Memory Efficient Systems for the Modern Data Processing Stack

by

Yifan Dai

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: Augest 5, 2025

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Michael M. Swift, Professor, Computer Sciences
Ming Liu, Assistant Professor, Computer Sciences
Jin-Yi Cai, Professor, Mathematics

To my family and friends.

Acknowledgments

I thank many people for their help and guidance throughout my Ph.D. life.

First and foremost, I express my most sincere gratitude to my advisors, Andrea and Remzi. Remzi's speech at the visiting day was so appealing and was one of the most important reasons for me to join their lab. They have formed my values in systems research. They have taught me to sit down and measure and explore the search space of a topic with patience, and then pick a set of interesting problems to solve with a new system.

Remzi has been an excellent guide in the exploration phase. He can always point out the details I missed and point to new directions. He encouraged me to be brave enough to break existing systems during explorations and implementations. He gave this advice in my first year and this advice led to one of the core mechanisms in my first project. I have never stuck and felt lost in building systems since then, with the belief that as long as I explore enough possibilities, I will always find a way.

Andrea is always organized and has helped us a lot to formulate the project after the exploration phase. I could not appreciate more having Andrea as my advisor when I started to write my paper on my own. She makes the project and the paper organized and gives clear directions for improvements. I was (and am now) not good at writing and every talk with her about writing and presentation improved me a bit. I feel that I still have so much to learn in writing from Andrea.

What I appreciate the most is that my advisors have created a solid and self-consistent environment for us. They recognize the efforts I have made. In this environment, I do not need to rush for the number of publications and I feel assured that I will be fine as long as I am making progress. I can sit down and try hard topics to improve myself and be less anxious about paper rejection. My Ph.D. life has been a hard but enjoyable experience because of Andrea and Remzi.

I am grateful to my committee members: Michael Swift, Ming Liu, and Jin-Yi Cai, for their

willingness to serve on my committee. Mike was also on my preliminary committee and gave valuable feedback on the proposal of my last project. Jin-Yi's course on complexity theory was inspiring and his lecture was so organized and detailed that even I, who struggled a lot in undergraduate theory courses, could follow with ease. I thank Shivaram Venkataraman, who was on my preliminary committee, for his feedback on my last project and his course on big data systems from which I learned a lot about large-scale systems.

I thank Jing Liu, my friend and former lab mate who graduated from our group last year. She helped a lot with the initial organization of the Symbiosis paper. We have had wonderful collaborations on 3 projects and even on other projects, discussions with her are always fruitful. She was a good partner in research to me because we are good at different things and think in different aspects. It was fun that we could improve each other's writing but struggled with our own writing. She has given me valuable advice on writing and presentation, second only to my advisors.

I thank Tyler Caraza-Harter for his advice in formulating the Kelvin project and his help in writing. It has been a pleasant experience during my last project as his ideas on many interesting engineering challenges fit me well. I thank Aishwarya Ganesan and Ram Alagappan for putting the Bourbon project together. I was completely lost as a first-year student then and learned a lot from them.

Apart from the above, I have had the privilege to work with outstanding lab mates and collaborators: Yien Xu, Anthony Rebello, Kan Wu, Shawn Zhong, Suyan Qu, Guanzhou Hu, Chenhao Ye, Kaiwei Tu, Vinay Banakar, Tingjia Cao, Brian Kroth, and many more. Discussions with them on research have always been fruitful; chatting with them about everyday life has been a great relief from the pressure of research. I remember the design of the core of my Kelvin project came from a casual dinner gathering with my lab mates.

I thank Naomi Geyer, Steven Ridgely, and Junko Mori for their help in my pursuit of Ph.D. Minor in Japanese, especially Professor Geyer's accommodation for this unusual case of mine. I had an interesting experience in Professor Ridgely's course on translation from Japanese to English as both are foreign languages to me, and I was surprised at how different languages can be from the fundamental logic. Professor Mori's course on Japanese linguistics opens a new direction for thinking about Japanese as well as other languages.

I feel fortunate to have Runyu Zheng as my best friend. Japanese story games have been my sole interest in leisure time and Runyu is the only one who can enjoy Japanese stories in the original language with me and feel the subtle emotions that would be lost by translation.

We share our feelings in the stories, and then in our own life. The conversations with her and the games we played together have made my life more colorful.

Finally, I express my gratitude to my parents. I did not have much chance to go back to China during my entire Ph.D. life due to COVID and visa issues, but they were with me for a few months in my final year and helped me through the hard times of job hunting and paper resubmission.

Contents

List of Tables

List of Figures

Abstract

The world has witnessed exponential growth of data. People interact with data every day through social media, e-commerce, and so on.

People have built software stacks to efficiently work with data. A modern data processing stack includes layers that are responsible for ingesting, storing, transforming, and utilizing data. With the data becoming larger, the data processing stack requires better memory efficiency. There has been extensive study from the systems community to optimize each layer of the stack, but communication between them is often overlooked, leading to hidden overheads across layers.

In this thesis, we introduce three aspects of study on memory efficiency of the data processing stack, both on optimizations within a single layer and those across layers. The first part of the thesis introduces a learned index for Log-structured Merge (LSM) Trees. The learned index is 0.5× to 0.75× smaller than the original index and improves the in-memory workload performance by 1.5× on average.

In the second part, we focus on the caching problem between the layer of storage engines and the layer of the underlying kernel. Both storage engines and the underlying kernel use data caching and they share the same memory quota, implicitly forming a two-level cache structure of which storage engines are often unaware. We introduce a framework that automatically optimizes cache allocation of storage engines by online cache simulation and dynamically adapts to different workloads. We incorporate our system into 3 popular key-value storage systems and provide a 1.5× gain on average.

The third part focuses on removing data copying and duplication in data pipelines on a single machine. Data communication between different nodes in a data pipeline currently requires full copying even if they are located on the same machine due to limited kernel support. We build a pipeline execution engine with co-design of new kernel mechanisms and container runtime. Our new system provides a 2× gain for real-world data pipelines.

We design and implement our solutions into real systems and yield benefits on real data processing workloads. We believe that our work will inspire the future development of the data processing stack.

Chapter 1

Introduction

In the past 20 years, the world has seen exponential growth of data, flowing from data storage such as data centers, data processing systems such as databases, to data consumers such as social media and machine learning models. The total amount of data generated worldwide is estimated to reach 200 zettabytes in 2025 [144], or 500 million TB daily.

While such a large scale was typically processed distributedly in the past [36, 130, 169], a large amount of such work can be done on a single machine with access to emerging single nodes with huge memories, in tens of terabytes [125, 141]. Processing data locally has several advantages, such as saving the data communication overhead and reducing the maintenance efforts of computing resources. How much work of the data processing stack that can be done locally depends on how much we can use the memory efficiently.

The data processing stack is a collection of technologies and tools that ingests, organizes, stores, and transforms data [124]. At the top of a typical modern data processing stack, we have applications in various areas such as data analytics and machine learning. Below that, we have databases and storage engines that are responsible for data storage, retrieval, and transformation. At the bottom layer, filesystems and the kernel process requests from databases and storage engines by communicating with the underlying hardware.

To improve the performance of the data processing stack, there has been extensive study from the system community to optimize each layer of the stack. For example,people have developed better algorithms for applications [168], new data structures for storage engines [77], and new kernel mechanisms to remove the scalability bottleneck [116].

However, as each layer tends to be optimized independently, communication between

them is often overlooked, leading to hidden overheads across layers. A typical type of such overhead is the inefficient use of memory. Caching, for example, is often built in every layer of the stack to utilize data locality, but the lack of coordination between these caches could result in sub-optimal policy choices and resource allocation [18, 159, 172]. Another example is data passing across layers. Data communication between different protection domains often requires copying, and this problem becomes more and more severe as the amount of data increases.

Thus, there lies huge opportunities in improving memory efficiency across layers in the modern data processing stack. We can optimize memory allocation mechanisms and policies according to specific access patterns from the upstream, or specific behaviors in the downstream. Moreover, co-design of the entire stack gives us a stronger ability to construct a more efficient data flow throughout the stack.

In this thesis, we introduce three aspects of study on memory efficiency of the data processing stack. In the first work, we focus on optimizing a single layer: the databases. We build a learned index for Log-structured Merge (LSM) Trees [98, 113, 123] and study the performance characteristics of a popular LSM-based storage engine (LevelDB [52]). We study the lifetime of tables in LSMs under various workloads and summarize rules for applying learning to LSMs. The learned index is much smaller than the original LSM-tree index and significantly improves the performance of in-memory workloads.

In the second work, we focus on the caching problem between the layer of storage engines and the layer of the underlying kernel. Both storage engines and the underlying kernel use data caching and they share the same memory quota, implicitly forming a two-level cache structure of which storage engines are often unaware. We introduce a framework that automatically optimizes cache allocation of storage engines by online cache simulation and dynamically adapts to different workloads.

The third work focuses on removing data copying and duplication in data pipelines on a single machine. Data communication between different nodes in a data pipeline currently requires a full copy of the communicated data even if the nodes are located on the same machine due to limited kernel support. We build a pipeline execution engine with co-design of the entire data processing stack, including new kernel mechanisms, new container runtime, and new resource management mechanisms.

Our studies solve real-world problems on memory efficiency of the modern data processing stack and build practical solutions. We believe that our work will inspire the future

development of components in the data processing stack.

1.1 A Learned Index for Log-Structured Merge Trees

Our first project focuses on improving in-memory workloads for databases. We build a learned index [77] for Log-structured Merge (LSM) Trees [98, 113, 123] and study the performance characteristics of a popular LSM-based storage engine (LevelDB [52]). Our system, BOURBON, significantly improves the indexing performance, which is the bottleneck for in-memory workloads, and reduces the index sizes by 0.5× to 0.75×.

Learned indexes apply machine learning to supplant the traditional index structure found in database systems, namely the ubiquitous B-Tree [32]. To look up a key, the system uses a learned function that predicts the location of the key (and value); when successful, this approach can improve lookup performance, in some cases significantly, and also potentially reduce space overhead. Since this pioneering work, numerous follow-ups [38, 50, 76] have been proposed that use better models, better tree structures, and generally improve how learning can reduce tree-based access times and overheads.

We now apply this new approach to the LSM trees. LSMs were introduced in the late '90s, gained popularity a decade later through work at Google on BigTable [27] and LevelDB [52], and have become widely used in industry, including in Cassandra [81], RocksDB [46], and many other systems [51, 106]. LSMs have many positive properties as compared to B-trees and their cousins, including high insert performance [34, 98, 117].

A major challenge of applying learned indexes to LSMs is that while learned indexes are primarily tailored for read-only settings, LSMs are optimized for writes. Writes cause disruption to learned indexes because models learned over existing data must now be updated to accommodate the changes; the system thus must re-learn the data repeatedly. However, we find that LSMs are well-suited for learned indexes. For example, although writes modify the LSM, most portions of the tree are immutable; thus, learning a function to predict key/value locations can be done once, and used as long as the immutable data lives. However, many challenges arise. For example, variable key or value sizes make learning a function to predict locations more difficult, and performing model building too soon may lead to significant resource waste.

Thus, we first study how an existing LSM system, WiscKey [98], functions in great detail (§2.2). We focus on WiscKey because it is a state-of-the-art LSM implementation

that is significantly faster than LevelDB and RocksDB [98]. Our analysis leads to many interesting insights from which we develop five *learning guidelines*: a set of rules that aid an LSM system to successfully incorporate learned indexes. For example, while it is useful to learn the stable, low levels in an LSM, learning higher levels can yield benefits as well because lookups must always search the higher levels. Next, not all files are equal: some files even in the lower levels are very short-lived; a system must avoid learning such files, or resources can be wasted. Finally, workload- and data-awareness is important; based on the workload and how the data is loaded, it may be more beneficial to learn some portions of the tree than others.

We apply these learning guidelines to build BOURBON, a learned-index implementation of WiscKey (§2.3). BOURBON uses piece-wise linear regression, a simple but effective model that enables both fast training (i.e., learning) and inference (i.e., lookups) with little space overhead. BOURBON employs *file learning*: models are built over files given that an LSM file, once created, is never modified in-place. BOURBON implements a cost-benefit analyzer that dynamically decides whether or not to learn a file, reducing unnecessary learning while maximizing benefits. While most of the prior work on learned indexes [38, 50, 77] has made strides in optimizing stand-alone data structures, BOURBON integrates learning into a production-quality system that is already highly optimized. BOURBON's implementation adds around 5K LOC to WiscKey (which has ~20K LOC).

We analyze the performance of BOURBON on a range of synthetic and real-world datasets and workloads (§2.4). We find that BOURBON reduces the indexing costs of WiscKey significantly and thus offers $1.23\times - 1.78\times$ faster lookups for various datasets. Even under workloads with significant write load, BOURBON speeds up a large fraction of lookups and, through cost-benefit, avoids unnecessary (early) model building. Thus, BOURBON matches the performance of an aggressive-learning approach but performs model building more judiciously. Finally, most of our analysis focuses on the case where fast lookups will make the most difference, namely when the data resides in memory (i.e., in the file-system page cache). However, we also experiment with BOURBON when data resides on a fast storage device (an Optane SSD) or when data does not fit entirely in memory, and show that benefits can still be realized.

1.2 Application and Kernel Cache Cooperation

We now dive into another crucial component in storage engines: caching. We design and build Symbiosis that optimizes the cache sizes of the storage engine applications with the knowledge of the underlying caching layer, the kernel page cache, and improves the overall cache efficiency of the data processing stack.

Key-value storage engines, such as LevelDB [53], RocksDB [46], and WiredTiger [107], are essential components in modern data-intensive applications. A crucial factor in the performance of key-value storage systems is the effectiveness of in-memory caching. Unlike the traditional database approach [136], in which raw devices or other "direct I/O" mechanisms are employed to avoid file system caching, today's key-value storage systems are often built on top of the file system, and thus will cache by default compressed data in the file system page cache. Furthermore, modern storage engines implement additional application-level caching structures where data is cached in uncompressed form. The effectiveness of these combined caches can dramatically affect overall performance; proper usage can improve performance by an order of magnitude.

Unfortunately, this two-level structure with data compression greatly complicates performance tuning. How large should the application cache for uncompressed data be? How much memory should be dedicated to kernel-level caching for compressed data? The proper answer to this question requires sophisticated knowledge of workload, machine configuration, OS behavior, compression costs, and other relevant details; as workloads change over time, the answer too may change.

We introduce *Symbiosis*, a system to coordinate application and kernel caches to maximize performance. The core component is an online approximate simulator used by a key-value store directly to adapt the size of the user-level cache. The simulator uses a modified form of ghost caching [42] to predict how different sized application caches will perform; Symbiosis uses these simulation results to periodically adjust the size of the application cache, thus improving performance. The online simulation includes novel optimizations to lower space overheads and handle nuanced kernel behaviors (such as prefetching), and guardrails to protect against unmodeled corner-case behaviors.

We show the utility of Symbiosis by incorporating it into 3 different key-value storage systems: LevelDB, WiredTiger, and RocksDB. Most of our work focuses on LevelDB [53]; through careful re-use of existing code (where appropriate), our modifications add roughly

1K lines to the code base. Across a range of read-heavy workloads, we show that Symbiosis improves LevelDB performance significantly (greater than 5×) as compared to unmodified LevelDB. We also show that our approach adapts effectively to workload changes and that the overheads are low.

Our other two implementations (in WiredTiger [107] and RocksDB [46]) demonstrate the generality of our approach. WiredTiger has a substantially different caching architecture than LevelDB, and yet we readily integrated Symbiosis into it with minor code alterations. In doing so, we also discovered a caching bug (acknowledged by the MongoDB team as major); we both fix the bug and show that Symbiosis improves performance. Finally, RocksDB can be configured to avoid the kernel cache; its two-level application-managed caching structure consists of a compressed cache of data read from disk and an uncompressed cache to service queries. We show Symbiosis works well when the application manages both caches directly, again improving performance.

1.3 Towards Zero-Copy Data Pipelines

Our last work focuses on improving memory efficiency for data pipelines. We build Kelvin by co-designing different layers of the data processing stack (user-space resource management, container runtime, and kernel support for shared memory) to eliminate data copying and duplication for data pipelines.

Data pipelines are a popular paradigm for data analysis and machine-learning workloads. Data pipelines are frequently implemented as DAGs (Directed Acyclic Graphs), where each node of a DAG describes a transformation to perform on the data [13, 132, 141, 169]. Edges represent data passing between nodes. Given fine-grained nodes, efficient communication along edges is critical for good performance [64, 127, 163]. Data passing along edges may occur via network, disk, or memory [36, 99, 100, 115, 169]; the specific medium depends on node placement and resource availability.

Recent workload and hardware trends [104, 140] demonstrate the feasibility of deploying most data pipelines on a single machine with ample memory, with memory as the sole and most efficient medium for data passing. Cloud virtual machines with 24 TB of RAM are now available [125]; in contrast, the largest (p99.9) datasets for OLAP workloads occupy a mere 0.25 TB [118]. Memory is increasingly affordable as well: from 2014 to 2023, cost has dropped from $4K to $1K for 1 TB of RAM [112]. The ideal way to pass in-memory data

between nodes in a DAG is by reference: if downstream processes in a data pipeline can virtually map upstream outputs into their virtual address spaces, copying and duplication overheads can be avoided.

Many challenges arise in practice when using shared memory for passing data between DAG nodes.

- **Challenge 1** In-memory formats often rely on pointers to virtual memory, but these pointers are only meaningful in the address space of a single process.
- **Challenge 2** Many libraries allocate non-shared memory, and sharing requires copying to shared memory.
- **Challenge 3** Node processes may be deployed in different containers, and accounting rules for "charging" containers for shared memory consumption are complex.
- **Challenge 4** Node output often extends node input, but copying is necessary if communication protocols lack a means for outputs to reference inputs.
- **Challenge 5** Sharing occurs at page granularity, but the commonality between inputs and outputs is often at row granularity.
- **Challenge 6** Data pipelines usually start by deserializing data from persistent storage to memory; without coordination, different pipelines will have their own copies of the same data in memory.

Simply using Linux shared memory and a zero-copy protocol such as Arrow [14, 147] solves the first challenge (pointers), but unfortunately the other challenges remain.

We build Kelvin, a data pipeline platform that avoids data copying and duplication for a wide range of scenarios running unmodified user code. Kelvin includes several components as solutions to the challenges above. The core component adds de-anonymization support to the Linux kernel to convert anonymous (i.e., non-shared) memory to shared memory without a copy, so that data produced by share-unaware code and libraries can be efficiently passed to other processes (Challenge 2). Around that, we build a new container to achieve both isolated execution and data sharing (Challenge 3), a shared IPC library to utilize the new kernel support to generate shared data without copying (Challenge 4 & 5), a cache based on shared memory to allow multiple pipelines to share a common input source (Challenge

6), a resource manager to manage the shared physical data, and a DAG executor built on OpenLambda [110] to utilize the existing mechanisms for containerized execution and basic resource accounting.

Kelvin's techniques provide substantial performance gains in a variety of scenarios by reducing both the time overhead of copying identical data and the memory overhead of duplicating identical data; this memory reduction allows significantly more DAG nodes to run in parallel. De-anonymization eliminates write-side copies, halving the latency of a single node outputting Arrow data that was loaded from a Parquet file. For a large batch of 2-node DAGs loading data from the same source, shared deserialization improves throughput by up to 38×. IPC inspection and resharing dramatically reduce physical output sizes for a variety of transformations, in some cases to practically zero new data. In combination, these features improve overall throughput by 1.2-28× for various complex mixes of DAGs.

1.4 Contributions and Highlights

A Learned Index for Log-Structured Merge Trees.

- We study the lifetime of LSM tables and the number of lookups they serve under various different workloads.
- We summarize a set of learning guidelines on how to perform learning on LSM trees.
- We design and implement a learned index for LSMs and dynamically decide whether to learn a new table according to the workload.

Application and Kernel Cache Cooperation.

- We identify the application-kernel two-level cache structure common for storage engines and highlight the performance gain and difficulty of optimal memory partitioning for the caches.
- We design and implement an offline cache simulator that accurately simulates the application-kernel cache structure and detailed admission and eviction behaviors of the kernel page cache.

- We design and implement Symbiosis to dynamically adjust cache sizes online according to the workload. Symbiosis is integrated into LevelDB, RocksDB, and WiredTiger and adds negligible overhead to the original systems.

Towards Zero-Copy Data Pipelines.

- We discover three different types of data copying and duplication in current data pipeline executions and identify the challenges to eliminate such data copying while achieving isolated execution and resource accounting.
- We add de-anonymization support to the Linux kernel to convert anonymous (i.e., non-shared) memory to shared memory without a copy so that data produced by share-unaware code and libraries can be efficiently passed to other processes.
- We create a new container implementation based on SOCK [110] that supports passing of shared Arrow data and leverages Linux cgroup accounting rules to expose control over swapping and eviction of intermediate data.
- We modify the Arrow communication library (Arrow IPC) to detect overlap between inputs and outputs, eliminate such overlap via references, and use new kernel support to de-anonymize as needed.
- We create a cache based on shared memory, allowing pipelines that load data from a common source to share a single in-memory copy of the data.
- We integrate all of the above into a new data pipeline platform, Kelvin, that avoids copying and duplication for a wide range of scenarios.

1.5 Overview

We briefly describe the contents of the chapters of this dissertation.

- **A Learned Index for Log-Structured Merge Trees.** In Chapter 2, we study the performance characteristics of LSM trees and present Bourbon, a learned index for LSMs.

- **Application and Kernel Cache Cooperation.** In Chapter 3, we study the application-kernel two-level cache structure and present Symbiosis to dynamically partition memory for the caches according to the workload.
- **Towards Zero-Copy Data Pipelines.** In Chapter 4, we identify various data copying and duplication in current data pipeline execution and build Kelvin to avoid them with co-design of the application and the kernel.
- **Related Work.** In Chapter 5, we describe previous work on machine learning for systems, LSM optimizations, cache management, and kernel methods in memory management.
- **Conclusions and Future Work.** In Chapter 6, we summarize the thesis and the lessons we learned during the work. We then discuss possible future directions to which our work can be extended.

Chapter 2

From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees

In this chapter, we build learned indexes for Log-structured Merge (LSM) Trees and study the performance characteristics of a popular LSM-based storage engine (LevelDB [52]). Our system, Bourbon, significantly improves the performance of in-memory workloads by accelerating the bottleneck, indexing, and reduces the index sizes by 0.5× to 0.75×.

We first present how LSMs function internally and extract statistics that are critical for learning (§2.2. With a suite of workloads with different access patterns and read/write rates, we study the lifetime of LSM tables and how many lookups these tables serve. We then formulate a set of guidelines on how to integrate learned indexes into an LSM with such statistics).

We then present the design and implementation of Bourbon which incorporates learned indexes into a real, highly optimized, production-quality LSM system (§2.3). We use piece-wise linear regression (PLR) [7, 71] to replace the indexing structure in LSM tables. We support variable-size values by incorporating key-value separation [98]. We apply a cost-benefit analyzer to determine whether it is worthwhile to learn a new table during runtime.

Through a variety of micro- and macro-benchmarks, we analyze Bourbon's performance in detail, demonstrating that Bourbon improves lookup performance by 1.23×-1.78× as compared to state-of-the-art production LSMs.

2.1 Background

We first describe log-structured merge trees and explain how data is organized in LevelDB. Next, we describe WiscKey, a modified version of LevelDB that we adopt as our baseline. We then provide a brief background on learned indexes.

2.1.1 LSM and LevelDB

An LSM tree is a persistent data structure used in key-value stores to support efficient inserts and updates [113]. Unlike B-trees that require many random writes to storage upon updates, LSM trees perform writes sequentially, thus achieving high write throughput [113].

An LSM organizes data in multiple *levels*, with the size of each level increasing exponentially. Inserts are initially buffered in an in-memory structure; once full, this structure is merged with the first level of on-disk data. This procedure resembles merge-sort and is referred to as compaction. Data from an on-disk level is also merged with the successive level if the size of the level exceeds a limit. Note that updates do not modify existing records in-place; they follow the same path as inserts. As a result, many versions of the same item can be present in the tree at a time. We refer to the levels that contain the newer data as *higher* levels and the older data as *lower* levels.

A lookup request must return the latest version of an item. Because higher levels contain the newer versions, the search starts at the topmost level. First, the key is searched for in the in-memory structure; if not found, it is searched for in the on-disk tree starting from the highest level to the lowest one. The value is returned once the key is found at a level.

LevelDB [52] is a widely used key-value store built using LSM. Figure 2.1(a) shows how data is organized in LevelDB. A new key-value pair is first written to the *memtable*; when full, the memtable is converted into an immutable table which is then compacted and written to disk sequentially as *sstables*. The sstables are organized in seven levels (L_0 being the highest level and L_6 the lowest) and each sstable corresponds to a file. LevelDB ensures that key ranges of different sstables at a level are disjoint (two files will not contain overlapping ranges of keys); L_0 is an exception where the ranges can overlap across files. The amount of data at each level increases by a factor of ten; for example, the size of L_1 is 10MB, while L_6 contains several 100s of GBs. If a level exceeds its size limit, one or more sstables from that level are merged with the next level; this is repeated until all levels are within their limits.

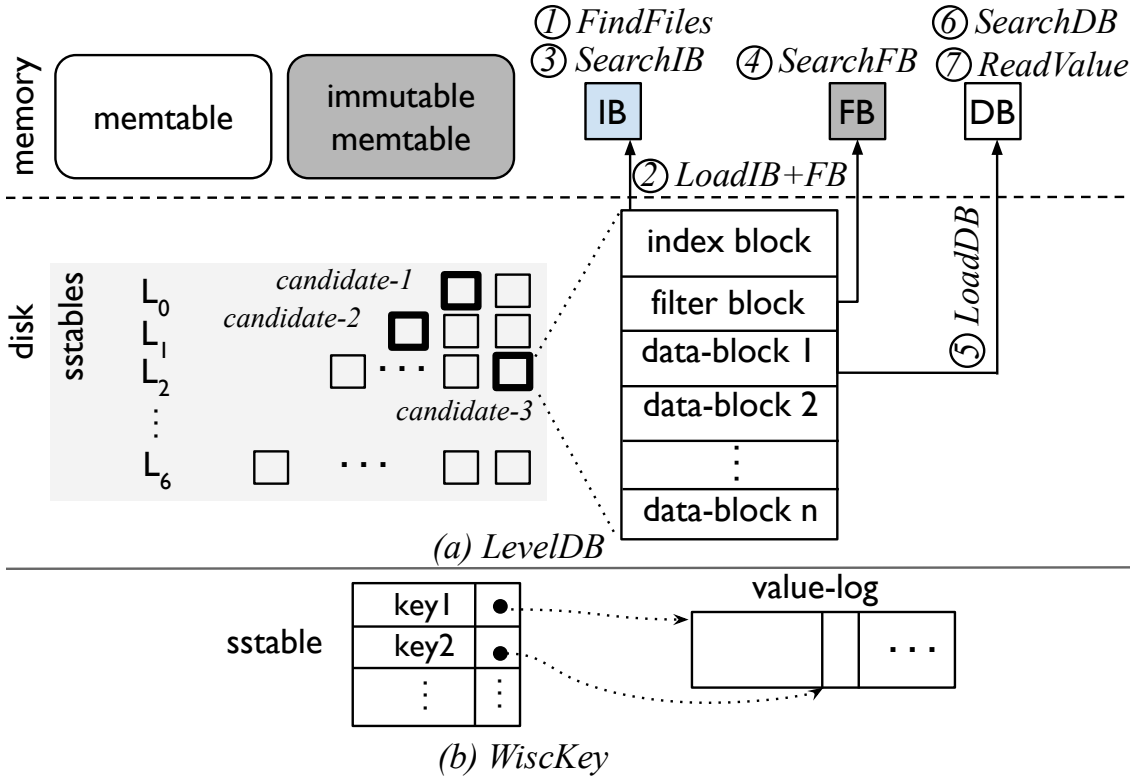


Figure 2.1: LevelDB and WiscKey. *(a) shows how data is organized in LevelDB and how a lookup is processed. The search in in-memory tables is not shown. The candidate sstables are shown in bold boxes. (b) shows how keys and values are separated in WiscKey.*

Lookup steps. Figure 2.1(a) also shows how a lookup request for key k proceeds. ① *FindFiles*: If the key is not found in the in-memory tables, LevelDB finds the set of candidate sstable files that may contain k. A key in the worst case may be present in all L_0 files (because of overlapping ranges) and within one file at each successive level. ② *LoadIB+FB*: In each candidate sstable, an index block and a bloom-filter block are first loaded from the disk. ③ *SearchIB*: The index block is binary searched to find the data block that may contain k. ④ *SearchFB*: The filter is queried to check if k is present in the data block. ⑤ *LoadDB*: If the filter indicates presence, the data block is loaded. ⑥ *SearchDB*: The data block is binary searched. ⑦ *ReadValue*: If the key is found in the data block, the associated value is read and the lookup ends. If the filter indicates absence or if the key is not found in the data block, the search continues to the next candidate file. Note that blocks are not always loaded from the disk; index and filter blocks, and frequently accessed data blocks are likely to be present in memory (i.e., file-system cache).

We refer to these search steps at a level that occur as part of a single lookup as an *internal lookup*. A single lookup thus consists of many internal lookups. A *negative internal lookup* does not find the key, while a *positive internal lookup* finds the key and is thus the last step of a lookup request.

We differentiate indexing steps from data-access steps; indexing steps such as *FindFiles*, *SearchIB*, *SearchFB*, and *SearchDB* search through the files and blocks to find the desired key, while data-access steps such as *LoadIB+FB*, *LoadDB*, and *ReadValue* read the data from storage. Our goal is to reduce the time spent in indexing.

Update steps. An update request is recorded directly to the memtable in Figure 2.1(a). When the memtable is full, it is converted to the immutable memtable. The previous immutable memtable is flushed to the first on-disk level L_0 and becomes an sstable. When each level exceeds a pre-defined size limit, a set of sstables on that level is compacted and becomes new sstables in the next level. An sstable's lifetime starts with a compaction from the upper level and ends with a compaction to the lower level. An sstable is immutable during its lifetime.

2.1.2 WiscKey

In LevelDB, compaction results in large write amplification because *both* keys and values are sorted and rewritten. Thus, LevelDB suffers from high compaction overheads, affecting foreground workloads.

WiscKey [98] (and Badger [2]) reduces this overhead by storing the values separately; the sstables contain only keys and pointers to the values as shown in Figure 2.1(b). With this design, compaction sorts and writes only the keys, leaving the values undisturbed, thus reducing I/O amplification and overheads. WiscKey thus performs significantly better than other optimized LSM implementations such as LevelDB and RocksDB. Given these benefits, we adopt WiscKey as the baseline for our design. Further, WiscKey's key-value separation enables our design to handle variable-size records; we describe how in more detail in §2.3.2.

The write path of WiscKey is similar to that of LevelDB except that values are written to a *value log*. A lookup in WiscKey also involves searching at many levels and a final read into the log once the target key is found. The size of WiscKey's LSM tree is much smaller than LevelDB because it does not contain the values; hence, it can be entirely cached in memory [98]. Thus, a lookup request involves multiple searches in the in-memory tree, and

the *ReadValue* step performs one final read to retrieve the value.

2.1.3 Optimizing Lookups with Learned Indexes

Performing a lookup in LevelDB and WiscKey requires searching at multiple levels. Further, within each sstable, many blocks are searched to find the target key. Given that LSMs form the basis of many embedded key-value stores (e.g., LevelDB, RocksDB [46]) and distributed storage systems (e.g., BigTable [27], Riak [106]), optimizing lookups in LSMs can have huge benefits.

A recent body of work, starting with learned indexes [77], makes a case for replacing or augmenting traditional index structures with machine-learning models. The key idea is to train a model (such as linear regression or neural nets) on the input so that the model can predict the position of a record in the sorted dataset. The model can have inaccuracies, and thus the prediction has an associated error bound. During lookups, if the model-predicted position of the key is correct, the record is returned; if it is wrong, a local search is performed within the error bound. For example, if the predicted position is $\mathtt{pos}$ and the minimum and maximum error bounds are δ_{min} and δ_{max}, then upon a wrong prediction, a local search is performed between $\mathtt{pos} - \delta_{min}$ and $\mathtt{pos} + \delta_{max}$.

Learned indexes can make lookups significantly faster. Intuitively, a learned index turns a $O(\log\text{-}n)$ lookup of a B-tree into a $O(1)$ operation. Empirically, learned indexes provide $1.5\times$ – $3\times$ faster lookups than B-trees [77]. Given these benefits, we ask the following questions: *can learned indexes for LSMs make lookups faster? If yes, under what scenarios?*

Traditional learned indexes do not support updates because models learned over the existing data would change with modifications [38, 50, 77]. However, LSMs are attractive for their high performance in write-intensive workloads because they perform writes only sequentially. Thus, we examine: *how to realize the benefits of learned indexes while supporting writes for which LSMs are optimized?* We answer these two questions next.

2.2 Learned Indexes: a Good Match for LSMs?

In this section, we first analyze if learned indexes could be beneficial for LSMs and examine under what scenarios they can improve lookup performance. We then provide our intuition as to why learned indexes might be appropriate for LSMs even when allowing writes. We

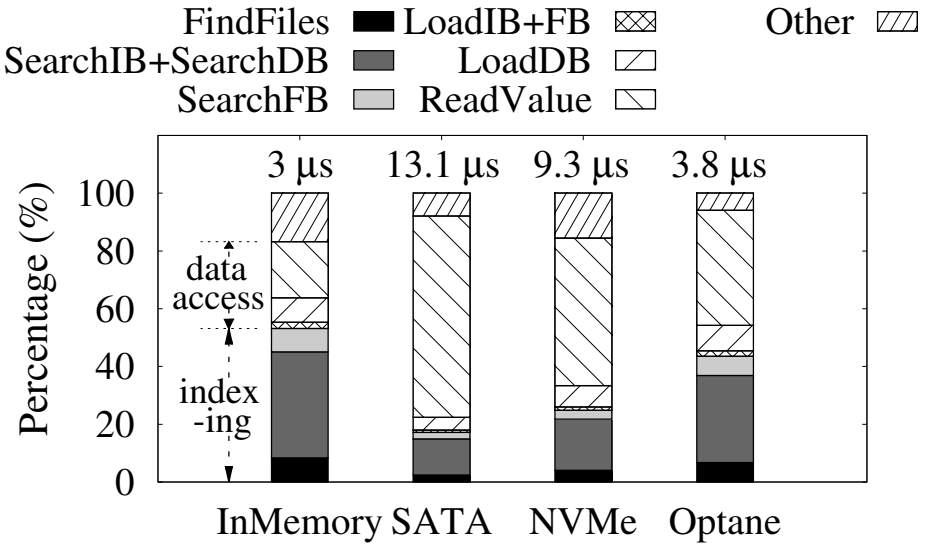


Figure 2.2: Lookup Latency Breakdown. *The figure shows the breakdown of lookup latency in WiscKey. The first bar shows the case when data is cached in memory from the beginning of the workload. The other three bars show the case where the dataset is originally stored on different types of SSDs. We perform 10M random lookups on the Amazon Reviews dataset [12]; the figure shows the breakdown of the average latency (shown at the top of each bar). The indexing portions are shown in solid colors; data access and other portions are shown in patterns.*

conduct an in-depth study based on measurements of how WiscKey functions internally under different workloads to validate our intuition. From our analysis, we derive a set of learning guidelines.

2.2.1 Learned Indexes: Beneficial Regimes

A lookup in LSM involves several indexing and data-access steps. Optimized indexes such as learned indexes can reduce the overheads of indexing but cannot reduce data-access costs. In WiscKey, learned indexes can thus potentially reduce the costs of indexing steps such as *FindFiles*, *SearchIB*, and *SearchDB*, while data-access costs (e.g., *ReadValue*) cannot be significantly reduced. As a result, learned indexes can improve overall lookup performance if indexing contributes to a sizable portion of the total lookup latency. We identify scenarios where this is the case.

First, when the dataset or a portion of it is cached in memory, data-access costs are

low, and so indexing costs become significant. Figure 2.2 shows the breakdown of lookup latencies in WiscKey. The first bar shows the case when the dataset is cached in memory; the second bar shows the case where the data is stored on a flash-based SATA SSD. With caching, data-access and indexing costs contribute almost equally to the latency. Thus, optimizing the indexing portion can reduce lookup latencies by about 2×. When the dataset is not cached, data-access costs dominate and thus optimizing indexes may yield smaller benefits (about 20%).

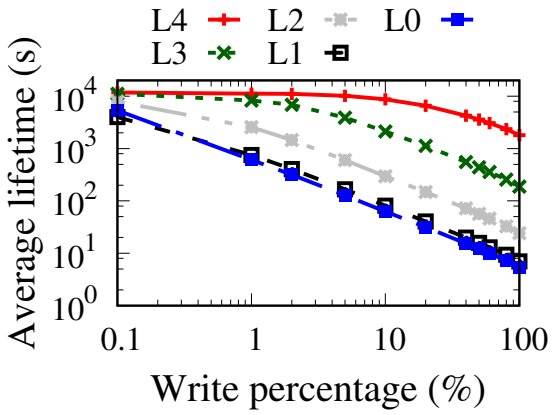
However, learned indexes are not limited to scenarios where data is cached in memory. They offer benefit on fast storage devices that are currently prevalent and can do more so on emerging faster devices. The last three bars in Figure 2.2 show the breakdown for three kinds of devices: flash-based SSDs over SATA and NVMe, and an Optane SSD. As the device gets faster, lookup latency (as shown at the top) decreases, but the fraction of time spent on indexing increases. For example, with SATA SSDs, indexing takes about 17% of the total time; in contrast, with Optane SSDs, indexing takes 44% and thus optimizing it with learned indexes can potentially improve performance by 1.8×. More importantly, the trend in storage performance favors the use of learned indexes. With storage performance increasing rapidly and emerging technologies like 3D Xpoint memory providing very low access latencies, indexing costs will dominate and thus learned indexes will yield increasing benefits.

Summary. Learned indexes could be beneficial when the database or a portion of it is cached in memory. With fast storage devices, regardless of caching, indexing contributes to a significant fraction of the lookup time; thus, learned indexes can prove useful in such cases. With storage devices getting faster, learned indexes will be even more beneficial.

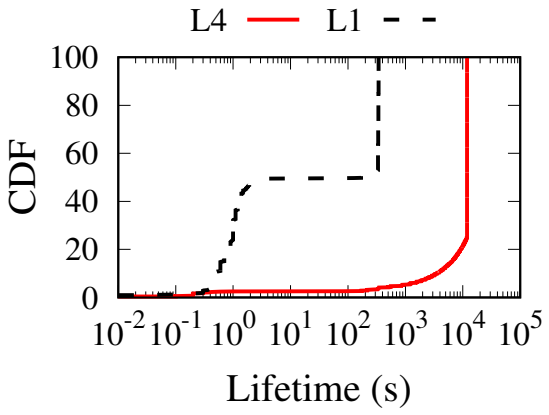
2.2.2 Learned Indexes with Writes

Learned indexes provide higher lookup performance compared to traditional indexes for read-only analytical workloads. However, a major drawback of learned indexes (as described in [77]) is that they do not support modifications such as inserts and updates [38, 50]. The main problem with modifications is that they alter the data distribution and so the models must be re-learned; for write-heavy workloads, models must be rebuilt often, incurring high overheads.

