ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Symbiosis: The Art of Application and Kernel Cache Cooperation

Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau
University of Wisconsin–Madison

Abstract. We introduce *Symbiosis*, a framework for key-value storage systems that dynamically configures application and kernel cache sizes to improve performance. We integrate Symbiosis into three production systems – LevelDB, WiredTiger, and RocksDB – and, through a series of experiments on various read-heavy workloads and environments, show that Symbiosis improves performance by $1.5\times$ on average and over $5\times$ at best compared to static configurations, across a wide range of synthetic and real-world workloads.

1 Introduction

Key-value storage engines, such as LevelDB [23], RocksDB [50], and WiredTiger [51], are essential components in modern data-intensive applications. These systems are deployed in numerous settings, including underneath relational databases [32, 52, 56], distributed storage systems [3, 18], graph processing engines [6, 13, 31, 55], stream processing systems [4, 12], and machine learning pipelines [30, 70].

A key factor in the performance of key-value storage systems is the effectiveness of in-memory caching. Unlike the traditional database approach [63], 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 (by default) will cache (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 greatly complicates performance tuning. How large should the application (uncompressed) cache be? How much memory should be dedicated to kernel-level (compressed) caching? 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.

In this paper, 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 [19] 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 three different key-value storage systems: LevelDB, WiredTiger, and RocksDB. Most of our work focuses on LevelDB, a popular LSM-based key-value storage system from Google [23]; 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 $5\times$) 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 [51] and RocksDB [50]) 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.

The rest of this paper is structured as follows. We introduce the cache partitioning problem and its significance (§2). Then, we conduct a simulation study of the general two-level cache partitioning problem to guide the design, approximations, and optimizations of Symbiosis (§3). We present Symbiosis's design and implementation, including its incorporation into LevelDB, WiredTiger, and RocksDB (§4). Finally, we perform an evaluation of our system (§5) 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.

2 Motivation and Framework

Databases and key-value stores utilize similar caching architectures (Figure 1). Irrespective of underlying data structure organization (log-structure-merge trees [23, 50] or B-trees [51, 61]), these systems use both a custom application-level cache and the underlying file system page cache.

To access a key-value pair, a request first queries an index-like structure, and, if successful, searches for the value in the

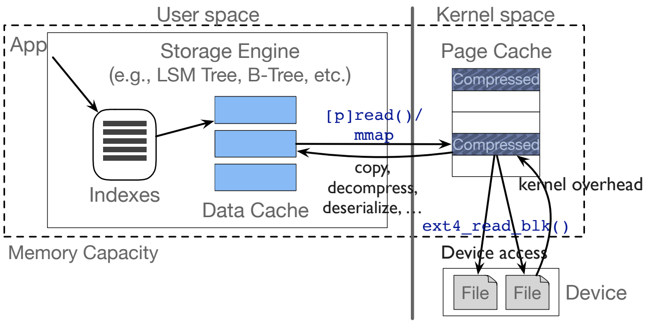


Figure 1: **The Cache Architecture across the Storage Stack**. *Modern applications commonly utilize storage engines (e.g., LevelDB) to manage on-disk data. A storage engine keeps compressed data on disk, and usually has separate index structures and an in-memory buffer for uncompressed data. The arrows depict the common read path.*

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 [14, 30].

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.

2.1 The Application-Kernel Cache Structure

We now describe the main properties of two-layer caching. 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 [23], 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 [51], 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 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.

2.2 Challenge: Memory Partitioning

Given this 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 [23] and B-tree-based WiredTiger [51]. 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 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

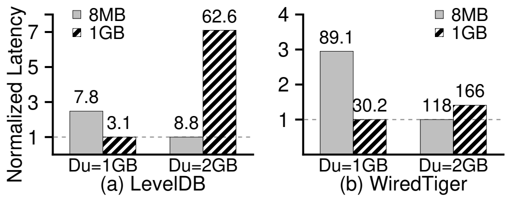


Figure 2: **Storage Engine Performance Varying Data Set Size**. *Each bar depicts one application cache size (8MB or 1GB); each pair of bars shows performance for a given dataset size. The y-axis is the latency normalized to the lowest value; numbers above are absolute latencies (us/op).*

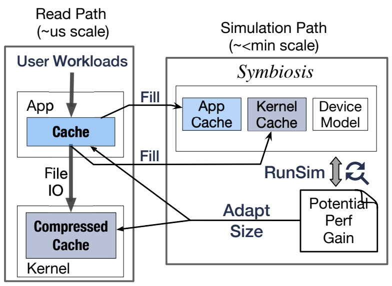


Figure 3: **Overview of Symbiosis**. *This figure shows the main components of Symbiosis and their interactions.*

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×.

