TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs

Guanyu Feng and Huanqi Cao, *Tsinghua University;* Xiaowei Zhu, *Ant Group;* Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen, *Tsinghua University*

https://www.usenix.org/conference/osdi22/presentation/feng

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs

Guanyu Feng[1], Huanqi Cao[1], Xiaowei Zhu[2], Bowen Yu[1], Yuanwei Wang[1], Zixuan Ma[1], Shengqi Chen[1], and Wenguang Chen[1]

[1]*Department of Computer Science and Technology & BNRist, Tsinghua University,* [2]*Ant Group*

Abstract

Out-of-core systems rely on high-performance cache subsystems to reduce the number of I/O operations. While the page cache in modern operating systems enables transparent access to memory and storage devices, it suffers from efficiency and scalability issues on cache misses, forcing out-of-core systems to design and implement their own cache components, which is a non-trivial task.

This study proposes TriCache, a cache mechanism that enables in-memory programs to efficiently process out-of-core datasets without requiring any code rewrite. It provides a virtual memory interface on top of the conventional block interface to simultaneously achieve user transparency and sufficient out-of-core performance. A multi-level block cache design is proposed to address the challenge of per-access address translations required by a memory interface. It can exploit spatial and temporal localities in memory or storage accesses to render storage-to-memory address translation and page-level concurrency control adequately efficient for the virtual-memory interface.

Our evaluation shows that in-memory systems operating on top of TriCache can outperform Linux OS page cache by more than one order of magnitude, and can deliver performance comparable to or even better than that of corresponding counterparts designed specifically for out-of-core scenarios.

1 Introduction

NVMe [45] Solid State Drives (NVMe SSDs) have drawn a wide range of interest because of their high I/O performance. The U.2 interface [48] and PCIe 4.0 standard [47] have also increased the storage density of NVMe SSDs in recent years. For instance, a dual-socket commodity server can mount an array of more than 16 NVMe SSDs to provide tens of TB of storage capacity, tens of millions of random IOPS, and dozens of GB/s of bandwidth while being 20–40 times cheaper than Dynamic Random Access Memory (DRAM).

Although NVMe SSD arrays can improve the aggregated performance and capacity of the system, they still suffer from block-wise I/O accesses and have latencies at least 100 times longer than those of DRAM. To efficiently process datasets that are significantly larger than available memory, out-of-core systems rely on cache sub-systems to maintain frequently operated data in memory. I/O operations can be merged or skipped on cache hits, bridging the performance gap between DRAM and SSDs.

Page cache [46] is a cache sub-system in modern operating systems (OS) that manages data on the granularity of pages (typically 4KB) across DRAM and SSDs. It enables in-memory applications to support out-of-core processing on SSDs without requiring any rewrite through swapping [44] or memory-mapping [43] based on virtual memory.

However, current implementations of page cache encounter issues related to scalability and performance on cache misses owing to global locking on internal data structures [29]. Recent literature [27, 34, 55] indicates that the heavy I/O stack, page faults, and context switching overheads also limit kernel swapping and I/O performance on fast storage devices such as NVMe SSD arrays.

Therefore, data-intensive applications such as databases and data processing systems [6, 12, 15, 20–22, 52, 54] usually design and implement their own user-space block caches (also known as buffer managers) that manage data by blocks (typically of a fixed size that is a multiple of the physical sector size). In contrast to OS page cache, block cache reduces context switching overhead by running mainly in the user space, and supports customization in terms of tuning block sizes and replacement policies to further improve performance.

Nevertheless, designing and implementing block caches and upper-level components imposes expensive development costs. Existing block caches in the user space usually ask users to explicitly acquire/release blocks [12, 20] or manipulate data through an asynchronous interface [54]. Developers often have to re-design and re-implement the entire system according to the API requirements of the block cache, which is non-trivial. To fill the gap between out-of-core performance and development costs, we investigate a new general cache mechanism that can transparently extend in-memory systems

for efficient out-of-core processing on NVMe SSDs without requiring any manual modification.

Efficient kernel-bypass I/O stacks, such as SPDK [50], can achieve good out-of-core performance in the user space by avoiding expensive kernel I/O operations to take advantage of the high IOPS from the NVMe SSD array. It inspires us to explore a user-space solution that can eliminate the overhead due to page faults and context switching. A solution in the user space is cross-platform, easy to deploy and customize, and avoids introducing potential security vulnerabilities caused by kernel modifications.

To ensure transparency for the user, a *virtual memory interface* is expected to fill the semantic gap between fine-grained memory accesses in existing in-memory programs and block-wise I/O operations on physical block devices as in the case of the virtual memory provided by OS. A virtual memory interface makes it possible for in-memory software to run on NVMe SSDs without requiring any modification if we can automatically redirect memory accesses to the block cache.

Several challenges need to be addressed to implement a user-space block cache with a virtual memory interface. First, the cache system requires good scalability to achieve high out-of-core performance so that it can fit in the hundreds of CPU cores and the tens of millions of SSD IOPS in use today. Second, it requires an efficient address translation mechanism that looks up in-memory addresses for cached blocks, to provide a virtual memory interface with fine-grained accesses. Such fine-grained accesses and per-access address translations pose a much more significant challenge than block lookups in current user-space block caches. Third, it requires a scheme to redirect the memory accesses of existing in-memory systems to the cache system without any manual modification.

To address the above challenges, we propose the following contributions:

- We build a scalable block cache based on a concurrency mechanism named *Hybrid Lock-free Delegation* that combines message passing based delegation with lock-free hash tables. It can utilize the NVMe SSD array with only a few server threads.
- We design a two-level *Software Address Translation Cache* (SATC) to support lightning-fast address translation in the user space, replacing human effort for writing block-aware code with an automatic mechanism by exploiting locality at runtime. SATC can accelerate software address translation by some orders of magnitude.
- We propose a pure software-based scheme to supervise memory accesses based on *compile-time instrumentation* and library hooking techniques. Existing in-memory applications can efficiently run on NVMe SSDs through the block cache without requiring any code modification.

Based on these techniques, we design and implement a user-transparent block cache providing a virtual memory interface, named TriCache. Our results show that TriCache enables in-memory programs to efficiently process out-of-core datasets without requiring manual code rewrite, by using various domains of application. TriCache can outperform OS page cache by some orders of magnitude, and can often reach or even exceed the performance of specialized out-of-core systems.

2 Background and Motivation

In this section, we briefly introduce the two types of general caches that can be used for out-of-core processing, OS page cache and user-space block cache, and use a motivating example to show the benefits as well as the challenges of a new approach that combines the advantages of both.

Page cache is a transparent cache for pages originating from storage devices [46]. Modern operating systems keep the page cache in unused portions of the main memory. Some accesses to storage devices can be handled by the page cache to improve performance. The page cache is implemented in kernels through virtual memory management and is mostly transparent to applications. Users can use a memory-mapping system call [43] to map a file to a segment in virtual memory, or rely on swapping [44] to swap out/in pages to/from disks on-demand, thus accessing storage just like memory.

While the memory interface of the page cache provides maximal user transparency for developing out-of-core applications [9, 26], its use can lead to severe performance bottlenecks, especially on cache misses when the backed storage is an array of high-performance NVMe SSDs. It results from various factors, including but not limited to its global locking in the kernel, the heavy I/O stack, page faults, and context switching overheads [27, 29, 34, 55]. Although some studies have attempted to modify the kernel to improve the performance of the page cache [27–29, 39, 41], it is challenging to apply the relevant modifications in the kernel space, which may introduce potential portability and security issues.

To this end, most out-of-core systems design and implement their own block caching components in the user space to mitigate and even eliminate the above issues. Like the OS page cache, a block cache manages a pool of pages in memory, and loads/evicts pages from/to disks upon user requests. The major difference is that the block cache runs mostly in user space and provides a block interface. Users first `pin` the blocks to be accessed in memory, then read/write data in corresponding blocks, and finally invoke `unpin` to mark the blocks that can be evicted or flushed to storage later, when needed according to the replacement policy [15, 20]. There are also some other forms of the block interface, such as asynchronous `read`/`write` routines with user-defined callbacks [54]. Block cache may be further customized for better performance according to the needs of the application. For example, it is unnecessary to support writing blocks back to the storage if cached contents are known to be read-only [12].