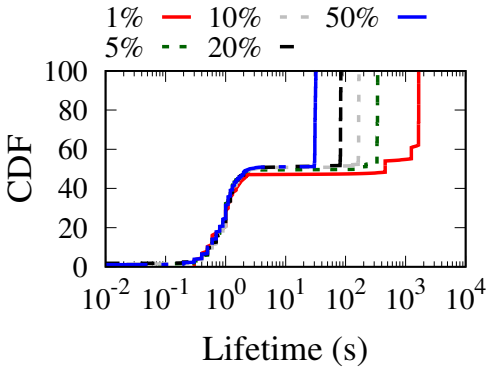
At first, it may seem like learned indexes are not a good match for write-heavy situations



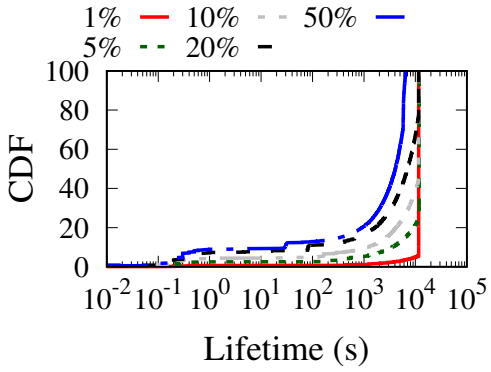
(a) Average lifetimes with varying write %



(b) Lifetime distribution with 5% writes



(i) Level 1



(ii) Level 4

(c) Lifetime distributions with varying write %

Figure 2.3: SSTable Lifetimes. *(a) shows the average lifetime of sstable files in levels* L_4 *to* L_0*. (b) shows the distribution of lifetimes of sstables in* L_1 *and* L_4 *with 5% writes. (c) shows the distribution of lifetimes of sstables for different write percentages in* L_1 *and* L_4*.*

for which LSMs are optimized. However, we observe that the design of LSMs fits well with learned indexes. Our key realization is that although updates can change portions of the LSM tree, a large part remains immutable. Specifically, newly modified items are buffered in the in-memory structures or present in the higher levels of the tree, while stable data resides at the lower levels. Given that a large fraction of the dataset resides in the stable, lower levels, lookups to this fraction can be made faster with no or few re-learnings. In contrast, learning in higher levels may be less beneficial: they change at a faster rate and thus must be re-learned often.

We also realize that the immutable nature of sstable files makes them an ideal unit for

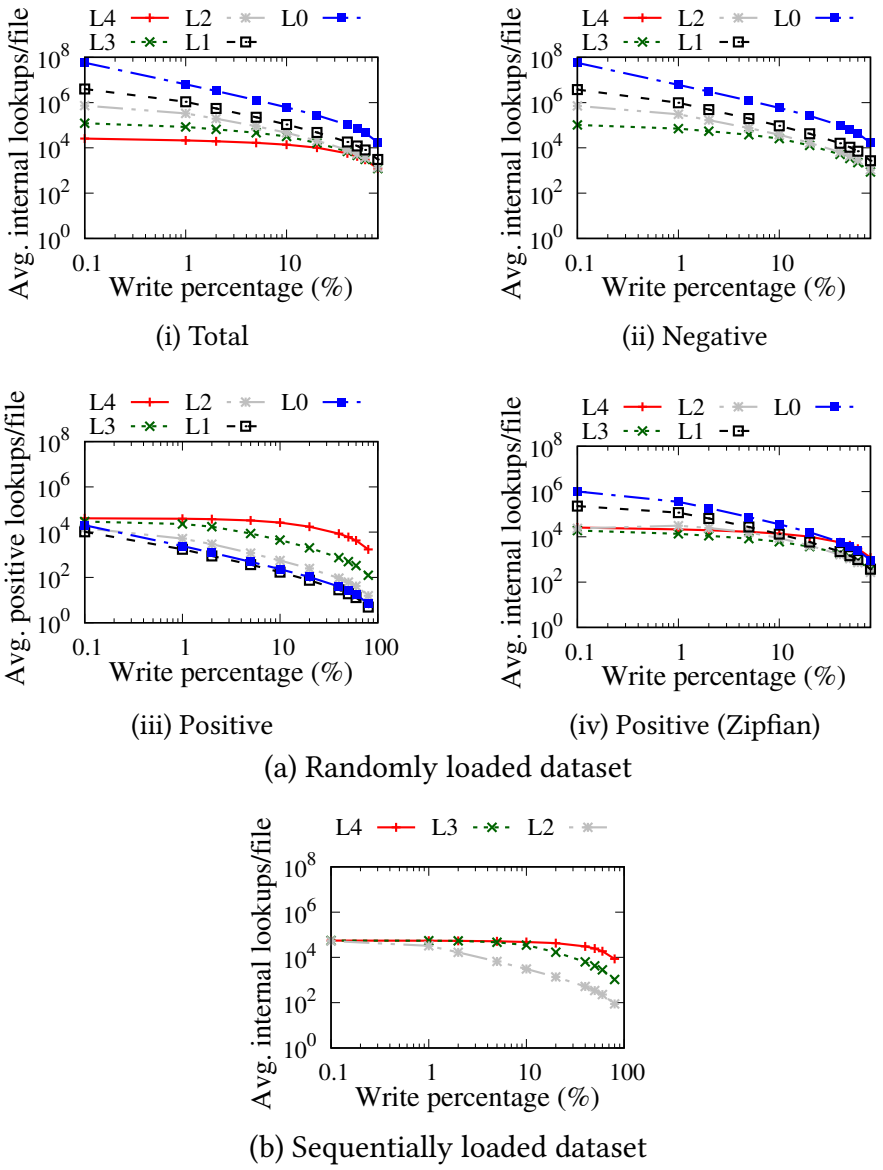


Figure 2.4: Number of Internal Lookups Per File. *(a)(i) shows the average internal lookups per file at each level for a randomly loaded dataset. (b) shows the same for sequentially loaded dataset. (a)(ii) and (a)(iii) show the negative and positive internal lookups for the randomly loaded case. (a)(iv) shows the positive internal lookups for the randomly loaded case when the workload distribution is Zipfian.*

learning. Once learned, these files are never updated and thus a model can be useful until the file is replaced. Further, the data within an sstable is sorted; such sorted data can be

learned using simple models. A level, which is a collection of many immutable files, can also be learned as a whole using simple models. The data in a level is also sorted: the individual sstables are sorted, and there are no overlapping key ranges across sstables.

We next conduct a series of in-depth measurements to validate our intuitions. Our experiments confirm that while a part of our intuition is indeed true, there are some subtleties (for example, in learning files at higher levels). Based on these experimental results, we formulate a set of *learning guidelines*: a few simple rules that an LSM that applies learned indexes should follow.

Experiments: goal and setup. The goal of our experiments is to determine how long a model will be useful and how often it will be useful. A model built for a sstable file is useful as long as the file exists; thus, we first measure and analyze sstable lifetimes. How often a model will be used is determined by how many internal lookups it serves; thus, we next measure the number of internal lookups to each file. Since models can also be built for entire levels, we finally measure level lifetimes as well. To perform our analysis, we run workloads with varying amounts of writes and reads, and measure the lifetimes and number of lookups. We conduct our experiments on WiscKey, but we believe our results are applicable to most LSM implementations. We first load the database with 256M key-value pairs. We then run a workload with a single rate-limited client that performs 200M operations, a fraction of which are writes. Our workload chooses keys uniformly at random. We have also run experiments on skewed workloads, which makes a difference for the number of internal lookups.

Lifetime of SSTables. To determine how long a model will be useful, we first measure and analyze the lifetimes of sstables. To do so, we track the creation and deletion times of all sstables. For files created during the load phase, we assign the workload-start time as their creation time; for other files, we record the actual creation times. If the file is deleted during the workload, then we calculate its exact lifetime. However, some files are not deleted by the end of the workload and we must estimate their lifetimes. If the files are created during load, we assign the workload duration as their lifetimes. If not, we estimate the lifetime of a file based on its creation time (c) and the total workload time (w); the lifetime of the file is at least $w - c$. We thus consider the lifetime distribution of other files at the same level that have a lifetime of at least $w - c$. We then pick a random lifetime in this distribution and assign it as this file's lifetime.

Figure 2.3(a) shows the average lifetime of sstable files at different levels. We make three

main observations. First, the average lifetime of sstable files at lower levels is greater than that of higher levels. Second, at lower percentages of writes, even files at higher levels have a considerable lifetime; for example, at 5% writes, files at L_0 live for about 2 minutes on an average. Files at lower levels live much longer; files at L_4 live about 150 minutes. Third, although the average lifetime of files reduces with more writes, even with a high amount of writes, files at lower levels live for a long period. For instance, with 50% writes, files at L_4 live for about 60 minutes. In contrast, files at higher level live only for a few seconds; for example, an L_0 file lives only about 10 seconds.

We now take a closer look at the lifetime distribution. Figure 2.3(b) shows the distributions for L_1 and L_4 files with 5% writes. We first note that some files are very short-lived, while some are long-lived. For example, in L_1, the lifetime of about 50% of the files is only about 2.5 seconds. If files cross this threshold, they tend to live for much longer times; almost all of the remaining L_1 files live over five minutes.

Surprisingly, even at L_4, which has a higher average lifetime for files, a few files are very short-lived. We observe that about 2% of L_4 files live less than a second. We find that there are two reasons why a few files live for a very short time. First, compaction of a L_i file creates a new file in L_{i+1} which is again immediately chosen for compaction to the next level. Second, compaction of a L_i file creates a new file in L_{i+1}, which has overlapping key ranges with the next file that is being compacted from L_i. Figure 2.3(c) shows that this pattern holds for other percentages of writes too. We observed that this holds for other levels as well. From the above observations, we arrive at our first two learning guidelines.

Learning guideline - 1: Favor learning files at lower levels. Files at lower levels live for a long period even for high write percentages; thus, models for these files can be used for a long time and need not be rebuilt often.

Learning guideline - 2: Wait before learning a file. A few files are very short-lived, even at lower levels. Thus, learning must be invoked only after a file has lived up to a threshold lifetime after which it is highly likely to live for a long time.

Internal Lookups at Different Levels. To determine how many times a model will be used, we analyze the number of lookups served by the sstable files. We run a workload and measure the number of lookups served by files at each level and plot the average number of lookups per file at each level. Figure 2.4(a) shows the result when the dataset is loaded in an uniform random order. The number of internal lookups is higher for higher levels, although a large fraction of data resides at lower levels. This is because, at higher levels,

many internal lookups are negative, as shown in Figure 2.4(a)(ii). The number of positive internal lookups is as expected: higher in lower levels as shown in Figure 2.4(a)(iii). This result shows that files at higher levels serve many negative lookups and thus are worth optimizing. While bloom filters may already make these negative lookups faster, the index block still needs to be searched (before the filter query).

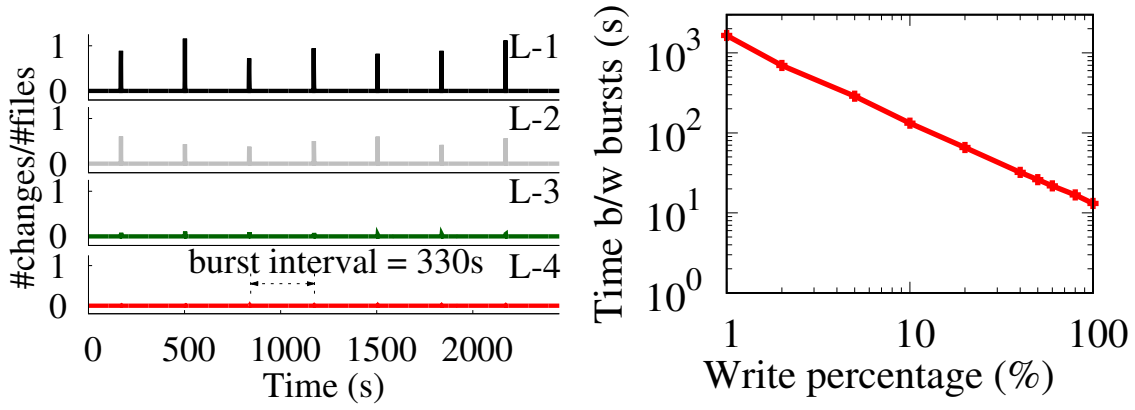
We also conduct the same experiment with another workload where the access pattern follows a zipfian distribution (most requests are to a small set of keys). Most of the results exhibit the same trend as the random workload except for the number of positive internal lookups, as shown in Figure 2.4(a)(iv). Under the zipfian workload, higher level files also serve numerous positive lookups, because the workload accesses a small set of keys which are often updated and thus stored in higher levels.

Figure 2.4(b) shows the result when the dataset is sequentially loaded, i.e., keys are inserted in ascending order. In contrast to the randomly-loaded case, there are no negative lookups because keys of different sstable files do not overlap even across levels; the *FindFiles* step finds the one file that may contain the key. Thus, lower levels serve more lookups and can have more benefits from learning. From these observations, we arrive at the next two learning guidelines.

Learning guideline - 3: Do not neglect files at higher levels. Although files at lower levels live longer and serve many lookups, files at higher levels can still serve many negative lookups and in some cases, even many positive lookups. Thus, learning files at higher levels can make both internal lookups faster.

Learning guideline - 4: Be workload- and data-aware. Although most data resides in lower levels, if the workload does not lookup that data, learning those levels will yield less benefit; learning thus must be aware of the workload. Further, the order in which the data is loaded influences which levels receive a large fraction of internal lookups; thus, the system must also be data-aware. The amount of internal lookups acts as a proxy for both the workload and load order. Based on the amount of internal lookups, the system must dynamically decide whether to learn a file or not.

Lifetime of Levels. Given that a level as a whole can also be learned, we now measure and analyze the lifetimes of levels. Level learning cannot be applied at L_0 because it is unsorted: files in L_0 can have overlapping key ranges. Once a level is learned, any change to the level causes a re-learning. A level changes when new sstables are created at that level, or existing ones are deleted. Thus, intuitively, a level would remain unchanged for an equal or shorter



(a) Timeline of changes

(b) Time between bursts for L4

Figure 2.5: Changes at Levels. *(a) shows the timeline of file creations and deletions at different levels. Note that #changes/#files is higher than 1 in* L_1 *as there are more creations and deletions than the number of files. (b) shows the time between bursts for L4 for different write percentages.*

duration than the individual sstables. However, learning at the granularity of a level has the benefit that the candidate sstables need not be found in a separate step; instead, upon a lookup, the model just outputs the sstable and the offset within it.

We examine the changes to a level by plotting the timeline of file creations and deletions at L_1, L_2, L_3, and L_4 in Figure 2.5(a) for a 5%-write workload; we do not show L_0 for the reason above. On the y-axis, we plot the number of changes divided by the total files present at that level. A value of 0 means there are no changes to the level; a model learned for the level can be used as long as the value remains 0. A value greater than 0 means that there are changes in the level and thus the model has to re-learned. Higher values denote a larger fraction of files changed.

First, as expected, we observe that the fraction of files that change reduces as we go down the levels because lower levels hold a large volume of data in many files, confirming our intuition. We also observe that changes to levels arrive in bursts. These bursts are caused by compactions that cause many files at a level to be rewritten. Further, these bursts occur at almost the same time across different levels. The reason behind this is that for the dataset we use, levels L_0 through L_3 are full and thus any compaction at one layer results in cascading compactions which finally settle at the non-full L_4 level. The levels remain static between these bursts. The duration for which the levels remain static is longer with a lower amount of writes; for example, with 5% writes, as shown in the figure, this period is about

5 minutes. However, as the amount of writes increases, the lifetime of a level reduces as shown in Figure 2.5(b); for instance, with 50% writes, the lifetime of L_4 reduces to about 25 seconds. From these observations, we arrive at our final learning guideline.

Learning guideline - 5: Do not learn levels for write-heavy workloads. Learning a level as a whole might be more appropriate when the amount of writes is very low or if the workload is read-only. For write-heavy workloads, level lifetimes are very short and thus will induce frequent re-learnings.

Summary. We analyzed how LSMs behave internally by measuring and analyzing the lifetimes of sstable files and levels, and the amount of lookups served by files at different levels. From our analysis, we derived five learning guidelines. We next describe how we incorporate the learning guidelines in an LSM-based storage system.

2.3 Bourbon Design

We now describe Bourbon, an LSM-based store that uses learning to make indexing faster. We first describe the model that Bourbon uses to learn the data (§2.3.1). Then, we discuss how Bourbon supports variable-size values (§2.3.2) and its basic learning strategy (§2.3.3). We finally explain Bourbon's cost-benefit analyzer that dynamically makes learning decisions to maximize benefit while reducing cost (§2.3.4).

2.3.1 Learning the Data

As we discussed, data can be learned at two granularities: individual sstables or levels. Both these entities are sorted datasets. The goal of a model that tries to learn the data is to predict the location of a key in such a sorted dataset. For example, if the model is constructed for a sstable file, it would predict the file offset given a key. Similarly, a level model would output the target sstable file and the offset within it.

Our requirements for a model is that it must have low overheads during learning and during lookups. Further, we would like the space overheads of the model to be small. We find that piecewise linear regression (PLR) [7, 71] satisfies these requirements well; thus, Bourbon uses PLR to model the data. The intuition behind PLR is to represent a sorted dataset with a number of line segments. PLR constructs a model with an error bound; that is, each data point d, which turns into the key of a key-value pair in the database, is guaranteed

to lie within the range $[d_{pos} - \delta, d_{pos} + \delta]$, where d_{pos} is the predicted position of d in the dataset and δ is the error bound specified beforehand.

To train the PLR model, BOURBON uses the Greedy-PLR algorithm [162]. Greedy-PLR processes the data points one at a time; if a data point cannot be added to the current line segment without violating the error bound, then a new line segment is created and the data point is added to it. At the end, Greedy-PLR produces a set of line segments that represents the data. Greedy-PLR runs in linear time with respect to the number of data points.

Once the model is learned, inference is quick: first, the correct line segment that contains the key is found (using binary search); within that line segment, the position of the target key is obtained by multiplying the key with the line's slope and adding the intercept. If the key is not present in the predicted position, a local search is done in the range determined by the error bound. Thus, lookups take $O(\log s)$ time, where s is the number of segments, in addition to a constant time to do the local search. The space overheads of PLR are small: a few tens of bytes for every line segment.

Other models or algorithms such as RMI [77], PGM-Index [48], or splines [73] may also be suitable for LSMs and may offer more benefits than PLR. We leave their exploration within LSMs for future work.

2.3.2 Supporting Variable-size Values

Learning a model that predicts the offset of a key-value pair is much easier if the key-value pairs are the same size. The model then can multiply the predicted position of a key by the size of the pair to produce the final offset. However, many systems allow keys and values to be of arbitrary sizes.

BOURBON requires keys to be of a fixed size, while values can be of any size. We believe this is a reasonable design choice because most datasets have fixed-size keys (e.g., user-ids are usually 16 bytes), while value sizes vary significantly. Even if keys vary in size, they can be padded to make all keys of the same size. BOURBON supports variable-size values by borrowing the idea of key-value separation from WiscKey [98]. With key-value separation, sstables in BOURBON just contain the keys and the pointer to the values; values are maintained in the value log separately. With this, BOURBON obtains the offset of a required key-value pair by getting the predicted position from the model and multiplying it with the record size (which is $keysize + pointersize$.) The value pointer serves as the

Workload	Baseline time (s)	File model		Level model	
		Time(s)	% model	Time(s)	% model
Mixed: Write-heavy	82.6	71.5 (1.16 ×)	74.2	95.1 (0.87 ×)	1.5
Mixed: Read-heavy	89.2	62.05 (1.44 ×)	99.8	74.3 (1.2 ×)	21.4
Read-only	48.4	27.2 (1.78 ×)	100	25.2 (1.92 ×)	100

Table 2.1: File vs. Level Learning. *The table compares the time to perform 10M operations in baseline WiscKey, file-learning, and level-learning. The numbers within the parentheses show the improvements over baseline. The table also shows the percentage of lookups that take the model path; remaining take the original path because the models are not rebuilt yet.*

offset into the value log from which the value is finally read.

2.3.3 Level vs. File Learning

Bourbon can learn individual sstables files or entire levels. Our analysis in the previous section showed that files live longer than levels under write-heavy workloads, hinting that learning at the file granularity might be the best choice. We now closely examine this tradeoff to design Bourbon's basic learning strategy. To do so, we compare the performance of file learning and level learning for different workloads. We initially load a dataset and build the models. For the read-only workload, the models need not be re-learned. In the mixed workloads, the models are re-learned as data changes. The results are shown in Table 2.1.

For mixed workloads, level learning performs worse than file learning. For a write-heavy (50%-write) workload, with level learning, only a small percentage of internal lookups are able to use the model because with a steady stream of incoming writes, the system is unable to learn the levels. Only a mere 1.5% of internal lookups take the model path; these lookups are the ones performed just after loading the data and when the initial level models are available. We observe that all the 66 attempted level learnings failed because the level changed before the learning completed. Because of the additional cost of re-learnings, level learning performs even worse than the baseline with 50% writes. On the other hand, with file models, a large fraction of lookups benefit from the models and thus file learning performs better than the baseline. For read-heavy mixed workload (5%), although level learning has benefits over the baseline, it performs worse than file learning for the same reasons above.

Level learning can be beneficial for read-only settings: as shown in the table, level learning provides 10% improvements over file learning. Thus, deployments that have only

read-only workloads can benefit from level learning. Given that BOURBON's goal is to provide faster lookups while supporting writes, levels are not an appropriate choice of granularity for learning. Thus, BOURBON uses file learning by default. However, BOURBON supports level learning as a configuration option that can be useful in read-only scenarios.

2.3.4 Cost vs. Benefit Analyzer

Before learning a file, BOURBON must ensure that the time spent in learning is worthwhile. If a file is short-lived, then the time spent learning that file wastes resources. Such a file will serve few lookups and thus the model would have little benefit. Thus, to decide whether or not to learn a file, BOURBON implements an online cost vs. benefit analysis.

2.3.4.1 Wait Before Learning

As our analysis showed, even in the lower levels, many files are short-lived. To avoid the cost of learning short-lived files, BOURBON waits for a time threshold, T_{wait}, before learning a file. The exact value of T_{wait} presents a cost vs. performance tradeoff. A very low T_{wait} leads to some short-lived files still being learned, incurring overheads; a large value causes many lookups to take the baseline path (because there is no model built yet), thus missing opportunities to make lookups faster. BOURBON sets the value of T_{wait} to the time it takes to learn a file. Our approach is never more than a factor of two worse than the optimal solution, where the optimal solution knows apriori the lifetime and decides to either immediately or never learn the file (i.e., it is two-competitive [68]). Through measurements, we found that the maximum time to learn a file (which is at most ~4MB in size) is around 40 ms on our experimental setup. We conservatively set T_{wait} to be 50 ms in BOURBON's implementation.

2.3.4.2 To Learn a File or Not

BOURBON waits for T_{wait} before learning a file. However, learning a file even if it lives for a long time may not be beneficial. For example, our analysis shows that although lower-level files live longer, for some workloads and datasets, they serve relatively fewer lookups than higher-level files; higher-level files, although short-lived, serve a large percentage of negative internal lookups in some scenarios. BOURBON, thus, must consider the potential benefits that a model can bring, in addition to considering the cost to build the model. It is

profitable to learn a file if the benefit of the model (B_{model}) outweighs the cost to build the model (C_{model}).

***Estimating* $\mathbf{C_{model}}$.** One way to estimate C_{model} is to assume that the learning is completely performed in the background and will not affect the rest of the system; i.e., C_{model} is 0. This is true if there are many idle cores which the learning threads can utilize and thus do not interfere with the foreground tasks (e.g., the workload) or other background tasks (e.g., compaction). However, BOURBON takes a conservative approach and assumes that the learning threads will interfere and slow down the other parts of the system. As a result, BOURBON assumes C_{model} to be equal to T_{build}. We define T_{build} as the time to train the PLR model for a file. We find that this time is linearly proportional to the number of data points in the file. We calculate T_{build} for a file by multiplying the average time to a train a data point (measured offline) and the number of data points in the file.

***Estimating* $\mathbf{B_{model}}$.** Estimating the potential benefit of learning a file, B_{model}, is more involved. Intuitively, the benefit offered by the model for an internal lookup is given by $T_b - T_m$, where T_b and T_m are the average times for the lookup in baseline and model paths, respectively. If the file serves N lookups in its lifetime, the net benefit of the model is: $B_{model} = (T_b - T_m) * N$. We divide the internal lookups into negative and positive because most negative lookups terminate at the filter, whereas positive ones do not; thus,

$$B_{model} = ((T_{n.b} - T_{n.m}) * N_n) + ((T_{p.b} - T_{p.m}) * N_p)$$

where N_n and N_p are the number of negative and positive internal lookups, respectively. $T_{n.b}$ and $T_{p.b}$ are the time in the baseline path for a negative and a positive lookup, respectively; $T_{n.m}$ and $T_{p.m}$ are the model counterparts.

B_{model} for a file cannot be calculated without knowing the number of lookups that the file will serve or how much time the lookups will take. The analyzer, to estimate these quantities, maintains statistics of files that have lived their lifetime, i.e., files that were created, served many lookups, and then were replaced. To estimate these quantities for a file F, the analyzer uses the statistics of other files at the same level as F; we consider statistics only at the same level because these statistics vary significantly across levels.

Recall that BOURBON waits before learning a file. During this time, the lookups are served in the baseline path. BOURBON uses the time taken for these lookups to estimate $T_{n.b}$ and $T_{p.b}$. Next, $T_{n.m}$ and $T_{p.m}$ are estimated as the average negative and positive model

lookup times of other files at the same level. Finally, N_n and N_p are estimated as follows. The analyzer first takes the average negative and positive lookups for other files in that level; then, it is scaled by a factor $f = s/\bar{s_l}$, where s if the size of the file and $\bar{s_l}$ is the average file size at this level. While estimating the above quantities, BOURBON filters out very short-lived files.

While bootstrapping, the analyzer might not have enough statistics collected. Therefore, initially, BOURBON runs in an always-learn mode (with T_{wait} still in place.) Once enough statistics are collected, the analyzer performs the cost vs. benefit analysis and chooses to learn a file if $C_{model} < B_{model}$, i.e., benefit of a model outweighs the cost. If multiple files are chosen to be learned at the same time, BOURBON puts them in a max priority queue ordered by $B_{model} - C_{model}$, thus prioritizing files that would deliver the most benefit.

Our cost-benefit analyzer adopts a simple scheme of using average statistics of other files at the same level. While this approach has worked well in our initial prototype, using more sophisticated statistics and considering workload distributions (e.g., to account for keys with different popularity) could be more beneficial. We leave such exploration for future work.

2.3.5 Bourbon: Putting it All Together

We describe how the different pieces of BOURBON work together. Figure 2.6 shows the path of lookups in BOURBON. As shown in (a), lookups can either be processed via the model (if the target file is already learned), or in the baseline path (if the model is not built yet.) The baseline path in BOURBON is similar to the one shown in Figure 2.1 for LevelDB, except that BOURBON stores the values separately and so *ReadValue* reads the value from the log.

Once BOURBON learns a sstable file, lookups to that file will be processed via the learned model as shown in Figure 2.6(b). ① *FindFiles*: BOURBON finds the candidate sstables; this step required because BOURBON uses file learning. ② *LoadIB+FB*: BOURBON loads the index and filter blocks; these blocks are likely to be already cached. ③ *ModelLookup*: BOURBON performs a look up for the desired key k in the candidate sstable's model. The model outputs a predicted position of k within the file (pos) and the error bound (δ). From this, BOURBON calculates the data block that contains records $pos - \delta$ through $pos + \delta$.[†] ④ *SearchFB*: The

[†]Sometimes, records $pos - \delta$ through $pos + \delta$ span multiple data blocks; in such cases, BOURBON consults the index block (which specifies the maximum key in each data block) to find the data block for pos.

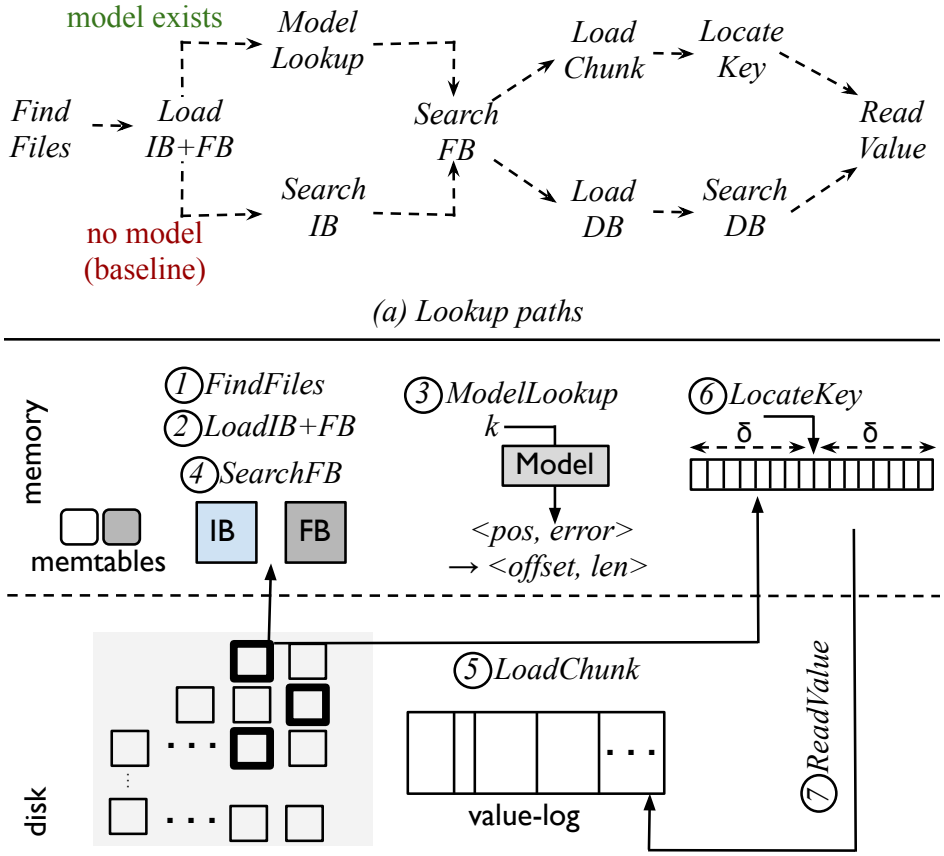


Figure 2.6: BOURBON Lookups. *(a) shows that lookups can take two different paths: when the model is available (shown at the top), and when the model is not learned yet and so lookups take the baseline path (bottom); some steps are common to both paths. (b) shows the detailed steps for a lookup via a model; we show the case where models are built for files.*

filter for that block is queried to check if k is present. If present, BOURBON calculates the range of bytes of the block that must be loaded; this is simple because keys and pointers to values are of fixed size. ⑤ *LoadChunk*: The byte range is loaded. ⑥ *LocateKey*: The key is located in the loaded chunk. The key will likely be present in the predicted position (the midpoint of the loaded chunk); if not, BOURBON performs a binary search in the chunk. ⑦ *ReadValue*: The value is read from the value log using the pointer.

Possible improvements. Although BOURBON's implementation is highly-optimized and provides many features common to real systems, it lacks a few features. For example, in the current implementation, we do not support string keys and key compression (although

we support value compression). For string keys, one approach we plan to explore is to treat strings as base-64 integers and convert them into 64-bit integers, which could then adopt the same learning approach described herein. While this approach may work well for small keys, large keys may require larger integers (with more than 64 bits) and thus efficient large-integer math is likely essential. Also, BOURBON does not support adaptive switching between level and file models; it is a static configuration. We leave supporting these features to future work.

2.4 Evaluation

To evaluate BOURBON, we answer the following questions to demonstrate BOURBONś consistent benefit in various micro- and macro-benchmarks:

- Q: Which portions of lookup does BOURBON optimize?

  A: SearchIB, SearchDB, and LoadData. (§2.4.1)

- Q: How does BOURBON perform with models available and no writes? How does performance change with datasets, load orders, and request distributions?

  A: BOURBON's model consistently provides 1.6× gain on average for various read-only workloads. (§2.4.2)

- Q: How does BOURBON perform with range queries?

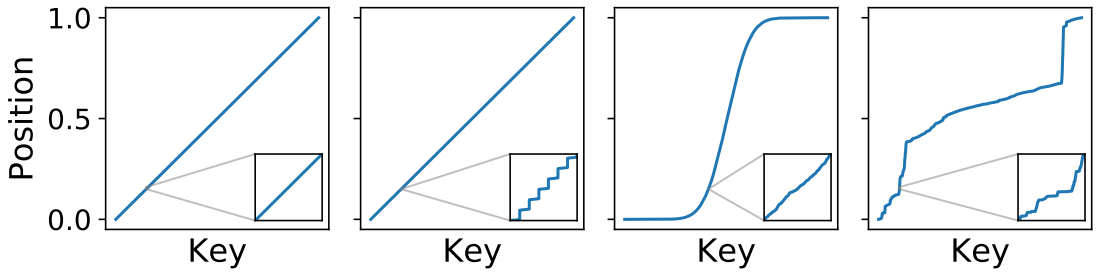
  A: BOURBON significantly accelerates the indexing performance, but has no effect on the following scans. (§2.4.3)

- Q: In the presence of writes, how does BOURBON's cost-benefit analyzer perform compared to other approaches that always or never re-learn?

  A: BOURBON provides benefits close to aggressive online learning with significantly lower costs. (§2.4.4)

- Q: Does BOURBON perform well on real benchmarks?

  A: BOURBON shows consistent gain for reads on real benchmarks and does not affect writes. (§2.4.5)



(a) Linear **(b)** Seg10% **(c)** Normal **(d)** OSM

Figure 2.7: Datasets. *The figure shows the cumulative distribution functions (CDF) of three synthetic datasets (linear, segmented-10%, and normal) and one real-world dataset (OpenStreetMaps). Each dataset is magnified around the 15% percentile to show a detailed view of its distribution.*

- Q: Is BOURBON beneficial when data is on storage?

  A: BOURBON offers an average gain of 1.15× with fast storage devices. (§2.4.6)

- Q: Is BOURBON beneficial with limited memory and no fast storage devices?

  A: BOURBON is beneficial for skewed workloads with limited memory and no fast storage devices. (§2.4.7)

- Q: What are the error and space tradeoffs of BOURBON?

  A: The model's memory consumption is higher if a lower guaranteed error bound is selected. $\delta = 8$ is optimal for our workloads. (§2.4.8)

Setup. We run our experiments on a 20-core Intel Xeon CPU E5-2660 machine with 160-GB memory and a 480-GB SATA SSD. We use 16B integer keys and 64B values, and set the error bound of BOURBON's PLR as 8. Unless specified, our workloads perform 10M operations. We use a variety of datasets. We construct four synthetic datasets: linear, segmented-1%, segmented-10% , and normal, each with 64M key-value pairs. In the linear dataset, keys are all consecutive. In the seg-1% dataset, there is a gap after a consecutive segment of 100 keys (i.e., every 1% causes a new segment). The segmented-10% dataset is similar, but there is a gap after 10 consecutive keys. We generate the normal dataset by sampling 64M unique values from the standard normal distribution $\mathrm{N}(0,1)$ and scale to integers. We also use two real-world datasets: Amazon reviews (AR) [12] and New York OpenStreetMaps (OSM) [4]. AR and OSM have 33.5M and 21.9M key-value pairs, respectively. These datasets

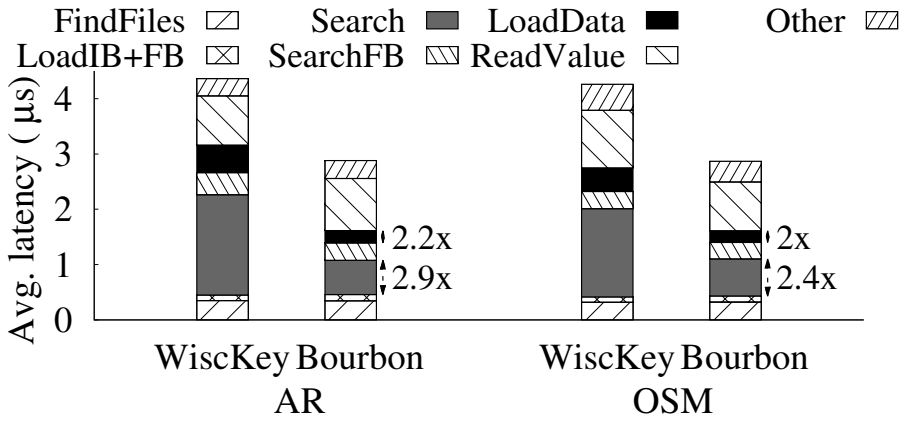


Figure 2.8: Latency Breakdown. *The figure shows latency breakdown for WiscKey and* Bourbon*. Search denotes SearchIB and SearchDB in WiscKey; the same denotes ModelLookup and LocateKey in* Bourbon*. LoadData denotes LoadDB in WiscKey; the same denotes LoadChunk in* Bourbon*. These two steps are optimized by* Bourbon *and are shown in solid colors; the number next to a step shows the factor by which it is made faster in* Bourbon*.*

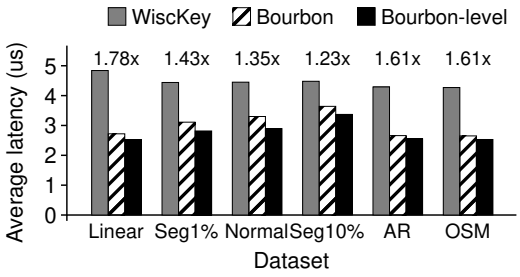
vary widely in how the keys are distributed. Figure 2.7 shows the distribution for a few datasets. The databases for these datasets vary from 2 GB to 5 GB according to the number of key-value pairs in the datasets. Most of our experiments focus on the case where the data resides in memory; however, we also analyze cases where data is present on storage.

2.4.1 Which Portions does Bourbon Optimize?

We first analyze which portions of the lookup Bourbon optimizes. We perform 10M random lookups on the AR and OSM datasets and show the latency breakdown in Figure 2.8. As expected, Bourbon reduces the time spent in indexing. The portion marked *Search* in the figure corresponds to *SearchIB* and *SearchDB* in the baseline, versus *ModelLookup* and *LocateKey* in Bourbon. The steps in Bourbon have lower latency than their baseline counterparts. Interestingly, Bourbon reduces data-access costs too, because Bourbon loads a smaller byte range than the entire block loaded by the baseline.

2.4.2 Performance under No Writes

We next analyze Bourbon's performance when the models are already built and there are no updates. For each experiment, we load a dataset and allow the system to build the models;



(a) Average lookup latency

Dataset	#segs	#segs/keys	latency (µs)
Linear	900	<0.01%	2.72
Seg1%	640K	1.0%	3.11
Normal	705K	1.1%	3.3
Seg10%	6.4M	10%	3.64
AR	129K	0.20%	2.66
OSM	295K	0.46%	2.65

(b) Number of segments

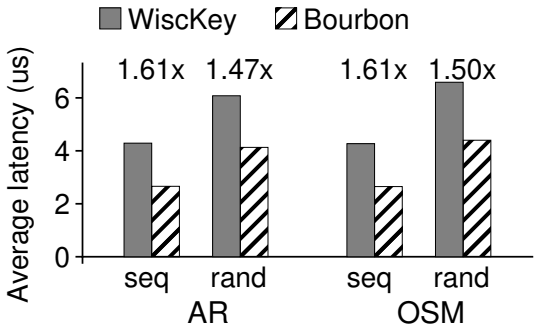
Figure 2.9: Datasets. *(a) compares the average lookup latencies of* Bourbon*,* Bourbon*-level, and WiscKey for different datasets; the numbers on the top show the improvements of* Bourbon *over WiscKey. (b) shows the number of segments for different datasets in* Bourbon*.*

during the workload, we issue only lookups.