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.

2.3 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 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* [19] 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 accuracy. We show how to strike this balance later (§4).

3 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).

3.1 Influential Factors

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

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 containers' resource limit [26, 38, 72], or enforcement by other mechanisms [71, 78]. 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.

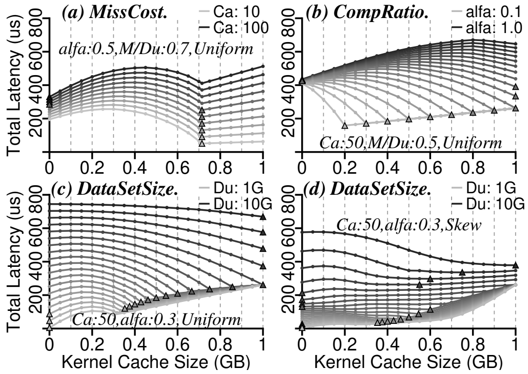
Data Size: The amount of compressed data that is stored on disk by an application is D_c; the corresponding uncompressed data size is D_u. We simulate 1GB $\leq D_u \leq$ 10 GB.

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 [73]; 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 [13]) and 0.5 (the default for RocksDB's *db_bench* [18]).

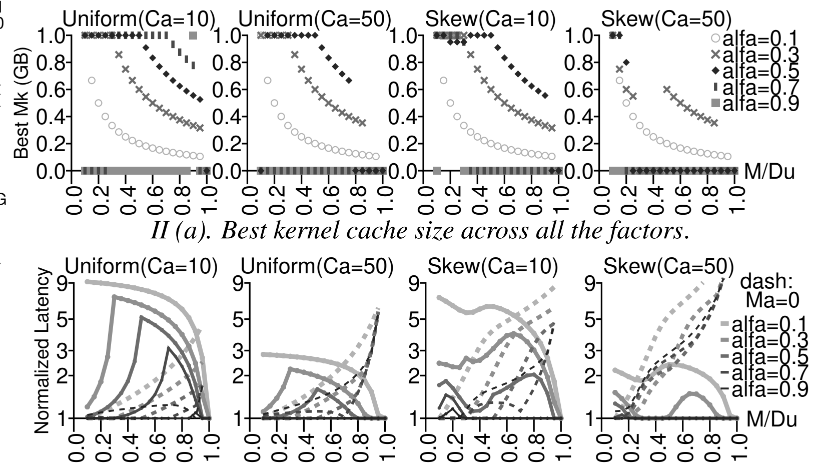
Retaining Data Size: (D_{mem}): We find the notion of a *retaining data size* useful: the size of the resulting data when it is all decompressed from memory. The minimum D_{mem} occurs when all of M is devoted to the uncompressed application cache; that is, $D_{mem}^{min} = M$. The maximum D_{mem} occurs 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 [9] 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 de-



I. 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.*



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

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

II. 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.*

Figure 4: **Simulation Results**.

pending on the compression algorithm in WiredTiger and is $< 10\mu s$ 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 $100\mu s$ for common devices. Again, the ratio of miss costs ($\frac{C_a}{C_k}$) matters and not their absolute values.

3.2 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: $L_e = (1-H_a) * (C_a + (1-H_k) * C_k)$. 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.

3.2.1 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 4 I(a) 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 4 II(a) 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 4 II(b) 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 4 I(b) 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 $M_k = \alpha * D_u = 0.2$.

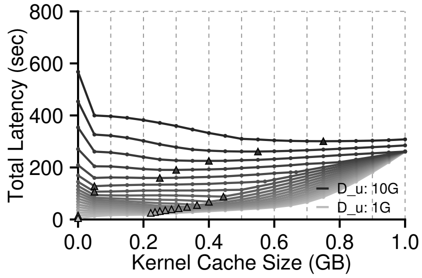


Figure 5: **Simulated performance under a Mixed (Read+Scan) workload**. *The legend describes parameters of the minimal and maximal value for the varying factor, DataSetSize (i.e., D_u). The triangle indicates the point of global minima. (Controlled factors: $C_a = 50, \alpha = 0.2$)*

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 4 II(a) confirms that larger kernel caches are more beneficial with smaller values of α and Figure 4 II(b) shows that the performance improvement is more dramatic with smaller α; the potential benefit of the kernel cache is high.

