.8	80	8
Intel PCIe DC-P4500 SSD	8.2	78.4	11.9	40	8
Intel SATA S4610 SSD	100	140	1.5	25	9
SATA ST2000NX0463 HDD	7000	7300	1.0	1	1

6.3 Challenges of Page-Level Multi-Tiered Storage

The goal of a multi-tiered storage architecture is to manage different storage devices in a manner that provides optimal performance, application functionality, and service-level guarantees. As discussed in Section 6.2, multi-tiered storage systems typically manage data at the granularity of a file or an object. When considering using multi-tiered storage systems for data systems that operate at the *page granularity* (e.g., relational database management systems, key-value value stores), they face a series of challenges. On the other hand, buffer management and classical eviction policies used to move data between memory and disk cannot capture the complex dynamics of a tiered storage design, since now data are not only *evicted* from memory, but also *migrated* between tiers, as shown in Figure 6·2. Overall, multi-tiered storage systems face the following challenges when supporting applications with page-level accesses:

Optimal Page Placement and Migration. Tiered systems use metadata of data objects to decide in which tier to store them and when to migrate them. The algorithms that are used to calculate optimal data placement for entire files are not applicable to single pages. For example, unlike files, all pages have the same size. Crucially, the number of pages is much higher than the number of files, making complex ML-based

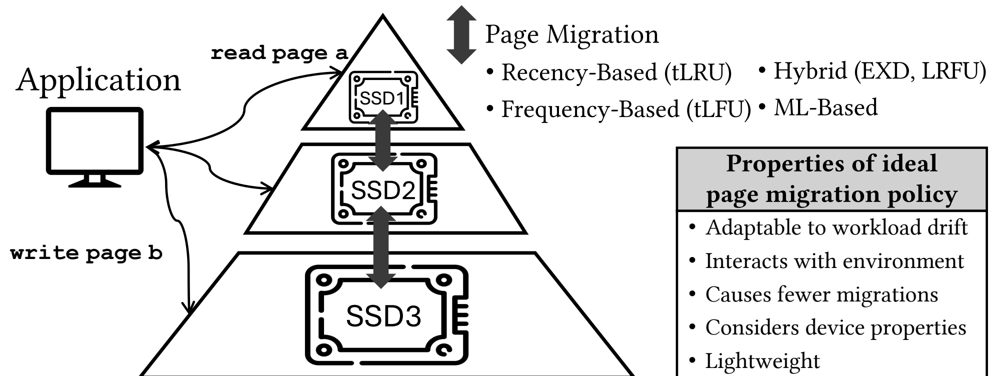


Figure 6·2: High-level overview of a page-level multi-tiered storage system consisting only of SSDs.

algorithms hard to use. On the other hand, standard caching algorithms, like LRU and LFU, can be adapted to perform data migration (page promotion and demotion), however, they will only use simple heuristics for their decisions, and they will typically struggle under workload drift.

Storage Device Properties. To the best of our knowledge, no prior data migration policy takes fully into account the storage device properties. In addition to access latency, capacity, and cost, modern storage devices (SSDs) typically exhibit a degree of read/write asymmetry and access parallelism. The devices can remain significantly underutilized without considering these properties, resulting in subpar performance. Existing migration policies do not faithfully model or exploit SSD characteristics.

Lightweight Migration Decisions. Following the first two challenges, we now see page-level migration policies for multi-tiered storage systems must strike a fine balance between being, on the one hand, complex enough to be able to capture workload access properties and drift as well as device properties, and, on the other hand, lightweight enough to be able to make decision at the page granularity, and as a result, at a much higher frequency.

Using Reinforcement Learning for Page Migration. While complex ML-based

approaches often come with a heavy overhead, we are dealing with a problem that can significantly benefit from learning based on the real-time execution and the state of the underlying system. To this end, we propose the usage of reinforcement learning (RL) to monitor the system's state and learn which migrations optimize performance. RL stands out as a better fit in this context because it can model the multi-tiered storage system with enough details using state variables to capture both workload and device characteristics. Furthermore, it can be designed to evaluate the reward of state transitions using lightweight value functions, balancing complexity and efficiency. We next discuss our RL-based design in detail starting with some introduction to RL and the notations we will use.

6.4 ReStore: RL-based Page Migration

Reinforcement Learning is particularly suited for dynamic environments like multi-tiered storage, where the optimal data migration strategy is often non-trivial due to changing workloads and storage constraints. RL can adapt to these variations by learning from the system's feedback, which allows it to make more informed and effective migration decisions over time. Moreover, ReStore is designed to be lightweight by using tailored value functions that minimize complexity while capturing key system properties like access patterns and device characteristics, enabling efficient decision-making with minimal overhead.

6.4.1 A quick primer on Reinforcement Learning

Reinforcement Learning is a widely used approach focused on decision-making through statistical estimations and environmental variables [180]. Its primary goal is to develop intelligent agents capable of learning from environmental interactions to make optimal decisions, adapt to new circumstances, and efficiently achieve defined objectives. RL accomplishes these objectives by solving the Markov Decision Process

(MDP), which is an environment formed by $< S, A, P, R, \gamma >$. Here S is states, A is actions, P is transition probabilities, R is rewards and γ is the discount factor. RL solves MDP by learning the best policy ($\pi : S \rightarrow A$) that prescribes the most appropriate action based on the current state. This determination of the best action derives from optimizing both the *state-value function* and *action-value function* under the chosen policy and action. The executing entity involved in these processes is commonly referred to as an agent. The agent continuously updates its *value functions*, learning from the system transitions from one state to another, enabling it to perform the best action based on the current state and its refined parameters.

RL for Tiered Storage. In the context of page management in multi-tiered storage systems, RL can be effectively utilized by defining the MDP components as follows: states (S) represent the current status of the storage tiers, actions (A) involve decisions on migrating pages between different tiers, transition probabilities (P) capture the likelihood of moving from one state to another based on the actions, and rewards (R) are determined by the performance metrics such as reduced access latency or improved I/O efficiency. By applying RL to this MDP framework, the system dynamically optimizes page placement, ensuring important pages reside in faster tiers while less important data is placed in slower, less costly storage, thus enhancing overall performance and cost-efficiency.

Basics and Nomenclature. RL trains intelligent agents to take *actions* in the MDP to achieve defined objectives. Formally, the MDP $< S, A, P, R, \gamma >$ is defined with S as the set of states with the Markov property (i.e., $\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_t, ..., S_1)$, t signifies time), A as the set of all actions, P as the state transition probability matrix with $P^a_{ss'} = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a)$, R is the reward function with $R^a_s = \mathbb{E}(R_{t+1}|S_t = s, A_t = a)$, and γ is the discount factor used in defining the return G_t, which is the total discounted reward from time t: $G_t = R_{t+1} + \gamma R_{t+2} + ... =$

Table 6.3: Our notation for modeling tiered storage with RL.

Term	Definition
t	Timestep
S	Set of states of an MDP
s	State
S_t	State at time t
A	Set of actions of an MDP
A_t	Action at time t
P	State transition probability matrix of an MDP
$P^a_{ss'}$	Transition probability from state s to s' taking action a
R	Set of rewards of an MDP
R_t	Reward at time t
R^a_s	Future reward from state s taking action a
G_t	Return from time t
γ	Discount factor of an MDP
π	Policy, to determine action based on state
$v(s)$	State-value function
$q(s,a)$	Action-value function
C^i_j	Fuzzy categories of FRB function
$\mu^i_{C_j}(x_j)$	Membership function of category C^i_j
a_j, b_j	Hyperparameters of membership function
p^i	Trainable parameters of FRB function
λ	Trace decay parameter of TD(λ)
α^i_n	Learning rate of TD(λ) updates
$z_n(s)$	Eligibility trace

$\sum_{i=0}^{\infty} \gamma^i R_{t+1+i}$. A policy $\pi : S \to A$ is used to determine the next action based on the current state. A policy $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$ is a distribution over actions given states, which defines the behavior of an agent. Therefore, the target of training an agent is to learn the best policy so that the agent takes correct action at each timestep, eventually attaining the objective. Table 6.3 shows in detail the nomenclature.

Value Functions. In RL, the objective is formulated as the return G_t, thus the training target of agent becomes to maximize the return at each timestep. To math-

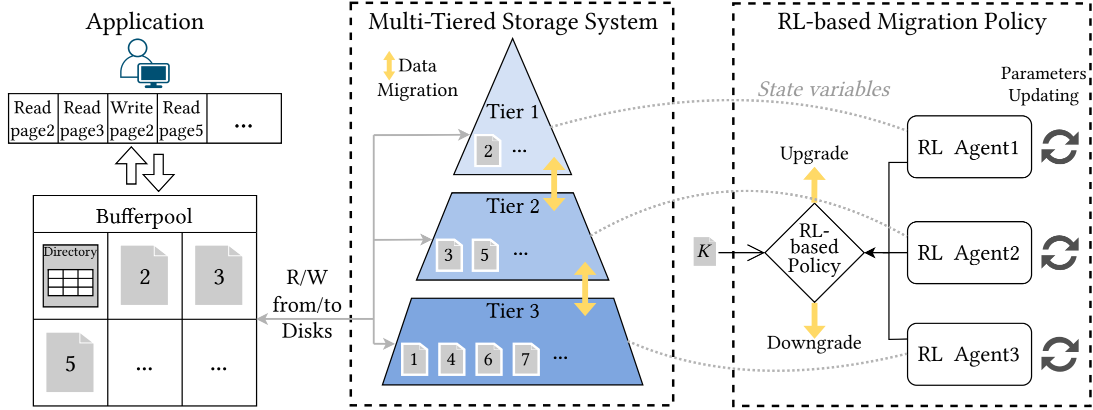


Figure 6·3: ReStore: A reinforcement learning based policy for data migrations in multi-tiered storage systems.

ematically express this target, state-value function $v_\pi(s)$ and action-value function $q_\pi(s,a)$ are defined, where $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$ represents the expected return starting from state s then following policy π, and $q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a)$ is the expected return starting from state s, taking action a, and then following policy π. The optimal value function implies the best possible performance in the MDP [171], and an optimal policy can be found by maximizing over the optimal action-value function $q_*(s,a)$, that is, $\pi_*(a|s) = 1$ if $a = argmax_{a \in A} q_*(s,a)$.

6.4.2 RL-based Tiered Storage Management

We now formulate the multi-tiered storage data migration problem with an RL-based framework. From a bird's eye view, we create an agent for every tier which manages data migration between neighboring tiers as shown in Figure 6·3. We configure the MDP environment as follows:

- states: variables representing the current status of a tier;
- actions: decisions on which page transferred to where;
- rewards: cost signal defined as the negative of response time.

These components allow us to define an MDP for multi-tiered storage and solve it to find the best migration policy using RL.

States. Defining the states requires careful consideration to encompass all significant properties and represent the status of a tier accordingly. As we earlier discussed, the essential features related to the data migration process include the page access information and the device status, which can be described by the access recency, frequency and R/W asymmetry and concurrency respectively. Therefore, we define the following state variables:

s_1: the average temperature of all pages in the tier. It is calculated using the formula $s_1 = \frac{1}{|K|}\sum_K temp(K)$, where $|K|$ is the total number of pages stored in the tier and $temp(K)$ is the temperature of page K given by the temperature model presented in Section 6.4.2, representing a combined information about the access recency and frequency. This variable indicates the percentage of hot/cold pages in the tier. A higher value of s_1 indicates a larger probability of pages in the tier to be requested and, at the same time, a larger expected number of accesses.

s_2: the current accumulated latency level in the tier. It is calculated as: $s_2 = number_of_queued_tasks \cdot avg_process_time$, where $avg_process_time = \frac{l_r}{k_r} + \frac{\alpha \cdot l_r}{k_w}$, l_r is the latency of read, α is the read/write asymmetry, and $k_r(k_w)$ is the read (write) concurrency. A higher value of s_2 denotes a longer waiting time in the tier, i.e., higher latency.

These state variables are used to represent the status of each tier at each timestep. They include file temperature, file access frequencies, and concurrency potential, which are metrics that have a strong impact in storage systems. Here, it is worth mentioning that this is a scalable framework, which implies the state variables can also be defined in other ways depending on use cases [213, 214].

Actions. The actions model the page movements between tiers. Since each page movement will change the distribution of pages in tiers, we define the actions upon each page being requested. In more detail, for a request of page K residing in Tier i, there are two possible actions: $A_t = 0$ means no movement of the page, $A_t = 1$ stands for upgrading the page to a faster tier j. In this definition, $A_t \in \{0, 1\}$ and $A = \{K, A_t\}$ where K is the page number.