2.4.2.1 Datasets

To analyze how the performance is influenced by the dataset, we run the workload on all six datasets and compare Bourbon's lookup performance against WiscKey. Figure 2.9 show the results. As shown in 2.9(a), Bourbon is faster than WiscKey for all datasets; depending upon the dataset, the improvements vary (1.23× to 1.78×). Bourbon provides the most benefit for the linear dataset because it has the smallest number of segments (one per model); with fewer segments, fewer searches are needed to find the target line segment. From 2.9(b), we observe that latencies increase with the number of segments (e.g., latency of seg-1% is greater than that of linear). We cannot compare the number of segments in AR and OSM with others because the size of these datasets is significantly different.

Level learning. Given that level learning is suitable for read-only scenarios, we configure Bourbon to use level learning and analyze its performance. As shown in Figure 2.9(a), Bourbon-level is 1.33× – 1.92× faster than the baseline. Bourbon-level offers more benefits than Bourbon because a level-model lookup is faster than finding the candidate sstables and then doing a file-model lookup. This confirms that Bourbon-level is an attractive option for read-only scenarios. However, since level models only provide benefits for read-only workloads and give at most 10% improvement compared to file models, we focus on Bourbon with file learning for our remaining experiments.



(a) Average latency

Dataset	Positive		Negative	
	#	Speedup	#	Speedup
AR	10M	2.15×	23M	1.83×
OSM	10M	1.99×	22M	1.82×

(b) Positive vs. negative internal lookups for randomly loaded case

Figure 2.10: Load Orders. *(a) shows the performance for AR and OSM datasets for sequential (seq) and random (rand) load orders. (b) compares the speedup of positive and negative internal lookups.*

2.4.2.2 Load Orders

We now explore how the order in which the data is loaded affects performance. For this experiment, we use the AR and OSM datasets and load them in two ways: sequential (keys are inserted in ascending order) and random (keys are inserted in an uniformly random order). With sequential loading, sstables do not have overlapping key ranges even across levels; whereas, with random loading, sstables at one level can overlap with sstables at other levels.

Figure 2.10 shows the result. First, regardless of the load order, BOURBON offers significant benefit over baseline (1.47× − 1.61×). Second, the average lookup latencies increase in the randomly-loaded case compared to the sequential case (e.g., 6µs vs. 4µs in WiscKey for the AR dataset). This is because while there are no negative internal lookups in the sequential case, there are many (23M) negative lookups in the random case (as shown in 2.10(b)). Thus, with random load, the total number of internal lookups increases by 3×, increasing lookup latencies.

Next, we note that the speedup over baseline in the random case is less than that of the sequential case (e.g., 1.47× vs. 1.61× for AR). Although BOURBON optimizes both positive and negative internal lookups, the gain for negative lookups is smaller (as shown in 2.10(b)). This is because most negative lookups in the baseline and BOURBON end just after the filter is queried (filter indicates absence); the data block is not loaded or searched. Given there are more negative than positive lookups, BOURBON offers less speedup than the sequential

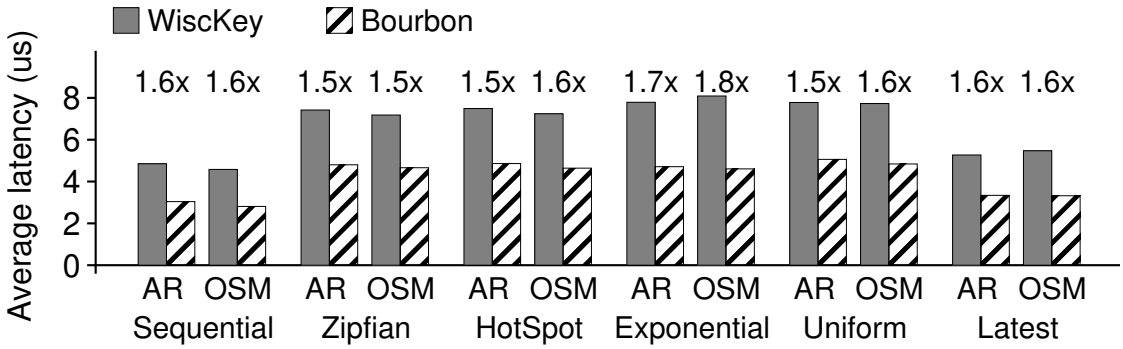


Figure 2.11: Request Distributions. *The figure shows the average lookup latencies of different request distributions from AR and OSM datasets.*

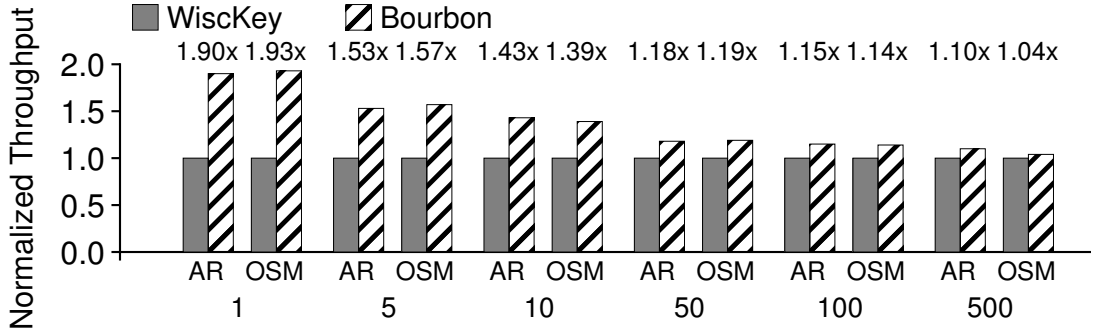


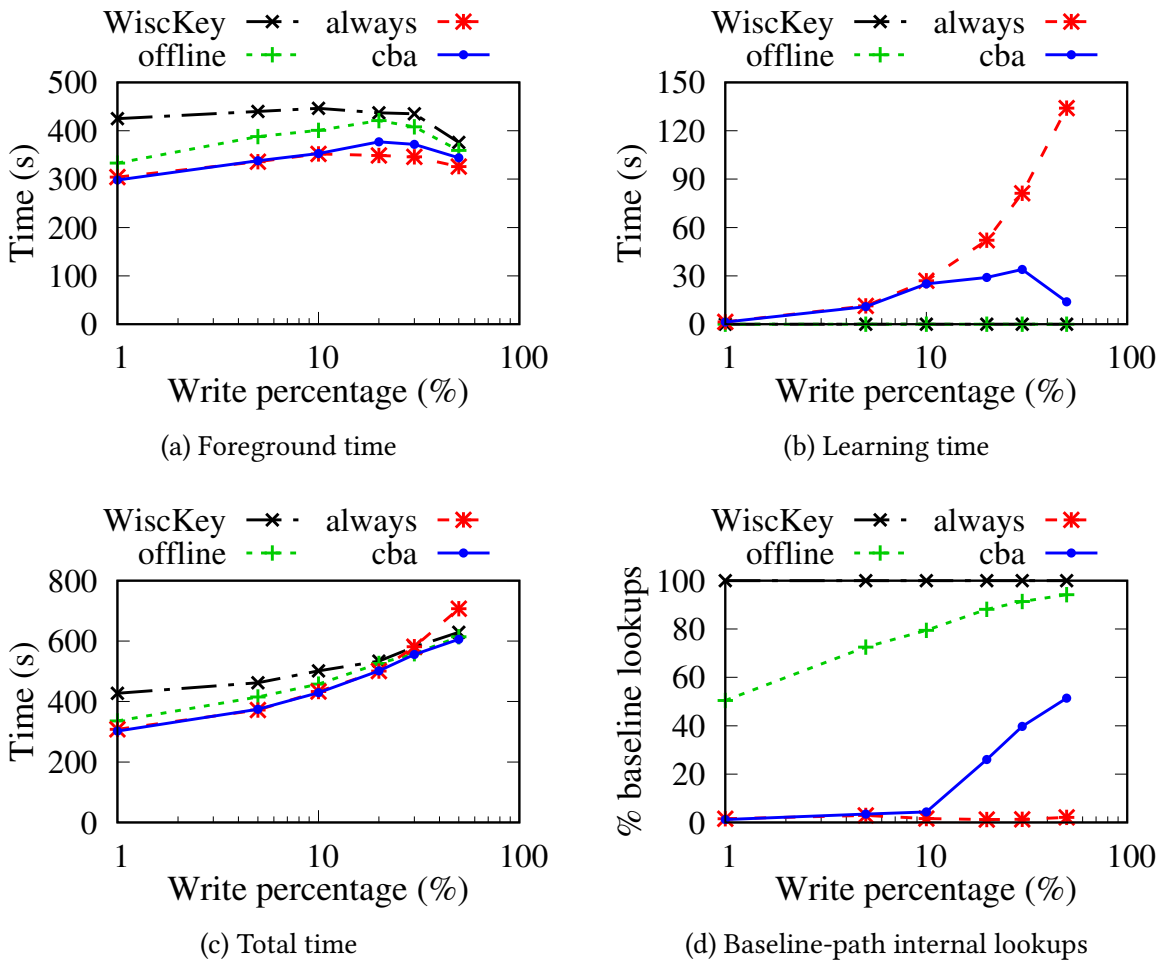
Figure 2.12: Range Queries. *The figure shows the normalized throughput of range queries with different range lengths from AR and OSM datasets.*

case. However, this speedup is still significant (1.47×).

2.4.2.3 Request Distributions

Next, we analyze how request distributions affect Bourbon's performance. We measure the lookup latencies under six request distributions: sequential, zipfian, hotspot, exponential, uniform, and latest. We first randomly load the AR and OSM datasets and then run the workloads; thus, the data can be segmented and there can be many negative internal lookups. As shown in Figure 2.11, Bourbon makes lookups faster by 1.54× – 1.76× than the baseline. Overall, Bourbon reduces latencies regardless of request distributions.

Read-only performance summary. When the models are already built and when there are no writes, Bourbon provides significant speedup over baseline for a variety of datasets, load orders, and request distributions.



(a) Foreground time

(b) Learning time

(c) Total time

(d) Baseline-path internal lookups

Figure 2.13: Mixed Workloads. *(a) compares the foreground times of WiscKey, BOURBON-offline (offline), BOURBON-always (always), and BOURBON-cba (cba); (b) and (c) compare the learning time and total time, respectively; (d) shows the fraction of internal lookups that take the baseline path.*

2.4.3 Range Queries

We next analyze how BOURBON performs on range queries. We perform 1M range queries on the AR and OSM datasets with various range lengths. Figure 2.12 shows the throughput of BOURBON normalized to that of WiscKey. With short ranges, where the indexing cost (i.e., the cost to locate the first key of the range) is dominant, BOURBON offers the most benefit. For example, with a range length of 1 on the AR dataset, BOURBON is 1.90× faster than WiscKey. The gains drop as the range length increases; for example, BOURBON is only 1.15×

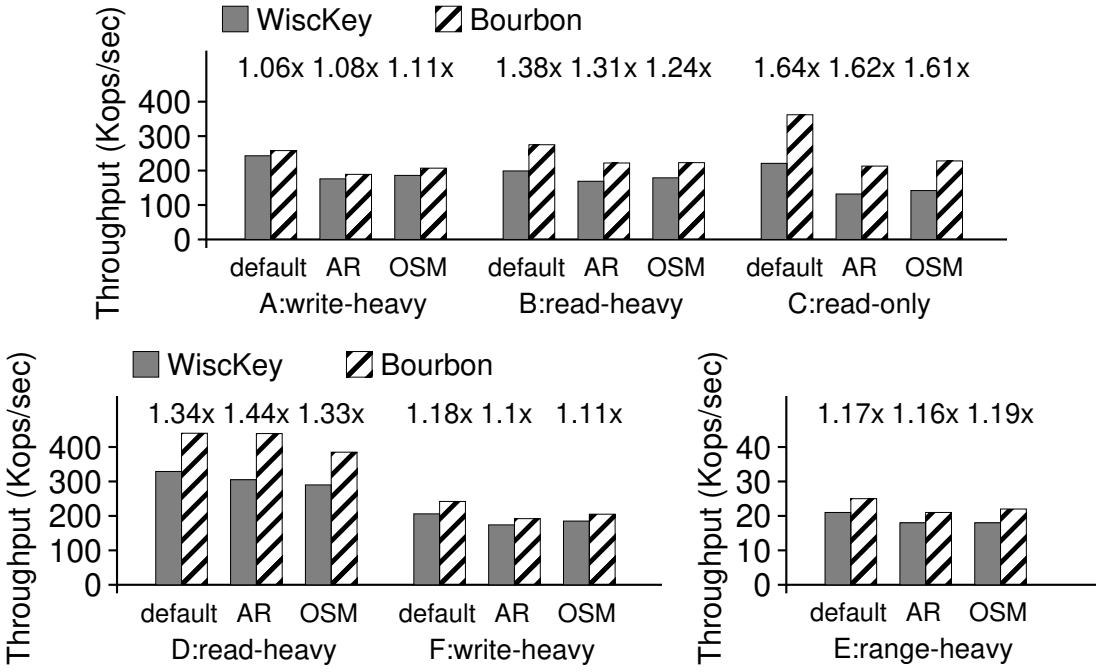


Figure 2.14: Macrobenchmark-YCSB. *The figure compares the throughput of* Bourbon *against WiscKey for six YCSB workloads across three datasets.*

faster with queries that return 100 items. This is because, while Bourbon can accelerate the indexing portion, it follows a similar path as WiscKey to scan subsequent keys. Thus, with large range lengths, indexing accounts for less of the total performance, resulting in lower gains.

2.4.4 Efficacy of Cost-benefit Analyzer with Writes

We next analyze how Bourbon performs in the presence of writes. Writes modify the data and so the models must be re-learned. In such cases, the efficacy of Bourbon's cost-benefit analyzer (cba) is critical. We thus compare Bourbon's cba against two strategies: Bourbon-offline and Bourbon-always. Bourbon-offline performs no learning as writes happen; models exist only for the initially loaded data. Bourbon-always re-learns the data as writes happen; it always decides to learn a file without considering cost. Bourbon-cba re-learns as well, but it uses the cost-benefit analysis to decide whether or not to learn a file.

We run a workload that issues 50M operations with varying percentages of writes on the AR dataset. To calculate the total amount of work performed for each workload, we

sum together the time spent on the foreground lookups and inserts (Figure 2.13(a)), the time spent learning (2.13(b)), and the time spent on compaction (not shown); the total amount of work is shown in Figure 2.13(c). The figure also shows the fraction of internal lookups that take the baseline path (2.13(d)).

First, as shown in 2.13(a), all BOURBON variants reduce the workload time compared to WiscKey. The gains are lower with more writes because BOURBON has fewer lookups to optimize. Next, BOURBON-offline performs worse than BOURBON-always and BOURBON-cba. Even with just 1% writes, a significant fraction of internal lookups take the baseline path in BOURBON-offline as shown in 2.13(d); this shows re-learning as data changes is crucial.

BOURBON-always learns aggressively and thus almost no lookups take the baseline path even for 50% writes. As a result, BOURBON-always has the lowest foreground time. However, this comes at the cost of increased learning time; for example, with 50% writes, BOURBON-always spends about 134 seconds learning. Thus, the total time spent increases with more writes for BOURBON-always and is even higher than baseline WiscKey as shown in 2.13(c). Thus, aggressively learning is not ideal.

Given a low percentage of writes, BOURBON-cba decides to learn almost all the files, and thus matches the characteristics of BOURBON-always: both have a similar fraction of lookups taking the baseline path, both require the same time learning, and both perform the same amount of work. With a high percentage of writes, BOURBON-cba chooses not to learn many files, reducing learning time; for example, with 50% writes, BOURBON-cba spends only 13.9 seconds in learning (10× lower than BOURBON-always). Consequently, many lookups take the baseline path. BOURBON-cba takes this action because there is less benefit to learning as the data is changing rapidly and there are fewer lookups. Thus, it almost matches the foreground time of BOURBON-always. But, by avoiding learning, the total work done by BOURBON-cba is significantly lower.

Summary. Aggressive learning offers fast lookups but with high costs; no re-learning provides little speedup. Neither is ideal. In contrast, BOURBON provides high benefits similar to aggressive learning while lowering total cost significantly.

2.4.5 Real Macrobenchmarks

We next analyze how BOURBON performs under two real benchmarks: YCSB [33] and SOSD [72].

Dataset	WiscKey latency (µs)	Bourbon	
		Latency(µs)	Speedup
Amazon Reviews (AR)	3.53	2.75	1.28×
NewYork OpenStreetMaps (OSM)	3.14	2.51	1.25×

Table 2.2: Performance on Fast Storage. *The table shows Bourbon's lookup latencies when the data is stored on an Optane SSD.*

2.4.5.1 YCSB

We use six workloads that have different read-write ratios and access patterns: A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), E (range-heavy, w:5%, range:95%), F (read-modify-write:50%, r:50%). We use three datasets: YCSB's default dataset (created using *ycsb-load* [5]), AR, and OSM, and load them in a random order. Figure 2.14 shows the results.

For the read-only workload (YCSB-C), all operations benefit and Bourbon offers the most gains (about 1.6×). For read-heavy workloads (YCSB-B and D), most operations benefit, while writes are not improved and thus Bourbon is 1.24× − 1.44× faster than the baseline. For write-heavy workloads (YCSB-A and F), Bourbon improves performance only a little (1.06× − 1.18×). First, a large fraction of operations are writes; second, the number of the internal lookups taking the model path decreases (by about 30% compared to the read-heavy workload because Bourbon chooses not to learn some files). YCSB-E consists of range queries (range lengths varying from 1 to 100) and 5% writes. Bourbon reaches 1.16× − 1.19× gain. In summary, as expected, Bourbon improves the performance of read operations; at the same time, Bourbon does not affect the performance of writes.

2.4.5.2 SOSD

We next measure Bourbon's performance on the SOSD benchmark designed for learned indexes [72]. We use the following six datasets: book sale popularity (amzn32), Facebook user ids (face32), lognormally (logn32) and normally (norm32) distributed datasets, uniformly distributed dense (uden32) and sparse (uspr32) integers. Figure 2.15 shows the average lookup latency. As shown, Bourbon is about 1.48× − 1.74× faster than the baseline for all datasets.

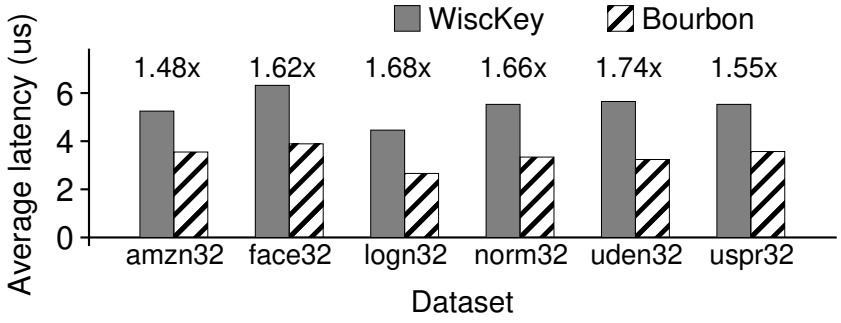


Figure 2.15: Macrobenchmark-SOSD. *The figure compares lookup latencies from the SOSD benchmark. The numbers on the top show* Bourbon*'s improvements over the baseline.*

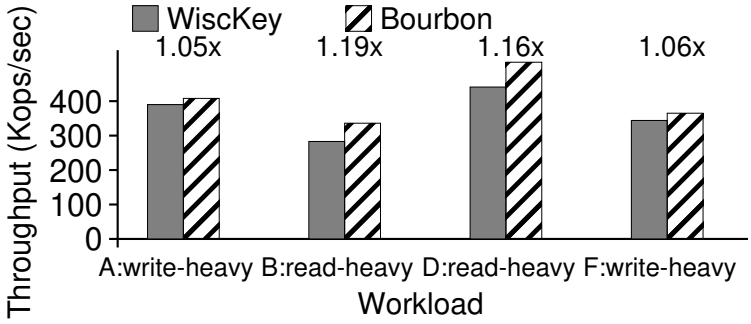


Figure 2.16: Mixed Workloads on Fast Storage. *The figure compares the throughput of* Bourbon *against WiscKey for four read-write mixed YCSB workloads. We use the YCSB default dataset for this experiment.*

2.4.6 Performance on Fast Storage

Our analyses so far focused on the case where the data resides in memory. We now analyze if Bourbon will offer benefit when the data resides on a fast storage device. We run a read-only workload on sequentially loaded AR and OSM datasets on an Intel Optane SSD, with a read latency of about 10µs. Table 2.2 shows the result. Even when the data is present on a storage device, Bourbon offers benefit (1.25× – 1.28× faster lookups than WiscKey). Figure 2.16 shows the result for read-write mixed YCSB workloads on the same device with the default YCSB datasest. As expected, while Bourbon's benefits are marginal for write-heavy workloads (YCSB-A and YCSB-F), it offers considerable speedup (1.16× – 1.19×) for read-heavy workloads (YCSB-B and YCSB-D). With the emerging storage technologies (e.g., 3D XPoint memory), Bourbon will offer even more benefits.

Workload	WiscKey latency (µs)	Bourbon	
		Latency(µs)	Speedup
Uniform	98.6	94.4	1.04×
Zipfian	18.8	15.1	1.25×

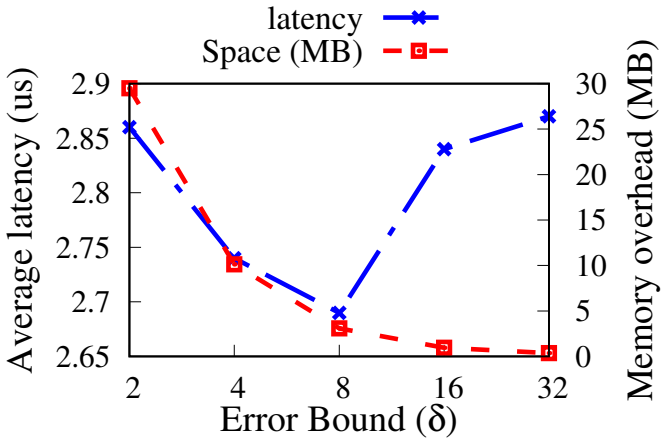
Table 2.3: Performance with Limited Memory. *The table shows Bourbon's average lookup latencies from the AR dataset on a machine with a SATA SSD and limited memory.*

2.4.7 Performance with Limited Memory

We further show that, even with no fast storage and limited available memory, Bourbon can still offer benefit with skewed workloads, such as zipfian. We experiment on a machine with a SATA SSD, with a read latency of over 100µs, and memory that only holds about 25% of the database. We run a uniform random workload, and a zipfian workload with consecutive hotspots where 80% of the requests access about 25% of the database. Table 2.3 shows the result. With the uniform workload, Bourbon is only 1.04× faster because most of the time is spent loading the data into the memory. With the zipfian workload, in contrast, indexing time instead of data-access time dominates because a large number of requests access the small portion of data that is already cached in memory. Bourbon is able to reduce this significant indexing time and thus offers 1.25× lower latencies.

2.4.8 Error Bound and Memory Consumption

We finally discuss the characteristics of Bourbon's ML model, specifically its error bound (δ) and memory consumption. Figure 2.17(a) plots the error bound (δ) against the average lookup latency (left y-axis) for the AR dataset. As δ increases, fewer line segments are created, leading to fewer searches, thus reducing latency. However, beyond $\delta = 8$, although the time to find the segment reduces, the time to search within a segment increases, thus increasing latency. We find that Bourbon's choice of $\delta = 8$ is optimal for other datasets too. Figure 2.17(a) also shows how memory consumption (right y-axis, compared to the size of the entire database) varies with δ. As δ increases, fewer line segments are created, leading to low memory consumption. Table 2.17(b) shows the memory consumptions for different datasets. As shown, for most of the datasets, the memory consumption compared to the total dataset size is little (less than 0.25%), which is 0.5× to 0.75× smaller than the original index.



(a) Error-bound tradeoff

Dataset	Memory Consumption	
	MB	%
Linear	0.02	0.0
Seg1%	15.38	0.21
Seg10%	153.6	2.05
Normal	16.94	0.23
AR	3.09	0.08
OSM	7.08	0.26

(b) Memory consumption

Figure 2.17: Error-bound Tradeoffs and Space Overheads. *(a) shows how the PLR error bound affects lookup latency and memory overheads; (b) shows the memory consumptions for different datasets.*

2.5 Conclusion

In this chapter, we examine if learned indexes are suitable for write-optimized log-structured merge (LSM) trees. Through in-depth measurements and analysis, we derive a set of guidelines to integrate learned indexes into LSMs. Using these guidelines, we design and build BOURBON, a learned-index implementation for a highly-optimized LSM system. We experimentally demonstrate that BOURBON offers significantly faster lookups for a range of in-memory workloads and datasets, and largely reduces the size of the indexes.

BOURBON is an initial work on integrating learned indexes into an LSM-based storage system. More detailed studies, such as more sophisticated cost-benefit analysis, general string support, and different model choices, could be promising for future work. In addition, we believe that BOURBON's learning approach may work well in other write-optimized data structures such as the B^{ϵ}-tree [22] and could be an interesting avenue for future work. While our work takes initial steps towards integrating learning into production-quality systems, more studies and experience are needed to understand the true utility of learning approaches.

Chapter 3

Symbiosis: The Art of Application and Kernel Cache Cooperation

In this chapter, we design and build Symbiosis to solve the cache partitioning problem for the common application-kernel two-level cache structure. Symbiosis optimizes the cache sizes of storage engines with the knowledge of the underlying layer, the kernel page cache, and improves the overall cache efficiency across the data processing stack.

We first introduce the cache partitioning problem and show its significance (§3.1). With a simple experiment, we show that configuring the right cache sizes can provide huge performance gains, but it is not trivial to get the right sizes for workloads with various access patterns.

We then conduct a simulation study of the general two-level cache partitioning problem to guide the design, approximations, and optimizations of Symbiosis (§3.2). We find that the optimal partitioning depends on a wide range of environment- and workload-related factors and it is hard to achieve a static solution.

We thus design and implement Symbiosis to dynamically adjust cache sizes online according to the workloads (§3.3). Symbiosis detects workload changes by monitoring cache performance and performs online cache simulations to determine the best cache configuration for the current workload. We apply a range of optimization techniques to achieve both high accuracy and low overhead (only about 1% time overhead and 0.1% space overhead) in online cache simulation. We integrate Symbiosis into LevelDB, RocksDB, and WiredTiger, each within 1K LOC.

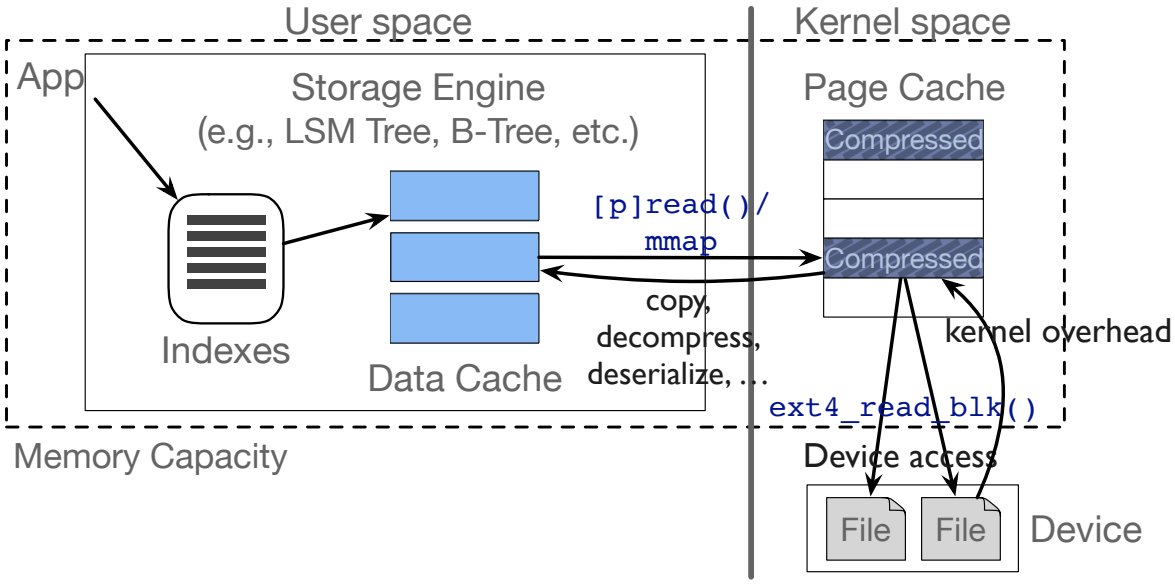


Figure 3.1: The Cache Architecture across the Storage Stack. *Modern applications commonly utilize storage engines (e.g., LevelDB) to manage on-disk data. A storage engine keeps compressed data on disk, and usually has separate index structures and an in-memory buffer for uncompressed data. The arrows depict the common read path.*

Finally, we perform an evaluation of our system (§3.4) using both synthetic and real workloads. We show that our approach improves performance, in some cases by an order of magnitude. We also show the costs of online simulation are not high and various optimizations work well. Overall, we show that Symbiosis is an effective approach to cache-size configuration for modern key-value storage systems.

3.1 Motivation and Framework

Databases and key-value stores utilize similar caching architectures (Figure 3.1). Irrespective of the underlying data structure organization (log-structure-merge trees [46, 53] or B-trees [107, 133]), these systems use both a custom application-level cache and the underlying file system page cache, forming an application-kernel two-layer cache structure.

To access a key-value pair, a request first queries an index-like structure, and, if successful, searches for the value in the user-level *application* cache. If the value is not present in the application cache, a file system read request is issued to fetch the data. This read request may be served by the *kernel page cache*, which holds a compressed version of the data. If the

file is not present in the kernel cache, the file system issues necessary I/Os to complete the request, and then caches the (compressed) data. In data-intensive workloads, memory used by the application and kernel caches constitutes a majority of the storage engine's memory usage [28, 63].

Most mainstream storage engines prefer the kernel page cache for buffering on-disk data, to utilize its robust performance under various workloads and to avoid the labor of implementing a sophisticated user-level device-friendly caching and prefetching approach. Thus, we focus our study on this application-kernel cache structure. However, some storage engines can be configured to manage their own second-level cache for compressed on-disk data (e.g., RocksDB). As we will see later, our techniques also work well on this (simpler) user/user configuration.

3.1.1 The Application-Kernel Cache Structure

This two-layer cache structure has several main properties. In the first layer, storage engines keep decompressed and deserialized data. These application caches store ready-to-use data to serve requests efficiently.

For example, LevelDB [53], the main storage engine we study, is an LSM-based key-value storage engine with a block-based application cache. Data blocks are variable-sized and not aligned. When a thread inserts an item and overflows the cache, it is responsible for performing evictions using LRU replacement. In contrast, WiredTiger [107], the underlying storage engine of the popular database MongoDB, is a B-Tree-based engine and has a significantly different caching mechanism. Instead of a unified cache structure, WiredTiger constructs an in-memory B-Tree representation and allows each B-Tree node to dynamically allocate memory to cache data. When the total amount of cached data reaches the limit, background threads are initiated to traverse the tree and perform evictions. Each node records last-access recency to approximate LRU replacement.

The second layer of this cache structure is a compressed cache that commonly utilizes the underlying OS kernel's page cache. Storage engines compress on-disk data to reduce device bandwidth and save space on disk; furthermore, by using the kernel page cache, one can leverage years of performance tuning that is present therein.

In Linux, the eviction algorithm is 2Q with a clock algorithm for each queue and involves sophisticated heuristics for promotion, demotion, and size partitioning among the queues.

In addition, Linux performs read-ahead to ensure high bandwidth utilization. The current read-ahead approach uses heuristics to determine which pages/when to prefetch (including basing its decisions on the cache presence of pages neighboring the target page), which can significantly affect the hit ratio in some scenarios.

To summarize, this two-level cache structure has several important characteristics. First, the application and kernel caches form a two-level caching scheme that shares the same memory quota (i.e., if one cache grows, the other must shrink). The kernel cache often stores compressed data, making it more efficient in terms of memory usage, while the application cache provides lower latency as its data is ready to be used, saving the cost of decompression and kernel crossing. Second, with data compression, the two caches store data in different forms, units, and alignments. One block in the application cache may correspond to several pages in the kernel page cache due to misalignment, which further complicates the management of the two caches and the optimization of overall performance.

3.1.2 Challenge: Memory Partitioning

Given the two-level caching architecture, a natural question arises: how should memory be allocated between the two caches, in order to maximize performance? To illustrate some of the complexities of this issue, we present the following motivating experiment. Here, we study the performance of different cache configurations in two representative storage engines, LSM-based LevelDB [53] and B-tree-based WiredTiger [107]. We run uniform random workloads with 1 GB of available memory. We use small data sets here to speed our analysis; as we will show later, results are nearly identical when data sets are scaled up.

We compare two extremes: one which devotes all available memory to the application cache, and the other which devotes all memory to the kernel cache. We show how performance varies across two different data set sizes (D_u), 1 GB and 2 GB (uncompressed); the compression ratio is 0.5. Figure 3.2 presents our results.

We see similar trends from both storage engines. When the data set size is 1 GB (and hence fits, uncompressed, into the application cache), devoting as much memory as possible to the application cache outperforms the kernel cache by 2.5× to 3×. In contrast, when the data set size is 2 GB (and hence fits compressed into the kernel cache, but is too large for the uncompressed application cache), the kernel cache outperforms the application cache, by up to 7×.

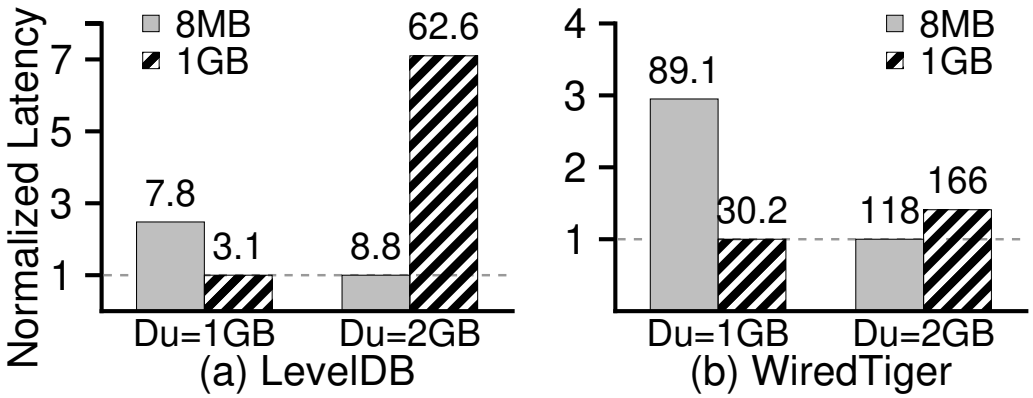


Figure 3.2: Storage Engine Performance Varying Data Set Size. *Each bar depicts one application cache size (8MB or 1GB); each pair of bars shows performance for a given dataset size. Total available memory is 1 GB. The y-axis is the latency normalized to the lowest value; numbers above are absolute latencies (us/op).*

The experiment demonstrates that cache configuration impacts performance significantly; no single configuration performs well across different workloads and settings. A deeper understanding of the performance characteristics of this two-level structure is required; a systematic approach that can coordinate the two caches to maximize performance is needed.

3.1.3 Cache Coordination with Symbiosis

To address this problem, we propose *Symbiosis*, a system to coordinate application and kernel caches to maximize performance across differing workloads and system configurations. Figure 3.3 presents an overview of the system architecture. A key element of Symbiosis is an online cache simulator that monitors performance levels given the current application/kernel configuration and determines necessary adaptations to improve performance. The simulator selectively applies *ghost caching* [42] to determine whether a different application cache size would be beneficial; if so, it changes the size of the application cache (and thus implicitly makes more or less memory available for the kernel cache).

Detailed online simulation can be prohibitively slow. Therefore, Symbiosis uses a simplified representation of the actual caching approaches used by real systems. The core challenge thus lies in determining how to abstract the essence of the cache sizing problem and adopt the right level of simplification, aiming for a balance between overhead and

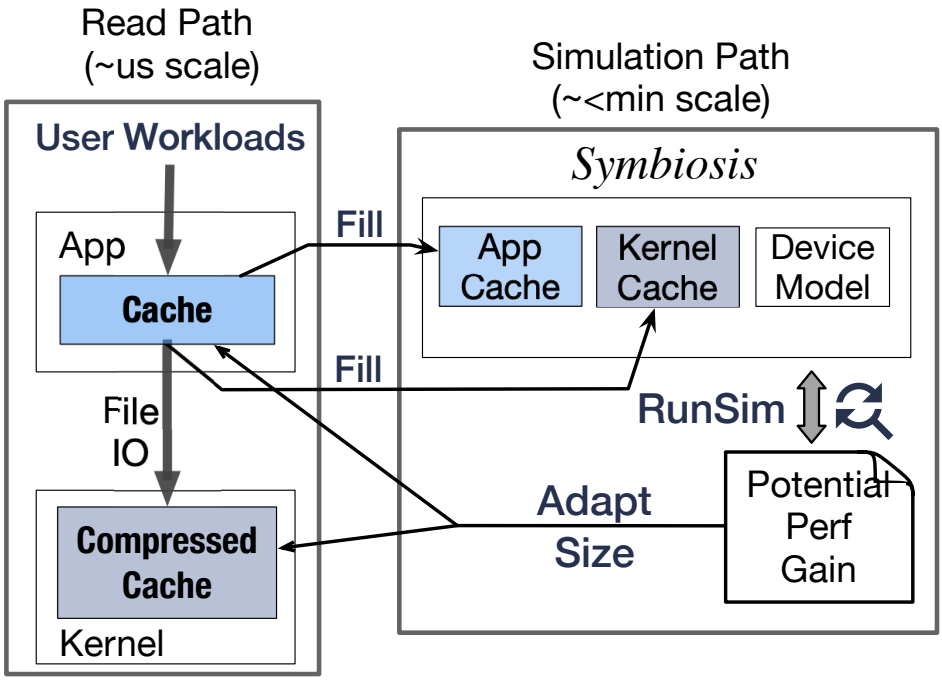


Figure 3.3: Overview of Symbiosis. *This figure shows the main components of Symbiosis and their interactions.*

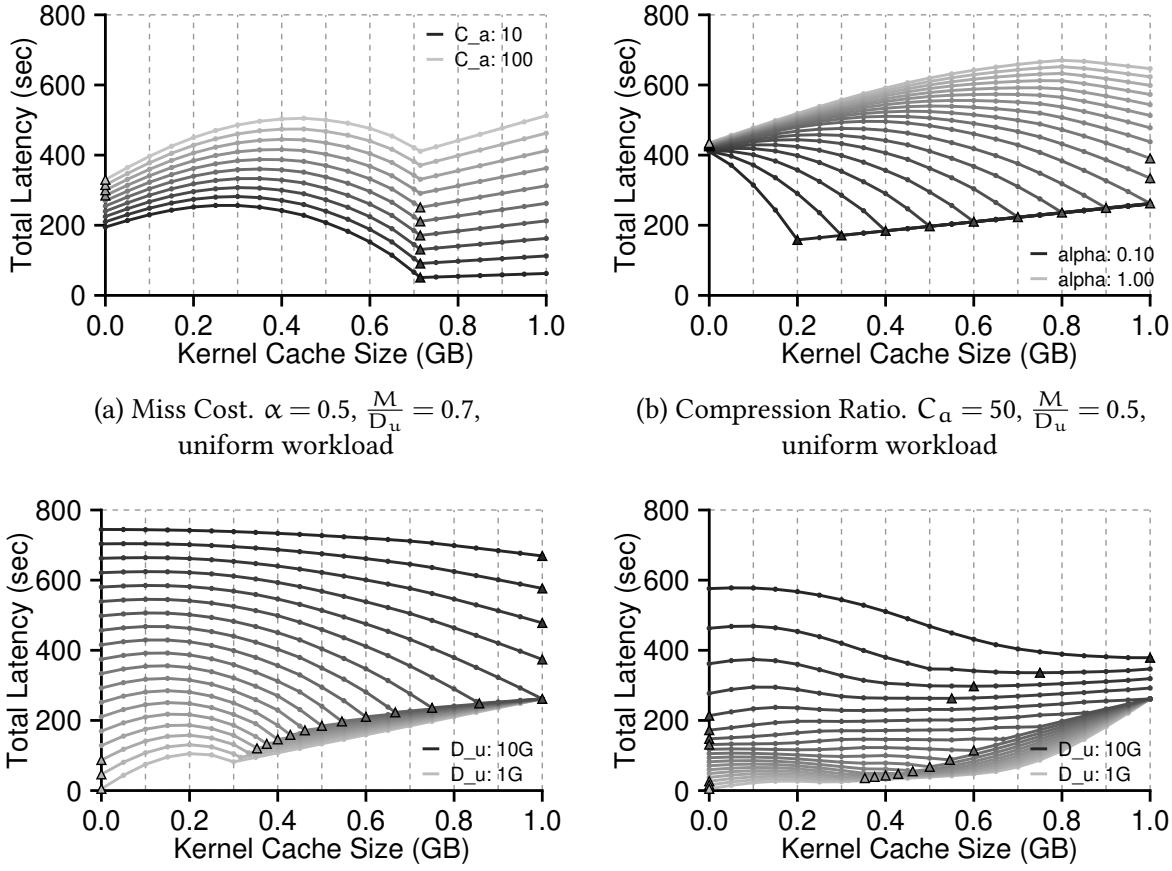
accuracy. We show how to strike this balance later (§3.3).