Data Size (D_u) **vs. Memory Capacity** (M): Figure 4 I(c) 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 4 II(a), the $\alpha = 0.7$ line in the first graph shows this trend best. As shown in Figure 4 II(b), 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 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 [13, 17]. We simulate a skewed workload containing a hotspot with locality as suggested by production RocksDB [13] in which 20% of the key space serves 80% of requests. Figure 4 I(d) shows that this skewed workload exhibits a significantly different performance curve from a uniform workload (Figure 4 I(c)). 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 4 II(a) shows this preference to the application cache, comparing the right half of graphs to the left half; Figure 4 II(b) 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 [13, 17]. We use the YCSB benchmark [17] 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 5 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.

3.3 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 4 II(b) 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 in online simulation can be tolerated.

4 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.

4.1 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 6 illustrates how Symbiosis integrates into existing storage engines. Symbiosis contains two main components:

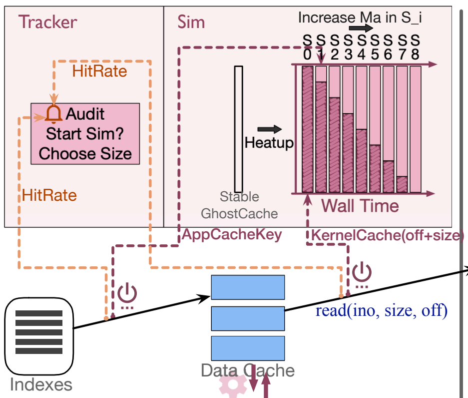


Figure 6: **Design of Symbiosis**. *Symbiosis is directly integrated into a storage engine. The orange dashed lines are the stats collection paths that are always active; the dashed red lines are the paths filling entries into the ghost cache, activated only in* Adapting State *and empty in* Stable State*. 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.*

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.

4.1.1 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.

4.1.2 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* [19,22,53,75], 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 ($> 30\%$).

4.2 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.

4.2.1 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 configuration, 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 §5.2.4.

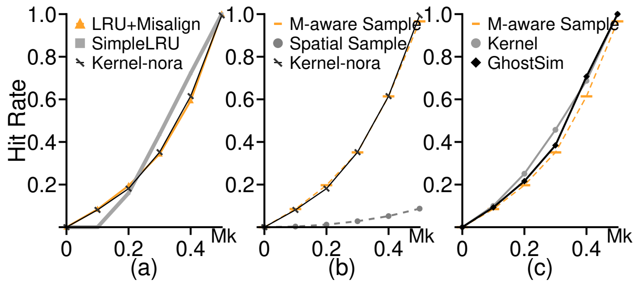


Figure 7: **KernelCache Simulation and Sampling**. *Kernel-nora and Kernel are the kernel cache implementations with and without read-ahead, respectively.*

4.2.2 Incremental reuse of a single Ghost Cache

We extend the idea of storing cache access metadata with a ghost cache [19, 22, 53, 75] 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 [48] (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 6 illustrates our choices of $<M_a, M_k>$ candidates. Our simulation results (Figure 4 II(a)) 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.

4.2.3 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 [67, 68]. 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 level while the application caches in application-defined blocks that misalign with pages; as a result, the independent reference model [2] 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 7 shows the hit rate curves for various kernel cache implementations and sampling approaches. Figure 7(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 7(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 7(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.

4.2.4 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.

4.2.5 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 [13, 17]. 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 [8], which is more challenging for prediction.

4.3 Multiple Implementations

We have integrated Symbiosis into three different storage engines: LevelDB [23], WiredTiger [51], and RocksDB [50]. 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.