Rewards. Reward $R_s^a = \mathbb{E}(R_{t+1}|S_t = s, A_t = a)$ is a function of state and action, representing the expectation of reward at the next timestep $t + 1$. In our case, since the global objective is to minimize the overall response time, we define the reward as the negative of the estimated future system response time, so that the return $G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i}$ represents the negative overall future system latency, and training RL agent to maximize the return is equivalent to minimize the system latency. More specifically,

$$\begin{aligned} R_s^a &= \mathbb{E}(R_{t+1}|S_t = s, A_t = a) \\ &= \begin{cases} \sum_{Tier} avg_IO_time \cdot \sum_{K \in Tier} \mathbb{E}(req_K), & \text{if } a = 1 \\ R_t, & \text{if } a = 0 \end{cases} \end{aligned} \tag{6.1}$$

where $\sum_{Tier}$ is the summation over all tiers, $avg_IO_time = w_r \cdot read_time + w_w \cdot write_time$ is the average I/O time of the tier, $\sum_{K \in Tier}$ is the summation over all pages in the tier, and $\mathbb{E}(req_K)$ is the expectation of future number of requests to the page. To approximate the future number of requests, we use the page temperature, i.e., $\mathbb{E}(req_K) \propto temp(K)$. By applying the law of large numbers and replacing average temperature of all pages in a tier $\frac{1}{|K|} \sum_K temp(K) = s_1$, the $\sum_{K \in Tier} \mathbb{E}(req_K)$ is then equal to $|K| \cdot exp(\frac{1+s_1}{2})$, where $|K|$ is the total number of pages in a tier and s_1 is the first state variable of that tier.

With the definitions of states, actions, and rewards, we formulate the data migra-

tion problem as an MDP. As mentioned earlier, RL solves MDPs by optimizing the value functions to provide the best action. We now discuss how the value functions are used in solving the MDP. The state-value function $v_\pi(s)$ of an MDP is the expected return starting from states s following the policy π, $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$. Since the reward is negative of response time, the expected return G_t represents the estimated future system response time. So, by maximizing the value function, we aim to find a policy that leads to the smallest possible future response time.

Value Function Representation

In simple RL problems, the value function can generally be represented using a straightforward table, such as a Q-table, where every state-action pair is explicitly stored and updated. However, in more complex environments with large or continuous state spaces, storing a table becomes impractical, and value functions can only be estimated. Given the continuity of our state variables and the high uncertainty of state space, we employ functional approximation techniques [93] to practically express the value function. There are various parameterized methods, such as linear approximation, polynomial fitting, and neural networks [75, 110, 189]. Our choice must accommodate the diverse scales of our state variables and input normalization.

In line with our goal of having a lightweight policy, we choose the Fuzzy Rule-Based function (FRB) [18] for the value function as it uses simple rules and linear approximation. The FRB function maps f from input vector $x \in \mathbb{R}^j$ to a scalar output y using a set of rules. A common form of a fuzzy rule is as follows:

$$\text{Rule } i\text{: IF } x_1 \subset C_1^i, x_2 \subset C_2^i, ..., x_j \subset C_j^i \text{ THEN } p^i,$$

where $x_1, ..., x_j$ are the components of x, $C_1^i, ..., C_j^i$ are fuzzy categories, and p^i is the output parameter of this rule. The output of the rule-based function $f(x)$ is then a

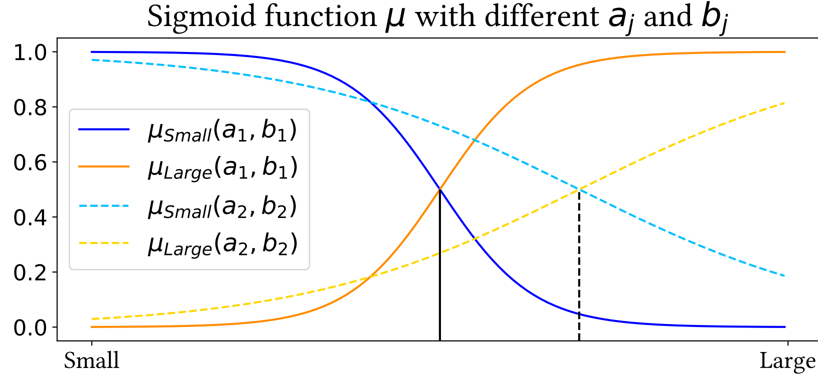


Figure 6·4: Membership function representing the categories 'Large' and 'Small', the value of the function stands for how likely a input is 'Large' or 'Small'.

weighted average of p^i:

$$y = f(x) = \frac{\sum_{i=1}^{N} p^i w^i(x)}{\sum_{i=1}^{N} w^i(x)} \tag{6.2}$$

where N is the number of rules, and $w^i(x)$ is the weight of rule i computed by $w^i(x) = \prod_1^j \mu_{C_j^i}(x_j)$. $\mu_{C_j^i}(x_j)$ is the membership function, which takes values in $[0, 1]$ and represent the measurement of an input variable x_j belonging to category C_j^i, i.e., a value closer to 1 means x_j highly probably belongs to C_j^i.

The FRB function is widely used for practical problems [38, 111]. It can model systems with different complexity, having anything between a large number of complex rules and a few simple rules. In our case, the input x is the set of state variables (s_1, s_2). The properties of these variables can be described by two categories, *Small* and *Large* of their values, i.e., $C_j^i \in \{Small, Large\}$. As for the membership function, we choose a Sigmoid function $\mu_{Large}(x_j) = 1/(1+a_j e^{-b_j x_j})$, $\mu_{Small}(x_j) = 1-\mu_{Large}(x_j)$. We set $b_j = 10/range(x_j), a_j = e^{b_j \bar{x}_j}$, where $range(x_j) = (\max x_j - \min x_j)$, and $\bar{x}_j$ is the expected value of x_j. With these parameters, we have $\mu_{Large}(\bar{x}_j) = \mu_{Small}(\bar{x}_j) = 0.5$. Figure 6·4 visualizes the two membership function pairs with different a_j, b_j.

This function is well-suited for scenarios where input ranges lack constraints, and differences between outliers with very large values or very small values are considered

less important than differences between centered data. The rationale behind selecting this membership function lies in its ability to render the policy sensitive to minor system changes, such as moving a few pages, but at the same time relatively robust against extensive system-wide modifications occurring at scale.

Based on the two categories and two state variables defined above, we formed a FRB function with four rules as our value function. Note that the number of rules equals to the number of categories to the power of the number of state variables, $2^2 = 4$ in our case.

$$v(s) = \frac{\sum_{i=1}^{4} p^i w^i(s)}{\sum_{i=1}^{4} w^i(s)} \tag{6.3}$$

where $s = (s_1, s_2)$ is the states variables, and $w^i(s) = \mu_{Small/Large}(s_1) \cdot \mu_{Small/Large}(s_2)$ is the multiplication of membership functions values at rule i.

Mapping the Data Migration problem

Having established the state variables and value function, we proceed to introduce their utilization in the control of data migration. For each page being requested, there are three scenarios:

- First, the page becomes hotter so that it will be upgraded to a faster tier.
- Second, the page is not hot enough to be upgraded so that it will stay in its current tier.
- Third, the page becomes hotter, but there is not enough space for the new hotter page in the faster tier, so the coldest page in the faster tier will be downgraded to the slower tier in order to make room for the new hotter page.

Reflecting on the data migration policy, it can be simplified as making a decision on whether the page should be upgraded or not based on its property and the current state of the involved tiers. Accordingly, the action space A can be written as $\{0, 1\}$

where $A_t = 1$ means current requested page should be upgraded to a faster tier and $A_t = 0$ implies should not. In this way, the action-value function $q(s, a)$ has two values for each s, which are $q(s, 0) = \mathbb{E}(G_t|S_t = s, A_t = 0) = \mathbb{E}(G_{t+1} = G_t|S_{t+1} = s) = v(s)$ ($A_t = 0$ means no migration and thus $S_{t+1} = S_t = s$), and $q(s, 1) = \mathbb{E}(G_t|S_t = s, A_t = 1) = \mathbb{E}(G_{t+1}|S_{t+1} = s')$ where s' is the next state due to the page migration decided by $A_t = 1$.

As indicated in Section 6.4.1, an optimal policy can be defined by maximizing over the optimal action-value function $q_*(s, a)$, where $q_*(s, a) = \max_{\pi} q_\pi(s, a)$. Therefore, to find the optimal policy is equivalent to maximize the action-value function. Combining the expressions of action-value function in the last paragraph, the RL-based data migration policy is formed in the following way:

A requested page K currently in tier i is upgraded from tier i to $j = i + 1$ if

$$v^i_{\text{up}} + v^j_{\text{up}} \geqslant v^i_{\text{not}} + v^j_{\text{not}} \tag{6.4}$$

where $v^{i/j}_{\text{up}}$ is the value function of tier i/j after page K is upgraded, and $v^{i/j}_{\text{not}}$ is before upgrade. The average temperature of the tier s_1 can be considered as an estimation of the number of requests for pages in the tier, while the value function $v(s)$ stands for the future cost. Hence, the sum of their multiplication represents an average weighted cost signal. In this way, Eq. (6.4) denotes that the workload latency is reduced, i.e., the overall system efficiency is increased.

Updating Value Function

The above-introduced policy relies on Eq. (6.4) that depends on value functions. Consequently, for the policy to consistently make optimal decisions across various states, continuous updates to the value functions are imperative during the system's transition from one state to another. In RL, there are multiple updating algorithms,

ranging from Monte-Carlo, Q-learning [157] to SARSA [157], and TD learning [179]. However, our application is a special case because, firstly, we use the same policy for both decision-making and future reward prediction. Secondly, since the action at each state is either to upgrade a page or not, the action-value function and the state-value function are equivalent. Therefore, we use an on-policy control approach to optimize our value functions, which is Temporal Difference (TD(λ)) learning.

TD(λ) is a well-known procedure for iteratively approximating the value function $v(s)$ under a given policy as follows:

$$\begin{aligned} \hat{v}(s) &= \hat{v}(s) + \alpha_n(R_n + \gamma\hat{v}(s_{n+1}) - \hat{v}(s_n))z_n(s) \\ \gamma &= e^{-\beta\tau_n}, z_n(s) = \lambda e^{-\beta\tau_n} z_{n-1}(s) + \mathbb{1}(s = s_n) \end{aligned} \tag{6.5}$$

where $\hat{v}(s)$ is the approximation of $v(s)$, α_n is the learning rate at the nth state, R_n is the rewards at state s_n, γ is the discounting factor (depending on the hyperparameter β), τ_n is the time the agent spends in state s_n, and z_n is the eligibility trace, which is initialized to 0 and updated by the formula above, and λ is the trace-decay parameter of TD(λ).

Recalling Eq. (6.3), the cost function is actually a linear combination of basis functions with parameters p^i. Thus, we can rewrite the $v(s)$ in the form $v(s,p) = \sum_{i=1}^{4} p^i\phi^i(s)$, where $\phi^i(s) = \frac{w^i(s)}{\sum_{i=1}^{4} w^i(s)}$. Therefore, the updating of $v(s)$ in Eq. (6.5) using TD(λ) equals to updating the parameters p^i. The updating formula can be rewritten for p^i in the following form:

$$
\begin{aligned}
p_{n+1}^i &= p_n^i + \alpha_n^i (R_n + \gamma \hat{v}(s_{n+1}) - \hat{v}(s_n)) z_n^i(s) \\
\alpha_n^i &= 0.1/(1 + 100 \sum_{t=0}^{n-1} \phi^i(s_t)) \\
R_n &= (1/X_n) \sum_{i=1}^{X_n} r_i e^{-\beta(t_{n,i} - t_n)} \\
z_n^i(s) &= \lambda \gamma z_{n-1}^i(s) + \phi^i(s_n)
\end{aligned}
\tag{6.6}
$$

where the learning rate α_n^i is initialized as 0.1 and decreases according to the accumulated basis function ϕ. The eligibility trace z_n^i is initialized as 0 and updated by Eq. (6.6). The reward R_n is given by the cost signal, where X_n is the total number of requests in state s_n, r_i is the response time of each request, $t_{n,i}$ is the arrival time of request i, and t_n is the time arrived at state s_n. This iteration process is proved to converge for MDPs when the basis functions $\phi^i(s)$ are linearly independent [189].

Temperature Model

In order to capture the information of frequency and recency from the workload access pattern, we define a *temperature* model for each page to represent its hotness level. It plays an essential role in both the definition of state variables and the action decisions given by Eq. 6.4. Specifically, we define a *hot-cold* function to represent these temperature changes. The *hot-cold* function includes the temperature-increasing process and the temperature-decreasing process, which are as follows:

- $Temperature\ increasing$: Page temperature will increase when there are more access requests to this page. We describe this process using an exponential formula: $temperature = 1 - \frac{0.5}{exp(0.2 \cdot req)}$, where req is the total number of requests to the page in a certain time window.

- *Temperature decreasing* : When a page has not been requested for a period, its temperature should drop because the importance and priority of this page have become lower. Therefore, we define the temperature decreasing rule to be: *After* 5 *timesteps without requests, page temperature will decrease* 0.05*, until it becomes* 0.0 *(the lowest temperature)*. As long as new requests come to a page, its temperature will start increasing, following the temperature-increasing rule above.

With this mechanism, we define the page temperature changing dynamics. Figure 6·5 shows an example of a temperature changing process of a page with 100 random requests in 1000 timesteps.