3.2 The Cache Partitioning Problem

Through offline simulations, we show the factors that influence how memory should be divided between the application and kernel caches. Our simulations demonstrate that the division of memory between application and kernel caches has a large impact on performance (e.g., up to 9×), and that the best division is highly dependent on a wide variety of factors, some of which are specific to the environment (e.g., application and kernel miss costs) and some of which can vary depending upon workload (e.g., the size of the data set, compression ratio, and application/kernel cache hit rates).

3.2.1 Influential Factors

We define a number of system and workload parameters that impact the best division of memory.



(a) Miss Cost. $\alpha = 0.5$, $\frac{M}{D_u} = 0.7$, uniform workload

(b) Compression Ratio. $C_a = 50$, $\frac{M}{D_u} = 0.5$, uniform workload

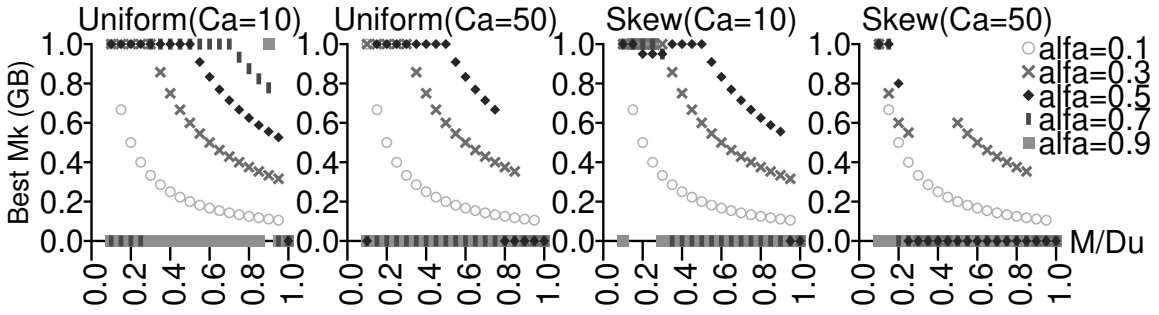
(c) Data Set Size. $C_a = 50$, $\alpha = 0.3$, uniform workload

(d) Data Set Size. $C_a = 50$, $\alpha = 0.3$, skewed workload

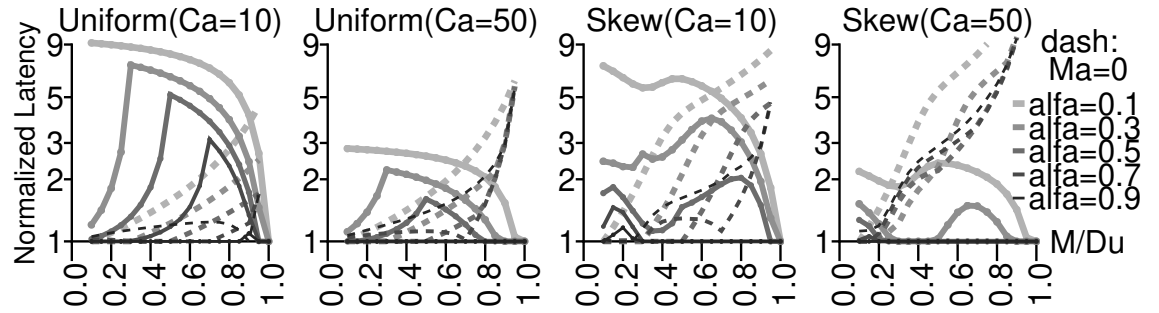
Figure 3.4: Simulation Results - Performance Varying One Factor. *In each subplot, the title indicates the varied factors across lines; the legend describes parameters of the minimal and maximal value for a factor (the rest is omitted). The triangle indicates the point of the global minima; the bold text depicts the controlled factors.*

Memory Cache Sizes: M depicts the total amount of memory that can be used for the application cache (M_a) and kernel cache (M_k); $M_a + M_k = M$. M can represent the total physical memory on a single machine, a container' resource limit [78, 155], or enforcement by other mechanisms [154, 164]. We arbitrarily fix M to 1 GB in the simulations, since only the relative size of memory to the data size matters, and not its absolute size.

Data Size: The amount of compressed data that is stored on disk by an application is D_c; the corresponding uncompressed data size is D_u. We simulate $1\text{GB} \leqslant D_u \leqslant 10$ GB.



(a) Best kernel cache size across all the factors.



(b) Performance gain. Two baselines: M_k=0 (solid) and M_a=0 (dashed).

Figure 3.5: Simulation Results - Best Configurations. *The title of each subplot means the workload and miss cost. We use* $\frac{M}{D_u}$ *from 0.1 to 1.0 (x-axis) and two miss costs* C_a*=10,50.*

Compression Ratio: (α, $0 < \alpha \leqslant 1$): The ratio of compressed data to decompressed data is α (i.e., $\alpha = \frac{D_c}{D_u}$). α is affected by the compressibility of the data and the specific compression algorithm [157]; for instance, in WiredTiger, we found that compressing a data set of $D_u = 1$ GB using four different compression algorithms (zstd, zlib, snappy, and lz4) takes between 9µs and 204µs and results in compression ratios between 0.36 to 0.51. We simulate values of α between 0.22 (observed in production [26]) and 0.5 (the default for RocksDB's `db_bench` [39]).

Retaining Data Size: (D_{mem}): We find the notion of a *retaining data size* useful: the size of cached data in both caches when it is all decompressed. The minimum D_{mem} occurs when all of M is devoted to the uncompressed application cache; that is, $D_{mem}^{min} = M$. The maximum D_{mem} occurs when all of M is devoted to the compressed kernel cache (i.e., $D_{mem}^{max} = \frac{M}{\alpha}$). A higher D_{mem} reduces device accesses.

Hit Rates: The hit rate of the application cache is H_a and the kernel cache is H_k. Hit rates are functions not only of the cache sizes, but also of access patterns and cache replacement policies. We examine uniform random, skewed, and mixed access patterns. Our simulations focus on LRU; note that improvements in replacement policies [21] are complementary to our approach as we aim to better use available memory regardless of the policy.

Miss Cost: Application miss cost is C_a and kernel cache miss cost is C_k. C_a is highly application dependent; empirically, we found C_a varied between 40µs and 250µs depending on the compression algorithm in WiredTiger and is < 10µs in LevelDB; thus, the simulation varies C_a from 10 to 100. The main factor influencing C_k is device performance; we set C_k to 100µs for common devices. Again, the ratio of miss costs ($\frac{C_a}{C_k}$) matters and not their absolute values.

3.2.2 Analysis

Our goal is to find the value of M_k that optimizes performance given the other system and workload parameters; our offline simulations do this by sweeping through the full range of valid values of M_k. To quantify the performance of the cache structure, we use average latency: $L_e = (1-H_a) * (C_a + (1-H_k) * C_k)$, omitting constant hit costs for both caches. Generally, as M_k increases, H_k increases, but H_a decreases; thus, the ideal hit rates for H_k and H_a depend on the relative values of C_k and C_a.

3.2.2.1 Uniform Workload

We begin simulations with a uniform workload as it leads to the most intuitive results. With a uniform workload and LRU replacement, the hit rate of a given cache is simply its size divided by the data size; specifically, $H_a = \frac{M-M_k}{D_u}$ where $0 \leqslant M_k \leqslant M$, and $H_k = \frac{M_k}{\alpha * D_u}$ where $0 \leqslant M_k \leqslant \alpha * D_u$. L_e can be calculated as a quadratic function of M_k with a negative quadratic term coefficient; thus, the two boundary points of the domain ($M_k = 0$ and $\min(M, \alpha * D_u)$) are two local minima, but which of the two is the global minimum depends on all factors, as we illustrate.

Miss Cost (C_a vs. C_k): We begin by showing the best kernel cache size as a function of miss costs. In our two-layer caching architecture, the ratio $\frac{C_a}{C_k}$ determines how much each

miss rate contributes to overall performance. While this ratio does not impact the cache configurations of the two local minima, it does influence which is the global minimum.

Figure 3.4a shows latency as a function of M_k, varying C_a from 10 to 100 (interval=10) and fixing $D_u = 1.43$ GB (i.e., $\frac{M}{D_u} = 0.7$) and $\alpha = 0.5$. For all values of C_a, the local minima are at $M_k = 0$ and $M_k = \alpha * D_u$, and the global minimum changes from 0 to $\alpha * D_u$ as C_a decreases (i.e., when $C_a < 60$). In general, when $0 < M_k < \alpha * D_u$, L_e is larger than at both extremes because both caches are non-zero and contain duplicates; when M_k grows beyond $\alpha * D_u$, L_e increases because the kernel cache already holds all compressed data. Additional M_k causes more application cache misses. With a higher C_a, the global minimum of M_k is smaller, as application cache misses are penalized more.

Figure 3.5a summarizes the best kernel cache size for different parameters, illustrating that different systems and workloads benefit from very different cache configurations, with best values of M_k from 0 to M and all points between. More specifically, the first two subplots show uniform workloads; comparing points across these first two subplots confirms that a higher value of C_a (i.e., $C_a = 50$ vs. $C_a = 10$) makes the best kernel cache size smaller. Figure 3.5b shows how much latency is improved when the cache system is configured correctly; specifically, the graphs compare latency with the best cache partition to two reasonable default cache configurations: $M_a = 0$ (dashed lines) and $M_k = 0$ (solid lines). For example, with a smaller C_a, latency can be nine times larger with a poor choice cache configuration (i.e., $M_k = 0$) than with the best choice.

Compression Ratio (α): Figure 3.4b shows the impact of α on the best kernel cache size, by varying α from 0.1 to 1 with an interval of 0.05 and setting $D_u = 2$ GB and $C_a = 50$; D_u is set larger than M so that it is not possible to cache all uncompressed data in memory.

Given a lower α (for a fixed D_u), a larger kernel cache tends to be better as it is more efficient with compressed data; with a low α, the kernel cache provides larger D_{mem}, avoiding more device accesses than the application cache. Specifically, with a very low α (i.e., the bottom line with $\alpha = 0.1$), latency drops sharply from $M_k = 0$ to $M_k = \alpha * D_u = 0.2$. Generally, while the latency at $M_k = 0$ remains the same, the latency at $M_k = \min(M, \alpha * D_u)$ decreases with smaller values of α; as a result, the global minimum changes from $M_k = 0$ to $M_k = \min(M, \alpha * D_u)$ when $\alpha < 0.65$.

Figure 3.5a confirms that larger kernel caches are more beneficial with smaller values of α and Figure 3.5b shows that the performance improvement is more dramatic with smaller α; the potential benefit of the kernel cache is high.

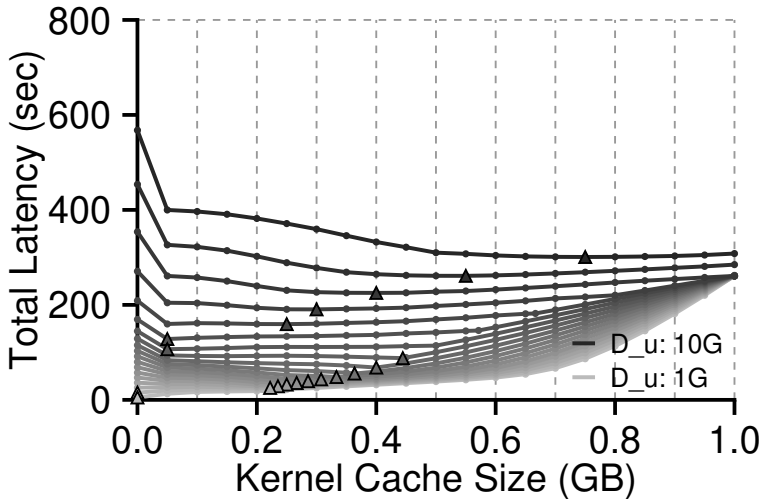


Figure 3.6: Simulated performance under a Mixed (Read+Scan) workload. *The legend describes parameters of the minimal and maximal value for the varying factor, DataSetSize (i.e.,* D_u*). The triangle indicates the point of global minima. (Controlled factors:* $C_a = 50, \alpha = 0.2$*)*

Data Size (D_u) **vs. Memory Capacity** (M): Figure 3.4c shows the impact of varying D_u from 1 GB to 10 GB (i.e., varying $\frac{M}{D_u}$ from 0.1 to 1.0) while $\alpha = 0.3$ and $C_a = 50$. While the two local minima for M_k (0 and $\min(M, \alpha * D_u)$) follow the studied trends of L_e, we make three specific observations. First, when D_u is very small, the application cache can fit all of the data uncompressed, so all memory should be devoted to the application cache ($M_k = 0$). Second, when D_u is much higher than M (e.g., when $D_u = 10$ GB), the impact of different values of M_k is smaller since most accesses miss both caches. Finally, as D_u grows larger than 2 GB, the global minimum changes from $M_k = 0$ to $M_k = \min(M, \alpha * D_u)$; for these values of D_u, the larger M_k is better because it leads to a larger D_{mem} at the cost of a lower H_a. In summary, the best M_k tends to be 0 for a very large or very small D_u, and $\min(M, \alpha * D_u)$ for a medium D_u.

In Figure 3.5a, the $\alpha = 0.7$ line in the first graph shows this trend best. As shown in Figure 3.5b, with a medium D_u, the performance gain over $M_k = 0$ is large and with a small D_u the gain over $M_a = 0$ is generally larger; with a very large D_u, the gain is small as all cache configurations perform similarly.

3.2.2.2 Non-Uniform Workload

While the hit rates (and thus the best values of M_k) can be precisely calculated for uniformly-random workloads, in practice, most real-world workloads are more complex [26, 33]. We simulate a skewed workload containing a hotspot with locality as suggested by production RocksDB [26] in which 20% of the key space serves 80% of requests. Figure 3.4d shows that this skewed workload exhibits a significantly different performance curve from a uniform workload (Figure 3.4c). The trend observed for a uniform workload, in which the best M_k grows with increasing D_u, does not hold for skewed workloads and the best M_k becomes highly unpredictable. Generally, for a skewed workload, a larger application cache is preferred since more accesses occur within a smaller hotspot and the same size of application cache provides a higher hit rate; this effect can be roughly viewed as effectively reducing D_u. Figure 3.5a shows this preference to the application cache, comparing the right half of graphs to the left half; Figure 3.5b confirms that the performance gain over $M_k = 0$ is smaller than for uniform workloads and that over $M_a = 0$ is larger.

Our second non-uniform workload contains a mix of *read* and *scan* operations, as commonly found in real deployments [26, 33]. We use the YCSB benchmark [33] to generate 90% reads and 10% scans with an 80/20 hotspot and a scan length uniformly distributed between 0 and 100 KB. The results in Figure 3.6 show that the trends are even more irregular: although the best M_k increases with decreasing $\frac{M}{D_u}$ (i.e., increasing D_u), the best M_k decreases significantly when $\frac{M}{D_u}$ decreases from 0.45 to 0.4, and never at the extreme points (i.e., 0 and M) when $\frac{M}{D_u} < 0.9$. In summary, the best cache configuration for a non-uniform workload is more difficult to predict with an offline simulation or model.

3.2.3 Discussion

Our simulations have shown that the best cache configuration is highly sensitive to factors such as memory capacity, compression ratio, and miss cost, which depend on data and hardware; non-uniform workloads further exacerbate the complexity. The performance gain curves in Figure 3.5b show that improvements compared to a default cache configuration can be significant, but that the best kernel cache size varies significantly. Statically determining the best configuration is impractical due to the dynamic nature of workloads, directing us to a runtime adaptive approach. Fortunately, although the amount of gain is difficult to predict, the curves are relatively smooth without abrupt changes, indicating that some inaccuracy

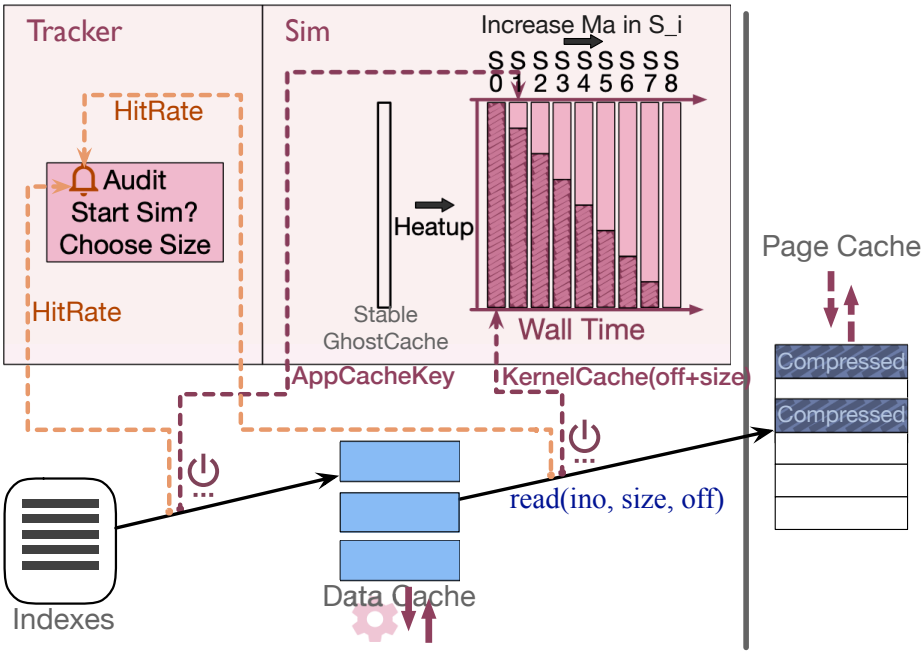


Figure 3.7: Design of Symbiosis. *Symbiosis is directly integrated into a storage engine. The orange dashed lines are the stats collection paths that are always active; the dashed red lines are the paths filling entries into the ghost cache, activated only in* Adapting State *and empty in* Stable State. *The information inside the GhostSim component illustrates how the ghost cache changes across the nine configurations during one simulation round. The size of the application cache (i.e., light red portion of a bar) is increased over time; the dark red portion represents the kernel cache.*

in online simulation can be tolerated.

3.3 Design and Implementation of Symbiosis

We present our design and implementation of Symbiosis, which performs online cache simulation to dynamically and adaptively configure two levels of cache for high performance. The key challenge is to achieve simulation accuracy and configuration coverage while maintaining high performance to minimize the impact on the foreground workload.

3.3.1 Design

SYMBIOSIS is an add-on module built into a storage engine that automatically adjusts the application cache size (M_a), implicitly changing the kernel cache size (M_k). Figure 3.7 illustrates how SYMBIOSIS integrates into existing storage engines. SYMBIOSIS contains two main components: *Tracker* and *GhostSim*. Tracker continuously audits application and kernel cache accesses to collect performance statistics; Tracker decides when to activate GhostSim to find a better $<M_a, M_k>$ and which specific candidate to adopt. GhostSim uses efficient online cache simulation to predict the performance of candidates.

We design SYMBIOSIS to achieve several goals. First, *low overhead*: incur negligible overhead for the in-memory read path, taking less than a few microseconds if a request hits in the caches. Second, *memory efficient*: minimize memory to reduce interference with the memory-constrained storage engine. Finally, *robust performance*: deliver superior performance in most cases, while guaranteeing baseline performance for arbitrary workloads.

To minimize the overhead of configuration exploration and changes, GhostSim is activated only when necessary. To lower our overhead and memory consumption, we maximize ghost cache reuse with a pipelined simulation of $<M_a, M_k>$ configurations in the order of increasing M_a. To reduce memory consumption and maintain high accuracy, we use sampling specifically tailored to our cache structure, accounting for misalignment and read-ahead in the kernel cache. Finally, to guarantee performance improvements, we apply a policy to guard against (uncommon) inaccurate simulation results.

3.3.1.1 Auditing by Tracker: Metric and States

SYMBIOSIS alternates between two states: *Stable* and *Adapting*. In the initial stable state, Tracker detects workload changes using the *expected latency*, calculated as $L_e = (1 - H_a) * (C_a + (1 - H_k) * C_k))$. L_e focuses on two major factors: H_a and H_c (and consequently the relative cache sizes) and the relative impact of each type of miss. Specifically, Tracker continuously audits the hit/miss result of each cache and calculates L_e with statically configured miss costs by offline measurement. Tracker periodically compares the current calculated L_e to the initial L_e for this round; if the difference is larger than a fixed threshold (currently 10%), Tracker considers it a workload change and enters the adapting state that starts a simulation round. Thus, GhostSim is activated only when necessary.

3.3.1.2 Simulating with GhostSim: Lifetime of a Round

The basic idea of the adapting state is to systematically generate several $<M_a, M_k>$ candidates, run simulations to predict their L_e's, and determine if the best of them has sufficient performance gain to be applied to the real system. GhostSim is responsible for efficiently predicting the performance of different cache configurations for the current workload. To simulate live workloads and predict their expected latency, GhostSim maintains a *ghost cache* [42, 49, 109, 159], filled with the same indices as in the embedded storage engine, but without the actual data. To minimize memory consumption and performance overhead, GhostSim simulates only one instance of ghost cache at a time, adopting a pipelined simulation of candidates in the order of increasing M_a to maximize ghost cache reuse. After collecting the L_e of each candidate $<M_a, M_k>$ through simulation, Tracker derives the potential gain of the best candidate configuration and applies it to the real system if the gain surpasses a certain threshold. The ghost cache entries are then discarded to save memory. Symbiosis waits for the real caches to warm up and generate a stable initial L_e as the reference point in the next period.

We strictly bound the ghost cache's space and time overhead with a collection of techniques (described below), as a naive full simulation incurs unacceptable memory consumption ($> 5\%$) and performance overhead ($> 30\%$).

3.3.2 GhostSim Optimization Techniques

We introduce four techniques to achieve sufficient simulation accuracy, memory efficiency, performance, and robustness; overall, we identify and solve new challenges for sampled ghost cache simulation raised by the unique interaction pattern of the two-level cache structure. First, we reset to a cache configuration during simulation that will perform reasonably for the current workload; second, we simulate a pipelined sequence of candidate configurations to achieve high coverage and efficiency; third, we use sampling to achieve accurate simulation with reduced memory; fourth, we guard against (uncommon) flawed simulation results that could occur due to not modeling all kernel caching features.

3.3.2.1 Initialization: Reset Policy

During *Adapting State*, GhostSim must use a cache configuration that performs reasonably for the live foreground workload; GhostSim either continues using the current cache config-

uration, or if L_e has increased (likely from an increase in D_u), it resets to the minimal default M_a used by the original storage engine (which increases D_{mem}). We show the benefits of this reset policy in Section §3.4.2.4.

3.3.2.2 Incremental reuse of a single Ghost Cache

We extend the idea of storing cache access metadata with a ghost cache [42, 49, 109, 159] to efficiently handle two-levels of caches while minimizing the memory footprint. Multiple first-level cache sizes can be simulated simultaneously with only the amount of memory required for the largest cache if the first-level cache follows the stack property [103] (e.g., LRU). However, the second-level cache sees different access patterns depending on the size of the first level, and thus has different contents when sized differently. Thus, simultaneous simulations of all second-level size candidates within one ghost cache instance is infeasible.

To efficiently simulate memory configurations with ghost caches, Figure 3.7 illustrates our choices of $<M_a, M_k>$ candidates. Our simulation results (Figure 3.5a) indicated that the best memory configuration could be anywhere within the search space; therefore, GhostSim forms the candidate set by dividing the search space into a fixed number of equal ranges (currently 8) without skipping candidates or stopping early; this provides relatively high coverage of the search space with reasonable convergence time. Since warming up each candidate ghost cache is a significant source of overhead, Symbiosis simulates each in the order of increasing M_a to maximize the reuse of ghost cache contents. Specifically, we keep the application ghost cache at its full size and simulate different M_a's using the stack property, so that when M_a is increased for the next candidate, the contents of the increased portion are already known. A short warm up for the kernel ghost cache after M_k is decreased is required to let its contents approach those of the next candidate's configuration.

3.3.2.3 Sampling with Misalignment and Read-ahead

Even with reuse, the memory consumed by a ghost cache is significant (e.g., 50 MB for 1 GB data). To reduce memory consumption, we incorporate a key-space sampling technique by hashing the indices so that one slot represents several keys [151, 152]. A sample ratio (R) of 0.01-0.001 minimizes memory usage while preserving accuracy.

Approximating H_k with sampling poses new challenges. An important difference between Symbiosis and other two-layer cache structures is that the kernel caches at the page

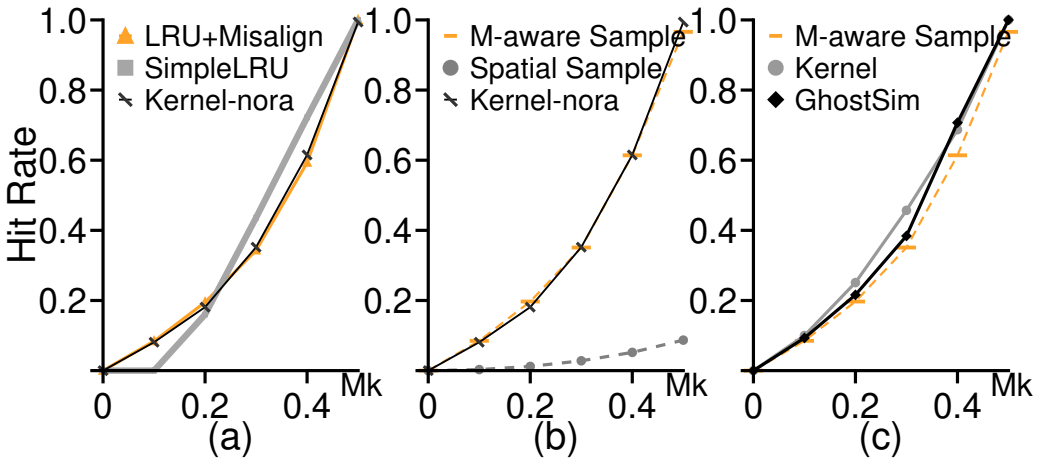


Figure 3.8: KernelCache Simulation and Sampling. *Kernel-nora and Kernel are the kernel cache implementations with and without read-ahead, respectively.*

level while the application caches in application-defined blocks that misalign with pages; as a result, the independent reference model [10] does not hold, as each request may access different targets in each layer and multiple contiguous targets in the kernel cache. Moreover, read-ahead strongly affects H_k, but a full simulation would be too costly.

We introduce different hashing approaches that accurately model these real-system effects. Figure 3.8 shows the hit rate curves for various kernel cache implementations and sampling approaches. Figure 3.8(a) shows a SimpleLRU simulator that caches in the unit of blocks instead of pages and thus does not take misalignment into account, deviates significantly from a kernel implementation that has read-ahead disabled (Kernel-nora). The LRU+Misalign simulation, which caches in the unit of pages and accounts for misalignment just as the kernel does, approximates the Kernel-nora line well. However, Figure 3.8(b) shows that spatial sampling ($R = \frac{1}{2}$) is not effective in the presence of misalignment, deviating from the Kernel-nora line. With misalignment, accessing a block across pages will read both pages into the cache, hitting neighboring blocks; spatial sampling's hashing scheme loses locality and cannot capture such behavior. We introduce *misalignment-aware sampling* that groups contiguous G application blocks before hashing to preserve locality; the M-aware Sampling line ($R = \frac{1}{2}$ and $G = 32$) approximates the Kernel-nora line well. Finally, to compensate for read-ahead, we adopt a heuristic that slightly increases the size of our modeled kernel cache. Figure 3.8(c) shows that this final version (GhostSim) approximates the Kernel better than M-aware Sampling.

Our sampling method produces similar hit rate curves with $R \geqslant \frac{1}{256}$; we choose $R = \frac{1}{64}$ due to the acceptable variance and sufficiency to realize a low-overhead online simulation. We confirm that our method broadly works well.

3.3.2.4 Guard against Unmodeled Cases and Fall Back

Although we have modeled misalignment between caches, GhostSim may be inaccurate in some workloads due to unmodeled kernel features such as read-ahead. Thus, Symbiosis only performs cache size adjustment if the predicted result improves latency by a threshold amount; we do not adapt away from settings that already works well. To understand why this approach is robust, consider a workload that performs strided access of one key per page. The kernel cache sees a linear access, triggers read-ahead, and thus achieves a high H_k, while GhostSim without read-ahead produces a low H_k. However, Symbiosis observes that the predicted L_e for all the candidate cache sizes is larger than the measured current L_e, and therefore rejects all simulation results.

3.3.2.5 Limitation and Discussion

We assume that workloads change infrequently. If the workload changes before a simulation round ends, Symbiosis detects the change, discards the current results, and starts over. If the workload changes repeatedly during simulation, Symbiosis stops the simulation as it is unable to finish and yield benefits. In our experimental environment, Symbiosis takes at most 45 seconds to detect and simulate new workloads.

Symbiosis generally offers larger and more robust benefits to existing storage engines in read-heavy workloads, which are observed as dominant in various studies [26, 33]. The idea of simulation-based cache size adaption can work with write-heavy workloads, yet will require additional research to realize in robust form. For example, LSM-based engines often schedule asynchronous background compaction in the write path; thus, speed differences in the foreground workload caused by different cache size configurations can lead to varying tree structures and thus different cache access traces. Further, write performance itself is less stable than read performance [19], which is more challenging for prediction.

3.3.3 Multiple Implementations

We have integrated Symbiosis into three different storage engines: LevelDB [53], WiredTiger [107], and RocksDB [46]. Modifying LevelDB to leverage Symbiosis required adding fewer than 1000 LoC to the 30000-LoC codebase. First, the required keys for the ghost cache are collected during the original processing of each request. Second, hit/miss statistics are recorded when accessing the application cache and inferred from timing when accessing the kernel cache. Third, LevelDB's `LRUCache` is modified to build the ghost cache utilizing the stack property, greatly reducing the amount of new code. Finally, a generic interface is added to the application cache to dynamically resize it to M_a and allow the kernel cache to automatically use the rest of the memory ($M - M_a$).

We have also ported Symbiosis to WiredTiger and RocksDB to demonstrate its generalizability. Despite the fact that WiredTiger's B-Tree-based engine has a completely different caching mechanism than LevelDB, the modifications required are similar to the four outlined above; the basic port added fewer than 100 LoC to WiredTiger and Symbiosis. Interestingly, as part of this porting process, we uncovered a bug in WiredTiger's cache eviction mechanism. Despite its claimed LRU-like behavior, the bug makes it evict data regardless of recency and its cache performance becomes extremely poor and unpredictable. This bug has been reported to MongoDB which recognized it as a major bug; we have added a workaround to restore the intended LRU policy, which significantly improves performance and enables Symbiosis to correctly simulate its cache behavior.

RocksDB is based on LevelDB and has a similar caching mechanism. To study Symbiosis's capability to handle an application-managed compressed data cache, we enable RocksDB's option to use its built-in compressed data cache and direct I/O. Whenever the application cache size is changed, we explicitly set the size of the compressed data cache to be all memory not used by the application cache (i.e., $M - Ma$). Due to RocksDB's similarity to LevelDB, the port required minimal effort.

3.4 Evaluation

To evaluate Symbiosis, we answer the following questions to demonstrate Symbiosiśs capability to adapt cache size configurations for a wide range of workloads with negligible overhead:

	Factors	Presented Space
Workloads	Data Set Size (GB)	5, 2.5, 1.67, 1.25, 1 ($M : D_u$= 0.2, 0.4, 0.6, 0.8, 1)
	Access Pattern	uniform, zipfian, hotspot{30,20,10}
Software	Compression Lib	snappy (default), zstd
	Storage Engine	LevelDB (default), RocksDB, WiredTiger
Hardware	CPU Freq.	HW1: Xeon 5128R (2.9 GHZ) HW2 [119]: Xeon D-1548 (2.0 GHz)
	Device Latency	HW1: OptaneSSD 900P (~ 10µs) HW2: Toshiba NVMe flash (~ 70µs)

Table 3.1: Factors for Static Workload. *Access patterns are generated by YCSB [33]. Zipfian has scattered hotspots over the key range to avoid space locality. Hotspot{30,20,10} means that 70%, 80%, and 90% of requests access 30%, 20%, and 10% keys in a contiguous range.*

- Q: How much better does Symbiosis perform than reasonable static cache size configurations (<M_a, M_k>) for different data set sizes (D_u), compression ratios (α), miss costs (C_a and C_k), and access patterns for different storage engines such as LevelDB, WiredTiger, and RocksDB?

  A: Symbiosis achieves as high of performance as the best static configuration and significantly outperforms other static configurations in all read workloads with different parameters and storage engines. Symbiosis also shows benefit for workloads with moderate writes. (§3.4.1)

- Q: How quickly does Symbiosis react to workload changes and how much overhead does Symbiosis incur for simulation and changing cache sizes?

  A: Symbiosis adapts to workload changes in 15.4 seconds in average and incurs negligible space and time overhead during online cache simulation. (§3.4.2)

- Q: How well does Symbiosis handle real-world workloads?

  A: Symbiosis's gain holds consistently for real-world workloads. (§3.4.3)

Setup. We use HW1 in Table 3.1 unless otherwise noted; the available memory M is fixed at 1 GB by cgroup. We evaluate Symbiosis by comparing it with two static configurations:

$M_a = 8$ MB (LevelDB's default) and M_a =1 GB ($M_k \approx 0$), referred to as $Static_{M_a=8MB}$ and $Static_{M_a=1GB}$, respectively.

3.4.1 Static Workloads

We first evaluate SYMBIOSIS under various static workloads, demonstrating that SYMBIOSIS finds a better $<M_a, M_k>$ for different data set sizes (D_u), compression ratios (α), miss costs (C_a and C_k), and access patterns. Table 3.1 shows the full range of factors. To vary α, C_a, and C_k, we use a secondary compression library (*zstd*) and hardware (HW2). We also evaluate its performance in WiredTiger and RocksDB to demonstrate its generalizability to different storage engines.

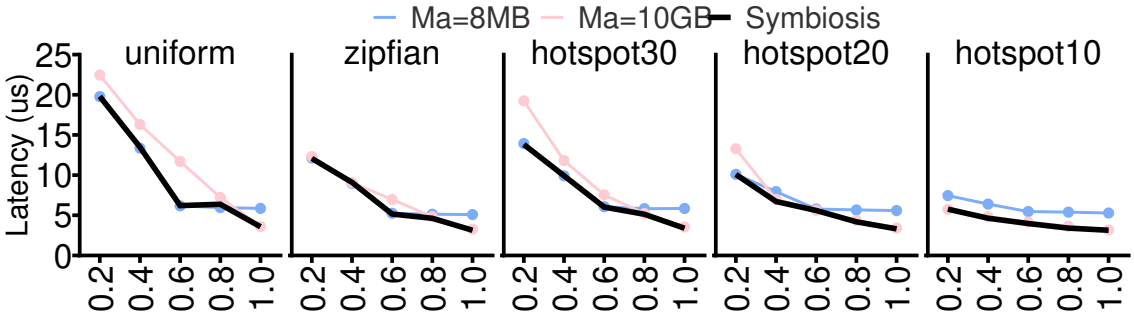
3.4.1.1 LevelDB Performance

Figure 3.9 compares the performance for LevelDB with SYMBIOSIS to the two static baselines as a function of $\frac{M}{D_u}$ for five access patterns on five different settings.

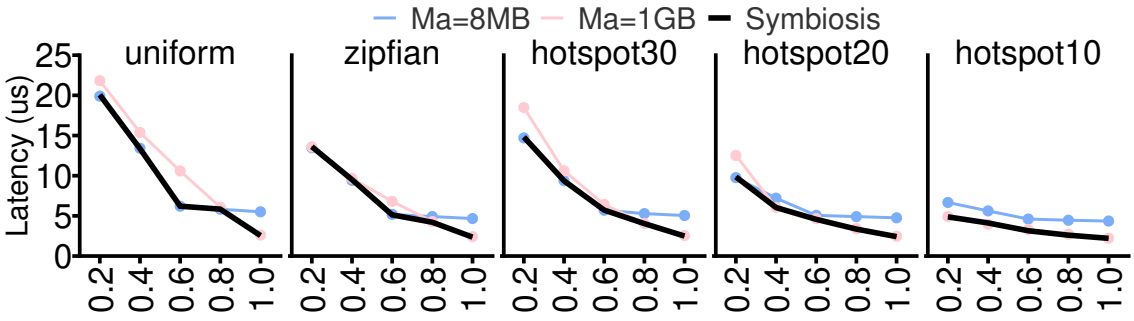
Large datasets and memory (a): To evaluate SYMBIOSIS in the context of modern data center machines with large amounts of memory, we begin with $M = 10GB$ and a range of large data sets (D_u=50, 25, 16.7, 12.5, 10 GB); we use the basic setting of HW1 and LevelDB's default compression ($\alpha = 0.5$). In all cases, SYMBIOSIS matches the performance of the better baseline. $Static_{M_a=8MB}$ tends to perform better when the data set is very large, and $Static_{M_a=1GB}$ when the data set size is small; the only exception is hotspot10, where the highly skewed accesses to the small hotspot should always reside in the application cache ($Static_{M_a=1GB}$). Again, SYMBIOSIS dynamically sizes the two caches to obtain the best observed performance.