Table 1: **Factors for Static Workload**. *Access patterns are generated by YCSB [17]. 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.*

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 [57]: Xeon D-1548 (2.0 GHz)
	Device Latency	HW1: OptaneSSD 900P ($\sim 10\mu s$) HW2: Toshiba NVMe flash ($\sim 70\mu s$)

5 Evaluation

We evaluate Symbiosis to answer the follow questions: (1) 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? (2) How quickly does Symbiosis react to workload changes and how much overhead does Symbiosis incur for simulation and changing cache sizes? (3) How well does Symbiosis handle real-world workloads?

Setup. We use HW1 in Table 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.

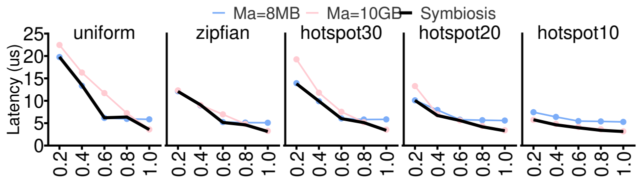
5.1 Static Workloads

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 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.

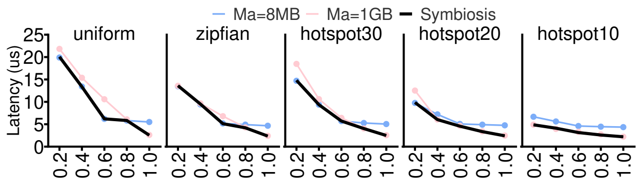
5.1.1 LevelDB Performance

Figure 8 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.

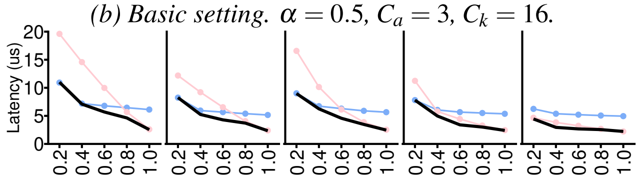
Large datasets and memory (a): To evaluate Symbiosis in the context of modern data center machines with large amounts of memory, we begin with $M = 10GB$ and a range of large data sets (D_u=50, 25, 16.7, 12.5, 10 GB); we use the basic setting of HW1 and LevelDB's default compression ($\alpha = 0.5$). In all cases, Symbiosis matches the performance of the better baseline. $Static_{M_a=8MB}$ tends to perform better



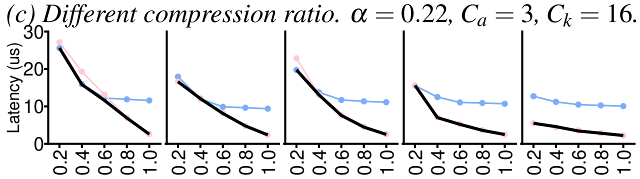
(a) Larger dataset and memory. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.



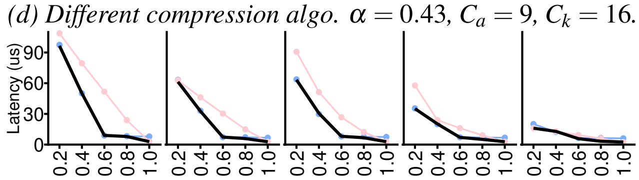
(b) Basic setting. $\alpha = 0.5$, $C_a = 3$, $C_k = 16$.



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



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

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). Thus, with better compression, Symbiosis achieves a larger performance increase over the worse baseline (commonly $> 1.2\times$) and some improvement 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, $Static_{M_a=1GB}$ usually performs better than $Static_{M_a=8MB}$ because $\frac{C_a}{C_k}$ is large (0.56) and $Static_{M_a=8MB}$ incurs the cost of the higher C_a. Symbiosis again always matches the performance of the better baseline, properly 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, $Static_{M_a=8MB}$ usually performs better than $Static_{M_a=1GB}$ 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>.

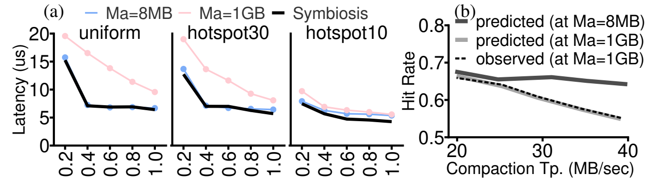
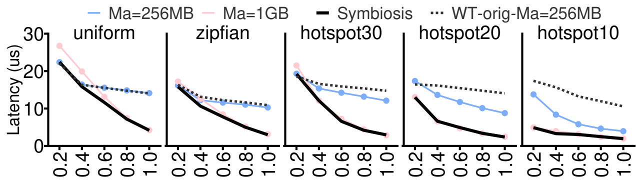