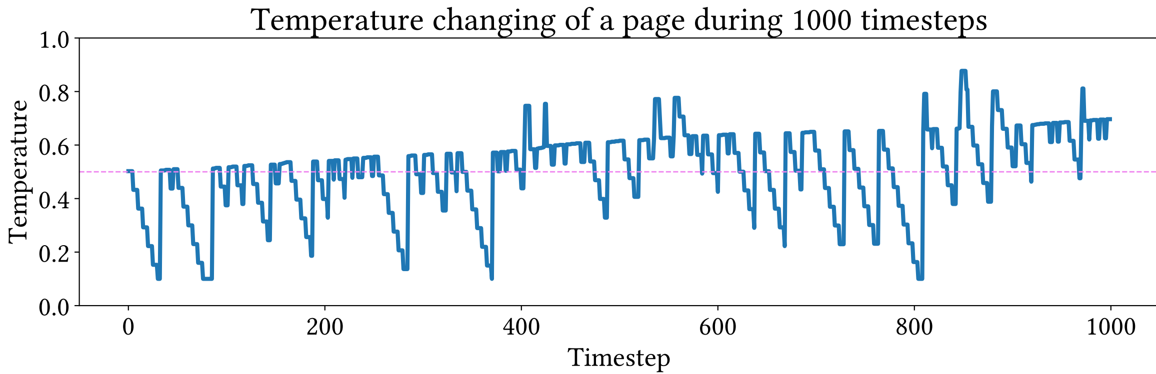


Figure 6·5: Example of our temparature model.

6.5 Emulating Multi-Tiered Storage

We develop a framework for a multi-tier storage system using C++ that simulates the tiers, storage devices and data migration policies. We implement five baseline data migration policies: tLFU (tiered variant of LFU), tLRU (tiered variant of LRU), EXD [50], LRFU [119], and TEMP (a migration policy based on the *temperature* model described in Section 6.4.2). We now discuss the architecture of our framework as presented in Figure 6·6.

Emulating Tiers. The tiers of the tiered storage system is represented by a class

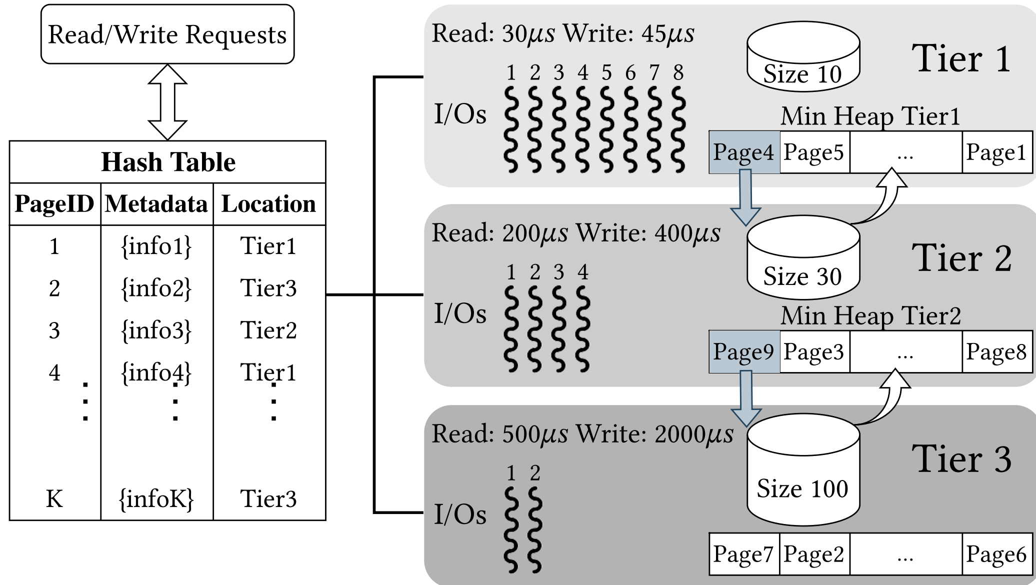


Figure 6·6: Architecture of our multi-tiered storage system.

`Tier`, which contains the information about the tier's maximum capacity, read/write speed, and concurrency. A hash table of `<page_id, page_metadata, location>` records the page distribution across tiers, and the `page_metadata` contains the features that each policy relies on. For example, the `page_metadata` for LFU stores access frequency, for LRU it stores recency (latest timestamp the page was requested), and for EXD/LRFU it includes both the EXD score/CRF value and the recency used to calculated them. In TEMP and ReStore, `page_metadata` includes the recency, frequency, and the *temperature*. After each operation, the `page_metadata` of the requested page is updated accordingly.

Emulating Page Migration. Upon each page access, each policy use its upgrade criteria to determine whether the page should be migrated to a faster tier. When the upper tier reaches it capacity limitation, the downgrade strategy of the policy identifies which page to downgrade to a slower tier to make space for the new page. Once a page is migrated, the corresponding entry `<page_id, location>` is updated. We now provide an overview of the upgrade and downgrade mechanisms:

tLFU **Upgrade:** if frequency of page K is larger than the lowest frequency in upper tier, then upgrade page K

Downgrade: when a tier is full, downgrade the LFU page.

tLRU **Upgrade:** if recency of page K is higher than the least recency in upper tier, then upgrade page K

Downgrade: when a tier is full, downgrade the LRU page.

LRFU **Upgrade:** if CRF value of page K is larger than the lowest CRF value in upper tier, then upgrade page K

Downgrade: when a tier is full, downgrade the LRFU page (with lowest CRF value).

EXD **Upgrade:** if EXD score of page K is larger than the lowest EXD score in upper tier, then upgrade page K

Downgrade: when a tier is full, downgrade the lowest EXD score page.

TEMP **Upgrade:** if temperature of page K is higher than the average temperature of upper tier, then upgrade page K

Downgrade: when a tier is full, downgrade the lowest temperature page.

ReStore **Upgrade:** if the RL-based criteria (condition (6.4) in Section 6.4.2) is satisfied, then upgrade page K

Downgrade: when a tier is full, downgrade the lowest temperature page.

A min-heap is maintained in the `Tier` class of each tier to identify pages for downgrading. The min-heap is based on the specific feature of each policy. For example, for LFU and LRU, a min-heap of frequency or recency values is maintained to guide page migration decisions. LRFU uses a CRF (Combined Recency and Frequency) value, defined as $\mathcal{C} = \sum_{i=1}^{k} \mathcal{F}(t - t_i)$, where $\mathcal{F}(x)$ is a weighting function, typically

$2^{-\lambda x}$, and $t - t_i$ represents the time gap since the last ith access. EXD balances recency and frequency using an exponential decay score S, updated on each access: $S = 1 + S \cdot \exp(-\alpha \cdot \Delta t)$, where $\Delta t = time_now - time_last_access$ is the time since the last access, and the parameter α governs the relative influence of frequency versus recency. TEMP operates as a function-based policy, focusing only on the *temperature* model discussed in Section 6.4.2. This policy is included to verify the efficacy of the *temperature* model in balancing recency and frequency and serves as a baseline to demonstrate that the effectiveness of ReStore comes not only from the *temperature* model but also from faithfully modeling storage devices with RL.

Emulating Storage Devices. A C++ thread pool simulates the concurrent read/write of each tier [168] the number of threads set as the number of concurrency. For example, for a device that throughput scales perfectly for four concurrent requests, the threadpool will use four threads to emulate each request. Note that our system has enough parallelism to emulate the concurrency of several devices at the same time. The asymmetry of the device is captured via the read/write latency values of each tier (device). While running a workload, a read (write) action is executed by detaching a task `std::this_thread::sleep_for(read_time (write_time))` to the corresponding `thread_pool`.

6.6 Experimental Evaluation

In this section, we evaluate the performance of ReStore and other baseline policies under different scenarios including varying workloads, tier capacities, and system conditions. The experimental results demonstrate the advantages of our RL-based migration strategies over traditional methods, with a particular focus on reduced migration overhead, adaptability and scalability.

6.6.1 Experimental Methodology

We design a comprehensive set of experiments with both classical and advanced data migration policies to thoroughly evaluate the effectiveness of ReStore. The performance of each policy is evaluated in experiments across both synthetic workloads and workloads adapted from real traces, with different characteristics, including access skew, read/write imbalance, and changing patterns. We also test on different device properties, such as different capacity, access latency, concurrency, and asymmetry.

Static Workloads. We test three static workloads with varying skew: *5hf90*, *10hf90*, and *10hf80*. The notation *xhfy* represents that $y\%$ of the accesses are performed on $x\%$ high-frequent (*hf*) pages. *5hf90* and *10hf90* differ in access concentration, while *10hf80* has lower skew. To isolate the effects of access skew, different high-frequency pages are used for *5hf90* and *10hf90*, but the same pages are used for *10hf90* and *10hf80*. Unless otherwise mentioned, these workloads have 50% reads and 50% writes, and they consist of 1,000,000 operations and 10,000 unique pages. We also test system scalability with increased page counts and number of operations.

Dynamic Workloads. To evaluate the performance of different policies under workload drift, which is common in real-world systems, we introduce four dynamic workloads: Workload A (*5hf90 + 10hf80*), Workload B (*5hf90+10hf80+20hf80*), Workload C (*5hf90*10*), and Workload D (*5hf90*50*). These workloads simulate shifting access patterns by combining multiple phases. For example, Workload A consists of a first phase with the *5hf90* pattern, followed by a second phase with the *10hf80* pattern. Further, *5hf90*n* refers to a workload with n phases, where each phase has a *5hf90* skew on a different 5% of the pages for each phase. All synthetic workloads are summarized in Table 6.4.

Tier Properties. We consider a three-tier storage system for all experiments. To assess the effect of tier capacity, we experiment with various configurations: Tier

Table 6.4: Our synthetic workloads.

Name	**Type**	**Description**
5hf90	static	Skewed 5% access on 90% pages
10hf90	static	Skewed 10% access on 90% pages
10hf80	static	Skewed 10% access on 80% pages
A	dynamic	*5hf90 + 10hf80*
B	dynamic	*5hf90 + 10hf80 + 20hf80*
C	dynamic	10 phases of *5hf90*
D	dynamic	50 phases of *5hf90*

Table 6.5: Standard benchmarks and real-world traces used.

Trace Name	**#unique_pages**	**#operations**
TPC-C	16842331	62925641
TPC-E	19274163	31884418
YCSB	61960292	92728323
Google Thesios	16417	95970071

1 with 1%/3%/6%/12% capacity, Tier 2 with 4%/8%/16%, and Tier 3 with 100% capacity. Unless otherwise specified, we simulate the fast, medium, and slow tiers as follows: Tier 1 has a read latency of $30\mu s$ and a read/write asymmetry of 1.5, giving it a write latency of $45\mu s$. Tier 2 is configured with a read latency of $200\mu s$ and an asymmetry of 2.0, while Tier 3 has a read latency of $500\mu s$ and an asymmetry of 4.0. These values reflect real-world SSD performance as outlined in Table 6.2. We also experiment with varying levels of concurrency and asymmetry, as well as without asymmetry (to simulate HDDs) for completeness.

Real-world Workloads. To further evaluate the robustness of each policy, we use real-world I/O traces from well-known benchmarks such as TPC-C [187], TPC-E [188], YCSB [205], and Thesios [143]. These traces are transformed into page-level

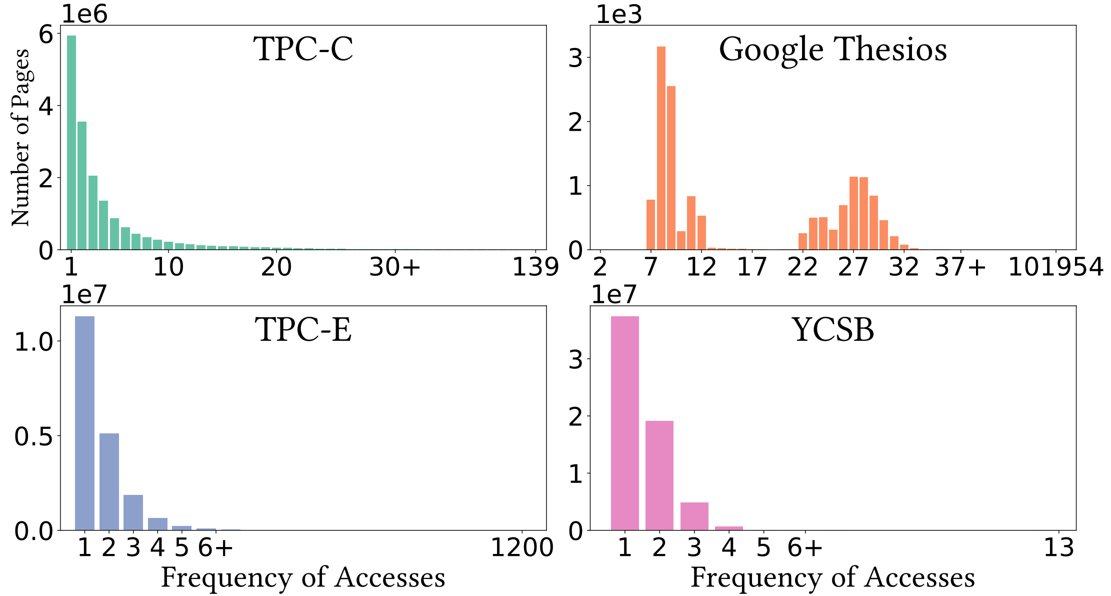


Figure 6·7: Distribution of page access frequencies in standard benchmarks and real-world workload traces.

workloads, assuming a 4KB page size. Each trace contains various access patterns, page sizes, and levels of skew. Table 6.5 provides a summary of page counts and the total number of operations, while Figure 6·7 visualizes the distribution of access frequencies in these traces.