Basic Setting (b): The setting is the same as (a), except to reduce the running time of our experiments, we use 1/10-th the data set sizes and $M = 1GB$. As desired, the full range of results are extremely similar to that of (a); thus, for efficiency, we use the smaller data set sizes and $M = 1GB$ in the remainder of our experiments.

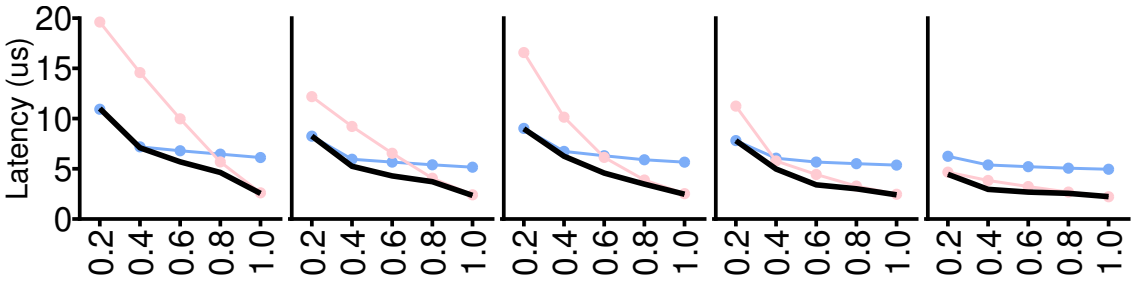
Different Compression Ratio (c): We change the compression ratio from $\alpha = 0.5$ in (b) to 0.22 in (c). With a smaller α, the performance gap between the two baselines increases, as noted in our offline simulations (§3.2). Thus, with better compression, SYMBIOSIS achieves a larger performance increase over the worse baseline (commonly $> 1.2\times$) and some im-



(a) Larger dataset and memory. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.



(b) Basic setting. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.

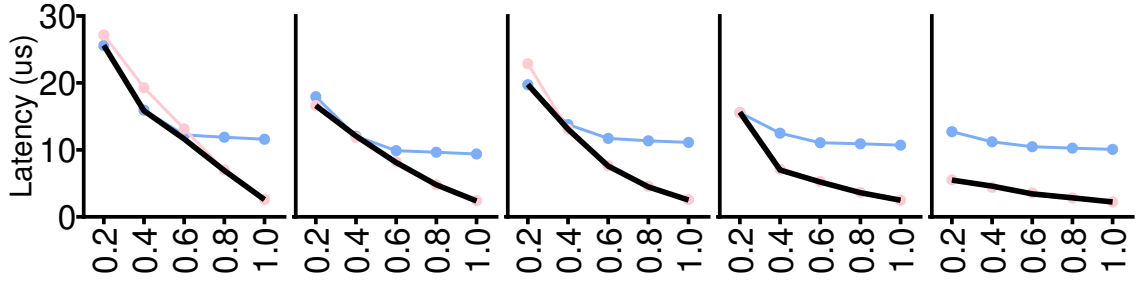


(c) Different compression ratio. $\alpha = 0.22$, $C_a = 3$, $C_k = 16$.

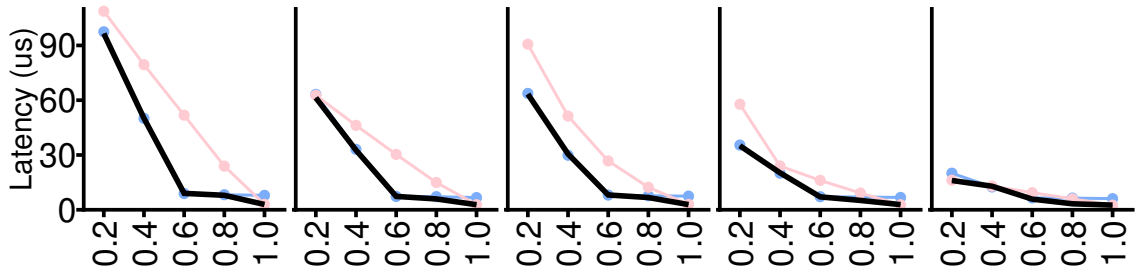
Figure 3.9: Performance under Static Workloads (Part 1). *X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $\texttt{Static}_{M_a=8MB}$, *similarly for Ma=1GB.*

provement over the best baseline (11.1% on average), especially when $M : D_u$ is within $[0.4, 0.8]$.

Different Compression Algorithm (d): We change the compression algorithm to alter α from 0.5 to 0.43 and C_a from 3 to 9. Now, $\texttt{Static}_{M_a=1GB}$ usually performs better than $\texttt{Static}_{M_a=8MB}$ because $\frac{C_a}{C_k}$ is large (0.56) and $\texttt{Static}_{M_a=8MB}$ incurs the cost of the higher C_a. Symbiosis again always matches the performance of the better baseline, properly



(d) Different compression algo. $\alpha = 0.43$, $C_a = 9$, $C_k = 16$.



(e) Different hardware (HW2). $\alpha = 0.5$, $C_a = 5$, $C_k = 80$.

Figure 3.9: Performance under Static Workloads (Part 2). *X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $\texttt{Static}_{M_a=8MB}$, *similarly for Ma=1GB.*

devoting most space to M_a, while correctly identifying the exceptions (e.g., the leftmost points in uniform and hotspot30).

Different hardware platform (e): We switch to HW2 so that device access is far slower than decompression ($\frac{C_a}{C_k} = 0.0625$). Now, $\texttt{Static}_{M_a=8MB}$ usually performs better than $\texttt{Static}_{M_a=1GB}$ because it avoids costly disk accesses, except for the hotspot10 workload where the cost of frequent application cache misses on the hotspot outweighs the benefit of reduced disk accesses. In several cases (e.g., $\frac{M}{D_u} = 0.8$), SYMBIOSIS performs significantly better than both baselines by properly balancing application cache misses and disk accesses, with an average gain of 6.9% over the better baseline.

Summary: In our LevelDB experiments, SYMBIOSIS achieves as high of performance as the better baseline and outperforms the other baseline by up to 5.77×. In some cases, SYMBIOSIS performs significantly better than both baselines (up to 1.32×), demonstrating the benefit of a fully flexible configuration of $<M_a, M_k>$.

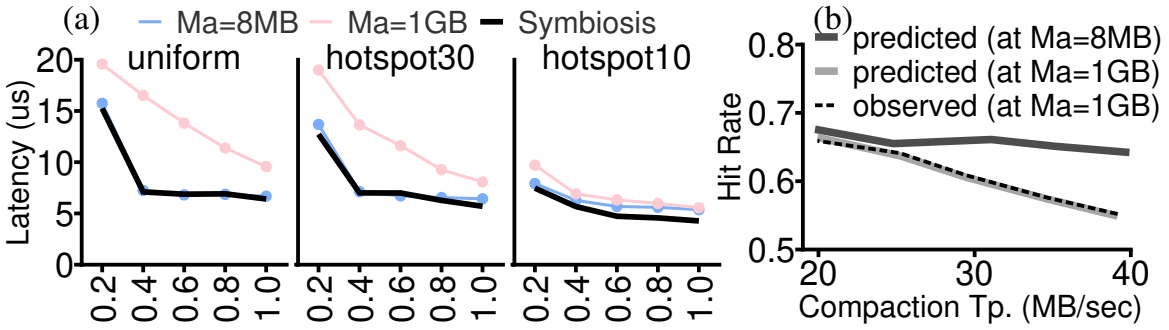


Figure 3.10: Static Workload with 20% Overwrites. *(a) X-axis is* $\frac{M}{D_u}$*. Ma=8MB means* $Static_{M_a=8MB}$*, similarly for Ma=1GB.* $\alpha = 0.22$*,* $C_a = 3$*,* $C_k = 16$*. (b) shows the predicted application cache hit ratio of the* $Ma = 1GB$ *configuration using cache traces from configuration* $Ma = 8MB$ *and* $Ma = 1GB$*, and the observed hit ratio when* $Ma = 1GB$*, under different compaction rates. The workload is uniform with 20% overwrite and* $M = D_u$*.*

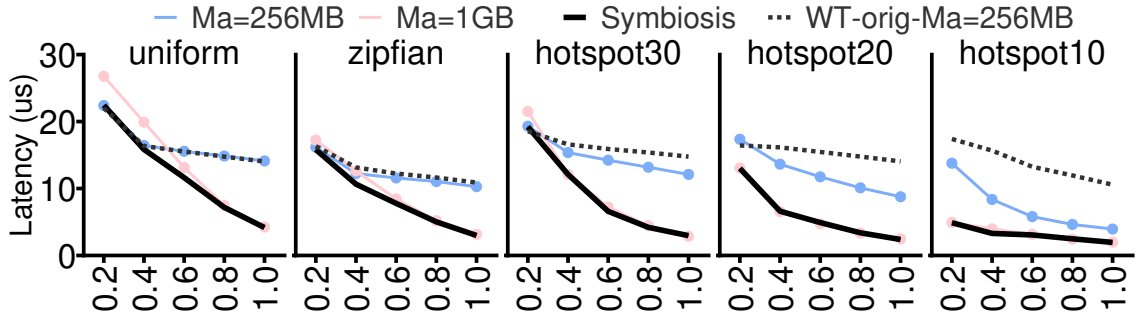
3.4.1.2 Workload with Writes in LevelDB

During simulations, Symbiosis uses cache access traces from the real system with a certain cache configuration, which deviates from the true cache access traces for other cache configurations when compaction exists. Figure 3.10(b) shows that Symbiosis's prediction is affected by such deviations under a large compaction rate. By limiting the compaction rate, the inaccuracy can be significantly reduced.

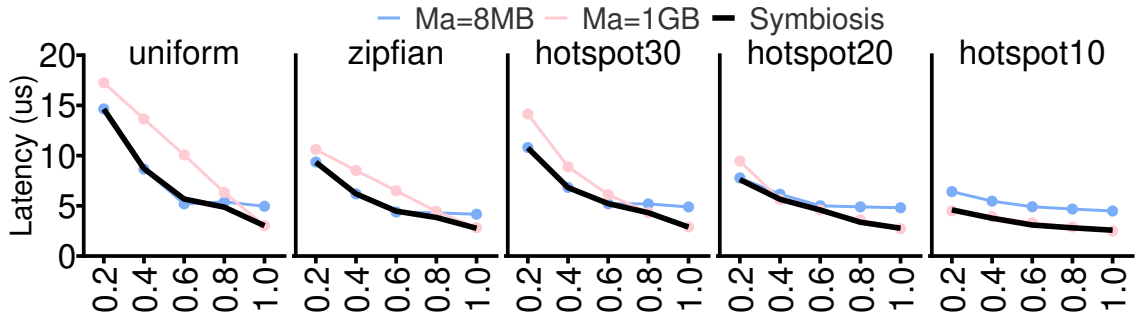
Figure 3.10(a) shows Symbiosis's performance with 20% overwrites. Compared to its read-only counterpart (Figure 3.9(c)), $Static_{M_a=1GB}$ performs worse than $Static_{M_a=8MB}$ even when $\frac{M}{D_u} = 1$ due to the immutable nature of LSM-tree that causes duplication with overwrites and makes the actual database size larger. Similarly, Symbiosis offers lower benefits, but still outperforms $Static_{M_a=8MB}$ when the workload is very skewed and D_u is small.

3.4.1.3 WiredTiger Performance

Figure 3.11(a) shows the performance benefits of incorporating Symbiosis into WiredTiger. As mentioned in §3.3.3, we began by modifying WiredTiger to correctly implement its claimed LRU-like behavior for its application cache; our modified version performs the same or better than the original version (*WT-orig Ma=256MB*) for all static workloads and is used in our baselines (*Ma=256MB* and *Ma=1GB*). WiredTiger has a significantly larger application cache miss penalty ($\frac{C_a}{C_k} = 1.25$) than LevelDB, so even with a very small compression ratio



(a) WiredTiger. $\alpha = 0.2$, $C_a = 20$, $C_k = 16$.



(b) RocksDB. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.

Figure 3.11: WiredTiger and RocksDB (Static Workload). *X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $\texttt{Static}_{M_a=8MB}$, *similarly for Ma=1GB. In (a), WT-orig-Ma=256MB is the original WiredTiger, while Ma=256MB, Ma=1GB, and* Symbiosis *uses our modified WiredTiger with LRU-like eviction policy.*

($\alpha = 0.2$), the baseline with a larger application cache (*Ma=1GB*) performs better than the other baseline for almost all workloads. Since WiredTiger's performance drops significantly when its cache size is less than its 256 MB default, Symbiosis searches for application cache sizes between 256 MB and 1 GB and outperforms or matches the better baseline, showing its capability on a completely different storage engine.

3.4.1.4 RocksDB Performance

Figure 3.11(b) shows the performance improvement when RocksDB uses Symbiosis to manage the sizes of its own decompressed and the compressed data cache. Making Symbiosis work with high accuracy is easier in this setting since we do not need to approximate complex kernel cache behavior. These results show a similar trend to that in Figure 3.9(a) where Symbiosis outperforms or matches the performance of the better baseline, demonstrating

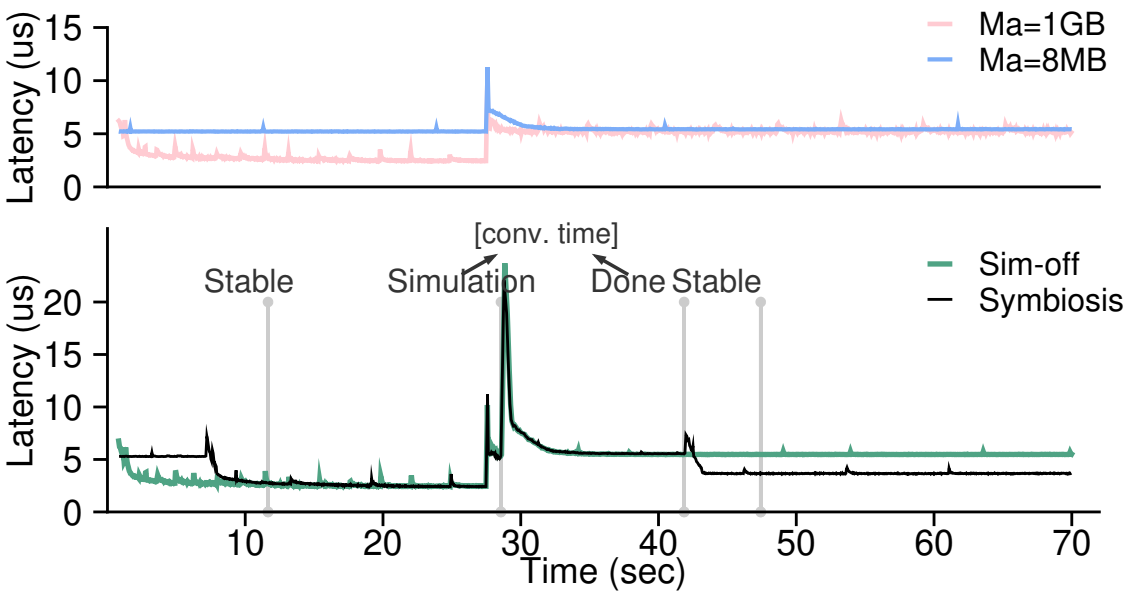


Figure 3.12: Timeline of Latency under a Dynamic Workload (hotspot20:1.0-2.0). *The workload changes are aligned at ~ 26sec, and we label state transfer of Symbiosis by the gray vertical lines. Sim-off means we turn off the simulation and shows the effect of only resetting the application cache size to default; its steady performance is the same as* $\texttt{Static}_{M_a=1GB}$ *before the change, and the same as* $\texttt{Static}_{M_a=8MB}$ *afterwards. (*$\alpha = 0.22$*)*

its capability to handle application-managed compressed data caches.

3.4.2 Dynamic Workloads

We demonstrate that Symbiosis adapts to workload changes with a reasonable convergence time and negligible overhead.

3.4.2.1 Example: LevelDB Behavior over Time

We begin by illustrating how Symbiosis within LevelDB behaves over time for a dynamic workload. Figure 3.12 presents the performance of Symbiosis (the bottom) and the two baselines (the top) for a workload with two phases; the access pattern in both phases is hotspot20 and $\alpha = 0.22$, but D_u varies from 1 GB to 2 GB.

The $\texttt{Static}_{M_a=8MB}$ baseline quickly obtains stable (but relatively poor) performance in the first phase, since the kernel cache can hold all the compressed data. When D_u increases, the latency increases while the kernel cache is warmed with the larger data set,

but eventually returns to its previous performance since the kernel cache can still hold all compressed data ($M_k \approx M > \alpha * D_u$ and $H_k = 1$).

The $Static_{M_a=1GB}$ baseline takes longer to warm the application cache in the first phase, but then achieves better performance since the application cache can hold all the decompressed data. When D_u increases, the latency increases because the application cache cannot contain all the data ($M_a < D_u$) and disk accesses are necessary.

SYMBIOSIS is able to obtain as good of performance as $Static_{M_a=1GB}$ in the first phase and better than both in the second. SYMBIOSIS starts with a default value for $M_a = 8$ MB while simulating cache configurations for $\sim 5sec$; after determining that $M_a = M$ delivers the best performance, it increases the application cache and matches the performance of $Static_{M_a=1GB}$ after the application cache is warmed at $\sim\ 12sec$. After SYMBIOSIS detects the significant increase in L_e at $\sim 28sec$, SYMBIOSIS defaults back to $M_a = 8$ MB and re-starts the simulations; the large initial overhead is due primarily to warming up the kernel cache (as shown by the Sim-off line which undergoes the same changes in cache configurations without simulation). Once the kernel cache is warmed, the simulation itself incurs negligible overhead (compared to $Static_{M_a=8MB}$) and finishes at $\sim 42sec$, at which point SYMBIOSIS changes to $M_a = 0.5M$, warms up the cache ~ 2 seconds, and then achieves the lowest latency.

3.4.2.2 Performance Gain and Dynamic Adaptation

To quantify the benefits, convergence time, and resulting cache configurations for a wide range of workloads with two phases, we construct a suite of 18 experiments varying D_u, access patterns, and α (0.22 and 0.5). We present the results with $\alpha = 0.22$ in Figure 3.13 ($\alpha = 0.5$ omitted for brevity) but consider both αs when discussing extremes and averages.

We use the example above to explain the metrics in Figure 3.13, which corresponds to hotspot20:1g→2g (the fifth bar group in the second row). Adjacent bars in the figure represent the two phases in an experiment. Latency is reported when performance is stable (e.g., in the example workload, latency is about 2.5µs for SYMBIOSIS and $Static_{M_a=1GB}$ for the first phase, and 5µs for $Static_{M_a=8MB}$; it is about 3.7µs for SYMBIOSIS and 5µs for $Static_{M_a=8MB}$ and $Static_{M_a=1GB}$ in the second phase). Convergence time represents the time to finish simulation (e.g., ~ 12 and 13 seconds for phase 1 and 2, respectively, shown by the time between the bars labeled as Simulation and Done in Figure 3.12). Finally, the

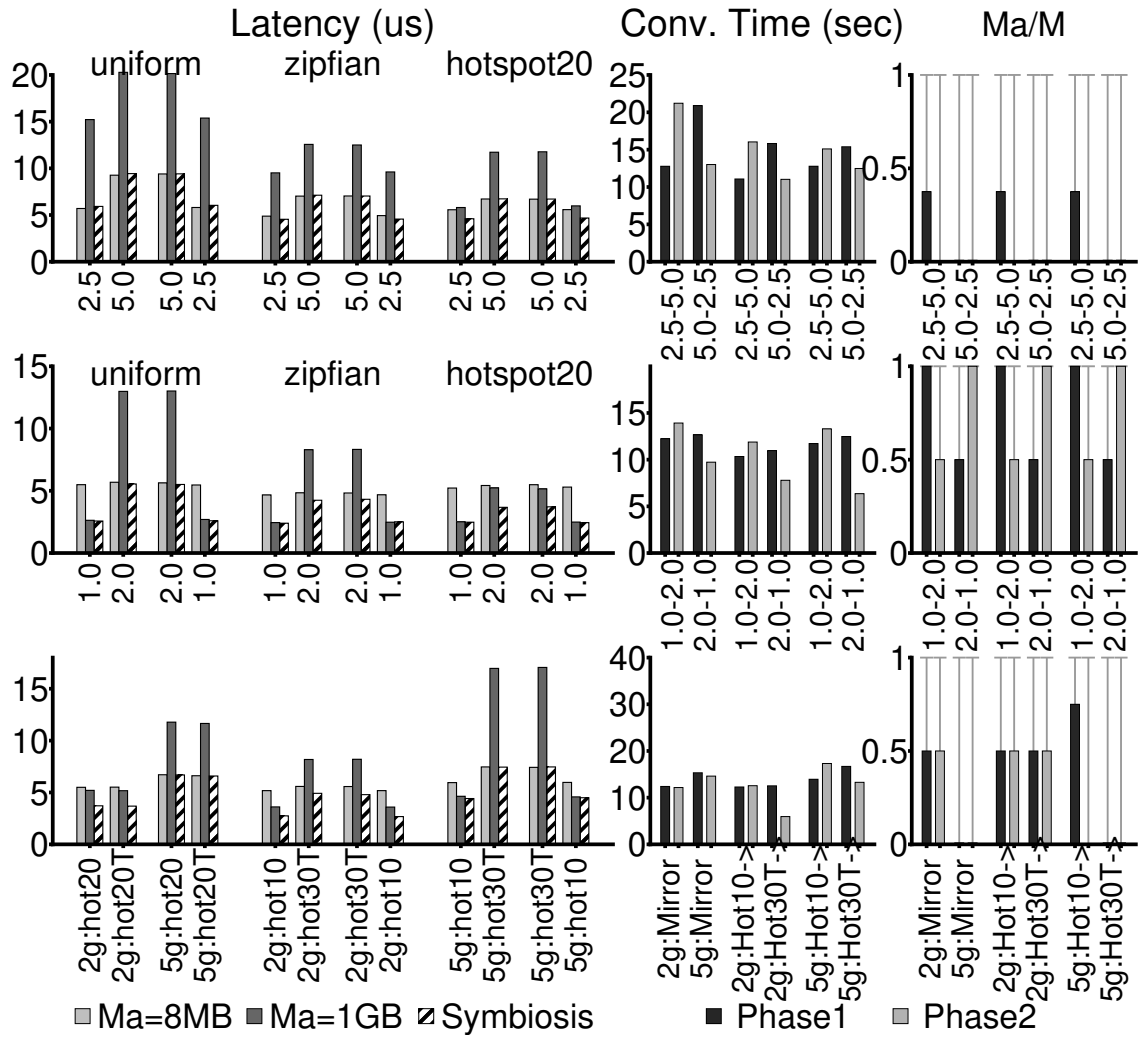


Figure 3.13: Performance under Dynamic Workloads ($\alpha = 0.22$). *In the Latency subplot, each group has three bars:* $\text{Static}_{M_a=8MB}$, $\text{Static}_{M_a=1GB}$ *and* Symbiosis. *Each adjacent bar group represents one workload1→workload2 change and the next group reverses the workloads. The first two rows contains 12 workloads where* D_u *varies (shown in the x-axis labels). The third row contains 2 workloads varying hotspot positions and 4 varying hotness and hotspot positions, each with a fixed* D_u. *For instance, 2g:Hot20 means a hotspot20 workload with* $D_u = 2$ *GB and 2g:Hot20-T mirrors the hotspot to the tail. 2g:Hot20→2g:Hot20-T is summarized as 2g:Mirror (hotspot change). The Conv. Time and Ma/M subplots only show the behaviors of* Symbiosis.

M_a/M subplot shows the best application cache size found by Symbiosis (e.g., 1 and 0.5 for the example workload).

Figure 3.13 shows that Symbiosis delivers good latency in all cases, at least as good as the

	p-95 Latency Median	p-95 Latency Max	p-99 Latency Median	p-99 Latency Max
Overhead (%)	8.6	14.5	15.3	52.0
Case	-	zipfian:1g→2g	-	uniform:1g→2g

Table 3.2: Tail Latency. *Overhead is the comparison to* $\texttt{Static}_{M_a=8MB}$. *(*$\alpha = 0.22$*)*

best baseline and sometimes better, with an average gain of 24% over $\texttt{Static}_{M_a=8MB}$, 42% over $\texttt{Static}_{M_a=1GB}$, and a best case of 42% over the better of the two (i.e., hotspot20:1.0→2.0). The average convergence time is 15.4 seconds with a worst case of 40 seconds; generally, more convergence time is required for larger D_u and D_c, and for less skewed workloads. During simulation, the worst overhead of Symbiosis is 15.1%, but this contains two portions: the larger is the overhead of possibly resetting M_a to the default and warming up the kernel cache; the smaller is the actual simulation overhead, which averages only 0.9% with a worst case of 3.4%. Finally, Symbiosis chooses different M_a values, typically scaling up M_a with a decrease in D_u and increase in skewness (and vice versa).

Adapting the size online and potential latency spike symptoms raises concerns of tail latency. As shown in Table 3.2, Symbiosis incurs reasonable tail latency overhead, with a 8.6% higher median p-95 latency and a 15.3% higher median p-99 latency compared to $\texttt{Static}_{M_a=8MB}$. Out of the 18 cases, 13 have less than 25% overhead for p-99 latency. The highest p-99 latency overhead is 52% in uniform:1g→2g. Extra device accesses due to cache size change cause the higher tail latency. Tail latency would be minimally impacted in workloads with a longer steady state or more device accesses.

3.4.2.3 Gradual Change

We show that Symbiosis also performs well in workloads with more gradual changes (Figure 3.14). During the workload, $\texttt{Static}_{M_a=8MB}$ holds all the data in the kernel cache; $\texttt{Static}_{M_a=1GB}$ cannot hold all the data in the application cache when $D_u = 2$ GB and performs worse, but then benefits from the shrink of D_u and finally eliminates device access when $D_u = 1$ GB and performs better than $\texttt{Static}_{M_a=8MB}$.

Symbiosis matches the performance of $\texttt{Static}_{M_a=8MB}$ at the beginning. Three simulations are triggered when the difference of L_e reaches the threshold for workload change detection, M_a is gradually increased according to the workload when simulations occur,

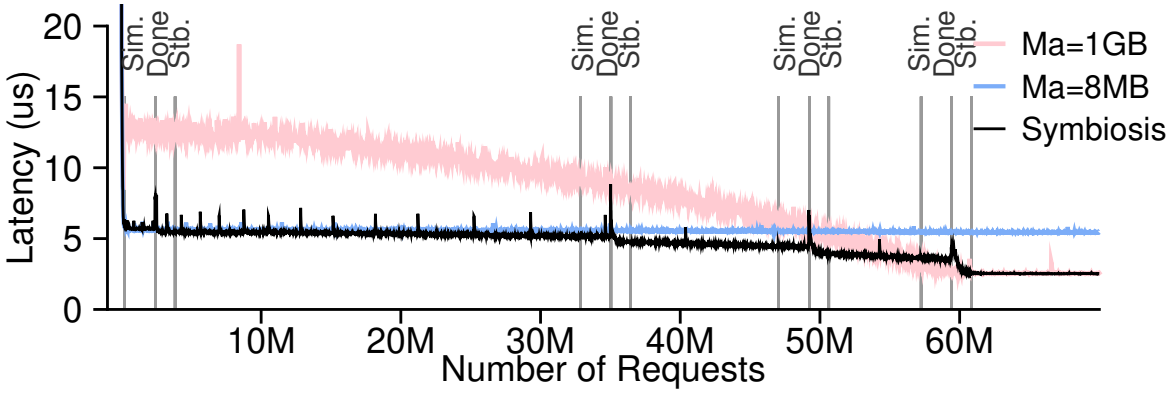


Figure 3.14: Timeline of Latency under a Dynamic Workload with Gradual Change. *The workload is uniform with $D_u = 2$ GB in the first 10M operations, $D_u = 1$ GB in the last 10M operations, and a uniform gradual change during the 50M operations in between. ($\alpha = 0.22$)*

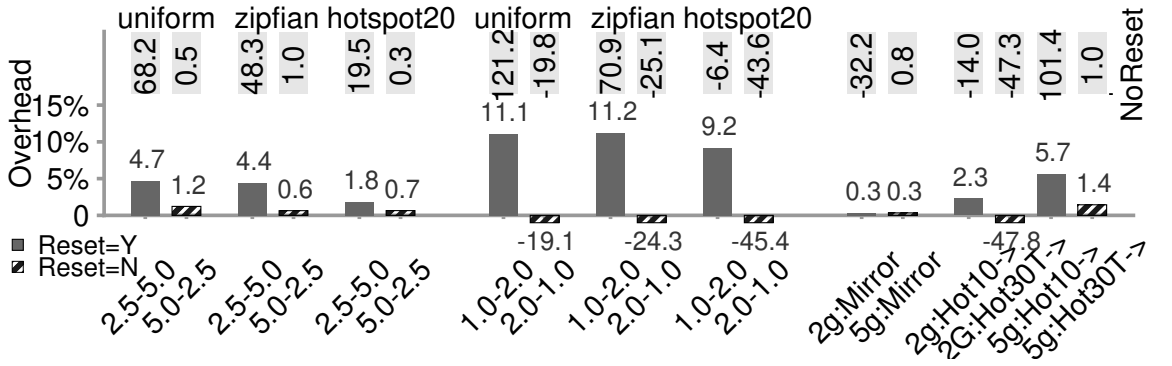


Figure 3.15: Overhead during Simulation ($\alpha = 0.22$). *The workloads are in the same order as in Figure 3.13. The bars are the overhead with the reset policy; dashed ones indicate no actual M_a change. Numbers in gray background are the overhead percentages without the reset policy.*

and the latency drops along with the shrink of D_u. Finally, $M_a = M$ is chosen when D_u approaches 1 GB and the performance of $\texttt{Static}_{M_a=1GB}$ is matched.

A gradual change of L_e is necessary for Symbiosis to match the change speed of workload. For workloads with faster changes beyond Symbiosis's threshold during simulation, simulations are halted until the workload stabilizes.

3.4.2.4 Effect of Optimization Techniques

We quantify the benefits of our techniques by comparing to a simplified version without the corresponding technique.

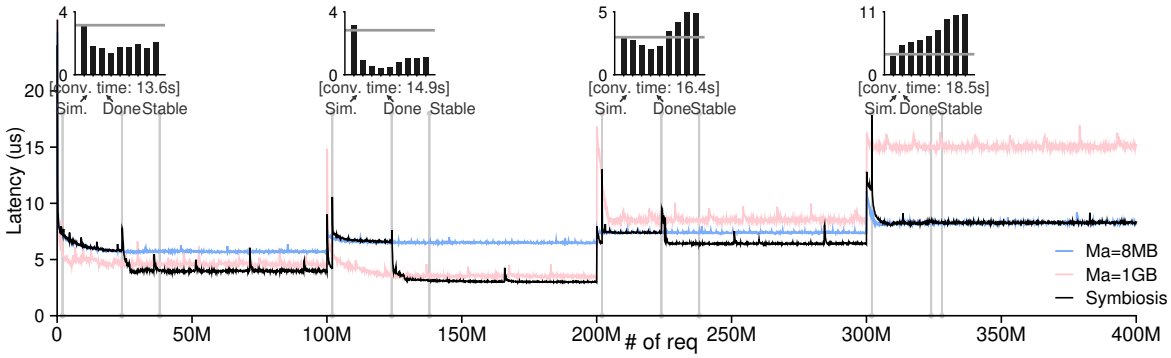


Figure 3.16: Request Latency versus the Request Sequence. *The 4 phases are composed of 2 workloads generated from RocksDB's* mix_graph *benchmark. Two versions of the first workload exhibit a decrease in* D_u*, with* $Key_{max} = 50M$ *and* $D_u = 5$ *GB in phase 1 and* $Key_{max} = 25M$ *and* $D_u = 2.5$ *GB in phase 2. Similarly, two versions of the second workload exhibit a increase in* D_u *s (phase 3:* $D_u = 2.5$ *GB and phase 4:* $D_u = 5$ *GB). The four small bar charts around the top illustrates the decision of Tracker; each chart is a simulation round. Each bar represents one simulated cache size setting (*$S_{\{0,...,8\}}$ *from* $M_a = 8$ *MB to* $M_a = 1$ *GB), y-axis is the* L_e *(expected latency), and the gray horizontal line shows the real system* L_e *at the time of simulation end. Tracker adopts the first three size changes, but rejects the last one; all four are good decisions. (*$\alpha = 0.22$*)*

Reset Policy: The reset policy (§3.3.2.1) aims for a cache size that performs reasonably while simulating, despite an arbitrary new workload. The overhead of Symbiosis compared to the $Static_{M_a=8MB}$ baseline during simulation is shown in Figure 3.15; large negative values occur when Symbiosis does not reset M_a to default due to a decrease in L_e and thus Symbiosis performs better than the baseline (e.g., the uniform:2g→1g experiment). As shown by the overhead numbers in gray background in Figure 3.15, Symbiosis without the reset policy performs poorly in many cases (e.g., up to 100×); therefore, the reset policy is better on average and beneficial for more stable performance.

Sampling: Sampling is essential for low overheads. The first and third row in Table 3.3 shows the memory consumption and operation overhead of Symbiosis with and without sampling. Without sampling, simulation consumes 51 MB of memory and adds 42% of overhead to every operation. Sampling significantly reduces the costs, consuming only 460 KB of memory and incurring only ~90ns per operation. Furthermore, sampling only adds the overhead over the 16.7 second simulation round – a negligible duration.

Incremental reuse of ghost cache: By comparing rows two and three in Table 3.3, we see that incremental reuse reduces both memory and time overhead by $> 3\times$, but at the cost of

a longer convergence time, compared to a design that simply uses one ghost cache instance for each candidate <M_a, M_k>. Thus, the incremental reuse design has the lowest impact on foreground workload and is most suitable.

3.4.3 Real World Workloads

We conclude by demonstrating that SYMBIOSIS handles complex and realistic workloads: performance is robust since only a size change that is predicted to sufficiently improve performance is adopted.

Two workloads generated from RocksDB's `mix_graph` benchmark [26] are used, the first with the supplied parameters in the last example in paper [26], and the second mimicking an interesting two hot key-range symptom in the paper, observed by Meta's ZippyDB Get workload. The benchmark models key-space localities and closely approaches real workloads in terms of storage I/O statistics.

Figure 3.16 shows the performance of LevelDB on four consecutive traces based on the two workloads. $\mathtt{Static}_{M_a=8MB}$ maintains relatively constant performance through the four phases with $H_k \approx 1$, as the kernel cache holds most of the compressed data across all phases. $\mathtt{Static}_{M_a=1GB}$ outperforms $\mathtt{Static}_{M_a=8MB}$ in the first and the second phase because the workload is very skewed (over 70% of requests access 1/30 of the data), and the gain of hitting in the application cache for most accesses outweighs the additional disk accesses for the data that does not fit; however, in the third and fourth phases, $\mathtt{Static}_{M_a=1GB}$ performs worse than $\mathtt{Static}_{M_a=8MB}$ as the workload becomes less skewed, with 80% of requests accessing 40% of the data, lowering H_a.

SYMBIOSIS finds a <M_a, M_k> as good as (and often better than) the better static configuration in every phase of the complex production workload. To illustrate why SYMBIOSIS is robust, the small bar charts show the predicted L_e of <M_a, M_k> candidates from $M_a \approx 0$ to $M_a = M$ and the real L_e (gray line) during each simulation. For each simulation, SYMBIOSIS resets $M_a = 8$ MB. In the first three phases, the best candidate is $M_a = \frac{3}{8}M$ and its L_e is much lower than the real L_e, so SYMBIOSIS applies it to the real system and outperforms both two baselines. In the last phase, the best candidate is $M_a = 8$ MB which is the default value that SYMBIOSIS currently takes, so it keeps the default M_a and matches the performance of the better baseline $\mathtt{Static}_{M_a=8MB}$.

Case	Memory Overhead (MB)	Operation Overhead (us/op)	Convergence Time (s)
Reuse & No Sampling	51	2.8 (42%)	22.9
No Reuse & Sampling	1.5	0.32 (4.8%)	7.35
Reuse & Sampling	0.46	0.09 (1.3%)	16.7

Table 3.3: Space and Time Overhead and Convergence Time of Various Simulation Settings. *Operation overhead compares to baseline LevelDB. Sample rate is* $\frac{1}{64}$.

3.5 Conclusion

We have introduced Symbiosis, a framework to enable robust cache adaptation for key-value storage systems. With the knowledge of the underlying kernel cache, Symbiosis optimizes the cache sizes of storage engine applications to improve the overall cache efficiency. With careful study of the performance space, we develop an online simulator which enables a live key-value storage system to adapt its application cache size and achieve high performance. Across a wide range of workloads and settings, we demonstrate the overall benefits of our approach, as shown through implementations in three production key-value storage systems: LevelDB, WiredTiger, and RocksDB. We open source our framework, workloads traces, modified systems, and utilities to facilitate further investigation [6].

Chapter 4

Kelvin: Towards Zero Copying and Duplication in Data Pipelines

In this chapter, we introduce Kelvin, a true zero-copy data pipeline execution engine that avoids data copying and duplication to the best of our knowledge. Kelvin co-designs different layers of the data processing stack (user-space resource management, container runtime, and kernel support for shared memory) to improve memory efficiency for data pipelines.

We first give background regarding DAG-based data pipelines, shared memory, zero-copy communication, and related challenges (§4.1). We identify three types of data copying and duplication caused by data origination, inline message data, and data deserialization, and related challenges such as resource accounting and isolated execution.

We then design Kelvin (§4.2) to solve these challenges and implement it on Linux (§4.3). The core component of Kelvin is a kernel module called DeAnon that zero-copy converts anonymous memory to shared memory by only modifying the metadata of relevant kernel data structures. Around the core, we build several components, including a shared IPC protocol that utilizes DeAnon to generate shared IPC data without copying and a resource manager that manages the physical data in a sharing environment. Our DAG executor is built upon OpenLambda [110] which provides the basic building blocks for containerized execution and resource management.

We evaluate Kelvin's performance with various micro- and macro-benchmarks. We demonstrate the benefit of Kelvin's components with micro-benchmarks and show that the

benefit remains with the ecosystem benchmark that runs end-to-end DAGs with various popular toolchains. Moreover, we use the Data Agent Benchmark for Multi-step Reasoning (DABstep) [8, 9] to show that Kelvin provides 1.2× to 2.3× performance gain on real, complex DAGs.

4.1 Background and Motivation

In this section, we first describe what DAG-based data pipelines are and how they fit in single machine (§4.1.1). We describe existing kernel support for shared memory (§4.1.2), on which user-space zero-copy techniques (§4.1.3) are based. In principle, these techniques should enable very efficient data passing. Unfortunately, the unique requirements of container-based DAG pipelines result in many scenarios involving copying or duplication (§4.1.4).