Figure 9: **Static Workload with 20% Overwrites**. *(a) X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $Static_{M_a=8MB}$, *similarly for Ma=1GB.* $\alpha = 0.22$, $C_a = 3$, $C_k = 16$. *(b) shows the predicted application cache hit ratio of the* $Ma = 1GB$ *configuration using cache traces from configuration* $Ma = 8MB$ *and* $Ma = 1GB$*, and the observed hit ratio when* $Ma = 1GB$*, under different compaction rates. The workload is uniform with 20% overwrite and* $M = D_u$.

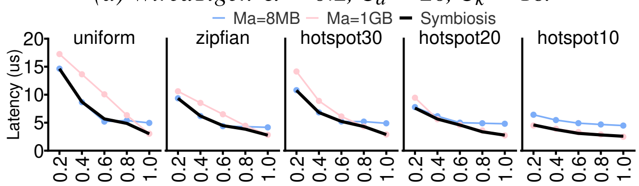
5.1.2 Workload with Writes in LevelDB

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 9(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 9(a) shows Symbiosis's performance with 20% overwrites. Compared to its read-only counterpart (Figure 8(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.



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



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

Figure 10: **WiredTiger and RocksDB (Static Workload)**. *X-axis is* $\frac{M}{D_u}$. *Ma=8MB means* $Static_{M_a=8MB}$, *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.*

5.1.3 WiredTiger Performance

Figure 10(a) shows the performance benefits of incorporating Symbiosis into WiredTiger. As mentioned in §4.3, 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 ($\alpha = 0.2$), 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.

5.1.4 RocksDB Performance

Figure 10(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 8(a) where Symbiosis outperforms or matches the performance of the better baseline, demonstrating its capability to handle application-managed compressed data caches.

5.2 Dynamic Workloads

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

5.2.1 Example: LevelDB Behavior over Time

We begin by illustrating how Symbiosis within LevelDB behaves over time for a dynamic workload. Figure 11 presents

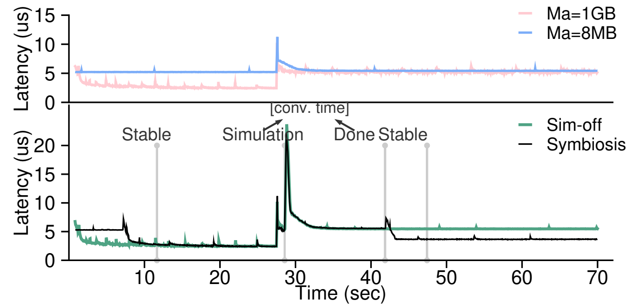


Figure 11: **Timeline of Latency under a Dynamic Workload (hotspot20:1.0-2.0)**. *The workload changes are aligned at* $\sim 26sec$*, and we label state transfer of Symbiosis by the gray vertical lines. Sim-off means we turn off the simulation and shows the effect of only resetting the application cache size to default; its steady performance is the same as* $Static_{M_a=1GB}$ *before the change, and the same as* $Static_{M_a=8MB}$ *afterwards.* ($\alpha = 0.22$)

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 $Static_{M_a=8MB}$ 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.

5.2.2 Performance Gain and Dynamic Adaptation

To quantify the benefits, convergence time, and resulting cache configurations for a wide range of workloads with two