While an efficient and scalable block cache can make full use of storage devices in terms of performance, its block interface requires a considerable amount of work to be put

```
size_t strlen_memory(char* str) {
  size_t len = 0;
  while (str[len] != '\0')
    ++len;
  return len;
}
```

```
size_t strlen_block(str_in_block s) {
  size_t len = 0;
  size_t block_id = get_block_id(s);
  size_t block_off = get_block_offset(s);
  char* raw_ptr = pin(block_id);
  while (raw_ptr[block_off] != '\0') {
    len += 1;
    block_off += 1;
    if (block_off == BLOCK_SIZE) {
      unpin(block_id);
      block_id += 1;
      block_offset = 0;
      raw_ptr = pin(block_id);
    }
  }
  unpin(block_id);
  return len;
}
```

Figure 1: Out-of-core implementations of `strlen` with memory (upper) and block (lower) interfaces

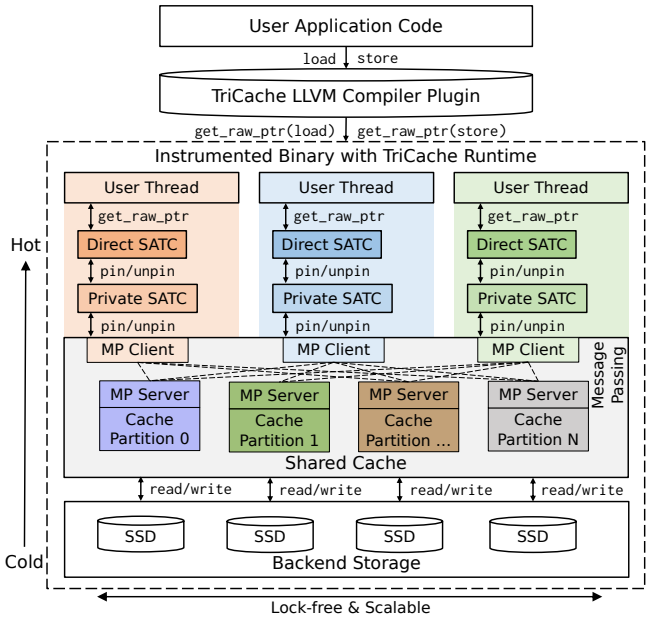


Figure 2: High-level architecture of TriCache

into use. Figure 1 illustrates this with a concrete example: calculating the length of a string. The upper part presents the implementation by using a memory interface, and the lower part shows an alternative version with a block interface. It is evident that the block version is far more complex than the memory version because system developers have to take care of more details, such as checking the block boundaries and making `pin`/`unpin` calls manually, while the memory version only needs to perform memory accesses.

It thus motivates us to explore a block cache providing a virtual memory interface in the user space, that can combine the advantages of high out-of-core performance and high user transparency from both types of caches. The user-space approach drops some functional capabilities of the OS page cache, such as sharing memory across processes with consistency guarantees. However, it allows us to redesign the cache sub-system towards new high-performance storage. Although the virtual memory interface forces applications to manipulate the cache synchronously and manage data in fixed-size blocks (rather than objects or rows), such an interface enables user transparency and saves developers considerable effort.

However, a user-space block cache with a virtual memory interface is not as easy as it might appear. Since every memory access now needs to involve a pair of `pin` and `unpin` calls to ensure that the data accessed reside in memory, as well as given that `pin` and `unpin` imply storage-to-memory address translation and concurrency control operations[1], we need optimizations in addition to those in current block cache designs to make `pin`/`unpin` as fast as possible.

[1] In case of cache misses, the victim blocks resident in memory need to be replaced with the requested blocks on storage; in case of cache hits, the reference counts need to be updated with locks/latches or atomic operations.

3 Design and Implementation of TriCache

In this section, we first present an overview of the system design of TriCache, and then describe its efficient multi-level block cache runtime in a bottom-up manner, including how to build a scalable block cache and reduce the cost of cache accesses in the user space to support transparent usage. Finally, we introduce how to automatically apply TriCache to in-memory applications via compiler techniques.

3.1 Overview of TriCache

Figure 2 shows the high-level architecture of TriCache. It consists of an LLVM compiler plugin and a runtime module.

TriCache LLVM Compiler Plugin first instruments each memory instruction, such as load and store, in the user application code, inserting a software address translation call (named `get_raw_ptr`) before the memory instructions. Upon execution, the instrumented binary calls the interface every time it tries accessing a storage address and retrieves a memory address pointing to data cached in memory. The translated address is then used as usual for the memory instruction.

TriCache Runtime is the core of TriCache (the dashed box in Figure 2). It is a multi-level block cache that supports fast address translation and provides a virtual memory interface. It implements `get_raw_ptr` to translate blocks to their corresponding cached memory addresses, manages the in-memory data cache for recently accessed blocks, handles I/O operations when the cache misses, and evicts blocks when the cache is full.

In the implementation of `get_raw_ptr`, TriCache Runtime introduces a two-level *Software Address Translation Cache*

(SATC) on top of the conventional block cache (*Shared Cache* in Figure 2). The first level is a directly mapped Direct SATC, and the second level is a set-associative Private SATC (under the three *User Threads* in Figure 2). They serve purposes similar to those of the hardware TLB, and accelerate address translations for hot blocks. We implement them as thread-local metadata caches for storage-to-memory address mappings. Direct SATC is responsible for efficient translation when operating the most recently used entries, while Private SATC aims to provide sufficient entry caching capacity and merge inter-thread operations. Meanwhile, the SATC employs a `pin`/`unpin` protocol (as mentioned in Section 2) to implement an inclusive two-level metadata cache. TriCache deploys SATC to automatically exploit localities in running programs for address translation and reduce the cost of runtime API calls, rather than relying on manually programming against blocks to reduce the number of API calls and amortize the runtime overheads.

Below SATC, TriCache Runtime manages data with Shared Cache (in the middle of Figure 2, gray background). Shared Cache is a full-featured block cache shared by multiple threads that maintains an in-memory cache pool for reading and writing the underlying storage. It manages a block table for all in-memory blocks and serves address translations when SATC misses. The block table exposes a `pin`/`unpin` interface to SATC as well, with the guarantee that recently used data pinned by SATC are not swapped out to external storage. To prevent scaling bottlenecks introduced by locking, the block space is partitioned, and each partition is owned by a single thread. Message passing based delegation is used to render critical operations (including block replacements and I/O accesses) single-threaded and lock-free. Moreover, Shared Cache can use kernel-bypass I/O stacks to eliminate context switching for I/O operations.

For Shared Cache, we propose a Hybrid Lock-free Delegation based concurrency control scheme. First, we distinguish between address translations and data accesses. Only address translations call `pin`/`unpin` remotely through message passing, while data accesses directly manipulate memory and rely on the CPU cache to ensure data consistency. The cached data are thus stored only in the Shared Cache and directly accessed by threads without any redundant memory copies. Second, we design and implement the per-partition block table as a concurrent lock-free hash table to further reduce inter-thread message passes. With this concurrent block table, only pinning operations that are missed in Shared Cache require a synchronous remote call.

In Figure 3, we present an example of a user program, a follow-up of Figure 1, instrumented by and then running with TriCache. The C program is first compiled to LLVM IR (Intermediate Representation) with Clang, with the memory read compiled to a load instruction. TriCache LLVM Compiler Plugin instruments the load instruction into two operations: one calls `get_raw_ptr` to retrieve the translated memory address, and the other loads the cached data. Upon execution, the `get_raw_ptr` call of TriCache Runtime results in an address translation operation sequence. If Direct SATC hits, the result is returned; otherwise, it pins the corresponding block in Private SATC. If the block is found in Private SATC, the result is returned; otherwise, it pins the corresponding block in Shared Cache. If Shared Cache is holding the block, the pin operation finds the memory address of the cached block in the concurrent block table. Otherwise, as invoked by a remote call, Shared Cache reads the block from storage and loads it into memory.