4.1.1 Background: DAG-based Pipelines on a Single Machine

Data pipelines are a popular paradigm for data analysis and machine-learning workloads. Data pipelines are frequently implemented as DAGs (Directed Acyclic Graphs), where each node of a DAG describes a transformation to perform on the data [13, 132, 141, 169]. Edges represent data passing between nodes. Given fine-grained nodes, efficient communication along edges is critical for good performance [64, 127, 163]. Data passing along edges may occur via network, disk, or memory [36, 99, 100, 115, 169]; the specific medium depends on node placement and resource availability.

Recent workload and hardware trends [104, 140] demonstrate the feasibility of deploying most data pipelines on a single machine with ample memory, with memory as the sole and most efficient medium for data passing. Cloud virtual machines with 24 TB of RAM are now available [125]; in contrast, the largest (p99.9) datasets for OLAP workloads occupy a mere 0.25 TB [118]. Memory is increasingly affordable as well: from 2014 to 2023, cost has dropped from $4K to $1K for 1 TB of RAM [112]. Processing data locally has several advantages, such as saving the data communication overhead and reducing the maintenance efforts of computing resources.

However, running multiple nodes locally calls for isolation between them. A strong isolation between nodes can provide better resource accounting and failure recovery and allow different nodes to have different library dependencies. Containers [20, 110] are widely

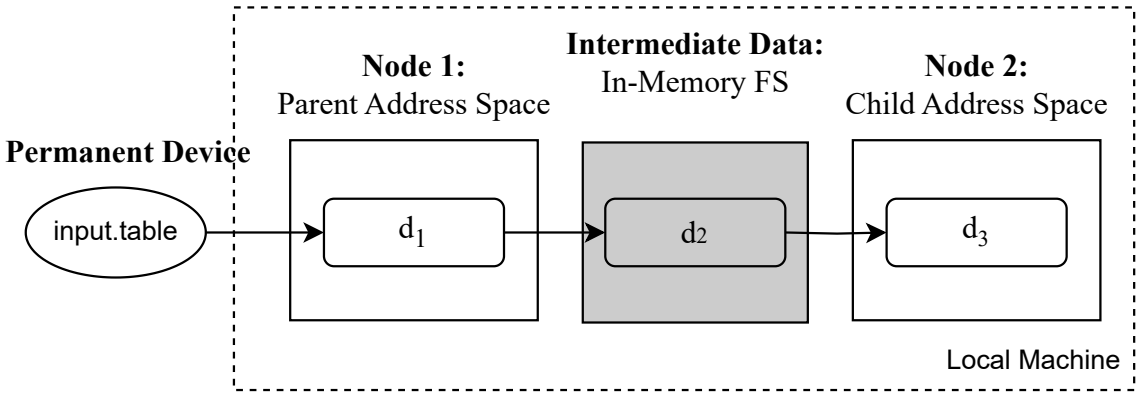


Figure 4.1: A Minimal DAG on a Single Machine.

used to provide lightweight isolation with kernel support. Different nodes have separated address spaces in containerized execution.

Figure 4.1 illustrates a minimal DAG on a single machine. The DAG contains two nodes. The parent node loads a table from a permanent device (could be remote) and generates the in-memory form (d_1) of the table. The parent node then passes the data to shared memory (d_2) so that the data is accessible by other nodes. The child node then puts the data in its own address space (d_3) and continues processing the data. Note that the address spaces of the two nodes and the shared memory are on the same physical memory of a single machine. Simply passing data by value would result in 3 local copies of the same data.

The ideal way to pass in-memory data between nodes in a DAG is by reference: if downstream processes in a data pipeline can virtually map upstream outputs into their virtual address spaces, copying and duplication overheads can be avoided.

4.1.2 Background: Kernel Shared Memory

Operating systems have implemented shared memory in a variety of ways. Implementations often interact with other systems, such as in-memory file systems, containers, and swap. Understanding these interactions is key to building systems that leverage shared memory for efficient data passing. In order to give specific examples, we describe Linux's support for shared memory.

Like most kernels, Linux gives every process its own virtual address space, internally implemented with (1) a page table that maps virtual pages (usually 4 KB) to physical memory

and (2) metadata describing features of different address ranges (e.g., permissions, and whether a range is mapped from a file). Memory sharing occurs when distinct processes have virtual pages mapping to the same physical pages.

Common sharing scenarios include bidirectional communication via writable pages and fast process fork via copy-on-write pages. These are not directly applicable to data pipelines, as DAG communication is unidirectional and nodes may have multiple parents (in contrast to process trees). DAG-specific use cases for shared memory include (1) avoiding unnecessary memory consumption that occurs when different processes have their own copies of the same data in physical memory, and (2) passing data faster by passing references instead of copying bytes of data.

Linux supports multiple memory-sharing APIs. The newer POSIX API integrates with an in-memory file system, *tmpfs*. Shared objects are represented as files in a tmpfs instance, typically mounted at /dev/shm. Sharing is achieved when multiple processes call `mmap` (or similar) to create file-backed virtual-memory areas (VMAs) referring to the same in-memory file. Generally, each process mapping the same physical data will do so at different virtual addresses; this has implications for pointer-based data structures.

Memory sharing also has implications for container accounting. Operating systems measure per-container memory consumption in order to take kill or swap actions when memory limits (if configured) are exceeded. An operating system must decide which container(s) to charge for physical data shared across containers. Seemingly obvious approaches lead to anomalies; for example, charging evenly across all containers means that when one container exits, consumption for other containers will spike, with a potential cascade of container kills and further consumption spikes.

On Linux specifically, containers and KVM-based micro-VMs rely on cgroups (control groups) to apply limits for memory and other resources to groups of processes. Linux's accounting rules for file-backed pages depend on the file system type. For tmpfs (which backs shared memory) a cgroup "owns" the file data written by processes in the cgroup, so those file bytes will count toward a container's memory consumption. If a cgroup is deleted, the charge is reassigned to the parent cgroup (cgroups are hierarchical).

Linux cgroups are tightly integrated with swap. Each cgroup has its own set of LRU queues for different types of pages, and under memory pressure, physical pages at the start of the queues are evicted from memory. Thus, cgroup limits will influence which in-memory files will be swapped to disk under memory pressure.

4.1.3 Background: User-Space Sharing

Kernel sharing support provides the building blocks necessary to construct share-friendly formats and protocols in user space. Different processes mapping the same data often do so at different addresses, so sharing usually requires data formats that avoid pointers, which are only valid in a particular address space. In this section, we describe a specific zero-copy (i.e., share-friendly) format optimized for analytics workloads: *Arrow*. The Apache Arrow project encompasses many related efforts, including: (1) a specification of a column-oriented tabular format, (2) a collection of implementations of the format, and (3) a communication protocol.

Tabular Format: Arrow uses indexes and null bitmaps to represent column data without pointers; tabular data is represented as a collection of columns. Indexing over contiguous arrays of fixed-length elements (e.g., float64s) is straightforward. Variable-length values such as strings, binary values, and lists are also stored contiguously in a value buffer (i.e., the first byte of a string at index N+1 immediately follows the last byte of a string at index N). A contiguous array of offsets is stored separately to provide indexing over these variable-length values. A null bitmap can indicate values at certain indexes should be considered null, regardless of the actual value stored. Sometimes, long strings or byte sequences may appear repeatedly in a column; Arrow supports an optional dictionary encoding where each unique value is stored once in a dictionary, and an array of numeric codes is used to reference specific values. Arrow data is uncompressed, immutable, and formatted such that computation can be performed on data in-place, often with SIMD instructions.

Implementations: The Apache Arrow project includes over 10 language-specific libraries; tools in the Arrow ecosystem may rely on these or directly implement the Arrow format. Cross-language calls mean that libraries and Arrow implementations need not match languages. For example, Polars, PyArrow, and DuckDB are Python packages that respectively depend on (1) an Apache Arrow implementation in Rust, (2) an Apache Arrow implementation in C++, and (3) an independent Arrow implementation in C++.

Communication: Arrow IPC (Inter-Process Communication) specifies a protocol for sending data between processes, perhaps over a network. The protocol defines several message types: a *schema* defines columns, a *dictionary message* assigns numeric codes to values, and a *record batch* contains inline data in the Arrow format. Using the regular Arrow format for record batches helps Arrow IPC avoid copies: once Arrow IPC messages are read

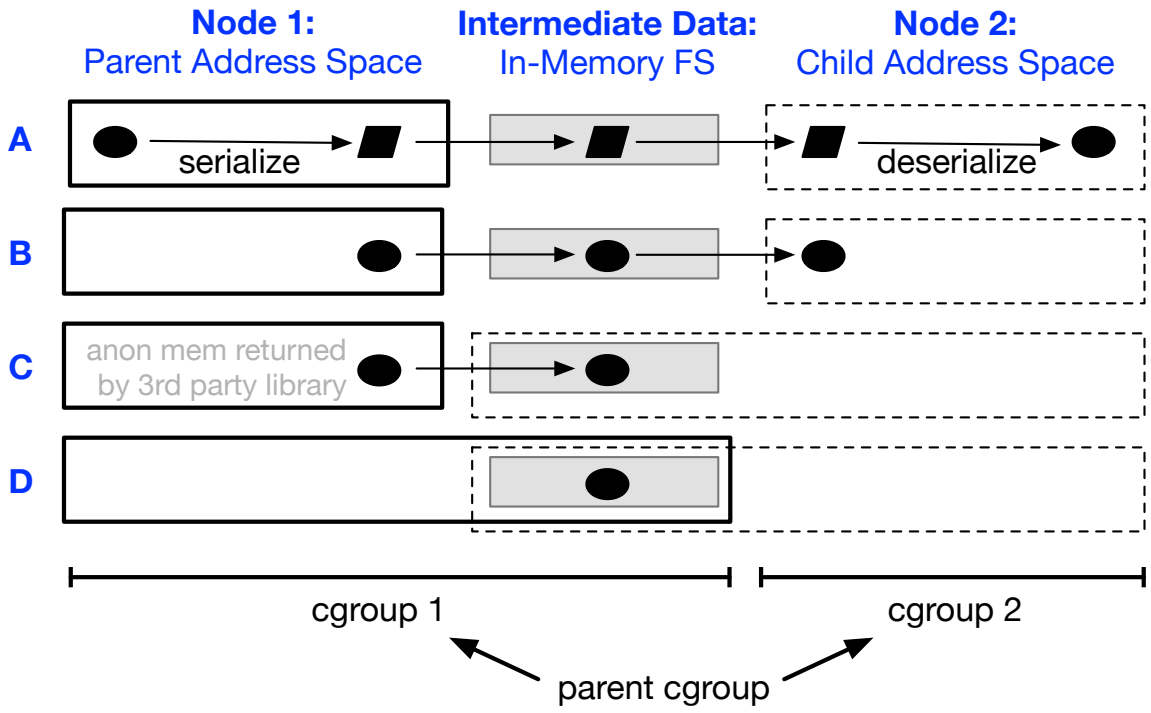


Figure 4.2: Sharing Challenges: Memory Management.

to memory, a tabular view may be constructed over the Arrow-formatted messages, in place.

4.1.4 Requirements and Challenges

In this work, our goal is to build a data pipeline platform based on one overarching principle: *share memory whenever possible to avoid copying and duplication.* We additionally impose three requirements: (1) developers may write arbitrary code for nodes, (2) that code may use any library in the broad Arrow ecosystem, without modification, and (3) code for each node must execute in its own container to limit resource consumption and provide isolation. We build upon Linux shared memory and Arrow's zero-copy protocols, but many obstacles to sharing must still be overcome.

Challenge 1: Pointers. Computational formats are often based on pointers, which do not work when shared data is mapped to different virtual addresses. Figure 4.2A shows a common scenario: frequent copying between between wire and computational formats and across mediums. Arrow avoids serialization by using the same format for wire data and

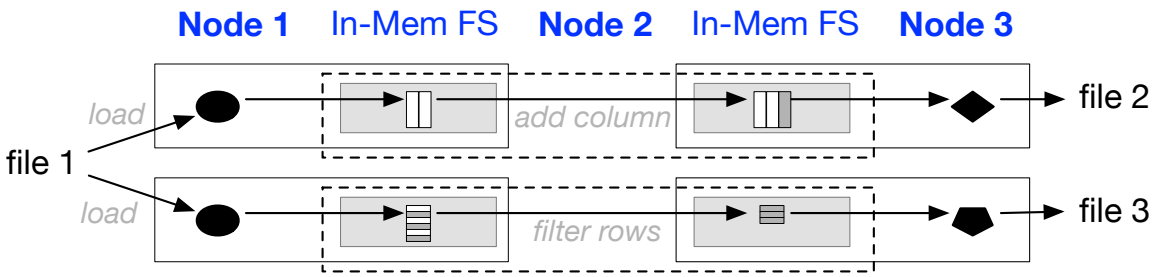


Figure 4.3: Sharing Challenges: Data Transformations.

computational data (Figure 4.2B). Even better, if the data is in an in-memory file system, readers can directly map the data to their address spaces (Figure 4.2C). Simply using Arrow and Linux `mmap` resolves the pointer challenge.

Challenge 2: Data Origination. Ideally, we would like a node producing data to directly populate shared memory (Figure 4.2D). Libraries that generate Arrow data may either (1) accept a reference to memory and populate it or (2) allocate memory and return a reference to it. The first case is amenable to sharing (the platform can allocate shared memory and pass a reference), but the latter is more common, perhaps because a library can better calculate the allocation size needed. Libraries that allocate memory for Arrow data typically do so via *malloc* (or similar), which allocates from *anonymous* memory (i.e., memory not backed by a file). Linux shared memory is backed by tmpfs files, so Arrow data in anonymous memory is not shareable without a copy.

Challenge 3: Containerization. Regardless of whether we achieve Figure 4.2C or Figure 4.2D (preferred), the intermediate tmpfs data will be charged to the container/cgroup of node 1. If the node 1 container exits, the charge will pass to the parent cgroup, making it difficult to track the overall memory consumption of the DAG or control swap.

Challenge 4: Inline Message Data. Node 2 in the top DAG of Figure 4.3 is adding a third column to a table that is calculated based on the values in the first two columns. Node 2 directly maps the outputs from node 1 into its own address space as input. Node 2 also directly populates shared memory with its outputs (this assumes the Data Origination problem has been addressed). Unfortunately, the message format (in this case, Arrow IPC) includes data inline rather than referencing data elsewhere; thus, Node 2 must copy two columns from the input to output, even though both are in shared memory. Adding a column is one example where overlap between inputs and outputs takes place; other

examples include slicing and concatenating.

Challenge 5: Data Granularity. Node 2 in the bottom DAG of Figure 4.3 is filtering input rows according to some condition to produce output rows. Although there is overlap between inputs and outputs, rows may be smaller than pages (often 4 KB), so shared memory cannot readily be used to create filtered and unfiltered views over the same physical rows without duplicating those rows.

Challenge 6: Deserialization. Copying is not the only way to end up with two copies of the same data in physical memory; duplicates also occur when multiple copies are created independently based on a common source. Figure 4.3 shows one common case: two independent DAGs each start by deserializing data from persistent storage (e.g., a Parquet file in a cloud bucket) to Arrow.

There are many challenges to using shared memory for data pipelines, and unfortunately using Linux support for sharing and a zero-copy format only solves the first problem, pointers. In this work, we explore techniques for addressing the remaining challenges.

4.2 Kelvin Design

We now introduce Kelvin, a new data pipeline platform that uses shared Arrow data to efficiently pass intermediate data between containerized DAG nodes with minimal copying and duplication. Using Arrow solves *Challenge 1* (Pointers) because Arrow data can be mapped to different virtual addresses without problems. Kelvin uses a mix of existing and novel techniques to overcome the five remaining obstacles to sharing (§4.1.4). We first describe how we build new kernel support for de-anonymizing data generated by unmodified libraries so that it can be efficiently shared across containers (§4.2.1). Next, we build zero-copy communication based on Arrow IPC that (a) leverages de-anonymization support to avoid copies to shared memory and (b) uses IPC inspection and dictionary sharing to avoid copying between inputs and outputs (§4.2.2). Finally, we describe how memory consumption is tracked and managed in a cohesive way across containers, source data, and intermediate data (§4.2.3).

4.2.1 Shared-Memory Mechanisms

We allow developers to write arbitrary code and use unmodified libraries in the Arrow ecosystem. This leads to *Challenge 2* (Data Origination). Libraries often allocate from

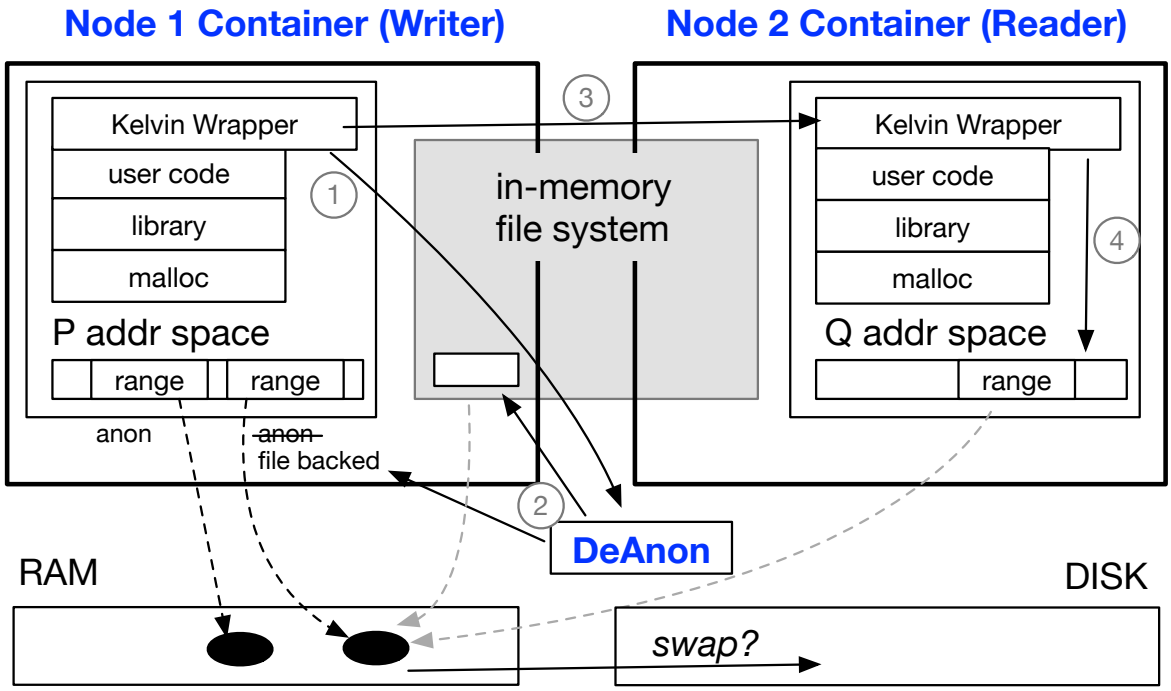


Figure 4.4: KELVIN Memory Management.

anonymous memory regions (via `malloc`); we want to avoid copying that anonymous data to shared memory.

Thus, we implement new kernel support for de-anonymizing memory in place. Specifically, a *DeAnon* kernel module converts anonymous memory to file-backed memory. DeAnon allocates new in-memory files and updates metadata in a process's address space to indicate that the specified range corresponds to a region of the new file. DeAnon transfers ownership of previously anonymous memory to the new files. This transfer does not involve copying actual data, and afterwards both the process's address space and the in-memory file will point to the same physical memory. DeAnon returns identifiers for the new files and offsets corresponding to the de-anonymized ranges.

Figure 4.4 shows how KELVIN uses DeAnon to pass data between two containerized processes without a copy. Both containers mount the same in-memory file system, in which files will be created via the de-anonymization process. In KELVIN, containers always start by executing KELVIN wrapper code, which assists with data passing and invokes the custom user-defined code. The user code is expected to return output back to the wrapper. The output typically references data in anonymous memory.

The write path proceeds as follows: (1) the wrapper invokes DeAnon, which (2) transfers ownership of the relevant anonymous memory to an in-memory file and updates the region so that it is file backed. DeAnon returns information for referencing the newly anonymized data, and (3) the wrapper includes that information in the container's output, which becomes the input for downstream nodes. In this case, process Q in Node 2 will use the received information to (4) map the in-memory file into its address space.

4.2.1.1 Resource Accounting for Containers

Regardless of which processes eventually map the in-memory file, space in the file is charged against the memory limit of the container that produced the file, as illustrated in Figure 4.4 by the placement of the file within both the file system and container boundaries. This leads to *Challenge 3* (Containerization): how can we track and control the consumption of in-memory files produced by containers?

In order to retain control over intermediate data after a node finishes running, we allow the process in the container to exit upon node completion, but retain the container itself until the whole DAG completes. We also borrow a technique, *limit dropping*, from an early version of Senpai [156]. With this approach, a container's memory limit is dropped to trigger swap. This allows us to select specific intermediate outputs (which still count toward the container's memory limit) that should be swapped to disk, if system memory runs low. Prior to running downstream nodes, the limit for an upstream container is raised again so that the data can be swapped back in as needed without triggering corresponding swap outs.

4.2.2 Zero-Copy Communication

The de-anonymization and container management techniques described (§4.2.1) provide a foundation on which to implement share-based communication protocols. For this purpose, Kelvin uses Arrow IPC, with modification. Using a protocol such as Arrow IPC leads to *Challenge 4* (Inline Message Data): including record batch data in IPC streams is a sensible approach for network-based communication, but on a local machine where shared memory is available, it results in writer-side copies from the data to the output stream, even if the data is in shared memory. Thus, we extend the Arrow IPC protocol, creating SIPC (Shared IPC), which can reference a range of an in-memory file.

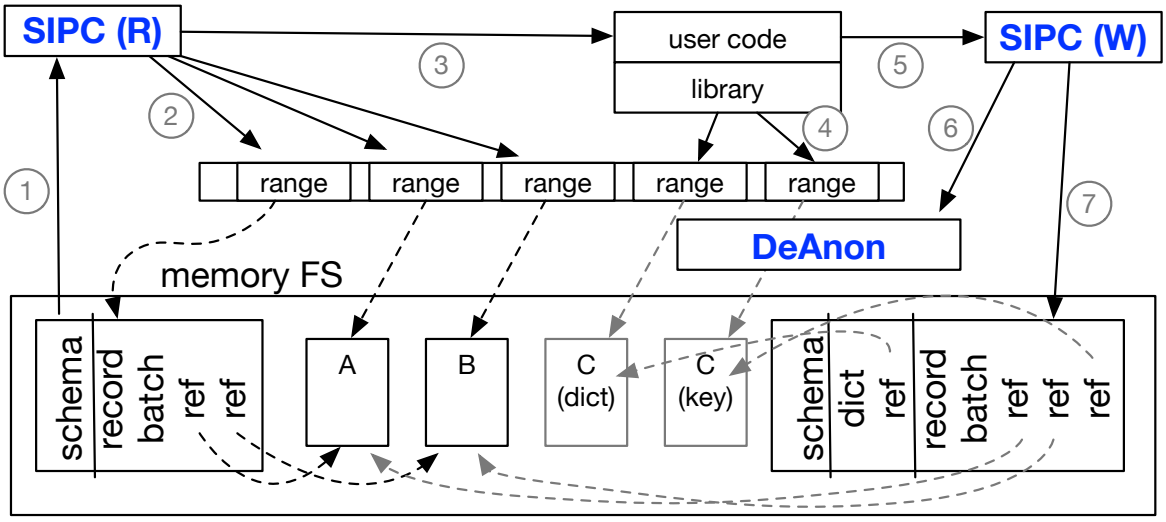


Figure 4.5: Shared Inter-Process Communication.

Figure 4.5 illustrates the read (R) and write (W) behavior of SIPC, when used by a function that is transforming inputs to produce outputs. Both inputs and outputs reside in an in-memory file system. Calls to SIPC are performed by the Kelvin Wrapper (Figure 4.4), not shown here for simplicity.

The transformation proceeds as follows: (1) a SIPC file is passed to the SIPC reader, which maps the file into the reader's address space. This file tends to be small because it references data in other files (instead of including data inline). The reader also (2) maps the referenced files into the address space and finally returns a tabular view over all the mapped data, which is (3) passed to the user code. The user code may perform arbitrary actions. In this example, the user code is concatenating two columns of input strings to create a third, dictionary-encoded column; the output includes three columns (two original and one new).

The Kelvin Wrapper receives the Arrow output data, which it (5) shares via SIPC. Given Kelvin's support for arbitrary user code, there is not an explicit data lineage that allows SIPC to identify the overlap between input and output. Instead, SIPC performs *IPC inspection*, comparing the memory addresses of buffers being written with the address ranges of previously mapped inputs. SIPC is able to then *reshare* those references, writing them to the output file. In this case, it references the entire A and B files, but overlaps could be partial (e.g., after slicing or other transformations). Other buffers passed for writing refer to anonymous memory; SIPC will (6) use DeAnon to de-anonymize these into in-memory files, then (7) write references to the new files in the output. In this case, the new column is

dictionary encoded, so the dictionary and record batch messages will both refer to newly shared data.

4.2.2.1 Dictionary Sharing

Some transformations (e.g., adding a column, slicing rows, concatenating tables) lead to coarse-grained overlap between inputs and outputs. Others (e.g., filtering or sorting rows) may lead to row-level overlap, leading to *Challenge 5* (Data Granularity). For regular encodings, Kelvin will unfortunately need to copy the data. For dictionary encodings, where values of a column are replaced by indexes into a dictionary of all distinct values, SIPC uses a *dictionary sharing* technique instead; references to input dictionaries are written to outputs, and only columns of dictionary lookup codes must be copied. Consequently, dictionary encoding will often be useful for Kelvin even if there are no repeated strings in a column (the traditional use case for dictionary encoding).

4.2.3 Resource Management

Source data, intermediate data, and running containers all consume memory, and Kelvin must take care to limit overall consumption to avoid excessive swapping and process kills. Memory consumption is especially hard to track when many containers share the same physical data. The Resource Manager (RM) is the subsystem of Kelvin that is responsible for tracking and limiting overall resource consumption.

4.2.3.1 Caching Deserialized Source Data

Copying creates multiple physical copies of the same data, but copies also occur when identical transformations in different jobs operate on the same source data. Specifically, how can we address *Challenge 6* (Deserialization)? Independent DAGs will often start by deserializing the same source data (e.g., Parquet files in persistent storage) to Arrow data. Kelvin's approach is to recognize such DAGs and restructure them so that different DAGs start from the same loader nodes. Different DAGs using the same data may not be launched at the same time, so we implement this common loading service as a deserialization cache (DeCache).

Figure 4.6 illustrates the DeCache. The cache consists of a special set of loader containers responsible for loading Parquet data from storage. These loaders use SIPC to efficiently share

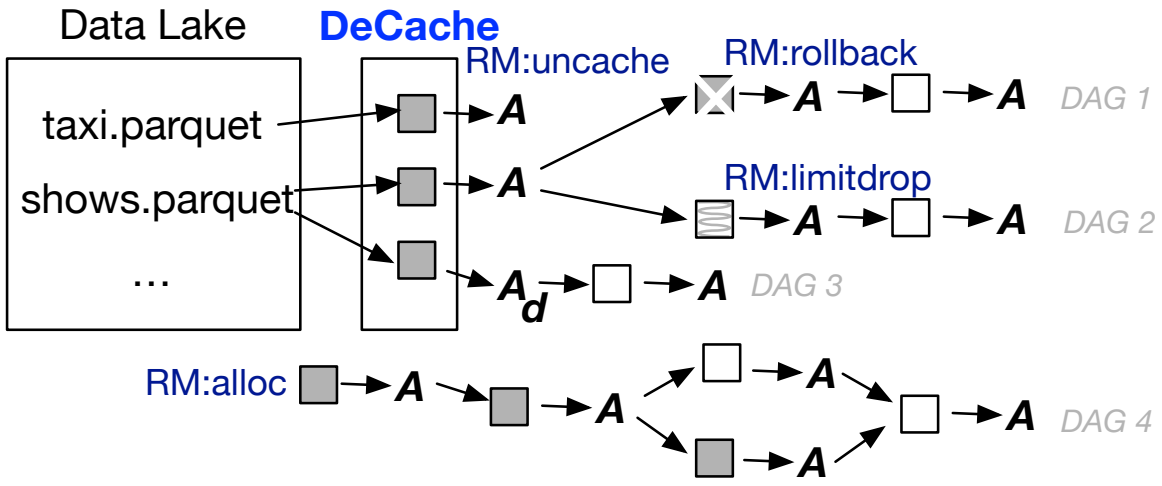


Figure 4.6: Resource Management. Finished nodes are gray.

deserialized data, the same way regular DAG nodes share. Unlike regular nodes, loaders are not deleted upon DAG completion because they may benefit other DAGs soon.

Different DAGs may wish to deserialize the same data with or without dictionary encoding on certain columns. In this case, different loader nodes will be used to create each representation (e.g., shows.parquet is deserialized with and without dictionary encoding in Figure 4.6).

4.2.3.2 Measurement and Mechanisms

The RM tracks dependency information with the help of SIPC. With resharing, intermediate data is represented as virtual Arrow tables referencing physical memory. Reference details are very relevant to eviction decisions (e.g., it may be necessary to delete outputs from multiple sandboxes to free physical memory with multiple references). SIPC inspects outgoing Arrow data and collects sharing information on behalf of the RM. This allows the RM to make informed decisions regarding eviction and garbage collection.

Figure 4.6 shows four actions (labeled with the "RM:" prefix) that the RM may perform to control overall memory consumption. First, the RM acts as an admission controller, deciding when to allocate memory to run new sandboxes (*RM:alloc*). Second, the RM can perform three types of eviction (*RM:uncache*, *RM:rollback*, and *RM:limitdrop*) to free memory as needed. Uncaching DeCache entries and rolling back DAG progress both involve deleting

completed containers and their Arrow outputs. In the latter case, the node will need to be re-executed later so that the DAG can eventually complete. Limit dropping involves reducing the container memory limit on a container to trigger swapout of the in-memory intermediate files produced by that node, as described earlier (§4.2.1). This is preferable to default swap behavior, which could swap out intermediate data needed by a downstream node that is about to be scheduled.

4.2.3.3 Admission and Eviction Policy

The RM allows new sandboxes to be created when the available memory (not reserved by running sandboxes nor consumed by intermediate data) can satisfy the memory requirements of a node. When multiple nodes are waiting for memory allocations, the RM assigns priority in depth-first order (i.e., the closest to finishing a DAG), as completing DAGs that are nearly finished enables deletion of all intermediate data for that DAG, freeing resources for other DAGs.

Admission control has the potential to cause deadlock if intermediate data has exhausted system memory and all DAGs are waiting for more memory to progress. Furthermore, too much intermediate data in memory could reduce the opportunity for more parallelism in DAG execution. Thus, the RM performs evictions using the knowledge of sharing details to free up memory as necessary. The RM chooses between memory-freeing actions as follows. First, the RM uncaches DeCache entries with no active references, if any. Then, the RM evicts the outputs of nodes with the lowest priority (as described based on depth). Outputs are evicted one by one until the available memory is larger than the memory requirement of the node scheduled to run next.

The RM has two mechanisms for evicting a non-cache node's output: *rollback* and *limit dropping*. The cost of rollback depends on how long it would take to recompute that output, and the cost of swapping depends on the output size. Thus, the RM adopts *adaptive eviction*, which either deletes or swaps the outputs of a function based on the ratio of its latency to the output sizes. The threshold should be tuned offline depending on the throughput of the swap device.

Challenges	Solutions
Data Origination	De-anonymization
Containerization	Container retainment & Limit dropping
Inline Message Data	Shared IPC
Data Granularity	IPC inspection & Dictionary sharing
Deserialization	DeCache

Table 4.1: Solutions for Challenges.

4.2.3.4 Design Summary

In this section, we describe our solutions to the 5 challenges from Section 4.1.4 in detail except for Challenge 1, which is solved by simply using Arrow and Linux `mmap`. The solutions are summarized in Table 4.1. In addition, we design mechanisms for resource management, which are building blocks for our solutions. We also design scheduling primitives for admission and eviction to utilize memory more efficiently for more parallelism.

4.3 Implementation: Kelvin on Linux

We now describe the implementation of five Kelvin subsystems on Linux: DeAnon (§4.3.1), SIPC (§4.3.2), Node Container (§4.3.3), DeCache (§4.3.4), and Resource Manager (§4.3.5).

4.3.1 DeAnon Kernel Module

We implement the DeAnon (de-anonymizer) subsystem as a new Linux module, written in C. DeAnon exposes two interfaces: `new_file()` allocates a tmpfs file to receive anonymous memory in the future, and `deanon(file_id, addr, len)` moves the anonymous memory in `[addr, addr+len)` to the end of the indicated tmpfs file (the operation resembles an append, without a copy).

A `deanon` call traverses the page table to identify pages in the specified range. For each page, DeAnon modifies the metadata of the page, the tmpfs file, and the virtual-memory area (VMA) to which the page belongs, such that the page appears to belong to the tmpfs file and the VMA appears to be backed by the tmpfs file. After the call, the calling process can access the data in this region normally, and other processes can map the tmpfs file into their address spaces via `mmap`. A `deanon` call commonly refers to a subset of a VMA; in this case, DeAnon splits the VMA as necessary.

Page tables require page-granular sharing, but DeAnon supports calls with offsets and lengths at byte granularity. If partial pages occur at the beginning or end of a shared range, DeAnon simply copies the partial pages to newly allocated pages in the in-memory files.

When a working set is larger than the available memory, Linux may swap out pages to a swapfile residing on a disk-backed file system. Both page tables and tmpfs files have an array of entries pointing to physical pages; when a page is swapped out, the corresponding entry is replaced by a swap entry. DeAnon may encounter swap entries when walking a page table for pages that have been swapped out. We optimize DeAnon for this situation. A simple approach is to swap in the pages first to avoid the issue entirely. Our optimized approach, called *direct swap*, directly inserts a swap entry into the tmpfs file, then modifies the metadata of the swap space and the original page table entry accordingly.

Prior to de-anonymization, memory ranges often correspond to malloc allocations from memory regions with write access enabled. Given these regions now map to physical memory that other processes will soon be able to access, it is important that any process exposing data with DeAnon does not take any action that would modify the data (e.g., if the process using de-anonymization were to free the corresponding memory, subsequent malloc uses could corrupt the data that was supposed to be shared). On the writer side, we do not share Arrow data until user code has returned; on the reader side, we map Arrow data with read-only access (Arrow data is meant to be immutable anyway).

4.3.2 SIPC Protocol

SIPC extends the Apache Arrow IPC implementation (v12.0.1) to use DeAnon and avoid copies. SIPC, like Arrow IPC, is responsible for writing three message types to a sink: *Schema*, *RecordBatch*, and *DictionaryBatch*. SIPC writes the schema to a sink (for us, a small in-memory file) by copying the data; schemas are usually small relative to data, so avoiding a copy would provide minimal benefit. Records and dictionaries, in contrast, may be large, growing with the number of rows and unique string values, respectively.

SIPC creates a single tmpfs file (with the help of DeAnon) for each column in the table, plus one more for all dictionaries (if any). For each RecordBatch (the data in one column), SIPC de-anonymizes the memory backing each partial column in the batch, transferring ownership to the appropriate file. Using different in-memory files for each column creates opportunities later to garbage collect some output columns that are not in use even while

other columns of the same table are still needed. Finer granularity would be possible (e.g., a new tmpfs file for every combination of batch and column), but we wish to avoid the overhead of creating too many small files.

Instead of copying the de-anonymized data to the sink, SIPC simply writes a tuple of three integers: (1) file identifier, (2) offset into file, (3) length of file range. On the read side, SIPC uses this information to make appropriate `mmap` calls and reconstruct a view of the table without copying. During read, SIPC also records the address ranges returned from `mmap`. Upon later writes, SIPC implements resharing by identifying output data that overlaps with the input ranges by `IPC inspection`. In this case, the output SIPC file references the same tmpfs files referenced by the input SIPC file; there is no need to de-anonymize a second time.

4.3.3 Node Container

For our container implementation, we use SOCK [110] with some modifications. SOCK allows each container to have its own set of dependencies, but when package versions happen to be the same, installations are efficiently shared between different containers. SOCK containers support zygote provisioning and reusing containers for multiple invocations, but these features provide minimal value for our use case, so we disable them to simplify memory accounting.

We modify SOCK to bind mount the /dev/shm tmpfs into every container instance to facilitate sharing (like Nightcore [64]). SOCK already uses cgroups to specify memory limits, but we expose this so Kelvin can implement limit dropping to swap out specific data. Given tmpfs memory for a file is charged to the cgroup of the process creating the file, we modify SOCK so that cgroups are retained after the process in a container exits.

We modify SOCK to provide Arrow-oriented entry points (instead of HTTP entry points). In particular, Kelvin communicates with a wrapper inside each container via `sendmsg` and `recvmsg` system calls over UNIX file sockets. These allow the passing of references to SIPC Arrow data (in tmpfs) in both directions. The wrapper is responsible for all interactions with SIPC, such that user code is only responsible for accepting and returning Arrow data.

4.3.4 DeCache: Shared Data Loading

We implement the DeCache in Go, leveraging existing components. When a DAG specifies that a node should load a specific Parquet file, we run a Node Container (§4.3.3) where the "user code" is actually a standard function we provide that uses PyArrow's `parquet.read_table` function to load the Parquet file to Arrow. The function accepts a `read_dictionary` argument specifying which columns should be deserialized using dictionary encoding (we construct this argument based on DAG configuration). Although Parquet loading is a special case, where it would be reasonable to implement a version that directly populates shared memory with Arrow data, for simplicity we use the `parquet.read_table` unmodified to produce Arrow data in anonymous memory, then use SIPC (§4.3.2) and DeAnon (§4.3.1) to efficiently expose it.

Due to resharing, DeCache outputs may be in use, even when there are currently no nodes directly consuming the data. Tracking such references and making eviction decisions are responsibilities of the resource manager (§4.3.5).

4.3.5 Resource Manager

The Resource Manager (RM) is implemented in Go and interacts with other subsystems to gather information and perform memory-management actions.

SIPC (§4.3.2) records references to tmpfs files that it has written to output, and it exposes this information to the share wrapper inside a node sandbox (§4.3.3), which in turn passes reference metadata back to the RM. The RM can thus associate Arrow outputs with sets of tmpfs files. Due to resharing and de-anonymization at column granularity, this is a many-to-many correspondence. Visibility into these relationships allows the RM to perform reference counting on tmpfs files and garbage collect a tmpfs file backing a column of data when there are no more unrun children that will use intermediate data referencing that column. DeCache outputs are handled differently: columns of loaded Parquet data are not immediately garbage collected upon a zero reference count because future DAGs may use the data. Data is garbage collected with a simple deletion of the relevant tmpfs file.

CPU	Two Intel Xeon Silver 4314 16-core CPU
RAM	256GB ECC DDR4-3200 Memory
DISK	Two 960GB Samsung PCIe4 x4 NVMe SSD
OS	Ubuntu 22.04, kernel version 5.15

Table 4.2: Hardware for Evaluation. Note that the actual memory limit of each experiment is enforced by cgroup, not the RAM size. Input parquet files reside on one disk and the other is used as the swap device. SMT is disabled on CPUs.

4.4 Evaluation

To evaluate Kelvin, we answer the following questions to demonstrate that Kelvin's components function as expected in various micro-benchmarks and Kelvin provides consistent benefit in end-to-end benchmarks and real-world data pipelines.