The *Ideal Static* Policy. We further define a '*Static*' policy for each workload as the *optimal baseline* considering the workload pattern is already known. For example, the *Static* policy for *5hf90* places the *5hf* pages in fast tiers (all in Tier 1, or if Tier 1 is full then put the rest in Tier 2), and the rest 95% pages in slower tiers (Tier 2 and Tier 3). This *Static* policy acts as the best possible placement with full workload-knowledge without any page migration.

6.6.2 Experimental Analysis

ReStore achieves superior performance, both in terms of latency reduction and resource utilization. We now provide a detailed analysis of our experimental evaluation.

ReStore Outperforms All Baselines in Static Workloads. In our first set of ex-

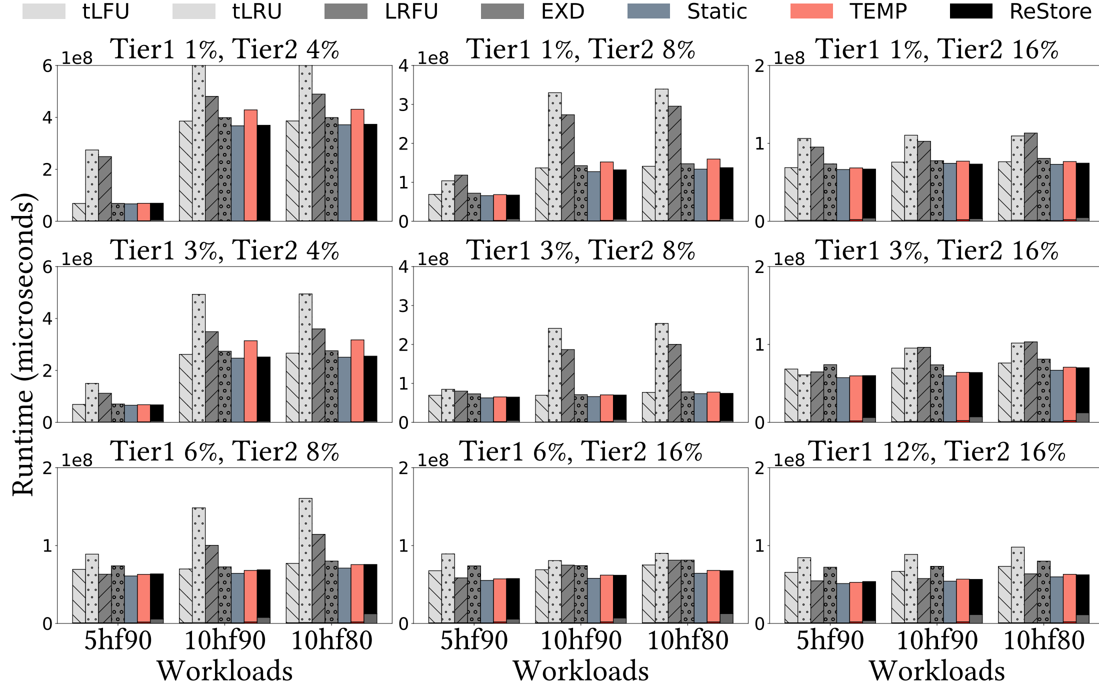


Figure 6·8: Overall runtime (CPU time + data access time) under different tier capacity configuration and three static workloads. In general, ReStore outperforms the other baseline policies. Frequency-based policy like tLFU performs well for this type of skewed workload. Recency-based policy like tLRU struggles to capture the long-term *hf* workload.

periments, we evaluate the performance of each policy for the static workloads (*5hf90*, *10hf90*, *10hf80*) under different tier capacity configurations. Figure 6·8 presents the overall latency of each workload under each policy as tier capacities change. We observe that overall latency decreases as the size of faster tier increases, i.e., the top row of Figure 6·8 (small fast tier) has the highest latency while the bottom row (large fast tier) has the lowest latency. The figure also shows that ReStore achieves the overall best performance. For example, ReStore achieves 5%, 40%, 23%, 7% and 13% lower runtime compared to tLFU, tLRU, LRFU, EXD and TEMP for *10hf90* workload with 1% Tier 1 and 4% Tier 2 capacity (middle group of bars in the top left sub-figure of Figure 6·8). ReStore demonstrates stable performance trends similar to the optimal *Static* policy as tier capacity increases for all three workloads.

The figure also shows that tLFU performs well under such skewed workloads with its reliance on frequency-based decisions allowing it to capture the long-term *hf* pattern in the workloads. We also observe that TEMP works particularly well when the faster tiers have higher capacity while EXD's performance does not depend on the tier capacity too much. This suggests that while the *temperature* model is highly efficient at identifying *hf* pages, it lacks resource optimization constraints, and is more sensitive to workload variations than the EXD score. We further notice that tLRU and LRFU has the worst performance in terms of runtime in these experiments. This is because recency-based policies incur a lot of data migrations in such skewed *hf* workloads which adversely affects the overall runtime. The latency bars of the figure also include CPU time (lower part of each bar) stacked with data access time. We notice that although ReStore has the highest CPU time (more details later) compared to other policies because of RL's high computation cost, *ReStore outperforms other baselines because of its optimal data placement decision based on both workload and device properties*. This in turn causes very few number of migrations compared to other policies which we now discuss.

ReStore Causes Fewer Migrations. Migration is a costly process involving both an upgrade and a downgrade operation, so minimizing the number of migrations is crucial. Figure 6·9 shows the number of page migrations for each policy from the previous experiment. The upper part of each bar represents migrations from Tier 3 to Tier 2, while the lower part shows migrations from Tier 2 to Tier 1. Note that the optimal *Static* policy causes no migrations, as pages are statically assigned to tiers. We observe that tLRU and LRFU cause the highest number of migrations, failing to capture the workload skew. The figure illustrates that ReStore results in significantly fewer migrations than other policies. Specifically, ReStore reduces migrations by up to 8×, 102×, 37×, 12× and 22× compared to tLFU, tLRU, LRFU, EXD, and TEMP,

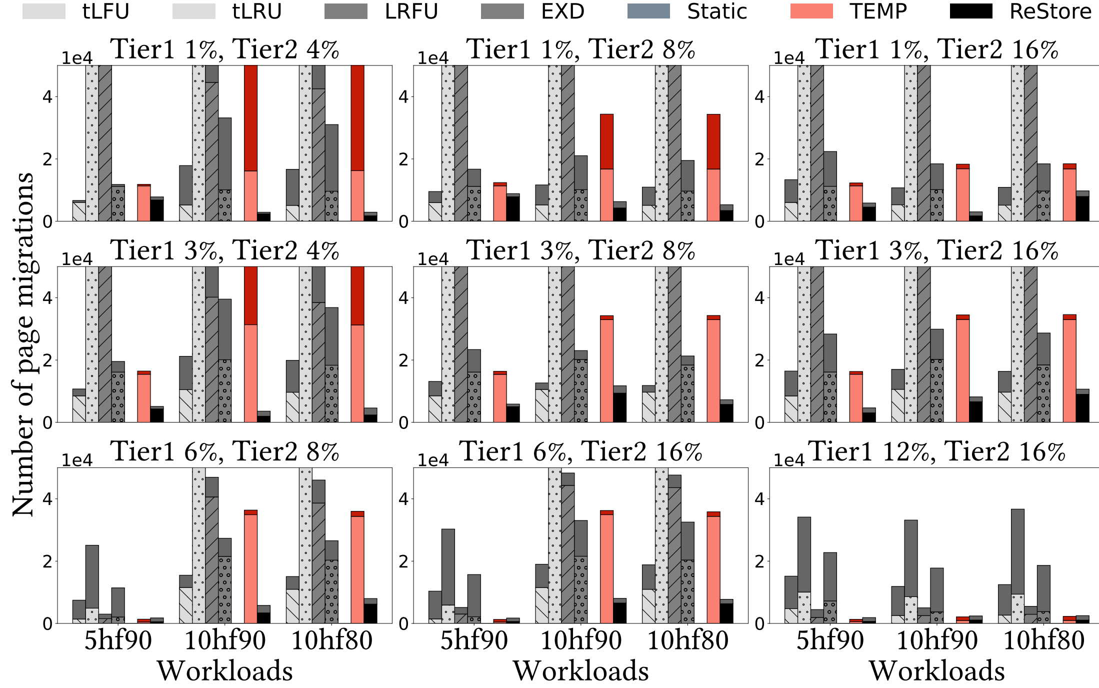


Figure 6·9: Number of page migrations, lighter colors are the number from Tier 2 to Tier 1, darker colors are from Tier 3 to Tier 2. ReStore causes the least number of migrations.

respectively. This demonstrates ReStore's effectiveness in minimizing migration overhead while optimizing data placement, striking a balance between performance and migration efficiency across all tiers.

ReStore can Adapt to Workload Drift. In this set of experiments, we evaluate ReStore and other policies under dynamic workloads using same tier configurations as in the previous experiments. Figure 6·10 presents the performance evaluation which shows that ReStore *significantly* outperforms other policies, particularly excelling in scenarios where access patterns change more frequently. For example, for 3% Tier 1 and 8% Tier 2 (center subfigure of Figure 6·10), ReStore achieves up to 2.81×, 1.03×, 3.51×, 4.28× speedup compared to LRFU (the next best rule-based policy) for workload A, B, C and D respectively. The performance trend remains similar for different tier capacities across different workload. We observe that ReStore has

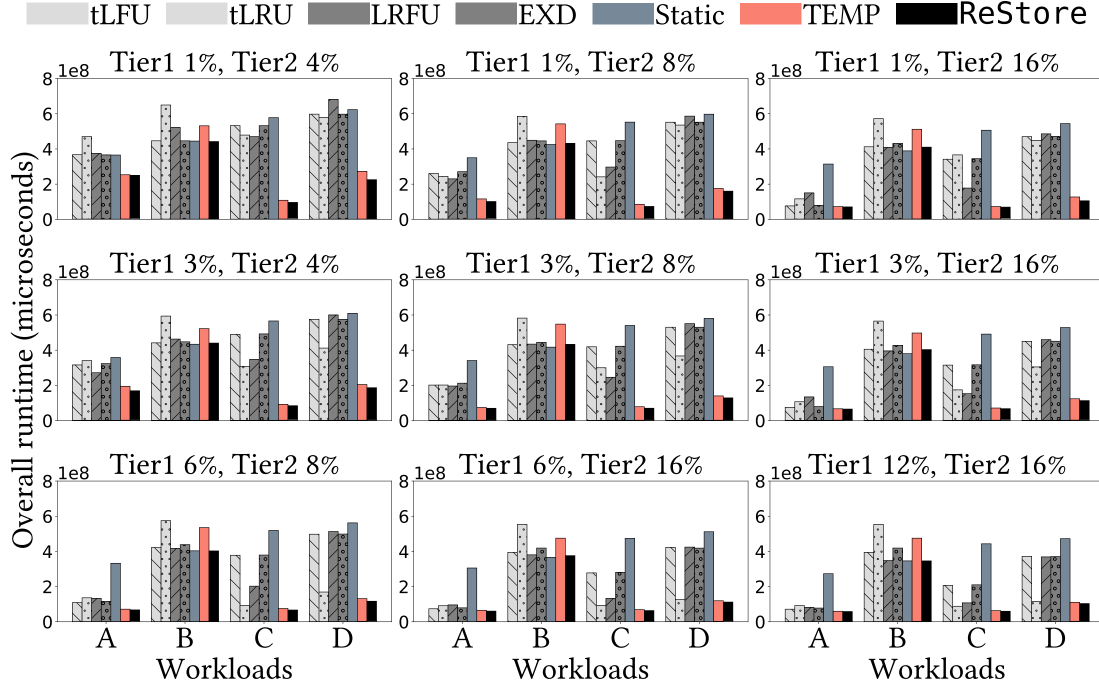


Figure 6·10: Total runtime under dynamic workloads. ReStore significantly outperforms other baselines with the greatest benefit in high drift scenarios (Workloads C & D).

the highest performance improvement for workload C and D which includes the highest amount of workload drift, showcasing ReStore's adaptibility. Overall, ReStore achieves up to 6.1×, 5.3×, 4.9×, 6.0×, and 1.4× speedup compared to tLFU, tLRU, LRFU, EXD, and TEMP, respectively. We further observe that recency-based policies, such as LRU, show a marked improvement compared to their performance on static workloads, reflecting their adaptability to short-term changes. On the other hand, LFU, which performs well on static workloads, struggles with workload drift but remains competitive in less volatile scenarios. TEMP and EXD exhibit higher robustness than LFU, with TEMP showing particular strength in more frequently changing workloads (workload C and D), validating the effectiveness of the *temperature* model. The Static policy struggles under workload drift, showing the importance of *effective* page migration.