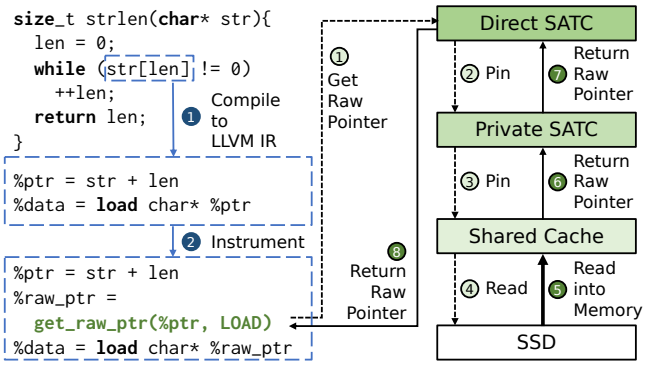


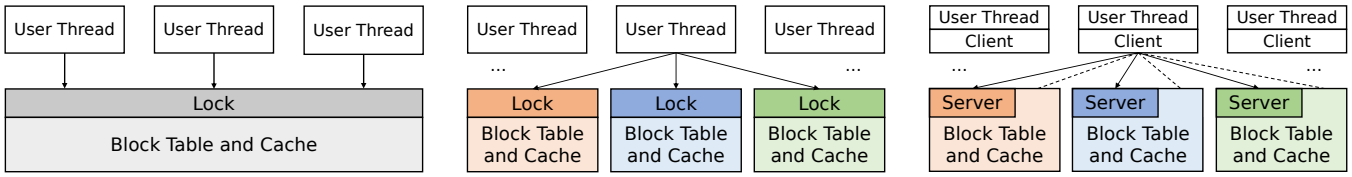
Figure 3: An example of a user program running on TriCache

3.2 Shared Cache

As the core module of TriCache, Shared Cache determines TriCache's throughput, especially its I/O performance. Therefore, good scalability is the primary design goal of Shared Cache for the effective use of hundreds of CPU cores, tens of NVMe SSDs, and millions of IOPS.

Design Decisions. Figure 4a shows a straightforward design used by the current Linux Kernel. It uses a global lock to protect the block table (or page table) and the cache. However, the single lock leads to heavy lock contention and is difficult to scale for high-performance storage devices [29]. The sharding technique can help mitigate the scalability issue, as shown in Figure 4b. The block cache [12, 29] can use a predefined function (usually hashing) to partition the blocks into several shards and then assign a lock to each shard. In addition, recent work proposes that well-designed delegation based on message passing can provide better scalability and hotspot tolerance than locks [23, 33, 53] on NUMA (Non-uniform Memory Access) architectures.

Therefore, we propose a Hybrid Lock-free Delegation for Shared Cache of TriCache, as shown in Figure 4c. The Shared Cache adopts a client-server model based on message passing (solid lines in Figure 4c). Each client-server pair shares a lightweight message queue with a size of two cache lines, similar to ffwd [33]. Each user thread corresponds to a client, and several dedicated servers handle requests from clients.



(a) Global Locking (b) Sharded Locking (c) Hybrid Lock-free Delegation

Figure 4: Different designs and concurrent mechanisms for the shared block cache

Each server is single-threaded, lock-free, and only responsible for managing a part of the blocks (e.g., partitioned by hashing block IDs). Multiple partitions and servers can achieve concurrency and scalability, and more servers can be added when a higher throughput is desired. In addition to message passing based delegation, clients can directly access per-partition block tables on cache hits to reduce server-side CPU consumptions (dashed lines in Figure 4c; more details are provided in the *Client-side Fast Paths on Cache Hits* paragraph below).

Metadata-only Delegation. When a user thread accesses block data, TriCache divides the block access into a metadata operation and a data operation. Metadata operations include address translations, reference count management, and evict policy enforcement. Data operations are memory accesses, such as load and store. In TriCache, only metadata operations are processed by servers through delegation while clients issue data operations by themselves.

A block is accessed in three stages. In the first stage, the client asks the server to cache the block in memory (`pin`) and translate the storage address to its address in memory. The server updates the metadata of the requested block, reads uncached blocks, and evicts unused blocks, without touching the actual data. In the second stage, after receiving the response, the client will directly perform its memory access on the translated memory address. In the last step, the client notifies the server that the block has been released and can be further evicted by the server (`unpin`).

This design eliminates redundant memory copies between servers and clients. Servers focus on metadata operations so that a few server threads can achieve good performance. Meanwhile, it helps TriCache provide the same consistency and atomicity guarantees as memory, which is necessary for user transparency and compatible with in-memory applications. CPU directly executes data operations on the client side via memory instructions, and cache coherence is ensured by hardware. TriCache only needs a memory fence to ensure that the updates are visible before evicting modified blocks.

Client-side Fast Paths on Cache Hits. We propose using concurrent block tables to avoid server-side synchronizations on cache hits. A client first tries to directly find a block in the block table and update the block reference counts (number of clients in use) by using atomic operations. If it succeeds,

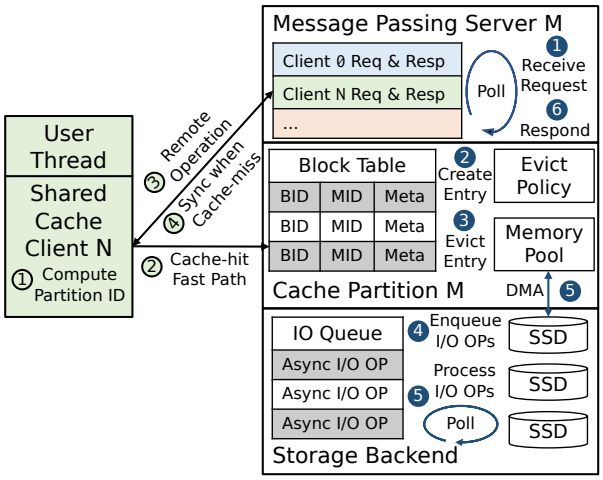


Figure 5: Shared Cache of TriCache

the client can translate the address from the concurrent block table by itself, thus skipping synchronous message passing. The client then sends an asynchronous message if this direct operation changes the reference count from 0 to 1 or conversely, to notify the server to update the evict policy for the block. Multiple asynchronous messages can be batched and processed together to amortize message passing overheads.

Workflow and Implementation. Figure 5 shows the workflow and implementation of Shared Cache. Each user thread corresponds to a Shared Cache Client and gets its unique Client ID. Each partition has a polling-based message passing server that is used to process requests sent from clients and return results to them. In addition, each partition maintains a concurrent block table for blocks cached in the memory, and each entry in the block table stores the Block ID (BID) of the block, Memory ID (MID) of its in-memory cache, and its metadata (Meta). The metadata include information on whether the block is available and whether it has been modified, and the reference count pertaining to the clients in use. We use the compact hashed block table similar to [51], in which each entry occupies only an average of 8 bytes, and uncached blocks do not occupy memory.

The cached blocks are indexed by their Memory IDs and stored in a Memory Pool. Meanwhile, an Evict Policy tracks all cached blocks with a zero reference count as they can be

safely evicted. Every time the cache is full, the Evict Policy chooses and evicts one or more blocks based on its statistics and strategies. TriCache uses the CLOCK algorithm [40] by default, and it is replaceable, allowing users to customize the policy based on their application characteristics. In addition, policy implementations in TriCache are completely single-threaded, so users do not need to consider any concurrency issues. At the bottom, an asynchronous I/O backend (Storage Backend) manages pending IO requests in an I/O Queue to continuously poll and process I/O operations. The I/O backend is also customizable and defaults to SPDK that is backed by user-space NVMe drivers. A kernel-space alternative based on Linux AIO is also supported as another candidate.