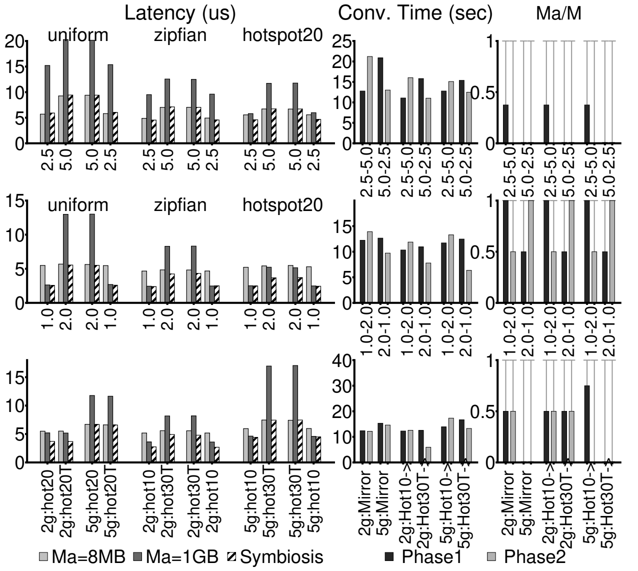


Figure 12: **Performance under Dynamic Workloads** ($\alpha = 0.22$). *In the Latency subplot, each group has three bars: $Static_{M_a=8MB}$, $Static_{M_a=1GB}$ and Symbiosis. Each adjacent bar group represents one workload1→workload2 change and the next group reverses the workloads. The first two rows 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.*

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 12 ($\alpha = 0.5$ omitted for brevity) but consider both αs when discussing extremes and averages.

We use the example above to explain the metrics in Figure 12, 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., $\sim$ 12 and 13 seconds for phase 1 and 2, respectively, shown by the time between the bars labeled as Simulation and Done in Figure 11). Finally, the M_a/M subplot shows the best application cache size found by Symbiosis (e.g., 1 and 0.5 for the example workload).

Figure 12 shows that Symbiosis delivers good latency in all cases, at least as good as the best baseline and sometimes better, with an average gain of 24% over $Static_{M_a=8MB}$, 42% over $Static_{M_a=1GB}$, 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).

Table 2: **Tail Latency**. *Overhead is the comparison to $Static_{M_a=8MB}$. ($\alpha = 0.22$)*

	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

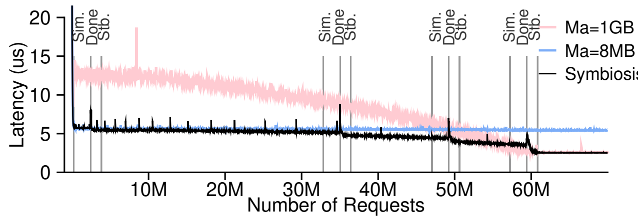


Figure 13: **Timeline of Latency under a Dynamic Workload with Gradual Change**. *The workload is uniform with $D_u = 2$ GB in the first 10M operations, $D_u = 1$ GB in the last 10M operations, and a uniform gradual change during the 50M operations in between. ($\alpha = 0.22$)*

Adapting the size online and potential latency spike symptoms raises concerns of tail latency. As shown in Table 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 $Static_{M_a=8MB}$. 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.

5.2.3 Gradual Change

We show that Symbiosis also performs well in workloads with more gradual changes (Figure 13). During the workload, $Static_{M_a=8MB}$ holds all the data in the kernel cache; $Static_{M_a=1GB}$ 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 $Static_{M_a=8MB}$.

Symbiosis matches the performance of $Static_{M_a=8MB}$ 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, 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 $Static_{M_a=1GB}$ is matched.

A gradual change of L_e is necessary for Symbiosis to match the change speed of workload. For workloads with

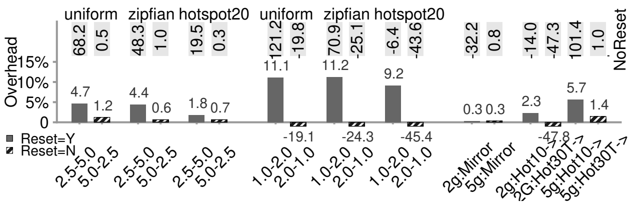


Figure 14: **Overhead during Simulation** ($\alpha = 0.22$). *The workloads are in the same order as in Figure 12. The bars are the overhead with the reset policy; dashed ones indicate no actual M_a change. Numbers in gray background are the overhead percentages without the reset policy.*

faster changes beyond Symbiosis's threshold during simulation, simulations are halted until the workload stabilizes.

5.2.4 Effect of Optimization Techniques

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

Reset Policy: The reset policy (§4.2.1) 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 14; 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 14, 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 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 $\sim$90*ns* 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, 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.

5.3 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 [13] are used, the first with the supplied parameters in the last example in paper [13], 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.