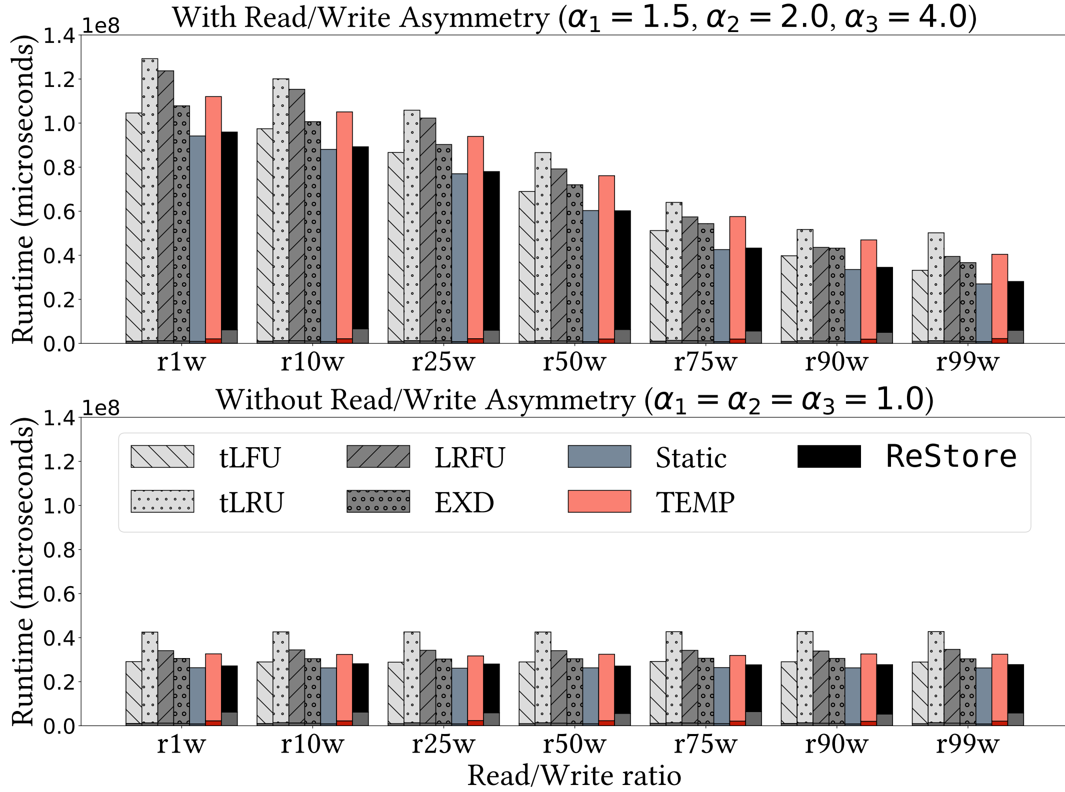


Figure 6·11: Performance under varying read/write ratio with asymmetry (top) and without asymmetry (bottom) for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore delivers stable performance.

ReStore is Robust for Different Read/Write Ratios. We now evaluate the robustness of ReStore and other policies as we vary the read/write ratio of the *5hf90* workload under both symmetric and asymmetric device configurations. Figure 6·11 presents the performance evaluation of different policies where the x-axis r1w-r99w, represent workloads with 1%-99% reads. The top figure shows the total runtime on devices with read/write asymmetry (e.g., SSDs), while the bottom shows performance on devices without asymmetry (e.g., HDDs). As expected, Figure 6·11 shows that in an asymmetric environment, latency increases with more writes. ReStore outperforms other baselines across all read/write ratios, demonstrating its stability. A careful look at both figures also show ReStore's resiliency to device asymmetry, maintaining stable performance across different read/write ratios, unlike other policies that exhibit

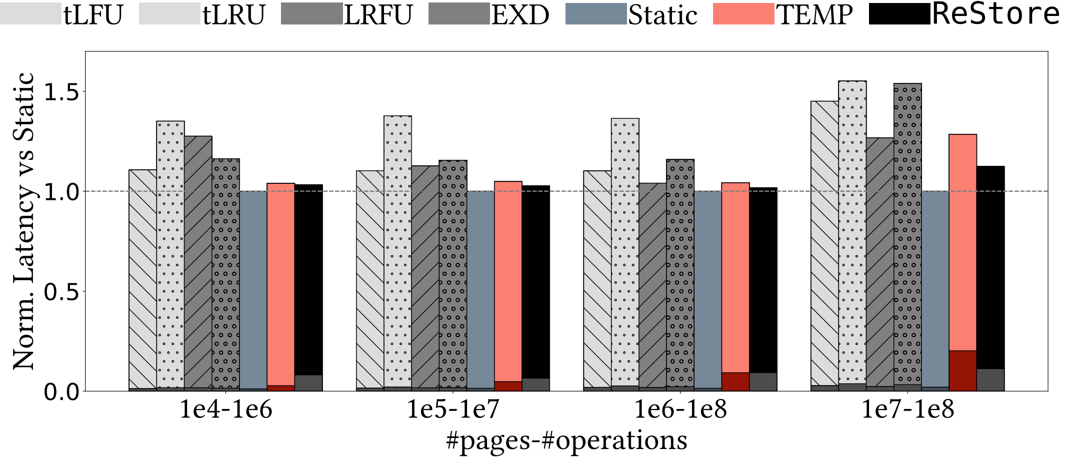


Figure 6·12: Normalized latency relative to *Static* of each policy as #pages and #operations increases for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore scales effectively with the data size.

slightly higher latency under asymmetric conditions. This robustness stems from ReStore's ability to account for read/write asymmetry in its decision-making, a crucial factor in heterogeneous storage environments.

ReStore Scales with Data Size. This set of experiments demonstrates the scalability of each policy as we increase the size of the *5hf90* workload by orders of magnitude. Specifically, we increase the number of pages from 10,000 to 10 million (a three-order magnitude increase) and the number of operations from 1 million to 100 million (a two-order magnitude increase). Figure 6·12 shows the normalized latency of each policy relative to the *Static* policy. We observe that ReStore scales effectively with data size. Specifically, its normalized latency increases slightly, from 1.03 to 1.12, as both the number of pages and operations grow significantly (first group of bars and fourth group of bars). In contrast, LFU experience normalized latency increase from 1.1 to 1.45, LRU from 1.35 to 1.55, EXD from 1.16 to 1.53, LRFU from 1.04 to 1.27, and TEMP from 1.04 to 1.28. These results highlight ReStore's ability to handle large-scale workloads while maintaining predictable performance.

ReStore Excels for Real-World Workloads. We evaluate ReStore and other

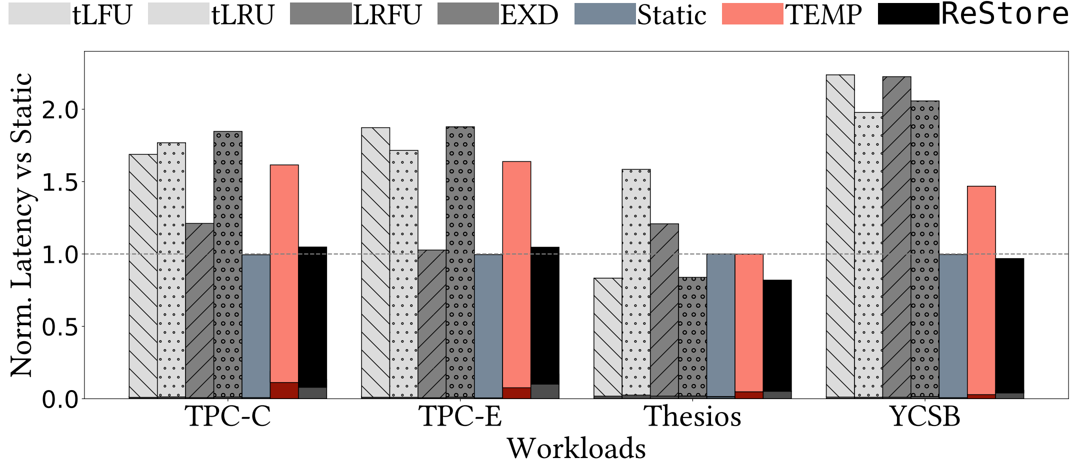


Figure 6·13: ReStore achieves the lowest normalized latency (relative to *Static*) under standard benchmarks and real-world traces, consistently outperforming all other baselines.

policies using standard benchmarks like TPC-C [187], TPC-E [188], YCSB [205], and Thesios [143], whose workload characteristics are detailed in Section 6.6.1. Given the varying number of pages in these benchmarks, we use different tier capacity settings: for TPC-C and TPC-E, the maximal capacity of Tier 1, Tier 2 and Tier 3 is 2M, 8M, and 20M pages; for Thesios, capacity of Tier 1, Tier 2 and Tier 3 is 1,000, 5,000, and 20,000 pages; for YCSB, Tier 1, Tier 2 and Tier 3 capacity is 10M, 20M, and 70M pages. Since the number of unique pages and number of operations differ, the runtimes vary significantly across workloads. Therefore, we present in Figure 6·13 the normalized latency of each policy relative to the *Static* policy. The figure shows that ReStore consistently outperforms other policies across all workloads, demonstrating its effectiveness for real-world scenarios. For example, ReStore achieves 2.3×, 2×, 2.2×, 2.1× and 1.5× lower runtime compared to tLFU, tLRU, LRFU, EXD, and TEMP for the YCSB workload, and similarly maintains superior performance across other workloads. We further observe that LFU performs well for Thesios, which closely resembles our static skewed workloads. While the performance differences between policies depend on specific trace properties, ReStore

Table 6.6: CPU time and memory overhead of each policy.

	Static	tLFU	tLRU	LRFU	EXD	TEMP	ReStore
CPU (μs)	0.892	1.081	1.187	1.272	1.196	1.762	3.625
Memory (MB)	1.01	1.19	1.12	1.43	1.40	1.67	1.73

maintains a clear advantage by dynamically adapting to diverse workload patterns.

CPU and Memory Overhead Analysis. While ReStore achieves significant performance improvements, it does so with an increased computational cost. Compared to the *Static* policy, LFU and LRU require additional memory to store the frequency or recency of each page, along with a heap to track the least frequently or least recently used page. Similarly, EXD uses a heap that records the EXD score. TEMP maintains a temperature model for each page based on the *hot-cold* function (discussed in Section 6.4.2), with a heap to quickly identify the coldest page. ReStore introduces a bit more overhead due to its three RL agents (one for each tier) each with parameters detailed in Section 6.4.2. Table 6.6 lists the average CPU time per access and memory usage for each policy when running the *5hf90* workload. We observe that ReStore has the highest CPU and memory usage overhead due to the computational complexity of its RL agents. However, as shown in Figures 6·8-6·13, the performance gains provided by ReStore far outweigh these additional CPU and memory costs, making the overhead negligible in light of the latency improvements.

6.7 Conclusions

Data migration has consistently been recognized as a fundamental challenge in multi-tiered storage systems. Existing methods either did not balance performance gain and migration cost or failed to consider device-specific characteristics. Leveraging the strengths of reinforcement learning in flexible state definitions and dynamically adjusting policies, we present ReStore, an efficient and effective data migration so-

lution for multi-tiered storage. Through carefully designed state variables, ReStore considers both workload patterns, such as access frequency and recency, and storage device properties, including read/write asymmetry and concurrency. The key benefit of our RL formulation is that the migration policy adapts to changes in the environment, including workload and device utilization. We demonstrate the superiority of our approach through experiments on various workloads with different sizes and access patterns, as well as different configurations. ReStore outperforms other baselines under skewed workloads, workload drift, and varying read/write ratio, while also scaling effectively, demonstrating its strong adaptivity and robustness.

Chapter 7

Concluding Remarks

The transition from traditional Hard Disk Drives (HDDs) to Solid-State Drives (SSDs) has redefined modern storage, offering significantly lower latency and higher throughput. However, existing data systems remain rooted in HDD-based assumptions, failing to harness SSD-specific properties such as read/write asymmetry and internal parallelism. This thesis addresses this gap by investigating SSD characteristics and integrating them into data system design, leading to optimizations that improve performance and device utilization.

Chapter 2 establishes the foundation by providing an in-depth analysis of SSD internals, explaining the origins of read/write asymmetry and internal concurrency. It also presents our benchmarking methodology to quantify these properties across five different SSD devices, highlighting their impact on system performance. Building on this, Chapter 3 introduces the Parametric I/O Model (PIO), which formalizes SSD behavior by incorporating α (asymmetry) and k (concurrency). This model enables better storage modeling and performance prediction. We demonstrate its benefits in terms of device utilization and provide five key guidelines for designing SSD-aware algorithms that optimize data placement and access patterns.

Chapter 4 explores SSD-aware data management at the DBMS bufferpool level. We propose ACE, an asymmetry- and concurrency-aware bufferpool manager that optimally batches writes and prefetches data in parallel to amortize high asymmetric write cost and exploit SSD concurrency. By integrating ACE into PostgreSQL with

multiple page replacement policies, we show that it reduces query runtimes by up to 32% while maintaining comparable write overhead.