When a user thread operates on a block, its client (*N* in the figure) first computes the Partition ID (*M* in the figure) by a predefined partition function. The client then searches for a valid block entry from the block table of Partition *M*, and if such an entry exists, the client tries increasing the reference count by using atomic operations. If the atomic operations succeed on cache hits (*Cache-hit Fast Path* in Figure 5), the client pins the block in memory and can directly query the memory address of the cached block. The client may further send an asynchronous request to the server if it is updating the reference count from 0 to 1, or the converse. The server then performs the corresponding actions according to the Evict Policy, such as enabling or disabling evictions of the block.

If the atomic operations fail on cache misses or when the block is being swapped in/out, the client requests a remote operation via synchronous message passing, immediately releases CPU resources, and waits for responses from the server (*Remote Operation* in Figure 5). After receiving the request, the server creates a block table entry and sets its valid bit to `false`. If the block table is full, the server evicts blocks according to the Evict Policy and sets their valid bit to `false`. The server then appends I/O operations for new blocks and the evicted blocks to the I/O queue. The I/O Backend processes the I/O requests by polling and controlling NVMe SSDs to perform DMA operations directly on the Memory Pool. And the server sets valid bits to `true` once the I/O requests have been processed, and it sends the memory addresses of the blocks to clients via message passing. After receiving the response, the client resumes and performs its memory accesses.

We use a micro-benchmark on a 128-core machine to test the effectiveness of TriCache Shared Cache. It can scale linearly to 256 threads (1/8 of the threads are servers), reaching 96.8M ops/s, and the hybrid mechanism provides an improvement of 52% compared with the delegation-only approach.

3.3 Software Address Translation Cache

The Shared Cache of TriCache provides scalable I/O performance and an efficient set-associative cache. However, block table lookups and atomic operations are required for each access on cache hits, still limiting the performance of TriCache.

Guiding Ideas. Considering the manual use of the block cache (e.g., Figure 1), users call the `pin` interface to get the in-memory address for a block, and then use the memory address to perform multiple operations; they finally call the `unpin` interface to release the block. Multiple read and write operations can be performed between a pair of manual `pin` and `unpin` operations to reduce the number of cache lookup operations. Users manually take advantage of data locality while investing extra effort in development.

In contrast, we design TriCache to automatically exploit locality to simulate manual coding without requiring human effort. We propose to build a two-level Software Address Translation Cache (SATC) on top of Shared Cache. The higher-level cache stores hotter data and provides faster access and smaller capacities than the lower-level cache, similar to the multi-level cache of the CPU and hierarchical storage [32, 49, 56]. Based on this idea, we now show how to implement the multi-level cache in software and where to divide the levels.

SATC Design. In our design, only the last-level cache manages data, and higher-level translation caches manage only metadata, such as modifying the reference counts of the blocks and translating block IDs to memory addresses. Managing metadata instead of data can help avoid redundant memory consumption, additional memory copies, and memory consistency issues caused by the multi-level design. The multi-level cache of TriCache is designed to be an inclusive cache, which means that all blocks in the higher-level cache are also present in the lower-level cache. With this inclusive policy, higher-level caches need to only interact with their next level. Moreover, TriCache guarantees that the capacity of higher-level caches is no greater than the lower-level cache, thus eliminating out-of-space errors from the lower-level cache when the higher-level cache requests to swap in blocks.

On top of the Shared Cache, we build a thread-local set-associative cache called Private SATC. When the Private SATC hits, the user thread uses its thread-local block table and evict policy, and only when the Private SATC swaps in/out blocks does the user thread need to operate on the Shared Cache. Private SATC is purposed to reduce Shared Cache operations for the hot data of its thread. Examples include thread-local hot data when each thread computes a segment of data independently, and hot elements shared by all threads when processing skewed data. Private SATC also helps reduce concurrent block table operations with cross-NUMA memory accesses and false sharing, which could take about 3–8 times higher latency than local memory accesses in our evaluation.

We further build a direct mapping cache called Direct SATC on top of the Private SATC to alleviate overheads due to hash table lookups and evict policy maintenance. Direct SATC maintains a few recently accessed pages in a fixed-size array to speed up address translation to a few bitwise operations and avoid having to update the evict policy for each access. The goal of Direct SATC is to cover multiple consecutive

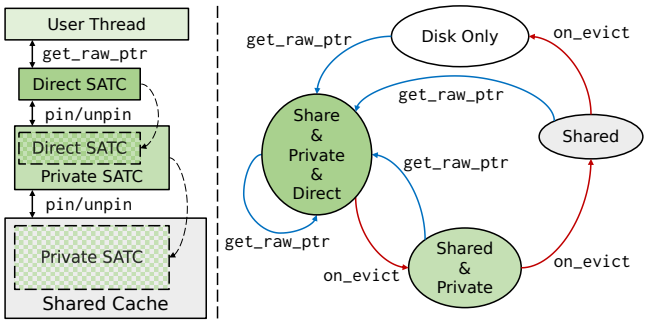


Figure 6: Software Address Translation Cache

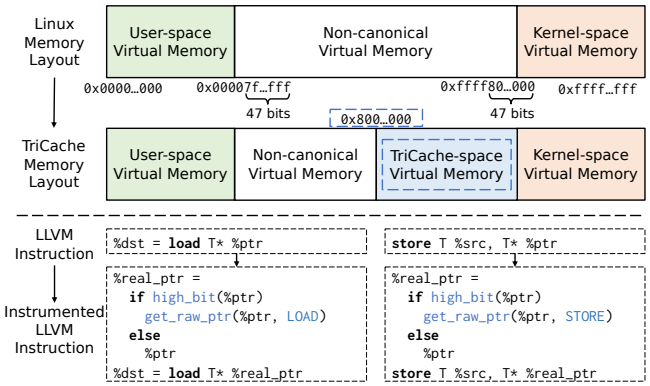


Figure 7: Memory layout and LLVM instrumentation

operations on the hottest blocks, such as sequential reads and writes, and displace manually written `pin`/`unpin` operations.

Implementation. As shown in Figure 6, we implement an inclusive multi-level cache with `pin` and `unpin` interfaces. Blocks in Direct SATC must exist in Private SATC, and blocks in Private SATC should also be present in Shared Cache. Private SATC calls the `pin` interface of Shared Cache to load blocks into the Private SATC, thus increasing reference counts to ensure that Shared Cache does not swap out the blocks. When blocks are evicted from Private SATC, it calls the `unpin` interface of Shared Cache to release the reference count. On top of Private Cache, Direct SATC also uses a scheme similar to Private SATC but provides a single `get_raw_ptr` interface implicitly combining a `pin` call and a following `unpin` call. It implies caching a block and translating the block ID into its raw address in memory. The raw address is valid until the next `get_raw_ptr` call because the subsequent access can evict any previous block from Direct SATC and possibly call the `unpin` interface of Private SATC or Shared Cache.

The right-hand side of Figure 6 presents the state machine maintained in TriCache. Starting from Disk Only state, the user thread loads blocks into Shared Cache, Private SATC, and Direct SATC by calling `get_raw_ptr`. When Direct SATC evicts blocks, Shared Cache and Private SATC still hold them. When the last thread in use evicts a block from its Private SATC, the block enters Shared Cache Only state. Any `get_raw_ptr` re-loads the block into all three levels of caches. If Shared Cache also evicts the block, it is removed from the in-memory cache and written back to storage when it is dirty, ending in Disk Only state.

In our implementation, the aggregated capacity of Private SATC entries is equal to that of Shared Cache entries, and the Direct SATC has a size 1/4 of that of the Private SATC.

Our evaluation shows that SATC can improve performance by tens of times over Shared Cache on real-world workloads. When SATC can absorb all accesses, TriCache can reach 57% and 91% of the in-memory performance for purely random and nearly sequential access patterns respectively, making it practical to operate the block cache at per-access granularity.