Table 3: **Space and Time Overhead and Convergence Time of Various Simulation Settings**. *Operation overhead compares to baseline LevelDB. Sample rate is $\frac{1}{64}$.*

Case	Memory Overhead (MB)	Operation Overhead (us/op)	Conv. 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

Figure 15 shows the performance of LevelDB on four consecutive traces based on the two workloads. $Static_{M_a=8MB}$ 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. $Static_{M_a=1GB}$ outperforms $Static_{M_a=8MB}$ 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, $Static_{M_a=1GB}$ performs worse than $Static_{M_a=8MB}$ 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 $Static_{M_a=8MB}$.

6 Related Work

Dynamic Cache Adaptation: As caching performance hinges on workload access pattern, prior work has explored how to dynamically adapt various aspects of cache management. Our work, 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 [5, 58, 69], cache allocation and partitioning [20, 28, 36, 39, 49, 54, 60, 64, 65, 82], and online cache performance approximation [37, 46, 59, 67, 68, 74]. For instance, SOPA [69] simulates different cache replacement policies to dynamically decide the best policy. ACME [5] simultaneously runs multiple cache replacement policies and updates their weights by the instant effectiveness. Recently, machine learning techniques were also explored [58].

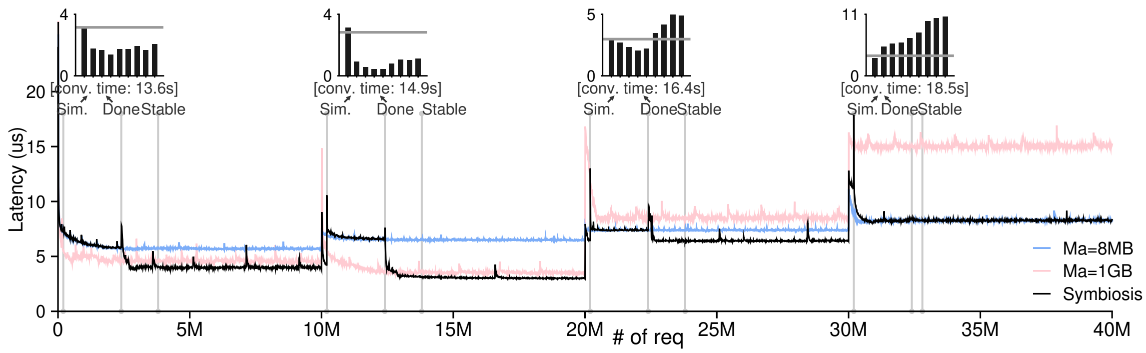


Figure 15: **Request Latency versus the Request Sequence**. *The 4 phases are composed of 2 workloads generated from RocksDB's mix_graph benchmark. Two versions of the first workload exhibit a decrease in D_u, with $Key_{max} = 50M$ and $D_u = 5$ GB in phase 1 and $Key_{max} = 25M$ and $D_u = 2.5$ GB 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 ($S_{\{0,\dots,8\}}$ 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. ($\alpha = 0.22$)*

Caching strategies designed for the properties of a given layer are necessary, such as for flash endurance [16, 27, 29, 53]. Our work, instead, considers compression, as it is widely-used in modern key-value storage engines. Recent research also incorporates compression in storage systems [43, 47, 77, 81], underscoring its importance.

Hierarchical Cache Management: Earlier works have distilled and tackled several major problems introduced by hierarchical cache management [79]: weak temporal locality in the second layer [83] due to the first layer's filtering effect, duplication of data that wastes capacity [7, 15, 75], and a lack of information in the second layer for decision making [7]. "Exclusiveness" is one of the main challenges. Either API changes for cooperation are required [24, 75] or some sort of hints from the upper layer needs to be propagated or derived [7, 45, 79, 80]. For instance, with DEMOTE [75], 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) [16, 33, 41, 42, 44, 76] and use cases (e.g., S3) [25, 35, 62] have led to new techniques to manage storage hierarchies and cache cooperation. For example, EDT [25] decides and adapts data placement between tiers of SSDs and HDDs according to workload, aiming to minimize power consumption. D3N [35] also adapts sizes for multi-level caching with a ghost cache, but aiming to alleviate network imbalance. A whole-stack programmable caching scheme is proposed [62] with APIs for size allocation of caches in layers within multi-tenant data center. The adaptation space of Symbiosis, which accounts for computation (compression), capacity, and IO, is enlarged by modern fast block devices.

Our approach 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.