Chapter 5 focuses on graph processing, presenting CAVE, the first out-of-core graph processing system designed to fully leverage SSD read parallelism. CAVE optimizes traversal operations by carefully scheduling concurrent I/Os, allowing it to execute multiple paths and process graph elements in parallel. Evaluations demonstrate that CAVE achieves up to an order-of-magnitude speedup over Mosaic and GridGraph and up to three orders of magnitude improvement over GraphChi. Chapter 6 extends SSD-aware system design to multi-tiered storage by introducing ReStore, a reinforcement learning-based migration policy. ReStore dynamically adapts to workload characteristics and device properties, significantly reducing migration overhead and improving system responsiveness. Experimental results show that it achieves up to 2.2× lower runtime and 10× fewer migrations compared to traditional policies.

Overall, this thesis demonstrates that a hardware-conscious approach to storage system design can unlock the full potential of SSDs. By carefully exploiting SSD properties, our proposed techniques enhance performance, improve resource utilization, and system performance. As SSD technology continues to evolve, current and future data systems and storage-intensive applications need to integrate such optimizations to ensure optimal performance and resource utilization.

One promising application of this approach is LSM-trees, a widely used data structure in modern databases. LSM-trees write immutable sorted runs to disk, triggering *compaction* when a level exceeds its threshold to redistribute data across the saturated level and the next one. Since multiple compactions run concurrently, each merging several sequential streams, a device-aware compaction scheduler leveraging PIO can optimize the number and selection of compactions to maximize SSD utilization. Another exciting direction is applying this approach to next-generation SSDs

like SmartSSD [138] and OpenSSD [20]. These devices incorporate programmable processing units, which can enable near-data processing techniques [135, 136, 154] to minimize unnecessary data movement. By offloading on-the-fly transformations to the storage device and leveraging properties like asymmetry and concurrency, end-to-end latency can be significantly reduced. Additionally, the software stack will need to be redesigned to take full advantage of near-storage computation, paving the way for more efficient query processing.

Bibliography

[1] Adaptive optimization for hpe 3par storeserv storage. https://www.hpe.com/psnow/doc/4aa4-0867enw. Accessed: 2024-10-14.

[2] Ibm integrated analytics system - tiered storage. https://www.ibm.com/docs/en/ias?topic=storage-tiered. Accessed: 2024-09-12.

[3] Umut A Acar, Arthur Charguéraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 67:1—-67:12, 2015.

[4] T M Tariq Adnan, Md. Saiful Islam, Tarikul Islam Papon, Shourav Nath, and Muhammad Abdullah Adnan. UACD: A Local Approach for Identifying the Most Influential Spreaders in Twitter in a Distributed Environment. *Soc. Netw. Anal. Min.*, 12(1):37, 2022.

[5] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[6] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 57–70, 2008.

[7] Rizik M. H. Al-Sayyed, Hussam N. Fakhouri, Ali Rodan, and Colin Pattinson. Polar particle swarm algorithm for solving cloud data migration optimization problem. *Mathematical Models and Methods in Applied Sciences*, 11:98, 2017.

[8] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *CoRR*, cond-mat/0, 2001.

[9] Waleed Ali, Sarina Sulaiman, and Norbahiah Ahmad. Performance improvement of least-recently-used policy in web proxy cache replacement using supervised machine learning. *International Journal of Advances in Soft Computing and Its Applications*, 6(1), 2014.

[10] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12(2/3):72–109, 1994.

[11] Malak Alshawabkeh, Alma Riska, Adnan Sahin, and Motasem Awwad. Automated storage tiering using markov chain correlation based clustering. In *2012 11th International Conference on Machine Learning and Applications*, volume 1, pages 392–397, 2012.

[12] Reddit Analysis. SSD to reach price parity by 2030 if current trend continue. https://www.reddit.com/r/DataHoarder/comments/17sljc1/as_requested_an_improved_chart_of_ssd_vs_hdd/.

[13] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. The Five minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2017.

[14] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. 27(4):3–11, March 2000.

[15] Manos Athanassoulis and Anastasia Ailamaki. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment*, 7(14):1881–1892, 2014.

[16] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path Processing using Solid State Storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 23–32, 2012.

[17] Manos Athanassoulis, Subhadeep Sarkar, Tarikul Islam Papon, Zichen Zhu, and Dimitris Staratzis. Building Deletion-Compliant Data Systems. *IEEE Data Engineering Bulletin*, 45(1):21–36, 2022.

[18] H.R. Berenji. Fuzzy q-learning: a new approach for fuzzy dynamic programming. In *Proceedings of 1994 IEEE 3rd International Fuzzy Systems Conference*, pages 486–491 vol.1, 1994.

[19] R Bishnoi, M Ebrahimi, F Oboril, and M B Tahoori. Improving Write Performance for STT-MRAM. *IEEE Transactions on Magnetics*, 52(8):1–11, 2016.

[20] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 359–373, 2019.

[21] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–12, 2015.

[22] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Efficient Algorithms with Asymmetric Read and Write Costs. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, pages 14:1–14:18, 2016.

[23] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks*, 30(1-7):107–117, 1998.

[24] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. Learn to Cache: Machine Learning for Network Edge Caching in the Big Data Era. *IEEE Wireless Communications*, 25(3):28–35, 2018.

[25] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, S A Wolf, A W Ghosh, J W Lu, S J Poon, M Stan, W H Butler, S Gupta, C K A Mewes, T Mewes, and P B Visscher. Advances and Future Prospects of Spin-Transfer Torque Random Access Memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010.

[26] Feng Chen, Binbing Hou, and Rubao Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)*, 12(3):13:1–13:39, 2016.

[27] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277, 2011.

[28] Jinchuan Chen, Yueguo Chen, Xiaoyong Du, Cuiping Li, Jiaheng Lu, Suyun Zhao, and Xuan Zhou. Big data challenge: a data management perspective. *Frontiers of computer Science*, 7:157–164, 2013.

[29] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):13:1—-13:39, 2018.

[30] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[31] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[32] Peng Cheng, Yutong Lu, Yunfei Du, Zhiguang Chen, and Yang Liu. Optimizing data placement on hierarchical storage architecture via machine learning. In

Network and Parallel Computing: 16th IFIP WG 10.3 International Conference, NPC 2019, Hohhot, China, August 23–24, 2019, Proceedings 16, pages 289–302. Springer, 2019.

[33] Youngdon Choi, Ickhyun Song, Mu Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon, Soehee Kim, Jaehwan Kim, Yong Jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, Hideki Horii, Jaewook Lee, Kisung Kim, Hansung Joo, Kwangjin Lee, Yeong Taek Lee, Jeihwan Yoo, and Gitae Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 46–48, 2012.

[34] Edward I Cohen, Gary M King, and James T Brady. Storage Hierarchies. *IBM Systems Journal*, 28(1):62–76, 1989.

[35] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[36] Michael Cornwell. Anatomy of a Solid-State Drive. *Communications of the ACM (CACM)*, 55(12):59–63, 2012.

[37] Kenneth M Curewitz, P Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 257–266, 1993.

[38] Robert Czabanski, Michal Jezewski, and Jacek Leski. *Introduction to Fuzzy Systems*, pages 23–43. Springer International Publishing, Cham, 2017.

[39] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP), Volume I: Architecture*, pages 56–63, 1993.

[40] John Rydning David Reinse, John Gantz. The Digitization of the World: From Edge to Core. *White Paper*, 2020.

[41] Dominic Davies-Tagg, Ashiq Anjum, Ali Zahir, Lu Liu, Muhammad Usman Yaseen, and Nick Antonopoulos. Data temperature informed streaming for optimising large-scale multi-tiered storage. *Big Data Mining and Analytics*, 7(2):371–398, 2024.

[42] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[43] I B M DB2. Configuring storage for performance. *https://www.ibm.com/docs/en/db2-for-zos/12?topic=performance-configuring-storage*, 2021.

[44] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proceedings of the VLDB Endowment*, 14(3):364–377, 2020.

[45] Ahmed Eldawy, Justin J Levandoski, and Per-Åke Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proceedings of the VLDB Endowment*, 7(11):931–942, 2014.

[46] Shimon Even and Guy Even. *Graph Algorithms (2nd ed.)*. Cambridge University Press, 2012.

[47] Mohd Zeeshan Farooqui, Mohd Shoaib, and Mohammad Zunnun Khan. A comprehensive survey of page replacement algorithms. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume*, 3, 2014.

[48] Blocks & Files. SSDs will crush hard drives in the enterprise, bearing down the full weight of Wright's Law. *https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/*, 2021.

[49] Fio. Flexible I/O tester. *https://fio.readthedocs.io/en/latest/*, 2021.

[50] Avrilia Floratou, Nimrod Megiddo, Navneet Potti, Fatma Özcan, Uday Kale, and Jan Schmitz-Hermes. Adaptive caching in big sql using the hdfs cache. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 321–333, New York, NY, USA, 2016. Association for Computing Machinery.

[51] John W C Fu and Janak H Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*, pages 54–63, 1991.

[52] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[54] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[55] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[56] Goetz Graefe. The five-minute rule twenty years later. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, page 6, 2007.

[57] Jim Gray and Goetz Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[58] Jim Gray and Franco Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time. *Tandem Computers - Technical Report*, 1986.

[59] Knuth Stener Grimsrud, James K Archibald, and Brent E Nelson. Multiple Prefetch Adaptive Disk Caching. *IEEE Trans. Knowl. Data Eng.*, 5(1):88–103, 1993.

[60] Yan Gu, Yihan Sun, and Guy E Blelloch. Algorithmic building blocks for asymmetric memories. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 112, pages 44:1—-44:15, 2018.

[61] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.

[62] Laura M Haas, Michael J Carey, Miron Livny, and Amit Shukla. Seeking the Truth About ad hoc Join Costs. *The VLDB Journal*, 6(3):241–256, 1997.

[63] Frank T Hady, Annie P Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[64] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 77–85, 2013.

[65] Herodotos Herodotou and Elena Kakoulli. Automating Distributed Tiered Storage Management in Cluster Computing. *Proceedings of the VLDB Endowment*, 13(1):43–56, 2019.

[66] IBM. Db2: Prefetching Data into The Bufferpool. *https://www.ibm.com/docs/en/db2/9.7?topic=management-prefetching-data-into-buffer-pool*, 2012.

[67] MySQL InnoDB. Configuring Buffer Pool Flushing. *https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-flushing.html*, 2021.

[68] Intel. Intel® Optane™ Technology. *https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html*, 2016.

[69] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A Boncz. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[70] Jon L. Jacobi. NVMe SSDs: Everything you need to know about this insanely fast storage. *https://www.pcworld.com/article/2899351/everything-you-need-to-know-about-nvme.html*, 2019.

[71] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[72] Song Jiang and Xiaodong Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA, 2002. Association for Computing Machinery.

[73] Ziyang Jiao, Janki Bhimani, and Bryan S Kim. Wear leveling in SSDs considered harmful. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, pages 72–78, 2022.

[74] Ziyang Jiao and Bryan S Kim. Generating realistic wear distributions for SSDs. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, pages 65–71, 2022.

[75] Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I Jordan. Provably efficient reinforcement learning with linear function approximation. In *Conference on learning theory*, pages 2137–2143. PMLR, 2020.

[76] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. KÙZU Graph Database Management System. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2023.

[77] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

[78] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Very Large Data Bases Conference*, 1994.

[79] Doug Joseph and Dirk Grunwald. Prefetching Using Markov Predictors. *IEEE Trans. Computers*, 48(2):121–133, 1999.

[80] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Trans. Consumer Electron.*, 54(3):1215–1223, 2008.

[81] Aarati Kakaraparthy, Jignesh M Patel, Kwanghyun Park, and Brian Kroth. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *Proceedings of the VLDB Endowment*, 13(4):519–532, 2019.

[82] Jeong-Uk Kang, Heeseung Jo, Jinsoo 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, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, pages 161–170, 2006.

[83] Mincheol Kang, Wonyoung Lee, and Soontae Kim. Subpage-Aware Solid State Drive for Improving Lifetime and Performance. *IEEE Trans. Computers*, 67(10):1492–1505, 2018.

[84] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 1091–1099, 2011.

[85] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access File System with DevFS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 241–256, 2018.

[86] Kamil Kedzierski, Miquel Moretó, Francisco J Cazorla, and Mateo Valero. Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[87] Sami Khuri and Hsiu-Chin Hsu. Visualizing the CPU scheduler and page replacement algorithms. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, pages 227–231, 1999.

[88] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 239–252, 2008.

[89] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. Rl-cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[90] Donald E. Knuth. *"Weak components", The Art of Computer Programming, Volume IV, Pre-Fascicle 12A: Components and Traversal*. 2022.

[91] Seongyun Ko and Wook-Shin Han. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 395–410, 2018.

[92] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[93] Michail G. Lagoudakis. *Value Function Approximation*, pages 1011–1021. Springer US, Boston, MA, 2010.

[94] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[95] Roland L Lee, Pen-Chung Yew, and Duncan H Lawrie. Data Prefetching In Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 28–31, 1987.

[96] 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), 2007.

[97] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, June 2014.

[98] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. TaP: Table-based Prefetching for Storage Caches. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 81–96, 2008.