3.4 Compile-time Instrumentation

With the help of SATC, TriCache opens up opportunities to provide a virtual memory interface and make TriCache fully transparent to users. To this end, we propose a purely user-space scheme based on compile-time instrumentation and library hooking techniques.

Memory Layout. We first modify the memory layout as shown in the upper part of Figure 7. In Linux, the current x86_64 virtual memory layout (with four-level page tables) consists of three main parts. User-space takes 47 bits at the beginning, kernel-space occupies 47 bits at the end, and most of the space in the middle is a hole of non-canonical virtual memory. We map the TriCache-managed disks into unused holes by block size, starting from 0x800...000 as TriCache-space virtual memory. Memory addresses in TriCache-space can be translated into an actual user-space memory address by calling the `get_raw_ptr` interface of TriCache Direct SATC.

Instrumentation. To enable transparent read and write operations on top of the TriCache block cache, we perform a translation before each memory operation (e.g. load, store, and atomic operations), so that TriCache-space addresses can be used just like user-space memory. TriCache does not require any manual code modifications with the help of compile-time instrumentation. The lower part of Figure 7 shows pseudo-codes for instrumenting load and store instructions in LLVM IR. TriCache instrumentation takes the highest bit of addresses to determine whether a memory address refers to user-space memory or TriCache-space virtual memory.

While instrumentation provides the virtual memory interface for TriCache-space memory, we still need to determine what data should be placed in TriCache-space memory. First, data on the stack is not necessary to enter the block cache. We perform a data flow analysis from LLVM `alloca` instructions to eliminate unnecessary instrumentation and overhead for the stack. Second, we set a runtime threshold for TriCache so that only memory allocations greater than the threshold will

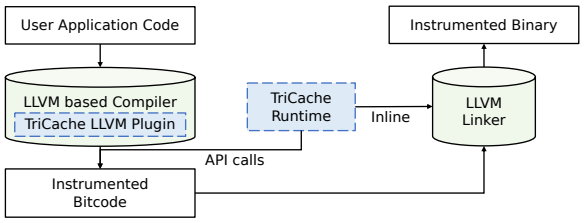


Figure 8: Compiling workflow of TriCache

belong to TriCache-space memory. In contrast, small chunks of data, usually short-term temporary data, remain in-memory allocation. Finally, TriCache supports limiting the total size of data allocated in memory by a predetermined memory quota. If in-memory data exceed the memory quota, TriCache is able to take over later allocations. Users can adjust the above runtime parameters to obtain a balanced trade-off between the memory usage and the performance.

Implementation. Figure 8 illustrates the compiling workflow of TriCache. The user application code is compiled by a compiler based on LLVM. The TriCache LLVM plugin instruments the code and generates instrumented LLVM IR bitcode. The plugin performs instrumentation after all the optimization passes, so it does not affect the compiler optimizations on applications, such as automatic vectorization. Also, TriCache supports vector instructions since TriCache leaves the CPU to perform memory operations.

Then, the bitcode links with the pre-compiled TriCache runtime (including `get_raw_ptr` implementations). TriCache forces to inline the cache-hitting implementations of Direct SATC and Private SATC through link-time optimization (LTO) to avoid intensive function call overheads.

The TriCache runtime also contains APIs on top of the virtual memory interface for manual optimizations, including `pin` and `unpin`. Optionally, users can optimize some bottlenecks of their applications through these APIs, such as using block-wise accesses and prefetching, while leaving other parts to transparently support out-of-core processing by instrumentation. In the TriCache runtime, some common utility functions, such as `memcpy` and `memset`, are already manually implemented by block-wise `pin` and `unpin` to reduce overheads of per-byte address translation from common components.

4 Evaluation

We set up our experiments on a dual-socket server equipped with two AMD EPYC 7742 CPUs (64 physical cores and 128 hyper-threads per CPU) and 512GB DDR4-3200 main memory. The storage devices are 8 PCIe-attached Intel P4618 DC SSDs which provide 51.2TB capacity, 9.6M 4KB-read IOPS, and 3.9M 4KB-write IOPS in total. The server runs Debian 11.1 with Linux kernel 5.10 and uses Clang 13.0.1 to compile TriCache and other systems.

In our evaluation, we limit the total available capacity of DRAM by `cgroups` to evaluate out-of-core performance. For TriCache and other systems with block caches, we ensure that the overall memory is less than the expected memory limit by adjusting the cache sizes. The SSDs are configured in SPDK mode for TriCache and as raw blocks for swapping. If the system requires a single filesystem, we construct a software RAID-0 by `mdadm` and use the XFS filesystem. TriCache launches 16 background threads (bound to 8 cores) for Shared Cache and uses 4KB blocks by default. And the total number of threads are searched to maximize performance over powers of two from the number of hardware threads (i.e. 256).

We first evaluate TriCache on four representative domains in terms of end-to-end performance: graph processing (Section 4.1), key-value store (Section 4.2), big-data analytics (Section 4.3), and transactional graph database (Section 4.4).

We then conduct a micro-benchmark, by using a configurable number of threads that issue load/store instructions. We adjust the hit rates and access patterns to explore circumstances in which TriCache outperforms OS page cache and to assess whether the design of TriCache provides a reasonable trade-off between in-memory (i.e. cache hit) and out-of-core (i.e. cache miss) performance (Section 4.5).

Finally, we use a series of breakdown experiments to evaluate the performance-related impact on TriCache, including I/O backends, hit rates and hit latency of SATC, and number of threads (Section 4.6).

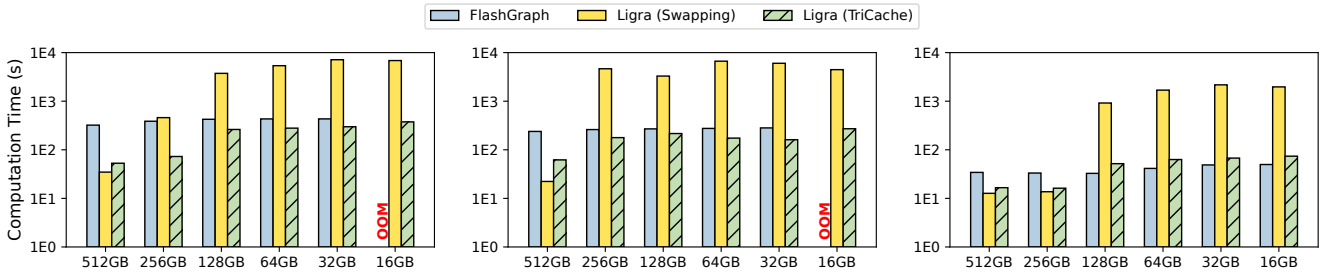
4.1 Performance on Graph Processing

Experimental Setup. Graph processing is a demanding workload for cache systems due to many small and random accesses on large datasets. We transparently apply TriCache to an in-memory graph processing framework Ligra[2] [37] and extend it to out-of-core. The baselines are Ligra with OS swapping and FlashGraph[3] [54], an efficient semi-external memory graph processing framework designed for SSDs.

Both Ligra and FlashGraph use 32-bit vertex IDs, and we force Ligra to use push mode to align with FlashGraph. For FlashGraph, we follow its recommended configuration of creating an XFS filesystem for each SSD block device and binding the device to the corresponding NUMA nodes. Meanwhile, FlashGraph is a semi-external memory graph engine that always stores vertex states in memory and edge lists on SSDs. We thus make TriCache to manage at least edge lists in the cache for a fair comparison.

We evaluate FlashGraph, Ligra on swapping, and Ligra on TriCache by three common graph algorithms: PageRank (PR), Weakly Connected Components (WCC), and Breadth-First Search (BFS). The dataset is a real-world graph dataset, UK-2014 [7, 8], with 788 million vertices and 47.6 billion edges. It requires more than 400GB for Ligra in-memory execution.