- Q: How much faster can data be passed if copies are avoided?

  A: The gain of copy avoidance is at least 2× under different memory limits. (§4.4.1.1)

- Q: Does resharing reduce memory consumption for deep DAGs?

  A: Resharing significantly reduces memory consumption and latency for applicable operations such as coarse-grained data appending/removal and certain other operations with dictionary-encoding. (§4.4.1.2)

- Q: Does the DeCache reduce memory consumption of functions, allowing more concurrent execution?

  A: DeCache largely reduces memory consumption and allows more parallelism when different DAGs share the same inputs. (§4.4.2)

- Q: Which admission and eviction techniques perform best for different scenarios?

  A: *Adaptive eviction* that selectively triggers *rollback* and *limit dropping* performs the best for various different workloads. (§4.4.3)

- Q: How does Kelvin perform on real workloads?

  A: Kelvin provides 1.2× to 2.3× gain on real data pipelines. (§4.4.4)

Table 4.2 describes our experimental setup. Unless otherwise stated, the memory limit for the system is 50 GB. The baseline we use is Kelvin with all our novel features disabled.

The baseline uses regular Arrow IPC and read-side memory maps for data passing (as in Figure 4.2C), so some would even describe our baseline as "zero-copy", despite some overlooked copies. The baseline Resource Manager (RM) only uses admission control, data passing, and reference counting, without the DeCache or advanced eviction.

We first evaluate each of Kelvin's subsystems with a targeted microbenchmark, then with a complex ecosystem benchmark with different libraries and memory usage patterns, described in Table 4.3. For all experiments, we load input Parquet files from local storage to avoid the noisy performance often associated with cloud storage (e.g., AWS S3).

4.4.1 DeAnon and SIPC

SIPC is the primary user of the DeAnon kernel module, so we evaluate these components in combination. We first perform experiments focused on the benefits of de-anonymization (§4.4.1.1), then explore SIPC's other capabilities related to resharing transformed data (§4.4.1.2). Finally, we run more complicated DAGs that use different packages in the Arrow ecosystem (§4.4.1.3).

4.4.1.1 DeAnon: Intermediate Data Copy Avoidance

When a parent node produces anonymous Arrow output for children to consume, Kelvin avoids write-side copying by de-anonymizing with DeAnon and using the SIPC to facilitate sharing. To quantify the resulting speedup, we run a single-function DAG that uses PyArrow to deserialize a 0.5 GB Parquet file (compressed) to about 4.0 GB of Arrow data (uncompressed), which is used for the output data. Additional memory is temporarily required during processing, such that peak memory during load is 5.8 GB.

Figure 4.7a shows the results with varying memory limits: DeAnon is 1.8× faster than the baseline for high memory limits (i.e., 10 GB) and 2.2× faster for low limits (i.e., 1 GB). Although loading from a Parquet file is more work than doing a copy, the former operation is parallelized over 24 threads, whereas writes to an IPC file are serialized.

For high memory limits, performance gains result from copy avoidance; for low limits, Figure 4.7b shows that the benefits result from reduced swapping. Without DeAnon, copying output temporarily duplicates the data in memory, surpassing cgroup memory limits sooner. Our *direct swap* technique (DeAnon line) can directly copy swap entries from a page table to

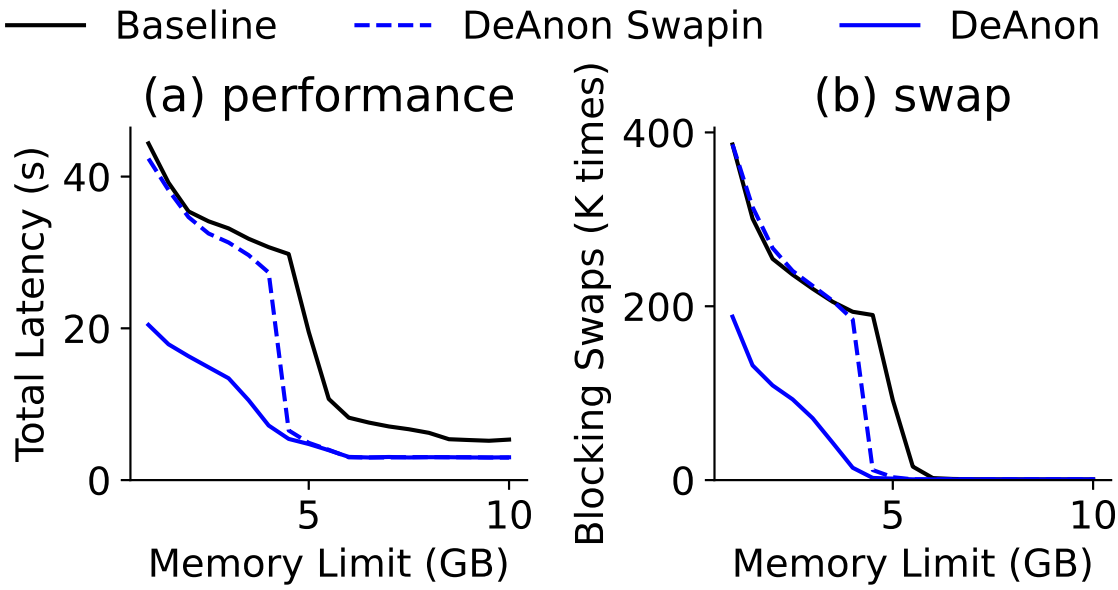


Figure 4.7: Copy Avoidance. Throughput and swapping are show with and without DeAnon for a single-node DAG.

a tmpfs file. When this critical optimization is disabled ("DeAnon swapin" line), we cannot significantly outperform the baseline under tight memory constraints.

4.4.1.2 Resharing: Internal Copy Avoidance

When SIPC reads input data, it records the address ranges where it maps specific tmpfs files that were generated by upstream nodes. During subsequent writes, SIPC is able to reshare data by reusing references to the inputs, without copying or making additional calls to DeAnon. In this section, we explore the impact resharing has on performance and the size of intermediate data for a variety of operations.

We run a variety of two-node DAGs, where the first node loads data, and the second performs a data transformation that produces an output that overlaps with its input. The first 4 transformations (add_col, concat, rm_col, and slice) use tables with 10 1-GB integer columns as inputs; the other DAGs use tables of 10 1-GB string columns of 100-byte strings with no repetition. In this subsection, baseline's copy of intermediate data does not count towards its output size.

Figure 4.8 shows the results. For subtractive cases (i.e., removing columns or slicing

to obtain a consecutive subset of rows), SIPC spends almost no time and produces almost no new physical data. For additive cases (i.e., adding columns or concatenating additional rows), time and space costs only apply to the new data, not reshared data.

The above cases constitute coarse-grained overlap (i.e., similarities between inputs and outputs consist of large ranges of identical consecutive bytes). In contrast, filtering and sorting are examples of fine-grained overlap: many of the input rows may appear as output rows, but with regular Arrow encodings there are not large contiguous ranges of overlap. However, when dictionary encoding is used ("dic" suffix on the filter and sort bars), SIPC allows us to reshare dictionaries themselves, even if resharing is not possible for the buffers of numeric codes referring to dictionary entries. We see with dictionary encoding (filter_dic and sort_dic), output sizes are negligible (because the dictionaries dominate the total size in our dataset), but there are no savings (relative to baseline) for regular encodings (filter and sort). The biggest performance gains are most pronounced for the dictionary encoding cases as well; though filter and sort are still somewhat better than baseline (sharing is faster than copying, even if it is slower than resharing).

In the *upper* operation, all strings in a column are converted to upper case. Unfortunately, no resharing is possible here, though it might intuitively seem possible. Arrow string arrays have both a value buffer (containing actual characters) and an offset buffer (indicating the starting offset of each string). If every string remained the same length after being converted to upper case, it would be possible to reshare the offset buffer, though not the value buffer; for short strings, resharing the offset buffer would be useful. Unfortunately, Arrow arrays use UTF-8 encoding, and a few characters have different byte lengths depending on case (e.g., "ß" is 2 bytes, but the upper case equivalent is 3). This example shows some character encodings are more reshare-friendly than others (e.g., the upper case optimization could be optimized to reshare offset buffers for UTF-16 strings).

We now explore the benefits of resharing for add_col with deeper pipelines. We run 10 experiments with 1 to 10 column-adding functions following one load function. Each column-adding function generates a column based on computations on two randomly chosen existing columns from the previous function. The input of the workload is a 2 GB table of 2 columns, and each new column contains 1 GB column of data. Each function outputs N+1 columns: N input columns and 1 new column.

Figure 4.9 shows the result. The cumulative output size for SIPC scales linearly because each added column is only sent to tmpfs once; latency also scales linearly. In contrast, in the

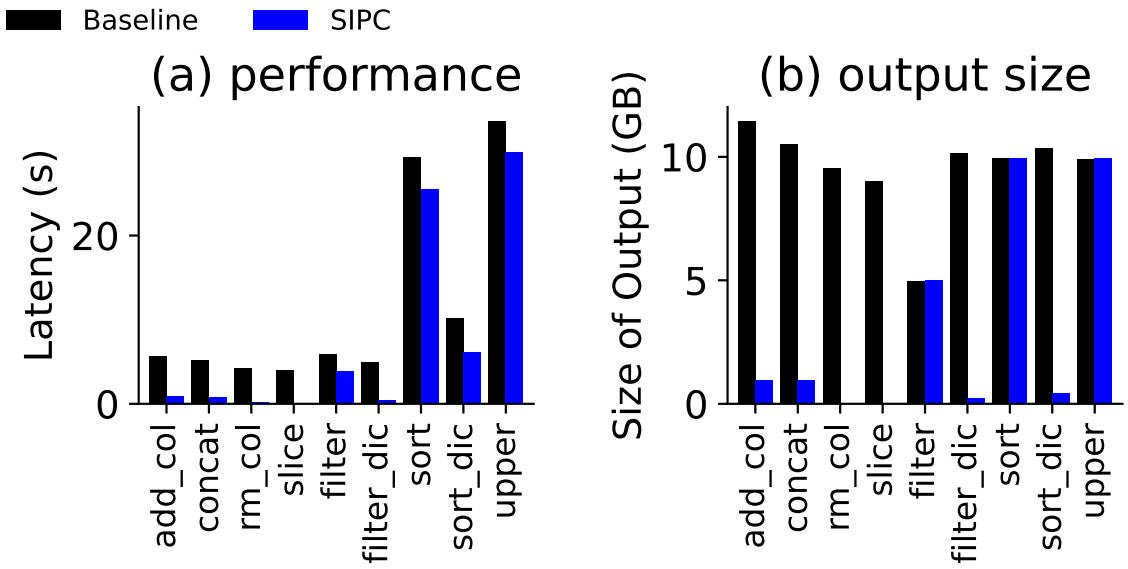


Figure 4.8: Resharing: Time and Space Benefits.

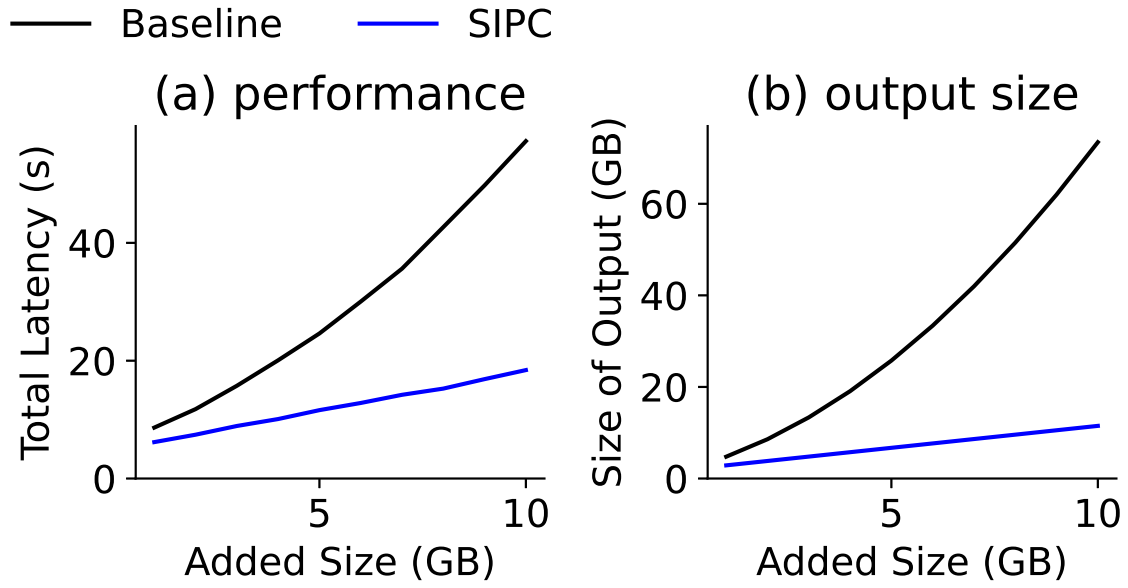
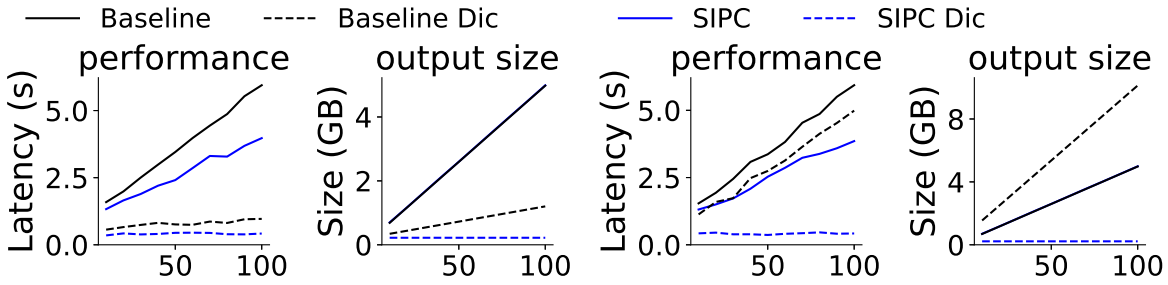


Figure 4.9: Latency of Col_Add of Different Sizes. The x-axis is the number of column adding function executed. The y-axis is the overall latency.

baseline (without resharing), the output from any node is rewritten in every downstream node, such that intermediate sizes continue to grow throughout the pipeline. Thus, cumula-



(a) Repeats

(b) No Repeats

Figure 4.10: Resharing Dictionaries. The x-axis is the string size in bytes. Each unique string appears 10 times in (a) and once in (b). Strings are dictionary encoded for dashed lines.

tive size (and time) scales superlinearly. Slicing exhibits the same scaling pattern as well (not shown).

We now explore the interactions between dictionary encoding and resharing in more detail. We generate a 10M-row dataset containing one column of `int32` values and 10 columns of strings of a given size. Each unique string value occurs 10 times in a column. After the Parquet data is loaded to Arrow (with or without dictionary encoding), a node runs a filter operation that will match half the rows, and outputs the results. Figure 4.10a shows the results. Without dictionary encoding, neither SIPC nor the baseline has the opportunity to reshare, so they produce output of identical size (SIPC does do this faster, though, because it can simply de-anonymize its Arrow data to produce the output). The Baseline and SIPC versions both benefit significantly (in terms of time and space) from dictionary encoding because each string can be recorded once instead of 10 times.

We perform the experiment again (Figure 4.10b), but this time generate the data such that each unique string occurs only once in a column. Now, for the Baseline, the output size is actually larger with dictionary encoding: not only is there no repetition to remove, but lookup codes (into a separate table) have increased the size. The Baseline performance still benefits from dictionary encoding, but only marginally.

In contrast, we observe that SIPC is able to reshare input dictionaries, producing intermediate outputs of negligible size extremely quickly. Kelvin's approach creates a compelling new reason to use dictionary encoding (besides removing redundancy in the data): supporting fine-grained resharing of values for operations such as filter and sort.

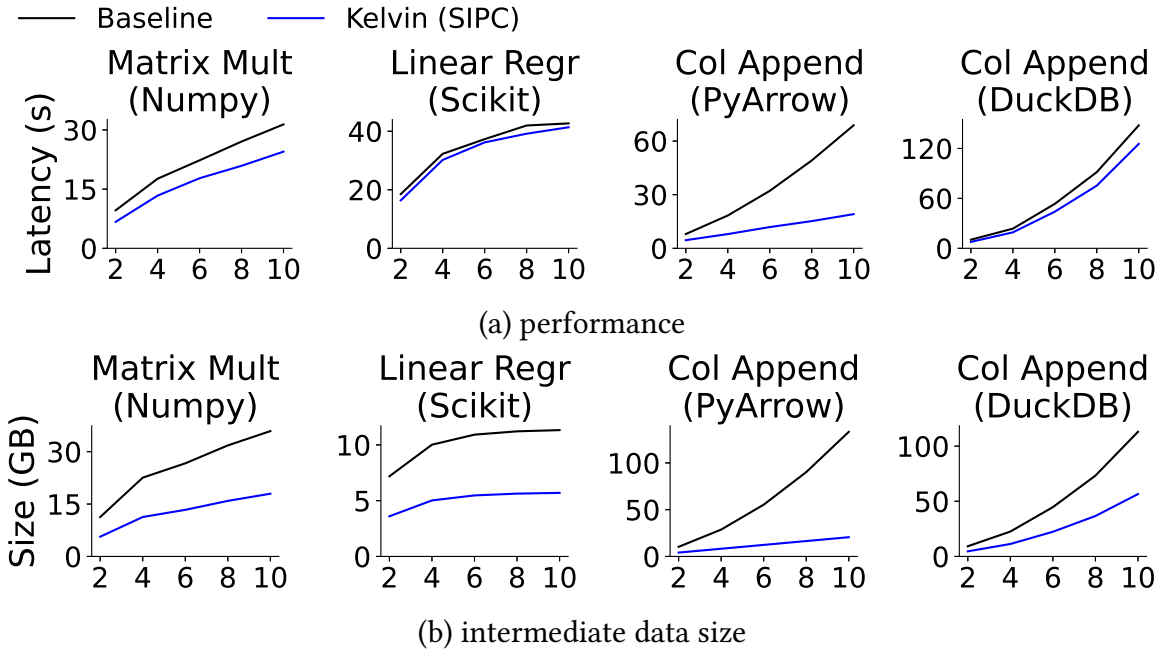
Name	Library	Memory Usage
Matrix Mult	Numpy	changing
Linear Regr	Scikit	decreasing
Col Append 1	PyArrow	increasing (w/ resharing)
Col Append 2	DuckDB	increasing

Table 4.3: Ecosystem Benchmark DAGs.

4.4.1.3 Ecosystem Benchmark

Our ecosystem benchmark contains 4 types of DAGs with different libraries and memory usage patterns, as shown in Table 4.3. All the DAGs are sequential chains (no branches), and the code is randomly generated for each node as follows. *Matrix Mult*: numpy is used to multiply the input table by a matrix to produce an output with the same number of rows and N columns (selected randomly between 1 and 10). Some numpy data (including this case) can be converted to Arrow data without a copy given both represent numbers consecutively in memory. *Linear Regr*: scikit-learn is used to perform a linear regression on the input data. We randomly choose half the columns as features and one other column as the label. The node splits the table into even-sized train/test sets. A linear regression model is trained on the training data, then used to add a prediction column to the test data, which is then used as the node's output. *Col Append 1*: PyArrow computes 4 new columns via simple operations over existing columns, horizontally concatenates the new columns to the input table, and returns the result. This naturally leads to resharing opportunities. *Col Append 2*: this is similar to the Col Append 1 pattern, but the projection adding the columns is implemented via a SQL query executed by DuckDB. Though DuckDB reads Arrow data without copying, a copy is required to transform a DuckDB table to an Arrow table to produce the output, so this workload type does not present resharing opportunities.

We evaluate SIPC with the ecosystem benchmark in Table 4.3, varying DAG length from 2 to 10. The latencies are shown in Figure 4.11a and the total amounts of intermediate data generated are shown in Figure 4.11b. SIPC has a steady performance improvement of 1.4× for the Matrix Mult DAG and generates half the intermediate data compared to baseline by eliminating intermediate data sharing. For Col Append (PyArrow), resharing takes effect so SIPC generates further less intermediate data than baseline by utilizing input-output sharing. SIPC has a latency proportional to DAG length because it only needs to generate data for newly added columns, while baseline shows superlinear latency growth as the data



(a) performance

(b) intermediate data size

Figure 4.11: Ecosystem Benchmark on SIPC. The x-axis is the DAG Length.

size becomes larger when DAGs are longer and baseline needs to copy the entire data.

For Col Append (DuckDB), resharing does not take effect and copying the entire data is costly. For Linear Regr, a large amount of time is spent in model training in user code (shared memory does not make help for this). Thus, SIPC's gains in these two DAGs are lower, at around 1.1×.

4.4.2 DeCache: Input Data Deduplication

We evaluate how the DeCache improves performance by reducing duplicate loads of the same Parquet file.

When multiple DAGs running in parallel share inputs, DeCache could save significant memory consumption and largely increase parallelism. Figure 4.12 runs a load function loading 1.5 GB data into memory, followed by a function that filters the table by rows. Multiple such DAGs (1 to 25) run in parallel and all the load functions load the same Parquet file. There is no admission control or eviction policy in this experiment. SIPC w/ DeCache detects the duplicated loads to the same input, runs the load only once, and caches the result to be used by all the following computation functions. Thus, there are no concurrent

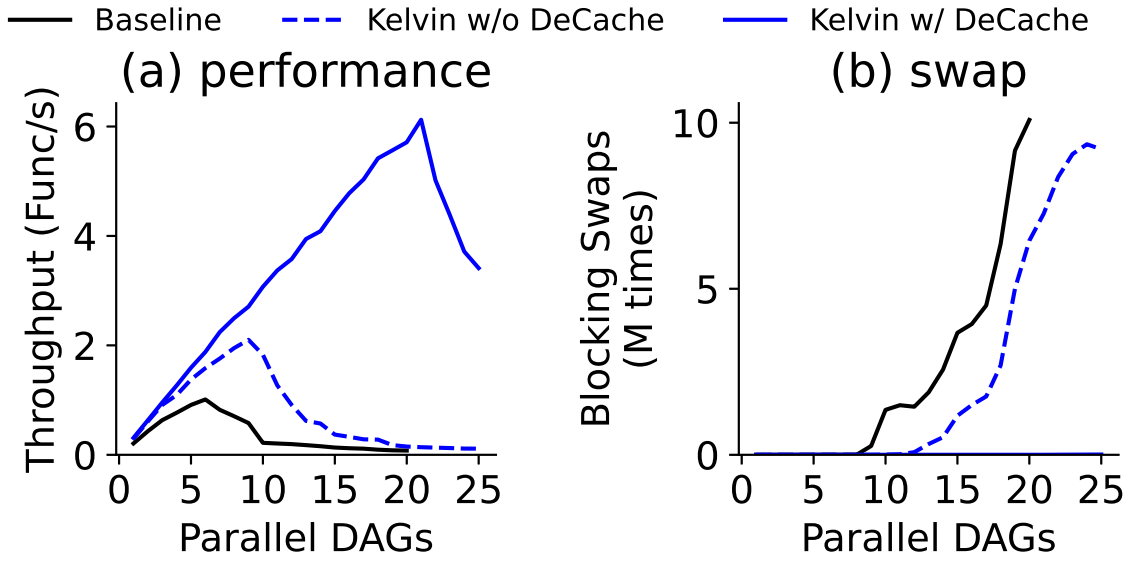


Figure 4.12: Performance of DAGs with the same Inputs. In (b), the y-axis is the number of foreground swap-in events in million times. Baseline crashes at x>20 because of OOM.

load steps eating up the memory and the performance is significantly improved when the number of parallel DAGs is larger, up to 38×.

DeCache also works when DAGs sharing inputs run sequentially. We benchmark DeCache with the ecosystem benchmark in Table 4.3, varying DAG length from 2 to 10 and running 5 DAGs back-to-back. From the second DAG, Kelvin with DeCache uses cached intermediate data directly. The result of Matrix Mult is shown in Figure 4.13. For any DAG length, DeCache provides a constant gain by saving the subsequent load functions. The results are similar for other DAGs in the ecosystem benchmark.

4.4.3 Eviction Mechanisms

We evaluate the mechanisms for eviction that are used by the Resource Manager: *rollback*, *limit dropping*, and *adaptive eviction*. The Node Container exposes the limit dropping option to the Resource Manager. We use cumulative DAGs with large depth, which have increasing memory requirements during the workload and stress eviction most.

Figure 4.14a runs a workload containing 15 cumulative sequential-chain DAGs, each containing 10 functions with the first function loading one 2 GB table and the remaining 9

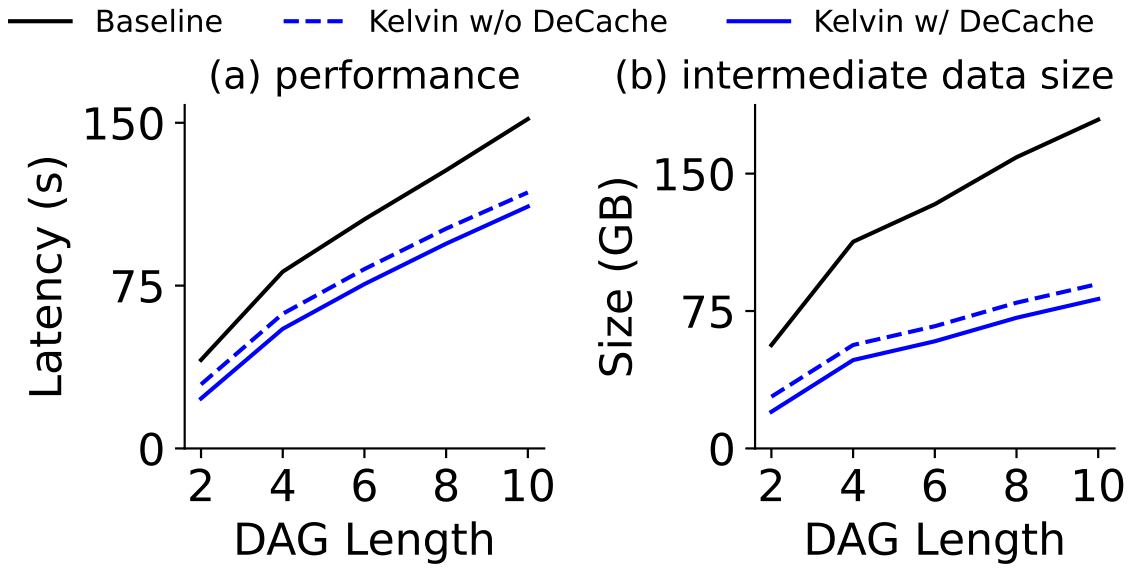


Figure 4.13: Matrix Mult on DeCache.

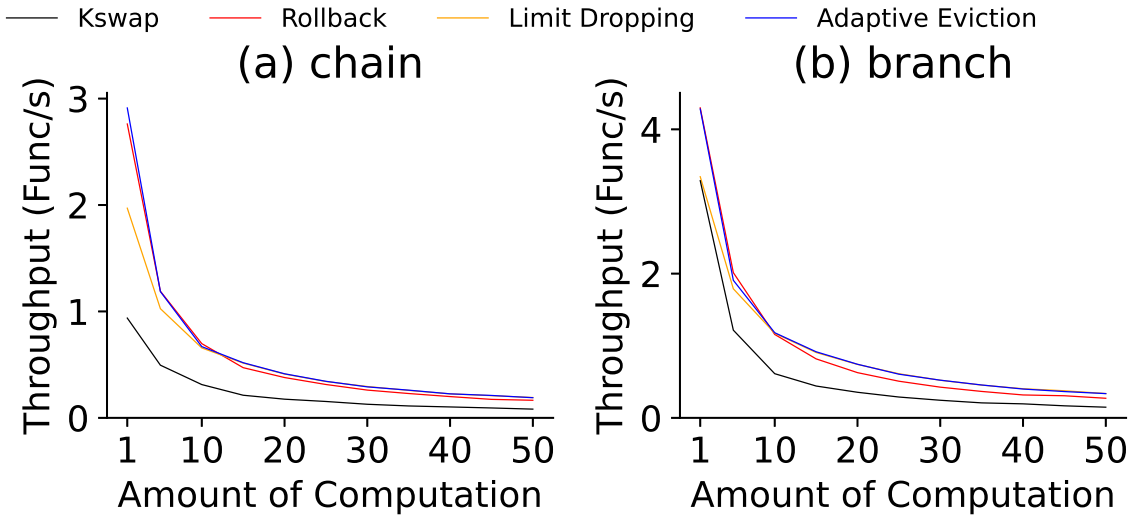


Figure 4.14: Performance of Different Eviction Methods. The x-axis is the amount of computation in a single function. The y-axis is the throughput of the entire workload.

appending 1 GB data to the output of the previous function based on computations of the existing data. The x-axis is the amount of computation in a single function, with unit of 1 being one simple addition of two 1 GB columns. The y-axis is the latency of finishing

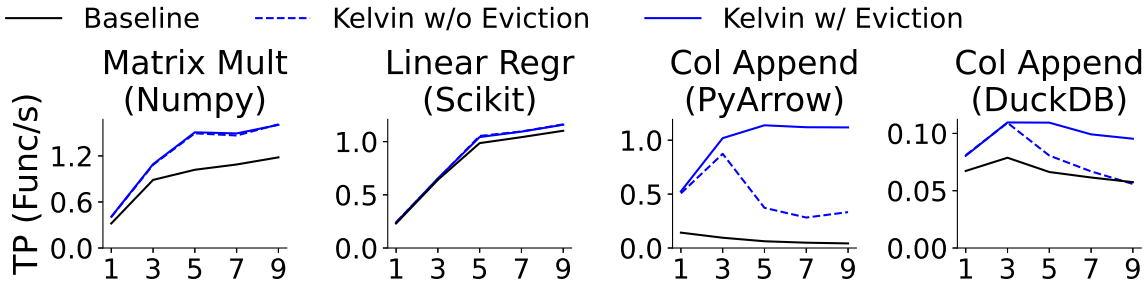


Figure 4.15: Synthetic Benchmark on Eviction Mechanisms. The x-axis is DAG Length.

the entire workload. *Kswap* is the baseline that only does admission control and resolves deadlocks with default kernel swapping behavior.

The *rollback* strategy performs 2.0-2.2× better than the *Kswap* baseline. Sometimes *limit dropping* (orange) outperforms *rollback* (red). When functions perform little computation (left side of the graph), *rollback* performs up to 1.3× better as re-running is cheap. For compute-intensive functions (right side), it is faster to temporarily swap out data rather than repeat a slow computation again in the future. *Adaptive eviction* (blue) chooses the better eviction method and achieves the best performance in all scenarios.

Figure 4.14b uses different cumulative DAGs, where each DAG branches out. Each preceding function has two succeeding functions and the total depth is 4; thus, one DAG contains a total of 15 functions. 15 DAGs are contained in one workload. We see similar trends as Figure 4.14a, that *rollback* performs 1.3-1.8× better than *kswap*, and that *adaptive eviction* performs the best under various conditions.

We now run the ecosystem benchmark on Kelvin, without and with eviction enabled (adaptive policy). We use DAGs of length 10, running 1 to 9 in parallel. Figure 4.15 shows the results. When many DAGs are running concurrently and intermediate data grows with each step (the "Col Append" workloads), enabling eviction improves throughput up to 4× (over Kelvin without eviction) or 28× (over baseline). The two implementations without eviction rely on generic kernel swapping. For workloads for which memory consumption does not steadily grow (Matrix Mult and Linear Regr), eviction is not important; simply running more nodes will free up memory as upstream outputs become no longer needed.

	DAG 1	DAG 2	DAG 3	DAG 4	DAG 5
Load	3	3	3	3	3
Preproc.	3	7	2	3	2
Compute	24	10	12	24	12
Total	30	20	15	30	17

Table 4.4: DABstep Workloads: Node Count by Type.

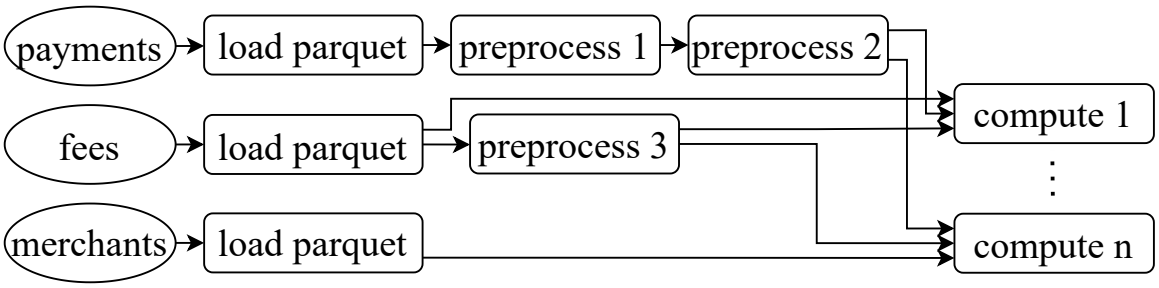
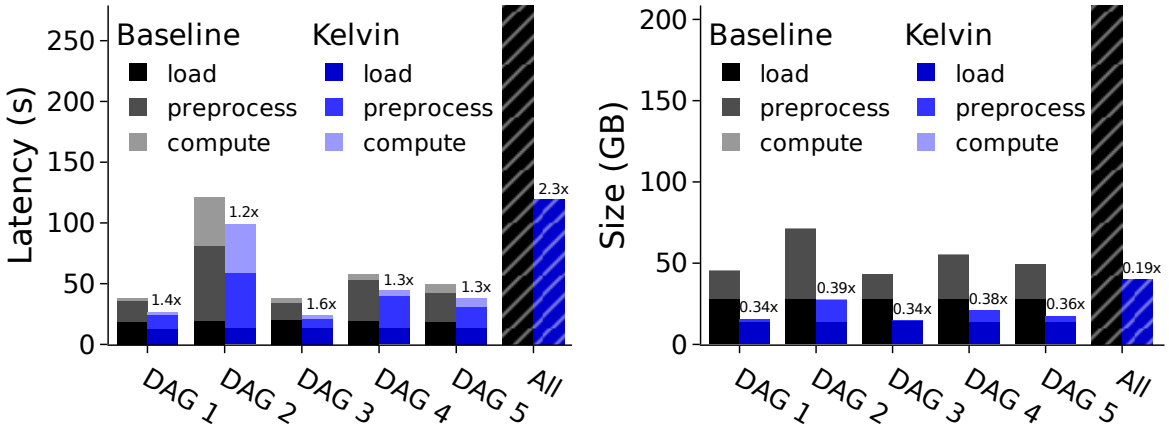


Figure 4.16: Sample DABstep DAG. Circles are parquet inputs. Boxes are functions. The load phase contains 3 nodes loading 3 tables. The preprocess phase contains 3 nodes, appending data to and filtering the input tables. The final computation phase involve tens of nodes in parallel.

4.4.4 DABstep: Evaluation with Real DAGs

We use the Data Agent Benchmark for Multi-step Reasoning (DABstep) [8, 9] to evaluate Kelvin's performance on real, complex DAGs. DABstep is built by Adyen [1] and Hugging Face [3] and contains analysis questions extracted from Adyen's real workloads related to payments and transactions. For example, a DAG may compute the expected change between fees F a merchant is paying in scenario A vs. a new scenario B, or list all the fees that apply to a certain scenario.

We were provided solutions to 5 hard questions sampled from the question list [8], and adapted these solutions to run as Kelvin DAGs. Table 4.4 shows the size of each DAG, and how many nodes are responsible for loading data, preprocessing, or computing results. Figure 4.16 shows a representative graph structure: all DAGs load the same 3 source tables in Parquet: a 5 GB *payments* table, a 20 MB *fees* table, and a 5 MB *merchants* table. We were asked not to reveal the correspondence between DAG solutions and specific DABstep questions, so as not to contaminate online sources used to train LLMs (Large Language Models) that specialize in writing data pipelines.



(a) performance

(b) intermediate data size

Figure 4.17: Performance of DABstep DAGs. Except for *All*, the latency and the intermediate data size are distributed to the load, preprocess, and compute phase.

Figure 4.17 shows the performance and the intermediate data size of running the 5 DAGs individually, plus one workload running all 5 simultaneously. The load and preprocess phases take a large portion of the total latency, and the compute phase is fast due to high parallelism.

For the 5 individual DAGs, in the load phase, Kelvin performs 1.4× faster and generates 0.5× intermediate data compared to the baseline because of removing intermediate data copying by de-anonymization. In the preprocess phase, Kelvin performs 1.5× faster and generates 0.2× intermediate data because of removing input-output copying and intermediate data copying when one preprocess node appends data to the original payments table. Kelvin does not optimize the compute phase as few output is produced.

Overall, Kelvin performs 1.2× to 1.6× faster for the 5 individual DAGs and generates only about 0.3× intermediate data. For the workload running all 5 together, Kelvin performs 2.3× faster because the load phase among the 5 DAGs runs only once thanks to the DeCache, saving computation time as well as memory for more parallelism.

4.5 Conclusion

Zero degrees Celsius is not the coldest theoretical temperature, and the term “zero-copy”, as commonly used, does not actually mean no copying. In this work, we consider often-overlooked occurrences of copying and duplication in data pipelines, and pursue an “absolute

zero" ideal. Towards this end, we introduced Kelvin, a new data pipeline execution tool that co-designs the user-space resource management, container runtime for containerized execution, and kernel support for shared memory. Kelvin improves the memory efficiency of data pipelines by avoiding write-side copies (from nodes) and input-to-output copies (within nodes). Kelvin further avoids duplication between different DAGs using the same inputs. Although some copying and duplication remain unavoidable, Kelvin's efficient handling of intermediate data improves overall throughput by 1.2-28× in complex workloads.

Chapter 5

Related Work

In this chapter, we discuss prior works related to Bourbon, Symbiosis, and Kelvin. We categorize them into machine learning for indexing and caching (§5.1), LSM-tree optimizations (§5.2), cache management (§5.3), kernel methods in memory management (§5.4), data pipeline techniques (§5.5), and other advancements for memory efficiency related to hardware (§5.6).

5.1 Machine Learning for Indexing and Caching

Learned indexes. The core idea of Bourbon, replacing indexing structures with ML models, is inspired by the pioneering work on learned indexes [77]. However, learned indexes do not support updates, an essential operation that a storage-system index must support. Recent research tries to address this limitation. For instance, XIndex [142], FITing-Tree [50], and AIDEL [88] support writes using an additional array (delta index) and periodic re-training, whereas Alex [38] uses a gapped array at the leaf nodes of a B-tree to support writes. APEX [96] further optimizes Alex for persistent memory. LISA [89] replaces R-tree for spatial data with sharded local models which support writes by periodic re-training. LeaFTL [139] applies learned indexes to Flash Translation Layer on SSDs to reduce its storage cost and handles updates by a Log-Structured Mapping Table which resembles the behavior of LSM trees.

Most prior efforts optimize B-tree variants, while Bourbon is the first to deeply focus on LSMs. Further, while most prior efforts implement learned indexes to stand-alone data

structures, Bourbon is the first to show how learning can be integrated and implemented into an existing, optimized, production-quality system. While SageDB [76] is a full database system that uses learned components, it is built from scratch with learning in mind. Our work, in contrast, shows how learning can be integrated into an existing, practical system. Finally, instead of "fixing" new read-optimized learned index structures to handle writes (like previous work), we incorporate learning into an already write-optimized, production-quality LSM.