[99] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[100] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, and Lihua Yue. CCF-LRU: a new buffer replacement algorithm for flash memory. *IEEE Trans. Consumer Electron.*, 55(3):1351–1359, 2009.

[101] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 285–300, 2017.

[102] Jihang Liu, Shimin Chen, and Lujun Wang. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[103] Ning Liu, Dong-sheng Li, Yiming Zhang, and Xiong-lve Li. Large-scale graph processing systems: a survey. *Frontiers Inf. Technol. Electron. Eng.*, 21(3):384–404, 2020.

[104] László Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2(1-46):4, 1993.

[105] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[106] Chenyang Lu. Aqueduct: Online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association.

[107] X. Lu, H. Najafi, J. Liu, and X. Sun. Chrome: Concurrency-aware holistic cache management framework with online reinforcement learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1154–1167, Los Alamitos, CA, USA, mar 2024. IEEE Computer Society.

[108] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. Operation-aware buffer management in flash-based systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2011.

[109] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 527–543, 2017.

[110] Hamid R. Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'09, page 1204–1212, Red Hook, NY, USA, 2009. Curran Associates Inc.

[111] L. Magdalena. Fuzzy rule-based systems. In J. Kacprzyk and W. Pedrycz, editors, *Springer Handbook of Computational Intelligence*, pages 203–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[112] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[113] Bo Mao, Suzhen Wu, and Lide Duan. Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 37(2):472–484, 2018.

[114] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 116–128, 2019.

[115] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9(1):526, 2014.

[116] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[117] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[118] Jai Menon, David A Pease, Robert M Rees, Linda Duyanovich, and Bruce Light Hillsberg. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[119] S. Min, D. Lee, C. Kim, J. Choi, J. Kim, Y. Cho, and S. Noh. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, dec 2001.

[120] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys*, 49(2):35:1—-35:35, 2016.

[121] Junoh Moon, Mincheol Kang, Wonyoung Lee, and Soontae Kim. Salvaging Runtime Bad Blocks by Skipping Bad Pages for Improving SSD Performance. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 576–579, 2022.

[122] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine M Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? and Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference, SYSTOR 2016, Haifa, Israel, June 6-8, 2016*, pages 7:1—-7:11, 2016.

[123] Suman Nath and Phillip B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. *Proceedings of the VLDB Endowment*, 1(1):970–983, 2008.

[124] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)*, 2007.

[125] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[126] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–306, 1993.

[127] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[128] R. Orlandic. Effective management of hierarchical storage using two levels of data clustering. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, pages 270–279, 2003.

[129] Tarikul Islam Papon. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 5644–5648, 2024.

[130] Tarikul Islam Papon and Manos Athanassoulis. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 2:1—-2:11, 2021.

[131] Tarikul Islam Papon and Manos Athanassoulis. The Need for a New I/O Model. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[132] Tarikul Islam Papon and Manos Athanassoulis. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1326–1339, 2023.

[133] Tarikul Islam Papon, Teona Bagashbili, and Manos Athanassoulis. ACE-in-Action: A Smart DBMS Bufferpool for SSDs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2025.

[134] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data (PACMMOD)*, 2(3):125:1—-125:26, 2024.

[135] Tarikul Islam Papon, Ju Hyoung Mun, Konstantinos Karatsenidis, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. Effortless Locality on Data Systems Using Relational Fabric. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 36(12):7410–7422, 2024.

[136] Tarikul Islam Papon, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. Relational Fabric: Transparent Data Transformation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 3688–3698, 2023.

[137] Hyoungmin Park and Kyuseok Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8):1298–1312, 2009.

[138] Kwanghyun Park, Yang-Suk Kee, Jignesh M. Patel, Jaeyoung Do, Chanik Park, and David J. DeWitt. Query Processing on Smart SSDs. *IEEE Data Engineering Bulletin*, 37(2):19–26, 2014.

[139] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 234–241, 2006.

[140] Stan Park and Kai Shen. A performance evaluation of scientific I/O workloads on Flash-based SSDs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–5, 2009.

[141] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, page 13, 2012.

[142] Roger A Pearce, Maya B Gokhale, and Nancy M Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11, 2010.

[143] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. Thesios: Synthesizing accurate counterfactual i/o traces from i/o samples. ASPLOS '24. Association for Computing Machinery, 2024.

[144] Stefan Podlipnig and László Böszörményi. A survey of Web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[145] Gregory Popovitch. The Parallel Hashmap. *https://github.com/greg7mdp/parallel-hashmap*, 2020.

[146] PostgreSQL. The Internals of PostgreSQL. *http://www.interdb.jp/pg/pgsql08.html*, 2015.

[147] PostgreSQL. PostgreSQL: The pg_prewarm module. *https://www.postgresql.org/docs/current/pgprewarm.html*, 2016.

[148] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 13:1–13:8, 2019.

[149] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[150] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. Phase-change random access

memory: A scalable technology. *IBM Journal of Research and Development*, 52(4-5):465–480, 2008.

[151] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[152] Jinting Ren, Xianzhang Chen, Duo Liu, Yujuan Tan, Moming Duan, Ruolan Li, and Liang Liang. A machine learning assisted data placement mechanism for hybrid storage systems. *Journal of Systems Architecture*, 120:102295, 2021.

[153] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[154] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 66–79, 2023.

[155] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 410–424, 2015.

[156] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013.

[157] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[158] Guido De Sandre, Luca Bettini, Alessandro Pirola, Lionel Marmonier, Marco Pasotti, Massimo Borghi, Paolo Mattavelli, Paola Zuliani, Luca Scotti, Gianfranco Mastracchio, Ferdinando Bedeschi, Roberto Gastaldi, and Roberto Bez. A 4 Mb LV MOS-Selected Embedded Phase Change Memory in 90 nm Standard CMOS Technology. *J. Solid-State Circuits*, 46(1):52–63, 2011.

[159] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 893–908, 2020.

[160] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)*, 48(3):8:1—-8:40, 2023.

[161] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Trans. Storage*, 1(3):316–345, August 2005.

[162] Jinho Seol, Hyotaek Shim, Jaegeuk Kim, and Seungryoul Maeng. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–146, 2009.

[163] Zhibing Sha, Zhigang Cai, François Trahay, Jianwei Liao, and Dong Yin. Unifying temporal and spatial locality for cache management inside SSDs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 1–6. IEEE, 2022.

[164] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United States Data Center Energy Usage Report. *Ernest Orlando Lawrence Berkeley National Laboratory*, LBNL-10057, 2016.

[165] Kai Shen and Stan Park. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 67–78, 2013.

[166] Milan M. Shetti, Bingzhe Li, and David H.C. Du. Machine learning-based adaptive migration algorithm for hybrid storage systems. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8, 2022.

[167] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 413–425, New York, NY, USA, 2019. Association for Computing Machinery.

[168] Barak Shoshany. BS::thread_pool: a fast, lightweight, and easy-to-use C++17 thread pool library. https://github.com/bshoshany/thread-pool, 2021.

[169] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146, 2013.

[170] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[171] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

[172] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. Sibyl: adaptive and extensible data placement in hybrid storage systems using online reinforcement learning. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 320–336, 2022.

[173] Smartmontools. Smart Monitoring Tools. *https://www.smartmontools.org/*.

[174] Alan Jay Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.

[175] SPDK. Storage Performance Development Kit (SPDK). *https://spdk.io*, 2016.

[176] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[177] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[178] V. Sundaram, T. Wood, and P. Shenoy. Efficient data migration in self-managing storage systems. In *2006 IEEE International Conference on Autonomic Computing*, pages 297–300, 2006.

[179] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 08 1988.

[180] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[181] Jianzhe Tai, Bo Sheng, Yi Yao, and Ningfang Mi. Live data migration for reducing sla violations in multi-tiered storage systems. In *2014 IEEE International Conference on Cloud Engineering*, pages 361–366, 2014.

[182] Andrew S Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[183] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 306–313, 1997.

[184] Luis Thomas, Sebastien Gougeaud, Stéphane Rubini, Philippe Deniel, and Jalil Boukhobza. Predicting file lifetimes for data placement in multi-tiered storage systems for hpc. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[185] Quoc-Cuong To, Juan Soto, and Volker Markl. A Survey of State Management in Big Data Processing Systems. *CoRR*, abs/1702.0, 2017.

[186] TPC. Specification of TPC-C benchmark. *http://www.tpc.org/tpcc/*, 2022.

[187] Transaction Processing Performance Council. Tpc-c benchmark. http://www.tpc.org/tpcc/, 2010. Last accessed on October 8, 2024.

[188] Transaction Processing Performance Council. Tpc-e benchmark. http://www.tpc.org/tpce/, 2010. Last accessed on October 8, 2024.

[189] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[190] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[191] P Venketesh and R Venkatesan. A Survey on Applications of Neural Networks and Evolutionary Techniques in Web Caching. *IETE Technical Review*, 26(3):171–180, 2009.

[192] Stratis D. Viglas. Adapting the B +-tree for asymmetric I/O. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7503 LNCS, pages 399–412, 2012.

[193] Jeffrey Scott Vitter and P Krishnan. Optimal Prefetching via Data Compression. *J. ACM*, 43(5):771–793, 1996.

[194] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 389–404, 2017.

[195] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 763–782, 2018.

[196] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z Sheng, Mehmet A Orgun, Longbing Cao, Francesco Ricci, and Philip S Yu. Graph Learning based Recommender Systems: A Review. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4644–4652, 2021.

[197] Shakthi Weerasinghe, Arkady Zaslavsky, Seng W. Loke, Alexey Medvedev, Amin Abken, Alireza Hassani, and Guang-Li Huang. Reinforcement learning based approaches to adaptive context caching in distributed context management systems. *ACM Trans. Internet Things*, 5(2), April 2024.

[198] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[199] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[200] John Wilkes, Richard A Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[201] Kan Wu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[202] Wenlei Xie, Guozhang Wang, David Bindel, Alan J Demers, and Johannes Gehrke. Fast Iterative Graph Computation with Block Updates. *Proceedings of the VLDB Endowment*, 6(14):2014–2025, 2013.

[203] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, 2016.

[204] 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 ACM International Systems and Storage Conference (SYSTOR), pages 6:1–6:11, 2015.

[205] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. Ssd-based workload characteristics and their performance implications. *ACM Trans. Storage*, 17(1), January 2021.

[206] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 335–346, 2004.

[207] Yun-Seok Yoo, Hyejeong Lee, Yeonseung Ryu, and Hyokyung Bahn. Page Replacement Algorithms for NAND Flash Memory Storages. In *Proceedings of the International Conference on Computational Science and Applications (ICCSA)*, Lecture Notes in Computer Science, pages 201–212, 2007.

[208] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting PCM asymmetries. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 282–293, 2013.

[209] Geng Yushui and Yuan Jiaheng. Cloud data migration method based on pso algorithm. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 143–146, 2015.

[210] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 148–155, 2010.

[211] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 183–193, 2015.

[212] Tianru Zhang. Autonomous hierarchical storage management via reinforcement learning. *Proceedings of the VLDB Endowment. ISSN*, 2150:8097, 2024.

[213] Tianru Zhang, Ankit Gupta, María Andreína Francisco Rodríguez, Ola Spjuth, Andreas Hellander, and Salman Toor. Data management of scientific applications in a reinforcement learning-based hierarchical storage system. *Expert Systems with Applications*, 237:121443, 2023.

[214] Tianru Zhang, Andreas Hellander, and Salman Toor. Efficient hierarchical storage management empowered by reinforcement learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5780–5793, 2022.

[215] Tianru Zhang, Tarikul Islam Papon, Teona Bagashvili, Manos Athanassoulis, and Salman Toor. Restore: A reinforcement learning approach for data migration in multi-tiered storage. *Under Preparation.*

[216] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 441–452, 2018.

[217] Peixiang Zhao and Jiawei Han. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment*, 3(1):340–351, 2010.

[218] Da Zheng, Disa Mhembere, Randal C Burns, Joshua T Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58, 2015.

[219] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

[220] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.

[221] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 375–386, 2015.

[222] Zichen Zhu, Siqiang Luo, Xiaokui Xiao, Yin Yang, Dingheng Mo, and Yufei Han. VC-Tune: Tuning and Exploring Distributed Vertex-Centric Graph Systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 3142–3145, 2022.

CURRICULUM VITAE

Tarikul Islam Papon

Contact Information

44 Washington Street, Apt 604
Brookline, MA 02445

E-mail: papon@bu.edu
Website: https://cs-people.bu.edu/papon

Research Interests

Databases, Data Management, Data Systems, Storage Systems, Flash Memory, Near-Data Processing, Disaggregated Memory and Storage.