[2] https://github.com/jshun/ligra [commit 7755d95]
[3] https://github.com/flashxio/FlashX [commit 2a649ff]



(a) PageRank (b) Weakly Connected Components (c) Breadth-First Search

Figure 9: Computation time of FlashGraph, Ligra on swapping, and Ligra on TriCache (lower is better)

In-memory Performance. Figure 9 shows the computation time of FlashGraph, Ligra on swapping, and Ligra on TriCache under different memory quotas. With 512GB of memory, Ligra can process all three algorithms in memory, and TriCache and FlashGraph can buffer all data in their cache. Under this setting, TriCache incurs overheads of only 34.4% for PageRank, 64.0% for WCC, and 23.5% for BFS. The in-memory performance shows that TriCache can provide efficient address translations and cache hits with its virtual memory interface, owing to the two-level SATC. Meanwhile, TriCache outperforms FlashGraph by 6.08×, 3.85×, and 2.06×, respectively. It illustrates that FlashGraph yields much higher in-memory overheads than TriCache because the block cache of FlashGraph involves redundant memory copies on cache operations with its `read`/`write` interfaces.

Out-of-core Performance. Under 256GB memory limitation, the caches start swapping in/out blocks/pages. Compared to the in-memory performance, Ligra on TriCache saves about half of memory and yields 47.7% performance on PageRank, 12.5% performance on WCC, and 78.4% performance on BFS. And TriCache's speedups over OS swapping and FlashGraph are 6.30× and 5.31× on PageRank, 26.1× and 1.46× on WCC, and 0.85× and 2.05× on BFS, respectively.

As the usable memory further decreases, I/O efficiency becomes the main factor affecting performance. For example, in the case of 64GB of memory, the performance of TriCache is 19.3×, 38.3×, and 26.8× better than that of swapping. Compared with FlashGraph, TriCache can still provide improvements of 54.8% and 58.3% on PageRank and WCC respectively, while the performance of Ligra with TriCache is 34.3% lower than FlashGraph on the BFS algorithm. This is because FlashGraph adopts two-dimension partition for out-of-core graph processing, resulting in a 50.1% cache hit rate that saves 2.68× of I/O volume compared to TriCache. Still, TriCache provides an average I/O bandwidth 1.78× better than FlashGraph and thus reduces the performance gap.

It is noteworthy that the semi-external memory FlashGraph cannot fit vertex states of PageRank and WCC with 16GB of memory. It leads to out-of-memory errors, whereas TriCache can operate the same dataset fully out-of-core.

The above results indicate that TriCache can extend an in-memory graph framework to support out-of-core processing without manual modification and can deliver performance comparable to a well-designed external memory framework. Meanwhile, TriCache outperforms OS swapping by up to 38.3× while providing the same user transparency.

4.2 Performance on Key-Value Stores

Experimental Setup. We use RocksDB[4] [12], a persistent key-value store widely used in production systems, for evaluation in this part. RocksDB organizes on-disk data in immutable Sorted Sequence Tables (SSTs). It provides a block-based table format on top of its user-space block cache, and a plain table format optimized for in-memory performance via `mmap`. We use TriCache to buffer RocksDB plain tables without manual modification and compare it with plain tables based on OS memory-mapped files and block-based tables on RocksDB's own cache.

We use the mixgraph [10] (prefix-dist) workload proposed by Facebook, which models production use cases at Facebook and emulates real-world workloads of key-value stores with hotness distribution and temporal patterns. The keys and values are 48 and 43 bytes on average, respectively, and there are 83% reads, 14% writes, and 3% scans. We generate 2 billion key-value pairs (consuming 180GB of space) and execute 100 million operations. Both plain and block-based tables use the hash index with a 4 bytes prefix. We set the sharding number of the RocksDB block cache to 1024 to avoid lock contentions on our 256-thread server and use the direct I/O mode for the RocksDB block cache. We also disable WAL to prevent log flushing from becoming a performance bottleneck.

In-memory Performance. Figure 10 illustrates the throughput of Plain Tables on TriCache, Plain Tables on `mmap` and Block-based Tables on the RocksDB user-space cache with different memory quotas. In memory, RocksDB Plain Tables with `mmap` provides the best performance, which is 4.28M ops/s. TriCache reaches about 53.5% throughput of `mmap`, and 73.7% throughput of the RocksDB block cache.

[4] https://github.com/facebook/rocksdb [tag v6.26.1]

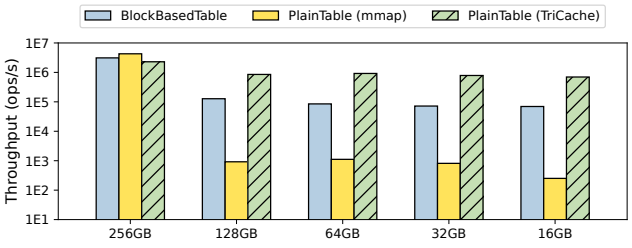


Figure 10: RocksDB throughput with varying memory quotas

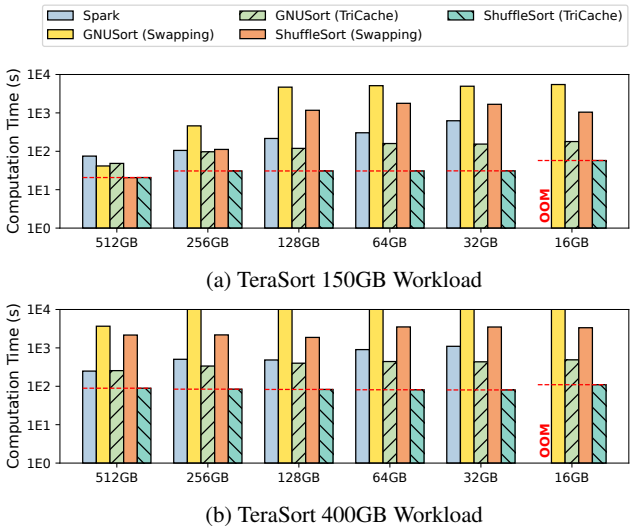
Out-of-core Performance. When RocksDB runs out-of-core, TriCache brings performance improvements of 2–3 orders of magnitude compared with `mmap`. Plain Tables with TriCache outperforms Block-based Tables by 6.69× under 128GB memory, 10.8× with 64GB, 10.9× with 32GB, and 10.0× with 16GB. Some performance benefits of TriCache come from the efficient I/O stack of SPDK, while the excellent scalability of Shared Cache is another key factor. For example, the RocksDB block cache can deliver a throughput of 122K ops/s with 256 threads. However, our eight NVMe SSDs require about 1024 I/O in-flight requests to maximize the I/O performance. Unfortunately, when the number of threads is increased from 256 to 1024, the throughput instead gradually drops. In the case of 1024 threads, RocksDB only provides 71.3% throughput of 256 threads. In contrast, the performance of RocksDB with TriCache improves by 2.15× from 256 threads to 1024 threads.

The in-memory performance indicates that user-transparent TriCache can provide similar performance as manually managed block cache in RocksDB. Meanwhile, TriCache has the potential to help existing systems with in-memory backends, such as RocksDB with Plain Tables, to achieve better out-of-core performance without any manual modifications.

4.3 Performance on Big-Data Analytics

Experimental Setup. TeraSort [1] is a representative application and an important performance indicator in the domain of big-data analytics [16]. Its typical distributed or out-of-core implementation consists of a shuffle phase followed by a sort phase. The shuffle phase produces parallel sequential reads and writes, which is I/O bound [16] and can stress sequential I/O throughput on cache systems. The sort phase requires the cache to buffer the working partition in memory and issues a vast number of string comparisons and copies that can examine the runtime overhead of cache systems.