Model choices for learned indexes. Duvignau et al. [41] compare a variety of piecewise linear regression algorithms. Greedy-PLR, which we utilize, is a good choice to realize fast lookups, low learning time, and small memory overheads. Neural networks are also widely used to approximate data distributions, especially datasets with complex non-linear structures [82]. However, theoretical analysis [94] and experiments [128] show that training a complex neural network can be prohibitively expensive. Similar to Greedy-PLR, recent work proposes a one-pass learning algorithm based on splines [73] and identifies that such an algorithm could be useful for learning sorted data in LSMs; we leave their exploration within LSMs for future work.

Learning-based cache replacement policies. The Glider [129] cache replacement policy improves cache hit rate by insights from its offline LSTM-based variant. Cacheus [120] utilizes a combination of experts and chooses the best fit with reinforcement learning. LRB [131] brings learned replacement algorithms to CDN cache. Cache replacement policies are orthogonal to Symbiosis because Symbiosis tunes sizes of caches and doesn't affect policies.

5.2 LSM-tree Optimizations

Prior work has built many LSM optimizations. Monkey [34] carefully adjusts the bloom filter allocations for better filter hit rates and memory utilization. Dostoevsky [35], HyperLevelDB [45], and bLSM [123] develop optimized compaction policies to achieve lower write amplification and latency. cLSM [56] and RocksDB [46] use non-blocking synchronization to increase parallelism. SpanDB [29] optimizes RocksDB on hybrid storage by parallel log writes to fast storage devices. MatrixKV [167] reduces write stalls caused by compactions by managing lower level tables on fast storage devices.Bourbon take a different yet complementary approach to LSM optimization by incorporating models as auxiliary index structures to

improve lookup latency, but each of the others is orthogonal and compatible with Bourbon's core design.

5.3 Cache Management

Dynamic cache adaptation. As caching performance hinges on workload access patterns, prior work has explored how to dynamically adapt various aspects of cache management. Symbiosis, sharing a similar motivation to effectively adapt to online workload changes, benefits from relevant innovations and operates within a more complex application-kernel cache structure.

In the scenario of a single-level cache where no cooperation is explicitly introduced, such efforts centered around dynamic replacement policies [16, 120, 153], cache allocation and partitioning [44, 60, 74, 79, 105, 111, 126, 138, 143, 171], and online cache performance approximation [75, 93, 121, 151, 152, 158]. For instance, SOPA [153] simulates different cache replacement policies to dynamically decide the best policy. ACME [16] simultaneously runs multiple cache replacement policies and updates their weights by the instant effectiveness. Recently, machine learning techniques were also explored [120, 129].

Caching strategies designed for the properties of a given layer are necessary, such as for flash endurance [31, 58, 61, 109]. Our work, instead, considers compression, as it is widely used in modern key-value storage engines. Recent research also incorporates compression in storage systems [86, 101, 161, 170], underscoring its importance.

Hierarchical cache management. Earlier works have distilled and tackled several major problems introduced by hierarchical cache management [165]: weak temporal locality in the second layer [172] due to the first layer's filtering effect, duplication of data that wastes capacity [18, 30, 159], and a lack of information in the second layer for decision making [18]. "Exclusiveness" is one of the main challenges. Either API changes for cooperation are required [55, 159] or some sort of hints from the upper layer need to be propagated or derived [18, 90, 165, 166]. For instance, with DEMOTE [159], the lower level deletes a block from its cache when it is read by the upper level. Achieving exclusiveness in the application-kernel cache structure with one compressed layer would be an interesting future work.

Evolving storage devices (e.g., NVM) [31, 65, 84, 85, 87, 160] and use cases (e.g., S3) [57, 70, 91, 135] have led to new techniques to manage storage hierarchies and cache

cooperation. For example, EDT [57] decides and adapts data placement between tiers of SSDs and HDDs according to workload, aiming to minimize power consumption. AutoNUMA [91] is the kernel mechanism to monitor data access by inserting artificial page faults and move hot data pages to higher tiers of the cache hierarchy. D3N [70] also adapts sizes for multi-level caching with a ghost cache, but aims to alleviate network imbalance. A whole-stack programmable caching scheme is proposed [135] with APIs for size allocation of caches in layers within multi-tenant data centers. The adaptation space of Symbiosis, which accounts for computation (compression), capacity, and IO, is enlarged by modern fast block devices.

Symbiosis only tunes the sizes of caches and is optimized for the application-kernel cache structure, without altering their interaction. Notably, it does not require modifications to the OS kernel. These advanced communication techniques and policies are complementary.

5.4 Kernel Methods in Memory Management

Kernel cache and application coordination. Deep understanding of kernel caching is crucial to performance optimization across the storage stack. The performance impact of kernel cache replacement policies and directory cache has been studied [24, 66, 149]. Butt *et al.* [25] builds a simulator studying kernel prefetching. Tricache [47] replaces the kernel page cache for performance and also emphasizes transparent cache management for applications. Lee *et al.* [83] enables application-specific kernel caching. Symbiosis, instead, utilizes simulation integrated into applications in a live system to adapt cache configuration.
Zero-copy I/O stacks. PASTE [59] and Fastmove [137] use DMA to avoid copying. Arrakis [116] redesigns the kernel as a control plane so that I/O data may reside solely in user space. zIO [134] deduplicates buffers for I/O transparently, with a key observation that applications often modify only small parts of the data.
Zero-copy IPC. The kernel page cache naturally serves as an inter-process shared-memory buffer for data on the filesystems, and DISCO [23] uses a similar mechanism to share data on filesystems across virtual machines. Data needs to reside on the filesystems to be shared through the page cache. Manipulating page tables is a well-known approach to avoid copying. Fbufs [40] uses this approach for IPC, but programs must identify what data to share in advance (in contrast to de-anonymization). IO-Lite [114] extends Fbufs, adding a *buffer aggregate* abstraction that (like resharing) allows some outputs to reference inputs. Kelvin applies resharing in the context of Arrow, where additional techniques are possible

(e.g., dictionary sharing and IPC inspection). X-kernel [62] assigns a dedicated process per message, or data flow, to perform operations on data within the kernel, but requires the data representation to match the data structure in the kernel or a transformation is required at the user/kernel boundary. RMMap [97] utilizes RDMA to provide an mmap-like abstraction that works for processes on different machines. RMMap further solves the pointer problem via address space planning (i.e., making sure the same physical data is mapped to the same virtual addresses). Such planning is an alternative to using a zero-copy format (e.g., Arrow). Nightcore [64], like Kelvin, mounts a tmpfs into multiple containers to facilitate communication. Nightcore focuses on low-latency RPCs between microservices; Kelvin is optimized for DAG workloads where children consume the large outputs of multiple parents. Arrow's experimental Dissociated IPC Protocol [15] proposes tags to indicate how message bodies should be interpreted; this could provide a standardized way to implement Kelvin's SIPC, where contents would be interpreted as references to tmpfs files.

Zero-copy kernel mechanisms. The Linux kernel has several existing and proposed features related to our work. KSM [37] scans physical memory and modifies page tables to deduplicate redundant pages it discovers (DeCache avoids duplication from the start). Senpai [156] introduced the limit-dropping idea for controlling swap, though Senpai's developers eventually abandoned the approach because the system could not respond quickly enough to surging memory needs. Limit dropping better suits Kelvin, where it is only used to swap out intermediate data produced by completed processes. A new *process_vm_mmap* [146] API has been proposed for Linux that would allow different processes to share VMAs (including anonymous ones). A Kelvin-like system could be built around such a primitive, though being able to directly access another address space would naturally be a privileged operation (a file-oriented approach such as de-anonymization supports better control over visibility). Another proposed API (*msharefs* [17]) would allow different processes to share page table entries (not just pages). We expect that Kelvin's SIPC could be slightly faster using such a feature.

5.5 Data Pipelines Techniques

Data pipeline platforms. Many classic data-processing platforms (e.g., MapReduce [36, 130] and Spark [169]) impose both a computational model and intermediate data format. These restrictions are conducive to optimization, but the need for flexibility has created use

cases for other data pipeline orchestration tools, including Apache Airflow [13], Luigi [132], and Metaflow [141]. These tools allow arbitrary code and data passing strategies, but copy avoidance is difficult for arbitrary data passing strategies. Kelvin takes a balanced approach, allowing arbitrary node code, but requiring Arrow for data passing. Kelvin is designed to serve the same workloads as an existing commercial product, Bauplan [80], which is also based on containers and Arrow. Kelvin's use of DeAnon and SIPC allows it to avoid some copying and duplication that are necessary in the original implementation, which relies on a generic Linux kernel and Arrow IPC.

Storage-based communication. Many data processing platforms leverage distributed file systems (e.g., HDFS) or cloud storage (e.g., S3) for intermediate data, but experience has shown that it is difficult to achieve fast and reliable performance this way [115]. SONIC [100] and SAND [11] use a hybrid method of remote storage and local file system based on online profiling and function placement. Fortunately, hardware improvements are catching up to workload sizes [118] such that most workloads can run on a single machine and take advantage of faster mediums [104, 140, 145].

5.6 Hardware-Related Advancements for Memory Efficiency

Processing in Memory. Processing-in-Memory (PIM) has been extensively examined in recent years, transitioning from theoretical concepts to practical products. PIM eliminates the data copying and movement from memory to the computing unit, significantly improving memory utilization and computation latencies.

Modern PIM approaches are divide into two categories: processing-using-memory exploiting DRAM operational principles for parallel operations, and processing-near-memory exploiting logic layers on modern memory chips [69, 108].

On the production side, Samsung's HBM-PIM technology [122] integrates Programmable Computing Units (PCU) directly into HBM. UPMEM [150] proposes a general-purpose PIM architecture applicable for different software implementations. On the software side, PIM-STM [95] introduces software transactional memory to address the synchronization challenges in PIM. Syncron [54] tackled the problem of coordinating computation across distributed PIM units.

Hardware Memory Compression. Hardware-based memory compression letmemory controller in the CPU transparently compresses and decompresses memory to increase the logical memory capacity of the system.

Many aspects of hardware memory compression systems have been studied. The seminal work, Pinnacle [148], established the architecture of hardware memory compression, with an static address translation table between uncompressed memory and compressed memory. Laghari *et al.* designs a memory allocator for compressed memory to improve its performance and stability. CRAM [168] proposes implicit-metadata mechanism that eliminates metadata access during compression.

Chapter 6

Conclusions

In this chapter, we summarize each part of this thesis (§6.1), discuss lessons learned while working on this thesis (§6.2), discuss future directions (§6.3), and conclude (§6.4).

6.1 Summary

This thesis consists of three projects, Bourbon, Symbiosis, and Kelvin, that improve the memory efficiency of the data processing stack from different angles.

6.1.1 Bourbon

Bourbon is a learned index for LSM-trees. Bourbon improves the performance of in-memory workloads for the database layer and provides a smaller index by careful study of LSM characteristics.

Through in-depth measurements and analysis, we derive a set of guidelines to integrate learned indexes into LSM-trees. We have found that learning is more beneficial to lower-level tables as they live longer, but it can also be beneficial to higher-level tables because they may serve more lookups under certain workloads. Being workload- and data-aware is the key to performance improvements.

We adopt the piecewise linear regression model because its one-pass learning phase and logarithmic inference time meet our requirements for low overheads. We support variable-size values by borrowing the idea of key-value separation from WiscKey. We deploy

a cost-benefit analyzer that uses past statistics of tables on the same level to determine whether to learn a model for a new table during runtime.

We build Bourbon upon WiscKey. With various microbenchmarks and macrobenchmarks, we show that Bourbon significantly improves the performance of in-memory workload by accelerating indexing and Bourbon's model is 0.5× to 0.75× smaller than WiscKey's indexes. Through microbenchmarks, we show that Bourbon consistently yields benefits for read workloads with different data distributions and request distributions while incurring no overhead to writes. For macrobenchmarks, Bourbon performs 1.06× to 1.64× faster on YCSB and 1.48× to 1.74× faster on SOSD.

6.1.2 Symbiosis

Symbiosis is a framework for robust cache size adaptation to different workloads for key-value storage systems. Symbiosis optimizes the cache sizes of storage engines with the knowledge of the underlying layer, the kernel page cache, and improves the overall cache efficiency across the data processing stack.

We first study the factors that affect the cache partitioning problem by extracting key statistics from the application cache and the kernel page cache. Our simulation shows that the best cache configuration is highly sensitive to factors such as memory capacity, data compression ratio, and miss cost, which vary significantly in different workloads.

We thus build Symbiosis to dynamically configure cache sizes according to the current workload by online cache simulation. Guided by our offline simulator, we develop optimization techniques such as incremental reuse of ghost caches and misalignment-aware sampling to achieve both high accuracy and low overhead in online cache simulation.

We demonstrate Symbiosisś benefit by various static and dynamic workloads. In static workloads, Symbiosis finds the best cache size configuration in different workloads, software, and hardware environments and yields an average of 1.5× gain over static configurations in read-heavy workloads. In dynamic workloads, Symbiosis is able to adapt to workload changes in all cases and incurs negligible space and time overhead in online cache simulation.

6.1.3 Kelvin

Kelvin is a new pipeline execution engine that avoids all kinds of data copying and duplication in data pipeline execution to the best of our knowledge. Kelvin co-designs different

layers of the data processing stack (user-space resource management, container runtime, and kernel support for shared memory) to improve memory efficiency for data pipelines.

We first identify three types of data copying and duplication common in current data pipeline systems: data copying from anonymous memory to shared memory for data communication, data copying from the inputs of a node to its outputs, and duplicated deserialized data when multiple DAGs share the same inputs.

To eliminate them in a containerized environment, we build several subsystems of Kelvin. DeAnon is a kernel module that converts anonymous memory to shared memory without copying data. SIPC is a container runtime that communicates data between nodes with shared memory and removes the copying between inputs and outputs within a node. DeCache caches the deserialized data form of input sources and directs DAGs with the same inputs to utilize the cache. In addition, the resource manager in Kelvin performs reference tracking and resource accounting of the underlying shared physical data.

Each of Kelvin's subsystems is evaluated with benchmarks consisting of different applications and memory usage patterns and is shown to achieve the design goal of eliminating the corresponding data copying and duplication. Kelvin is further evaluated with real-world data pipelines from the DABstep benchmark and shows 1.2× to 2.3× performance gain.

6.2 Lessons Learned

We list several methodologies for systems research learned from the works in this thesis.
Memory efficiency is critical to performance, even if there is enough memory. When memory is not enough, it sounds trivial that using the limited resources efficiently will certainly be helpful for performance. In this case, Symbiosis automatically gives more memory to the cache for compressed data so that we effectively cache more data in memory and reduce device I/O. In Kelvin, eliminating the extra data copies reduces swaps caused by memory overflow.

But people often overlook the effect of saving computation and being friendly to the hardware CPU cache when we improve memory efficiency given enough memory. Even in data processing workloads and machine learning workloads, which are considered computation-bounded, reducing the copies in producing the output data with Kelvin could provide 1.2× gain solely by saving the computation of copying. When optimizing the online simulator in Symbiosis, we also noticed that, though the simulator takes only up to 2% of the available

memory, reducing its memory consumption by optimizing the data structures and carefully reclaiming the memory used by the simulator after simulation can largely reduce the cost of online simulation. This is because a process with a smaller memory footprint generally utilizes the small hardware CPU cache better. Engineering details such as fragmentation in memory pages and the kernel's page reclamation policy could also affect the actual memory footprint of user-space caching. As a developer of low-level systems such as operating systems and the kernel, we should pay close attention to memory efficiency in both high-level design and low-level implementation.

Optimize software with knowledge across layers. There are huge opportunities in performance optimization across different layers in the modern data processing stack, as modern software is often developed independently, well-optimized on its own, and lacks communication in between. For example, though popular storage engines embed sophisticated caching within themselves, they are either unaware of the underlying kernel page cache [133], or only recommend static configurations [53, 107] such as 25/75 or 50/50 partitions (which turn out to be sub-optimal in most workloads we've tested). Symbiosisś performance gain thus comes from the knowledge of the underlying page cache and the dynamic nature of the two-layer cache partitioning problem.

Kelvin, on the other hand, optimizes the underlying kernel with the knowledge of the applications' needs. Kelvin builds necessary kernel mechanisms to support efficient execution of popular DAG-style applications for data processing and co-designs the applications to utilize the new mechanisms. As software evolves much faster than the underlying systems nowadays, there are huge potentials in application-kernel co-design that build specialized systems for new applications.

Measure carefully first. Good systems research often starts with detailed measurements of existing systems, discovering problems in existing systems and guiding the initial design of improvements. In Bourbon, a large-scale analysis of table lifetime in WiscKey leads to the precious learning guidelines that lay the foundation of learning policies under write workloads. In Symbiosis, we conducted a study on various types of storage engines and identified the universal problem of cache partitioning by collecting cache statistics from both the application and the kernel cache. Detailed measurements into the kernel cache also lead to the high accuracy of our offline simulator, which guides the design and optimization of the final online simulator.

Reuse existing systems. There are two advantages in reusing existing systems and only

building add-ons that implement the core design of a systems research project. First, this enables a more fair comparison, that we can use the original systems as the baseline and compare them to our modified system to highlight the benefits of the changes we have made. In Bourbon, we keep the database write path of WiscKey intact and only make changes to the indexing part of the read path to show that we add no overhead to writes. In Symbiosis, we reuse the original cache-related policies in LevelDB and only change the sizing.

The second advantage is that reusing saves enormous amounts of engineering efforts and allows us to only focus on the core functionalities of a research project. Bourbon and Symbiosis adds about 5K and $<$ 1K LOC to the original Wisckey and LevelDB code base (around 20K LOC), respectively. Especially for Symbiosis, we have also ported it to RocksDB and WiredTiger for additional evaluation within only two weeks. Kelvin is built upon OpenLambda, reusing all the existing logic for basic containerized execution and only focusing on the logic of DAG scheduling and resource accounting.

But, be brave to break the systems for performance. When we were building Bourbon, the original plan was to change only the Search DB phase in Wisckey (explained in Figure 2.6), but we soon found out that after the model size grows as the data set gets more complicated, the performance benefit of replacing the binary search with model inference becomes marginal. We thus come up with the final design that completely replaces the logic to find tables in the database and the indexing into a single table.

It is critical to identify the correct scope of changes we want to make to the original systems. The previous example shows that breaking the systems too little could result in insufficient return, but changing the systems too much would unexpectedly break the assumptions of other parts of the systems. In Bourbon, we end up giving up level models upon workloads with writes, because level models treat all tables on one level as a unit, which is not true in write workloads as compactions frequently change small parts (tables) of each level. In Kelvin, the original design of DeAnon was to let the data sharer record the anonymous pages it would like to share and then create another mapping in the sharee's page table to these pages. This creates multiple mappings from unrelated processes to the same anonymous page, which breaks the assumption of the swap subsystem in Linux, makes these shared pages unswappable, and downgrades the performance with limited memory. To avoid touching the much more complicated swap subsystem, we come to the design that changes the anonymous mapping in the sharer process to shared memory-based file mapping, where both the states before and after sharing are legal in vanilla Linux and

support swapping natively. The state transition breaks assumptions of the underlying data structures in Linux, but is carefully guarded by locks from being visible to other processes. Moreover, the sharee can map to the shared data using the original `mmap` system call instead of another modified interface. This way, we precisely restrict our intrusion into the kernel to achieve our design goal while minimizing the chances for bugs.

6.3 Future Work

In this section, we discuss how we can extend the projects in this thesis and the directions our research points to.

Optimize write workloads in application-kernel caching. Symbiosis optimizes the partitioning of memory between the application cache and the kernel page cache, which are the caches for read workloads. Though Symbiosis can handle read-write mixed workloads with moderate amounts of writes, Symbiosis only optimizes the read requests in such cases and does not affect writes. For write workloads, newly written data and other data structures come in to compete for memory. Solving the problem of partitioning memory among these data structures to optimize write requests would pave the way for Symbiosisś idea to real-world deployment.

One potential challenge would be the variety of ways of handling writes in different storage engines. For read requests, storage engines with different data structures, such as WiredTiger (B-tree) and RocksDB (LSM-tree), share a similar logic: use indexing structures to find the target data pages and load and search on them. Caching is only involved in loading data pages and thus Symbiosis can be ported to these storage engines with minimal effort. But they differ largely in handling writes, and Symbiosis would need to be customized differently to match each storage engine's write path. For example, for RocksDB, writes are first buffered in memory and read requests need to query such memory buffers first. Thus, the memory buffers, the application cache, and the kernel cache form a three-layer cache structure and share the same memory quota, largely complicating Symbiosisś online simulation. Moreover, LSM-trees may prefer pinning certain lower-level tables in memory to accelerate lookups and compactions when there are write requests. A more sophisticated design is required to take such policy choices into account.

More study on implicit kernel policies. Symbiosis tunes the size of the kernel page cache in user space by changing the memory used by the application. A hidden policy is

demonstrated here that the kernel page cache would implicitly use up all the memory left by the application. There are plenty of other hidden policies in the kernel, with a high potential to significantly affect performance, for us to discover, study, and optimize. For example, the kernel page cache utilizes a variation of 2Q [67, 92] as its eviction policy. The admission policy of the page cache, on the other hand, is coupled with the readahead algorithms for each filesystem. A large number of parameters and policies are hard-coded here, such as the trigger of 2Q balancing, the activation policy of 2Q, and the trigger and the window size of readahead.

It is very likely that the one set of hard-coded numbers and policies does not fit the needs of different kinds of applications. We first need to expose them to be tunable. Unlike the size of the page cache that can be implicitly tuned in the user space, explicit access to the kernel data structures may be necessary. Fortunately, with modern tools like eBPF [43], this can be achieved with little intrusion into the kernel code. We can then further dynamically tune such parameters and policies according to the workload of applications.

A unified kernel interface for memory sharing between trusted processes. DeAnon in Kelvin utilizes the existing POSIX shared memory mechanism to share anonymous memory of a process. DeAnon can be viewed as a workaround to the kernel's enforced isolation between processes' address spaces. Such isolation is necessary for security reasons, but in the big data era, we may need to revisit this fundamental policy for efficient movement of data across different (trusted) application processes.

A possible solution could be letting the processes of a new type of (trusted) user share each other's address spaces and manually control the lifetime of their generated data in memory. This allows seamless sharing of data between these processes and application-controlled lifetime management with more knowledge of data dependencies in the user space.

A unified user-space protocol for sharing memory buffers between processes with Arrow. Kelvin needs to modify the Arrow IPC protocol to pass data on POSIX shared memory generated by DeAnon without copying, because vanilla Arrow always copies data when generating IPC data regardless of where the data resides. A more generic protocol that zero-copy passes data, which resides not only on shared memory but also on GPU and other new memory media such as remote memory, would be beneficial for modern data processing applications utilizing new types of memory resources. Arrow's experimental Dissociated IPC Protocol [15] is a starting point on this direction.

More sophisticated learning algorithms for learned indexes. BOURBON only supports numerical keys with fixed length. Current studies on learned index only focus on numerical keys. One way towards general string keys may be treating them as base-64 integers. Potential problems would be the efficiency of model training and inference on large numerical values. In BOURBON, the numerical values are treated as 32-bit integers, and we have seen large overhead when using 64-bit integers, also seen in a later study of learned indexes [102]. But 32-bit integers could only represent ASCII strings of length 8, and those of length more than 16 would even need 128-bit or larger integers. Efficient large-integer math is likely required for efficient training and inference in such cases.

6.4 Closing Words

Data is everywhere in our everyday life. People have built a software stack to process data and with the amount of data increasing, memory efficiency of the data processing stack becomes more and more critical.

In this thesis, we have introduced three aspects of study on memory efficiency of the data processing stack, especially on optimizations across layers of the stack. We first studied the performance characteristics of a popular LSM-based storage engine and built a learned index for LSMs, reducing the index sizes and improving the performance of in-memory workloads. We then focused on the caching problem between the layer of storage engines and the layer of the underlying kernel, and built an add-on module to storage engines that automatically optimizes cache allocation by online cache simulation. Last, we applied kernel methods to remove data copying and duplication in data pipelines and co-designed a pipeline execution engine with new user-space runtimes and kernel-space mechanisms.

Our studies have built practical solutions to real-world problems of the data processing stack. We hope that our work will inspire the future development of the stack, such as designing a coherent cache structure throughout the stack, or a unified user-space and kernel-space interface for memory sharing across processes.

Bibliography

[1] Adyen. https://www.adyen.com/.

[2] BadgerDB. https://github.com/dgraph-io/badger.

[3] Huggingface. https://huggingface.co/.

[4] Open Street Maps. https://www.openstreetmap.org/#map=4/38.01/-95.84.

[5] Running a Workload. https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload.

[6] Symbiosis Repository. https://github.com/daiyifandanny/Symbiosis, 2023.

[7] Jayadev Acharya, Ilias Diakonikolas, Jerry Li, and Ludwig Schmidt. Fast Algorithms for Segmented Regression. *arXiv preprint arXiv:1607.03990*, 2016.

[8] Adyen and Hugging Face. Dabstep. https://huggingface.co/datasets/adyen/DABstep.

[9] Adyen and Hugging Face. Data agent benchmark for multi-step reasoning. https://medium.com/adyen/data-agent-benchmark-for-multi-step-reasoning-dabstep-70e913c339dc.

[10] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of Optimal Page Replacement. *J. ACM*, 18(1):80–93, January 1971.

[11] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX '18)*, Boston, MA, July 2018.

[12] Amazon. Amazon Customer Reviews Dataset. `https://registry.opendata.aws/amazon-reviews/`.

[13] Apache. Apache Airflow. `https://airflow.apache.org/`.

[14] Apache. Arrow. `https://github.com/apache/arrow`.

[15] Apache. Dissociated ipc protocol. `https://arrow.apache.org/docs/format/DissociatedIPC.html`.

[16] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive Caching Using Multiple Experts. In *In Proceedings in Informatics*, pages 143–158, 2002.

[17] Khalid Aziz. memshare. `https://lore.kernel.org/lkml/a1d6a3de-502e-4114-a603-01710e30428e@oracle.com/T/`.

[18] Lakshmi N. Bairavasundaram, M. Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[19] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX '19)*, Renton, WA, July 2019.

[20] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[21] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[22] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to b^ϵ-trees and write-optimization. *;login: Operating Systems and Sysadmin*, (5):23–28, Oct 2015.

[23] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, page 412–447, November 1997.

[24] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[25] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. In *Proceedings of the 2005 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, Banff, Canada, June 2005.

[26] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Virtual Conference, February 2020.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, WA, November 2006.

[28] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: a Cloud-cost Optimized Self-designing Key-value Storage Engine. *Proceedings of the VLDB Endowment*, 15(1):112–126, 2021.

[29] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, February 2021.

[30] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[31] Wonil Choi, Bhuvan Urgaonkar, Mahmut Kandemir, Myoungsoo Jung, and David Evans. Fair Write Attribution and Allocation for Consolidated Flash Cache. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Virtual Event, March 2020.

[32] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), June 1979.

[33] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[34] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD '17)*, Chicago, IL, May 2017.

[35] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[37] Linux Kernel Developer. Kernel sampage merging. `https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html`.

[38] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. ALEX: An Updatable Adaptive Learned Index. *arXiv preprint arXiv:1905.08898*, 2019.

[39] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[40] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, December 1993.

[41] Romaric Duvignau, Vincenzo Gulisano, Marina Papatriantafilou, and Vladimir Savic. Piecewise linear approximation in data streaming: Algorithmic implementations and experimental analysis. *arXiv preprint arXiv:1808.08877*, 2018.

[42] Maria R. Ebling, Lily B. Mummert, and David C. Steere. Overcoming the Network Bottleneck in Mobile Computing. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 34–36, 1994.

[43] The eBPF community. Dynamically program the kernel for efficient networking, observability, tracing, and security. `https://ebpf.io/`.

[44] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA-18)*, Vienna, Austria, February 2018.

[45] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. HyperLevelDB. https://github.com/rescrv/HyperLevelDB, 2013.

[46] Facebook. RocksDB. `http://rocksdb.org/`.

[47] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*, Carlsbad, CA, July 2022.

[48] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index. *Proceedings of the VLDB Endowment*, 13(10):1162–1175, Jun 2020.

[49] Michael R. Frasca and Ramya Prabhakar. SRC: Virtual i/o Caching: Dynamic Storage Cache Management for Concurrent Workloads. In *International Conference on Supercomputing (ICS '11)*, Tucson, Arizona, May 2011.

[50] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*, Amsterdam, Netherlands, June 2019.

[51] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-size Data*. O'Reilly Media, Inc., 2011.

[52] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[53] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[54] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–276. IEEE, 2021.

[55] Binny S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[56] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.

[57] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, CA, February 2011.

[58] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash Caching on the Storage Client. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013.

[59] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[60] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, Santa Clara, CA, July 2015.

[61] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Trans. Storage*, 12(2), 2016.

[62] Norman C. Hutchinson, Larry L. Peterson, and Herman Rao. X-kernel: An open operating system design. In *Proc Second Workshop Workstation Oper Sys WWOS II*, 1989.

[63] Stratos Idreos and Mark Callaghan. Key-Value Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, Portland, OR, June 2020.

[64] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, Virtual Event, April 2021.

[65] Dejun Jiang, Yukun Che, Jin Xiong, and Xiaosong Ma. uCache: A Utility-Aware Multilevel SSD Cache Management Policy. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 391–398, 2013.

[66] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[67] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[68] Anna R Karlin, Kai Li, Mark S Manasse, and Susan Owicki. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5):41–55, 1991.

[69] R Kaur, A Asad, and F Mohammadi. A comprehensive review of processing-in-memory architectures for deep neural networks. *Computers*, 13(7):174, 2024.

[70] Emine Ugur Kaynar, Mania Abdi, Mohammad Hossein Hajkazemi, Ata Turk, Raja R. Sambasivan, David Cohen, Larry Rudolph, Peter Desnoyers, and Orran Krieger. D3N: A multi-layer cache for the rest of us. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 327–338, 2019.

[71] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An Online Algorithm for Segmenting Time Series. In *Proceedings 2001 IEEE international conference on data mining*, 2001.

[72] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A Benchmark for Learned Indexes, 2019.

[73] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. *arXiv preprint arXiv:2004.14541*, may 2020.

[74] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-Side SSD Caching for Storage Performance Control. In *2015 IEEE International Conference on Autonomic Computing (ICAC '15)*, Grenoble, France, July 2015.

[75] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Estimating Application Cache Requirement for Provisioning Caches in Virtualized Systems. In *Proceedings of the 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, July 2011.

[76] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *Proceedings of 9th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2019.

[77] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*, Houston, TX, June 2018.

[78] Kubernetes.io. What is cgroup v2. `https://kubernetes.io/docs/concepts/architecture/cgroups/#cgroup-v2`.

[79] Jaewon Kwak, Eunji Hwang, Tae-Kyung Yoo, Beomseok Nam, and Young-Ri Choi. In-Memory Caching Orchestration for Hadoop. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–97, 2016.

[80] Bauplan Lab. Bauplan. `https://www.bauplanlabs.com/`.

[81] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Montana, Oct 2009.

[82] Stéphane Lathuilière, Pablo Mesejo, Xavier Alameda-Pineda, and Radu Horaud. A comprehensive analysis of deep regression. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[83] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. P2Cache: An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications. In *15th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*, Boston, MA, July 2023.

[84] Eunji Lee and Hyokyung Bahn. Caching Strategies for High-Performance Storage Media. *ACM Trans. Storage*, 10(3), 2014.

[85] Eunji Lee, Hyojung Kang, Hyokyung Bahn, and Kang G. Shin. Eliminating Periodic Flush Overhead of File I/O with Non-Volatile Buffer Cache. *IEEE Transactions on Computers*, 65(4):1145–1157, 2016.

[86] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[87] Chu Li, Dan Feng, Yu Hua, and Fang Wang. Improving RAID Performance Using an Endurable SSD Cache. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 396–405, 2016.

[88] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A Scalable Learned Index Scheme in Storage Systems. *arXiv preprint arXiv:1905.06256*, 2019.

[89] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2119–2133, New York, NY, USA, 2020.

[90] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-Tier Cache Management Using Write Hints. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[91] Linux Kernel Organization. Autonuma. `https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git?h=tiering-0.8/`.

[92] Inc. Linux Kernel Organization. Linux Page Replacement Policy. `https://www.kernel.org/doc/gorman/html/understand/understand013.html`.

[93] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. eMRC: Efficient Miss Ratio Approximation for Multi-Tier Caching. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[94] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In *Advances in Neural Information Processing Systems (NIPS '14')*, pages 855–863, 2014.

[95] André Lopes, Daniel Castro, and Paolo Romano. Pim-stm: Software transactional memory for processing-in-memory systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, 2024.

[96] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: a high-performance learned index on persistent memory. *Proceedings of the VLDB Endowment*, 15(3):597–610, November 2021.

[97] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/deserialization-free state transfer in serverless workflows. In *Proceedings of the EuroSys Conference (EuroSys '24)*, Anthens, Greece, April 2024.

[98] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[99] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An Extensible Shuffle Architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, 2023.

[100] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX '21)*, Virtual Conference, July 2021.

[101] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using Transparent Compression to Improve SSD-Based I/O Caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, Paris, France, April 2010.

[102] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, September 2020.

[103] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9, 1970.

[104] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *USENIX Workshop on Hot Topics in Operating Systems*, 2015.

[105] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[106] Mathias Meyer. The Riak Handbook, 2012.

[107] MongoDB. MongoDB WiredTiger. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[108] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Rachata. A modern primer on processing in memory. *arXiv preprint arXiv:2012.03112*, 2020. Updated January 2025.

[109] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-RAC: SSD Friendly Caching for Data Center Workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[110] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[111] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[112] Our World in Data. Historical price of computer memory and storage. `https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage?time=2010..latest&facet=metric`, 2024.

[113] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[114] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[115] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, Portland, OR, June 2020.

[116] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[117] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shangai, China, October 2017.

[118] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. In *Proceedings of the 50th International Conference on Very Large Databases (VLDB 50)*, Guangzhou, China, August 2024.

[119] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[120] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[121] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seattle, WA, November 2014.

[122] Samsung Semiconductor. PIM: Processing-In-Memory Technology. `https://semiconductor.samsung.com/technologies/memory/pim/`, 2023.

[123] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, AZ, May 2012.

[124] Amazon Web Service. Aws data pipeline documentation. `https://docs.aws.amazon.com/data-pipeline/`.

[125] Amazon Web Services. Ec2 high memory instances with 18tib and 24tib of memory are now available with on-demand and savings plan purchase options. `https://aws.amazon.com/about-aws/whats-new/2022/10/ec2-high-memory-instances-18tib-24tib-memory-available-on-demand-savings-plan-purchase-options/`, 2022.

[126] Peter Shah and Keith Smith. Method for using service level objectives to dynamically allocate cache resources among competing workloads. https://patents.google.com/patent/US9836407B2/en, 2017.

[127] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. Operationalizing machine learning: An interview study. *arXiv*, 2022.

[128] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[129] 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.

[130] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[131] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, February 2020.

[132] Spotify. Luigi. `https://github.com/spotify/luigi`.

[133] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[134] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation (OSDI '22)*, Carlsbad, CA, July 2022.

[135] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '15)*, Kohala Coast, HI, August 2015.

[136] Michael Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 24(7), 1981.

[137] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. Revitalizing the forgotten On-Chip DMA to expedite data movement in NVM-based storage systems. In *Proceedings of the 21th USENIX Conference on File and Storage Technologies (FAST '23)*, Santa Clara, CA, February 2023.

[138] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.*, 28(1), 2004.

[139] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, 2023.

[140] Jacopo Tagliabue. You do not need a bigger boat: Recommendations at reasonable scale in a (mostly) serverless and open stack. In *Fifteenth ACM Conference on Recommender Systems*, RecSys '21, page 598–600, 2021.

[141] Jacopo Tagliabue, Hugo Bowne-Anderson, Ville Tuulos, Savin Goyal, Romain Cledat, and David Berg. Reasonable scale machine learning with open-source metaflow. *ArXiv*, abs/2303.11761, 2023.

[142] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 308–320, 2020.

[143] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA-10)*, Bangalore, India, January 2010.

[144] Petroc Taylor. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028. `https://www.statista.com/statistics/871513/worldwide-data-created/`.

[145] Jordan Tigani. Big data is dead. `https://motherduck.com/blog/big-data-is-dead/`, 2023.

[146] Kirill Tkhai. process_vm_mmap. `https://lore.kernel.org/linux-mm/155836082337.2441.15115541609966690918.stgit@localhost.localdomain/T/`.

[147] Matthew Topol. *In-Memory Analytics with Apache Arrow: Perform fast and efficient data analytics on both flat and hierarchical structured data.* 2022.

[148] R.B. Tremaine, T.B. Smith, M. Wazlowski, D. Har, Kwok-Ken Mak, and S. Arramreddy. Pinnacle: Ibm mxt in a memory controller chip. *IEEE Micro*, 21(2):56–68, 2001.

[149] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[150] UPMEM. UPMEM: True Processing-In-Memory Acceleration Solution. `https://www.upmem.com/`.

[151] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of the USENIX Annual Technical Conference (USENIX '17)*, Santa Clara, CA, July 2017.

[152] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[153] Yang Wang, Jiwu Shu, Guangyan Zhang, Wei Xue, and Weimin Zheng. SOPA: Selecting the Optimal Caching Policy Adaptively. *ACM Trans. Storage*, 6(2), 2010.

[154] Johannes Weiner. PSI - Pressure Stall Information. `https://www.kernel.org/doc/html/latest/accounting/psi.html`, April 2018.

[155] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, February 2022.

[156] Johannes Weiner and Dan Schatzberg. Transparent memory offloading: more memory at a fraction of the cost and power. `https://engineering.fb.com/2022/06/20/data-infrastructure/transparent-memory-offloading-more-memory-at-a-fraction-of-the-cost-and-power/`.

[157] Wikipedia. Data compression. `https://en.wikipedia.org/wiki/Data_compression`.

[158] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing Storage Workloads with Counter Stacks. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[159] Theodore Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[160] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual Conference, February 2021.

[161] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. ZExpander: A Key-Value Cache with Both High Performance and Fewer Misses. In *Proceedings of the EuroSys Conference (EuroSys '15)*, Bordeaux, France, April 2015.

[162] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. Maximum Error-bounded Piecewise Linear Representation for Online Stream Approximation. *The VLDB journal*, 23(6), 2014.

[163] Doris Xin, Litian Ma, Shuchen Song, and Aditya G. Parameswaran. How developers iterate on machine learning workflows - a survey of the applied machine learning literature. *ArXiv*, abs/1803.10311, 2018.

[164] Tim Xu. Quality of Service for Memory Resources. `https://kubernetes.io/blog/2021/11/26/qos-memory-resources/`, 2021.

[165] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Trans. Comput. Syst.*, 29(2), 2011.

[166] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All Replacement for a Multilevel Cache. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, CA, February 2007.

[167] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020.

[168] Vinson Young, Sanjay Kariyappa, and Moinuddin K. Qureshi. Cram: Efficient hardware-based memory compression for bandwidth enhancement. *arXiv*, 2018.

[169] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[170] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*, Philadelphia, PA, USA, June 2022.

[171] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards Practical Page Coloring-Based Multicore Cache Management. In *Proceedings of the EuroSys Conference (EuroSys '09)*, Nuremburg, Germany, April 2009.

[172] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.