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 have been studied [10, 34, 66]. Butt *et al.* [11] build a simulator studying kernel prefetching. Tricache [21] replaces the kernel page cache for performance and also emphasizes transparent cache management for applications. Lee *et al.* [40] enable application-specific kernel caching. Our work, instead, utilizes simulation integrated into applications in a live system to adapt cache configuration.

7 Conclusion

We have introduced Symbiosis, a framework to enable robust cache adaptation for key-value storage systems. 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 [1].

8 Acknowledgement

We thank Sam H. Noh (shepherd), anonymous reviewers, and the ADSL group for their comments and suggestions. This material was supported by funding from NSF CNS-1838733. Jing Liu was supported by a Meta PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

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

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

[3] Apache. Cassandra. http://cassandra.apache.org/.

[4] Apache. Kakfa. http://kafka.apache.org/.

[5] 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.

[6] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, June 2013.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and X Sean Wang. LogKV: Exploiting Key-Value Stores for Event Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[13] 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.

[14] 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.

[15] 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.

[16] 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)*, Lausanne, Switzerland, March 2020.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. IOCost: Block IO Control for Containers in Datacenters. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Lausanne, Switzerland, February 2022.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] ArangoDB Inc. What's the Latest with ArangoDB? `https://github.com/arangodb/arangodb`, 2022.

[32] PingCap. Inc. TiDB Introduction. `https://docs.pingcap.com/tidb/dev/overview`, 2022.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] Meta. RocksDB. http://rocksdb.org/.

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

[52] Arjun Narayan and Peter Mattis. Why we built CockroachDB on top of RocksDB. https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/, 2019.

[53] 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.

[54] 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.

[55] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. ArkDB: A key-value engine for scalable cloud storage services. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*, Virtual Conference, June 2021.

[56] Andy Pavlo. It is too expensive/time-consuming to build a DBMS from scratch. https://twitter.com/andy_pavlo/status/1523666179247595520, 2022.

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

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

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

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. *Proceedings of Machine Learning and Systems*, 2:199–215, 2020.

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

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.

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

A Artifact Appendix

Abstract

Symbiosis is a framework for key-value storage systems that dynamically configures application and kernel cache sizes to improve performance. This artifact includes code for Symbiosis-integrated LevelDB, RocksDB, and WiredTiger, the offline simulator, scripts to run these applications, and several workload traces used in the paper.

Scope

Our artifact is fully functional, including all the features and optimizations mentioned in the design and three implementations (i.e., Section 4). We provide several traces used in our evaluation. Running the three storage engines with and without Symbiosis in similar hardware settings can support our findings of the cache-sizing problem and the effectiveness of Symbiosis's design and techniques.

The offline simulator can run various size configurations and workloads; its kernel cache can be configured to mimic the kernel cache behaviors (e.g., 2Q and read-ahead). Running the simulator experiments with the same configuration as Section 3 is expected to exactly reproduce the results, supporting our findings about the impacting factors and performance gain under various workloads.

Contents

We describe the contents of the subdirectories in the root of the repository as below:

- `leveldb` contains Symbiosis-embedded LevelDB that is used to reproduce experiments in Section 5.1.1, Section 5.2.2, and Section 5.3.
- `wiredtiger` contains Symbiosis-embedded WiredTiger that is used to reproduce experiments in Section 5.1.3.
- `rocksdb` contains Symbiosis-embedded RocksDB that is used to reproduce experiments in Section 5.1.4.
- `simulator` contains the cache simulator (in Python) used in Section 3.
- `traces` includes all the traces for the experiments mentioned above.
- `scripts` includes the scripts to run the experiments mentioned above. Detailed instructions can be found in `ae_readme.txt`.

Hosting

The artifact is hosted on https://github.com/daiyifandanny/Symbiosis, on branch *main* with commit id *36e3ea7*.

Requirements

Offline Simulations (Section 3):

- Software: Python 3.8. Python package numpy and simpy.

Performance Evaluation (Section 5):

- Library: sdt, zstd, and snappy. Installation guide can be found in `ae_readme.txt`.
- System: Linux kernel 5.11 and Ubuntu 20.04.
- Hardware: Hardware listed in Table 1, especially an OptaneSSD, is necessary for reproducing the exact results. With different hardware, offline calibration of the application and kernel cache miss costs is required; the result (in microsecond) needs to be set in `leveldb/util/adapter.h`.