We generate two TeraSort workloads, 1.5B records (about 150GB) and 4B records (about 400GB). For TriCache, we first use the parallel sort based on multi-way merge sort in GNU `libstdc++` [38] (named GNU Sort) to implement an out-of-core TeraSort, which requires only a single function call. We also implement a shuffle-based parallel sort by partitioning the first byte of the keys (named Shuffle Sort), which takes 15 additional lines of C++ code. Compared with the multi-way merge sort, the shuffle-based parallel sort mainly issues sequential read/write I/O operations, so it is more friendly to out-of-core processing. We use TriCache to manage memory allocations during sorting and compare TriCache with OS swapping. For Shuffle Sort, we configure the page size of TriCache to 128KB to maximize the sequential I/O performance. We also use a widely used big-data framework Spark[5] [52] as a baseline, which supports both scale-up and scale-out processing.



(a) TeraSort 150GB Workload

(b) TeraSort 400GB Workload

Figure 11: Computation time for TeraSort workloads with different memory quotas (lower is better)

In-memory Performance. Figure 11 shows the computation time of TeraSort. On the 150GB dataset, both GNU Sort and Shuffle Sort occupy about 300GB of memory and fit in 512GB of memory. In this case, Shuffle Sort is 2.01× faster than GNU Sort. Meanwhile, the overheads of TriCache amount to only 14% for GNU Sort and nearly zero (less than 1%) for Shuffle Sort. The reason is that the Shuffle Sort algorithm mainly generates sequential reads and writes for each thread, which can be well handled by thread-local Direct SATC and Private SATC. Compared with Spark, GNU Sort and Shuffle Sort on TriCache is faster by 1.55× and 3.62×, respectively.

Out-of-core Performance. When the memory quota is less than 256GB for the 150GB workload, TriCache can provide tens of times speedups over swapping, up to 39.3× for GNU Sort at 128GB memory and 57.8× for Shuffle Sort at 64GB memory. Meanwhile, the performance of Shuffle Sort with TriCache is up to 20.2× better than Spark at 32GB memory.

For the 400GB dataset, both algorithms keep executing out-of-core. Shuffle Sort on TriCache is faster than swapping by

[5] https://github.com/apache/spark [tag v3.2.0]

up to 43.6× at 32GB memory and outperforms Spark by up to 13.7× with the same amount of memory. GNU Sort based on swapping is 41.2× slower than TriCache Shuffle Sort with 512GB memory and about 128× slower when the memory quota is less than 256GB because of its sub-optimized algorithm and the limited performance of the OS page cache.

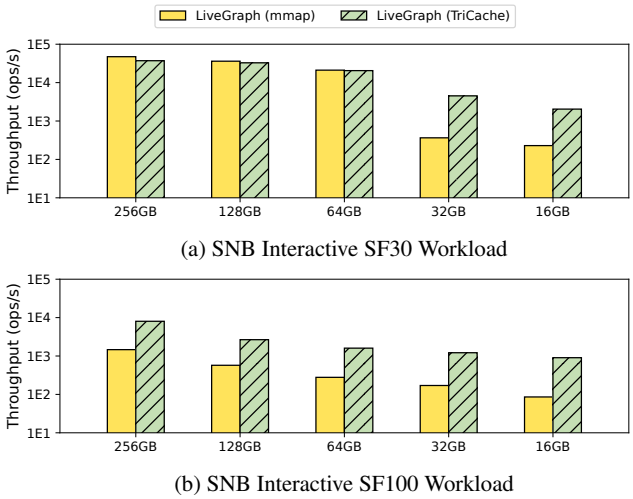
Compared with the in-memory processing of the 150GB dataset, Shuffle Sort with TriCache saves 90% memory with 32GB memory, while its processing time is only 49.3% longer than the processing time at 512GB. We also compared the distributed Spark and TriCache-based scale-up solutions. We use four servers with the same hardware configuration and connect them with 200Gb HDR Infiniband NIC. TriCache under 32GB memory outperforms in-memory distributed Spark by 7.20× using Shuffle Sort and 1.33× with GNU Sort on the 400GB workload. So TriCache with NVMe arrays can use less memory and provide nearly in-memory performance for TeraSort. In addition, TriCache can nearly utilize the peak bandwidth of our 8 NVMe SSDs, reaching 44GB/s for read-only operations and 31GB/s for mixed read/write operations.

In summary, developers can write in-memory programs (e.g., less than 20 lines of C++ code for Shuffle Sort), and TriCache then helps them to fully utilize the high-performance NVMe SSD array, especially when the algorithm is friendly to out-of-core processing.

4.4 Performance on Graph Database

For workloads in graph databases, we evaluate TriCache on LiveGraph[6] [57], an efficient transactional graph database based on OS memory-mapped files. LiveGraph treats memory-mapped files as in-memory data and relies on atomic memory accesses and cache consistency to support transactional queries. It can examine whether a user-transparent block cache is able to provide the same semantics as in-memory operations. We replace the memory-mapped files with TriCache and compare it with the original LiveGraph. We evaluate their performance on the LDBC SNB interactive benchmark, which simulates user activities in a social network and consists of 14 complex-read queries, 7 short-read queries, and 8 update queries. As the SNB driver occupies part of the memory, we limit LiveGraph to use up to 256GB memory and generate two workloads: SF30 and SF100 datasets. With LiveGraph, these datasets take about 100GB and 320GB memory, respectively. SNB clients request 1.28M operations for the SF30 workload, and 256K operations for the SF100 workload during the benchmark run.

Figure 12 shows the SNB throughputs of LiveGraph on TriCache and `mmap`. When the dataset can fit into a 256GB memory, the instrumentation and user-space cache of TriCache incur only 21% runtime overheads on the SNB benchmark. As the memory quota gradually decreases, the advantage of TriCache becomes increasingly prominent, e.g., TriCache outperforms `mmap` by 12.4× at 32GB memory as its scalable Shared Cache and the efficient I/O backend supply much higher throughputs. For SF100, LiveGraph keeps running in out-of-core states. TriCache improves the throughput by 5.48× compared with `mmap` at 256GB memory, and the speedup can grow up to 10.5× at 16GB memory.



(a) SNB Interactive SF30 Workload

(b) SNB Interactive SF100 Workload

Figure 12: Throughput of LiveGraph on TriCache and `mmap`

We then take a closer look at the latency metrics when running SF100 with 256GB of memory. TriCache cuts the average latency on complex queries by 11.5×, on short queries by 1.79×, and on update queries by 21.1× (geometric means). The P999 tail latency of TriCache keeps 10.9× lower than `mmap` on complex queries and 1.35× lower on short queries. Meanwhile, TriCache shortens the P999 latency of update queries to 34.6× shorter than the original LiveGraph because TriCache is additionally aware of thread locality while `mmap` is not. Although TriCache and `mmap` are both user-transparent, the Private SATC of TriCache can automatically hold recently updated data for writer threads in memory even when writers are waiting for group commits. On the contrary, `mmap` may evict these dirty pages under memory pressure. Our design helps LiveGraph to reduce tail latencies on update operations.

4.5 Micro-benchmarks

We conduct two custom multi-threaded micro-benchmarks which issue random memory-load instructions. The first generates random accesses in 8-bytes (named 8B Random workload), which can stress the systems in the case of completely random memory accesses. We control the random pattern to generate operations with different hit rates of block caches, and we also adjust the hit rate of Private SATC to examine its performance impact. The second randomly chooses 4KB pages and sequentially accesses each page in 8-byte words (named 4KB Random workload) to evaluate the performance when a page is accessed multiple times.