Professional Experience

- PhD Researcher at **Boston University**, MA, USA — Sept '19 - Present
- Preceptor at **Harvard University**, MA, USA — Jan '25 - Present
- Course Instructor at **Boston University**, MA, USA — Spring 2024
- Research Scientist Intern at **Intel Corporation**, CA, USA — May '22 - Sept '22
- Research Intern at **Microsoft Research**, WA, USA — May '21 - Aug '21
- Lecturer at **CSE, BUET**, Dhaka, Bangladesh — Oct '15 - Aug '19

Education

Ph.D. in Computer Science — 2019 - Present
Boston University
Topic: SSD-Aware Data Systems Design
Supervisor: Prof. Manos Athanassoulis

M.Sc. in Computer Science and Engineering — 2016 - 2019
Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh.
Thesis: Design and Development of A Deep Learning Based Application for Detecting Diabetic Retinopathy
Supervisor: Prof. Ashikur Rahman

B.Sc. in Computer Science and Engineering — 2010 - 2015
Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh.
Thesis: Photoplethysmographic Analysis of Optical Signals : A Single Device to Measure All the Vital Signs
Supervisor: Prof. Ashikur Rahman

Awards and Scholarships (Selected)

- **SIGMOD Travel Scholarship**, SIGMOD — 2024
- **Best Demo Award**, VLDB — 2023
- **Travel Award**, IEEE ICDE — 2023
- **Research Excellence Award**, Boston University — 2022
- **Dean's Fellowship**, Boston University — Fall 2019
- **Dean's Honor List** in all 4 levels of Undergrad — 2010-2015
- **University Merit Scholarship** in all 8 semesters of Undergrad — 2010-2015
- **Champion** of the Intra University Database Project Show, BUET — 2013
- **Champion** of the Intra University Robotics Contest, BUET — 2012
- **Champion** of the Intra University Logic Olympiad, BUET — 2012

Publications

1. **Tarikul Islam Papon**, Teona Bagashbili, Manos Athanassoulis. *ACE-in-Action: A Smart DBMS Bufferpool for SSDs*, In Proceedings of the ACM on Management of Data (**SIGMOD**), 2025 (Demonstration).
2. **Tarikul Islam Papon**, Taishan Chen, Shuo Zhang, Manos Athanassoulis. *CAVE: Concurrency-Aware Graph Processing on SSDs*, In Proceedings of the ACM on Management of Data (**SIGMOD**), 2024.
3. **Tarikul Islam Papon**, Ju Hyoung Mun, Konstantinos Karatsenidis, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, Manos Athanassoulis. *Effortless Locality on Data Systems using Relational Fabric*, IEEE Transactions on Knowledge and Data Engineering (**TKDE**), 2024.
4. **Tarikul Islam Papon**. *Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry*, In Proceedings of the IEEE International Conference on Data Engineering (**ICDE**) PhD Symposium, 2024.
5. **Tarikul Islam Papon**, Abdul Wasay. *Silhouette: Toward Performance-Conscious and Transferable CPU Embeddings*, Workshop on ML for Systems at the Conference on Neural Information Processing Systems (**ML-for-Systems@NeurIPS**), 2023.
6. Subhadeep Sarkar, **Tarikul Islam Papon**, Dimitris Staratzis, Zichen Zhu, Manos Athanassoulis. *Enabling Timely and Persistent Deletion in LSM-Engines*, ACM Transactions on Database Systems (**TODS**), 2023.

7. Ju Hyoung Mun, Konstantinos Karatsenidis, **Tarikul Islam Papon**, Shahin Roozkhosh, Denis Hoornaert, Ulrich Drepper, Ahmed Sanaullah, Renato Mancuso, Manos Athanassoulis. *On-the-fly Data Transformation in Action*, In Proceedings of the **VLDB** Endowment, 2023. (Demonstration)

8. Renato Mancuso, Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, **Tarikul Islam Papon**, Manos Athanassoulis. *Software-Shaped Platforms*, In Proceedings of the Cyber-Physical Systems and Internet of Things Week (**CPS-IoT Week Workshops**), 2023.

9. **Tarikul Islam Papon**, Manos Athanassoulis. *ACEing the Bufferpool Management Paradigm for Modern Storage Devices*, In Proceedings of the IEEE International Conference on Data Engineering (**ICDE**), 2023.

10. **Tarikul Islam Papon**, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, Manos Athanassoulis. *Relational Fabric: Transparent Data Transformation [Vision]*, In Proceedings of the IEEE International Conference on Data Engineering (**ICDE**), 2023.

11. Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, **Tarikul Islam Papon**, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, Manos Athanassoulis. *Relational Memory: Native In-Memory Accesses on Rows and Columns*, In Proceedings of the International Conference on Extending Database Technology (**EDBT**), 2023.

12. Manos Athanassoulis, Subhadeep Sarkar, **Tarikul Islam Papon**, Zichen Zhu, Dimitris Staratzis. *Building Deletion-Compliant Data Systems*, IEEE Data Engineering Bulletin (**IEEE DEBull**), 2022.

13. T. M. Tariq Adnan, Md. Saiful Islam, **Tarikul Islam Papon**, Shourav Nath, Muhammad Abdullah Adnan. *UACD: A Local Approach for Identifying the Most Influential Spreaders in Twitter in a Distributed Environment*, Social Network Analysis and Mining (**SNAM**), 12(1), 1-21, Springer, 2022.

14. **Tarikul Islam Papon**, Manos Athanassoulis. *A Parametric I/O Model for Modern Storage Devices*, In Proceedings of the International Workshop on Data Management on New Hardware (**DaMoN**), 2021.

15. **Tarikul Islam Papon**, Manos Athanassoulis. *The Need for a New I/O Model*, Conference on Innovative Data Systems Research **(CIDR)**, 2021. (Abstract)

16. Subhadeep Sarkar, **Tarikul Islam Papon**, Dimitris Staratzis, Manos Athanassoulis. *Lethe: A Tunable Delete-Aware LSM Engine*, In Proceedings of the ACM **SIGMOD** International Conference on Management of Data, 2020.

17. Sadia Tamanna Khan, Ashef Ainan Baksh, **Tarikul Islam Papon**, Muhammad Ashraf Ali. *Rainwater Harvesting System: An Approach for Optimum Tank Size*

Design and Assessment of Efficiency, International Journal of Environmental Science and Development (**IJESD**), 8(1), 37-43, 2017.

18. **Tarikul Islam Papon**, Ishtiyaque Ahmad, Nazmus Saquib and Ashikur Rahman. *Non-invasive Heart Rate Measuring Smartphone Applications using Onboard Cameras: A Short Survey.* The 1st International Conference on Networking Systems and Security (**NSysS**), January 5-7, 2015, Dhaka, Bangladesh.

19. Nazmus Saquib, **Tarikul Islam Papon**, Ishtiyaque Ahmad and Ashikur Rahman. *Measurement of Heart Rate Using Photoplethysmography.* The 1st International Conference on Networking Systems and Security (**NSysS**), January 5-7, 2015, Dhaka, Bangladesh.

Research Talks

1. "Optimizing Data Systems for Modern Storage and Memory Technology", Oct 2024, *Guest Lecture@COSI 127B*, Brandeis University, Waltham, USA.

2. "CAVE: Concurrency-Aware Graph Processing System on SSDs", June 2024, *SIGMOD*, Santiago, Chile.

3. "Concurrency-Aware Graph Processing System for SSD", May 2024, *NEDB Day*, Boston, USA.

4. "ACEing the Bufferpool Management Paradigm for Modern Storage Devices", Apr 2023, *ICDE*, Anaheim, USA.

5. "The Parametric I/O Model & its Applicability on DBMS Bufferpool", Mar 2023, *Guest Lecture@CS 561*, Boston University, USA.

6. "Can we make Asymmetry/Concurrency-Aware Bufferpool Manager for SSDs?", December 2022, DIAS Group, EPFL, Lausanne, Switzerland.

7. "Asymmetry/Concurrency-Aware Bufferpool Manager for Modern Storage Devices", April 2022, University of Wisconsin-Madison, USA.

8. "A Parametric I/O Model for Modern Storage Devices", June 2021, *DaMoN Workshop*, China.

9. "The Case for a Parametric I/O Model", May 2020, *MiDAS Seminar*, Boston University, USA.

Media Interview

- "ACEing the Bufferpool Management Paradigm for Modern Storage Devices", Disseminate: The Computer Science Research Podcast, June 2023.

Posters

1. *ReStore: RL-Based Data Migration for Multi-Tiered Storage*, Poster at North East Database (NEDB) Day, Brandeis University, Waltham, USA, 2025.
2. • *CAVE: Concurrency-Aware Graph Processing System for SSD*, • *ZNS SSDs: Architectural Insights and Performance Analysis*, Posters at North East Database (NEDB) Day, Boston University, Boston, USA, 2024.
3. • *ACE Bufferpool Management for SSDs*, • *Relational Fabric: Effortless Locality for Data-intensive Systems*, • *Relational Memory: Native In-Memory Accesses on Rows and Columns*, • *Building Robust LSM-based Data Stores*, Posters at North East Database (NEDB) Day, Northeastern University, Boston, USA, 2023.
4. *Relational Memory: Native In-Memory Stride Access*, Poster & Presentation at DevConf. US 2022, Boston, USA.
5. *Lethe: A Delete-Aware LSM Engine*, Poster at North East Database Day 2020.

Teaching Experience

Boston University (Course Instructor)

- CS 561 (Data Systems Architectures) — Spring 2024

Boston University (Teaching Fellow)

- CS 210 (Computer Systems) — Fall 2023
- CS 561 (Data Systems Architectures) — Spring 2023, Spring 2021
- CS 460 (Introduction to Database Systems) — Fall 2021
- CS 460 (Introduction to Database Systems) — Fall 2020

Bangladesh University of Engg. and Tech. (Lecturer) — 2015 - 2019

Theory Courses

- CSE 453 (High-Performance Database Systems) — Jan 2019
- CSE 433 (Digital Image Processing) — Jan 2018, Jul 2016
- CSE 317 (Numerical Methods) — Jul 2017
- CSE 307 (Software Engineering & Information System Design) — Jan 2017

Lab Courses

- CSE 218 (Numerical Methods Lab) — Jan 2019
- CSE 410 (Computer Graphics Lab) — Jan 2019
- CSE 322 (Computer Networks Lab) — Jul 2018, Jan 2016

- CSE 208 (Data Structures & Algorithms II Lab) Jul 2018
- CSE 108 (Object Oriented Programming Lab) Jul 2018, Jul 2017
- CSE 308 (Software Engineering Lab) Jan 2018, Jan 2017
- CSE 206 (Digital Logic Design Lab) Jan 2018, Jan 2016
- CSE 102 (Structured Programming Language Lab) Jan 2018
- CSE 216 (Database Lab) Jul 2017, Jul 2016
- CSE 402 (Artificial Intelligence Lab) Jul 2016
- CSE 404 (Digital System Design Lab) Jul 2016, Jan 2016
- CSE 214 (Assembly Language Lab) Jul 2016

Supervision & Mentorship (Selected)

- Aditya Chowdhri & Panos Koutsoukos, *Undergrad Intern.* Spring 25, Fall 24
  Topic: Benchmarking Large-Scale Graph Processing Systems.
- Yiyun Zheng, *Undergraduate Visiting Intern.* Fall 2024
  Topic: Sieve Page Replacement Policy for Database operations.
- Prisha Shah, *High-school RISE Intern.* Summer 2024
  Topic: Evaluation of Sieve Page Replacement Policy.
- Ronin Bae, *High-school Research Intern.* Spring, Summer 2023
  Topic: Visualizing the ACE bufferpool in action.
- Shuo Zhang & Taishan Chen, *Undergrad Research Intern* Summer, Fall 2022
  Topic: Concurrent Graph/Tree Traversal in Modern Storage Devices.
- Subin (Rachael) Kim, *High-school Research Intern.* Fall 2020 - Spring 2021
  Topic: Visualizing the impact of the Parametric I/O Model.
- Zheng Hui, *Undergrad Research Intern* Fall 2020
  Topic: Designing a concurrency-aware graph traversal algorithm.

Professional Services

- **PC Member**, SIGMOD Demo 2025
- **Journal Reviewer**
  - ACM Transactions on Knowledge Discovery from Data (TKDD)
  - Transactions on Database Systems (TODS)
  - Journal of Systems Architecture (JSA)
  - Journal of Big Data

  - IEEE BigData

- **Shadow PC Member**, EuroSys 2024
- **Society Member**, Association for Computing Machinery (ACM), ACM Special Interest Group on Management of Data (SIGMOD), Institute of Electrical and Electronics Engineer (IEEE)
- **Member**, Sponsorship and Finance Committee, International Conference on Net-working, Systems and Security (NSysS 2016 – 2018)

Consultancy Projects (Selected)

- Testing and Certification of the Flora Bank Core Banking System (2018-2019)
- Development of Android Based I-Books & Associated Services for Bangladesh Technical Education Board and Madrasah Education Board (2016-2019)
- Development, Supply & Installation of Multi-Language Training Software titled "*Vashaguru*" for Sheikh Russell Digital Labs in Bangladesh (2017-2018)
- XI Class Admission in Bangladesh (Session: 2016-2017, 2017-2018)
- Processing of Results of IEB Election (2017)

Technical Skills

- **Programming Languages:** C, C++, C#, Java, Python, Golang
- **Database Management Systems:** RocksDB, MonetDB, PostgreSQL, MySQL, Oracle, SQL Server, MongoDB, Neo4j
- **Distributed Computing Frameworks:** Apache Hadoop, Spark, Giraph
- **Machine Learning:** TensorFlow, PyTorch, Keras
- **Hardware:** Verilog, Vivado (Xilinx), Arduino, AVR, Rasberry Pi