[6] https://github.com/thu-pacman/LiveGraph [commit eea5a40]

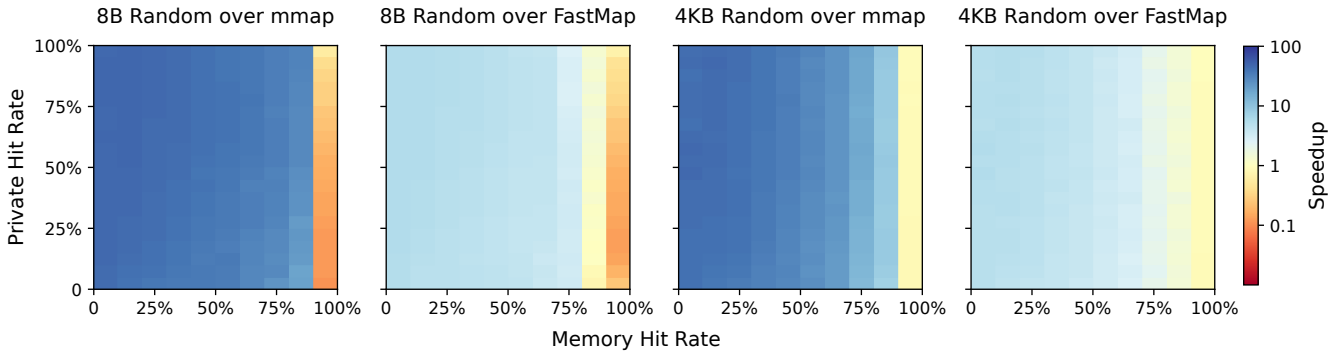


Figure 13: The speedup of TriCache compared with mmap and FastMap on 8B Random and 4KB Random workloads

We compare TriCache with Linux mmap and FastMap [29] (both given a hint of random accesses, MADV_RANDOM). FastMap optimizes the mmap path in the Linux kernel, including sharding locks (discussed in Section 3.2) and batching TLB invalidations. It mainly aim to mitigate the scalability limitation of Linux mmap. For FastMap, we downgrade the kernel version to 4.14 as FastMap relies on it, and configure a RAID-0 with mdadm as suggested by the authors of FastMap, and leverage FastMap to manage bare block devices.

Figure 13 shows TriCache's speedup compared with mmap and FastMap with 8B Random and 4KB Random workloads.

For 8B Random workloads, the performance of TriCache is about 11% of the in-memory (with mmap) performance in the worst case, when the memory hit rate is 100% and Private SATC hardly hits. Under the same 100% memory hit rate, when the hit rate of Private SATC grows up to 100%, TriCache can attain 57% of the in-memory (with mmap) performance, with a performance improvement of up to 6.07×.

Once the memory hit rate drops to 90%, TriCache can provide improvements of 18.6× to 31.5× over mmap whose performance is severely limited by lock contentions in the kernel. At the same time, TriCache outperforms FastMap by 1.22× on average. As the memory hit rate gradually decreases, the advantage of TriCache becomes increasingly significant. For instance, TriCache outperforms mmap by 33.6× and FastMap by 3.34× with an 80% hit rate. When the memory hit rate reaches 10%, TriCache performs 45.0× to 47.2× better than mmap, and 5.38× to 5.60× better than FastMap. In this case, TriCache provides 12.4 million random accesses per second and fully saturates our 8 NVMe SSDs. However, FastMap can only support 2.22 million accesses per second with all the hardware cores, where this is equivalent to about the I/O performance of only two NVMe SSDs. This indicates that FastMap cannot accommodate currently available high-performance NVMe SSD arrays because it still suffers from the heavy I/O stack of the kernel, page faults, and context switching overheads [27, 34, 55].

For 4KB Random workloads, Direct SATC can mainly absorb the in-memory overheads of TriCache. The performance of TriCache reaches 84% to 91% of the in-memory (with mmap) performance when the memory hit rate is 100%. With a 90% memory hit rate, TriCache can provide a speedup of 8.43× on average over mmap and 1.46× over FastMap. Under a 10% memory hit rate, TriCache can provide 12.3 million random accesses per second, which outperforms mmap by 43.1× and FastMap by 5.08× on average.

Table 1: Performance slowdown relative to TriCache

	Linux AIO	W/O Direct	W/O Private	Shared Only
PageRank	1.15×	2.75×	1.03×	40.1×
RocksDB	2.16×	1.27×	1.02×	22.0×
ShuffleSort	1.69×	1.87×	4.67×	10.1×
GNUSort	2.36×	2.51×	4.25×	57.9×
LiveGraph	1.21×	1.07×	1.01×	7.55×

4.6 Performance Breakdown

We select five cases under 64GB of memory for the breakdown experiments: PageRank, RocksDB, Shuffle and GNU Sort for the 400GB Terasort dataset, and LiveGraph for the SNB SF100 workload.

SPDK and Linux AIO. TriCache currently supports SPDK and Linux AIO as its storage backend. In the default configuration, TriCache uses SPDK to handle I/O operations. We compared the performance of these two backends. The first column in Table 1 shows the performance slowdown when using the AIO backend compared with the SPDK backend.

In terms of the (geometric) average, SPDK performs 1.64× better than Linux AIO, demonstrating that the user-space NVMe driver enables better IO performance. Nevertheless, SPDK has some drawbacks, such as high programming complexity, deployment difficulties, and not easy for supporting multiple applications. Luckily, TriCache hides SPDK programming details from users, allowing users to code in-memory programs and achieve efficient out-of-core performance. Moreover, the design of TriCache is not coupled with SPDK and can provide comparable performance with Linux AIO. If using or deploying SPDK is not feasible, AIO can serve as a reasonable alternative backend for use in TriCache.

Breakdown Analysis of SATC. The three columns on the right side of Table 1 break down the performance impact of SATC on TriCache by gradually removing SATC levels. *W/O Direct* disables Direct SATC, *W/O Private* disables Private SATC, and *Shared Only* uses only Shared Cache by removing both SATC levels.

According to the performance degradation listed in Table 1, SATC is an essential component contributing to the good performance of TriCache. The slowdown that occurs by disabling SATC (*Shared Only*) is 20.8× on average for the five cases. And even when the memory quotas are less than 1/5 of the working set (i.e., running out-of-core), SATC still yields a speedup of 40.1× for PageRank, 10.1× for Shuffle Sort, and 57.9× for GNU Sort.

Meanwhile, both Direct SATC and Private SATC are indispensable to TriCache. Without Direct SATC, the performance of PageRank is degraded by 2.75× because accessing each edge incurs a heavy overhead due to hash table lookups and evict policy maintenance. However, PageRank is not sensitive to Private SATC because the size of the dataset is more than 5× larger than the available memory, and the edges are visited only once for each iteration. For the shuffle phase of Shuffle Sort, the performance drops by 5.39× without Private SATC but remains almost the same (only 4.8% slower) without Direct SATC. The reason is that string copies constitute the bottleneck in the shuffle phase and are optimized by the compiler to `memcpy`, which is implemented by manually calling `pin`/`unpin` in the TriCache runtime. For GNU Sort, removing Direct SATC and Private SATC degrades the performance by 2.51× and 4.25×, respectively.

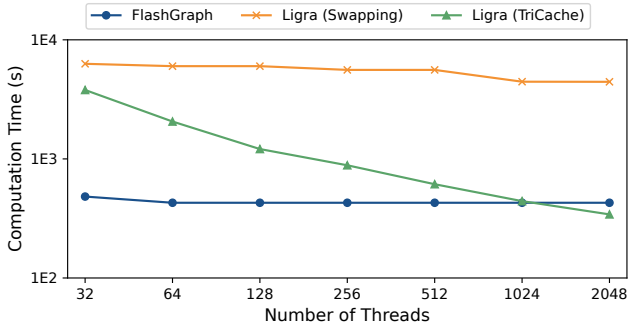
Multi-level Cache in TriCache. Next, we use PageRank, Shuffle Sort, and GNU Sort to further examine the design of the multi-level cache in TriCache. Table 2 lists the miss rates for each level of the cache, the average hit cycles (*HitC.*) for Direct SATC and Private SATC, and the average access cycles (*Acc.C.*) for Shared Cache.

According to the miss rates listed in Table 1, Direct SATC and Private SATC can handle most memory accesses. The miss rate of Direct SATC is less than 5% for all the three workloads, and the miss rate of Private SATC is less than 1% for Shuffle Sort and GNU Sort. The results show that SATC can cover most accesses to meet the above performance.

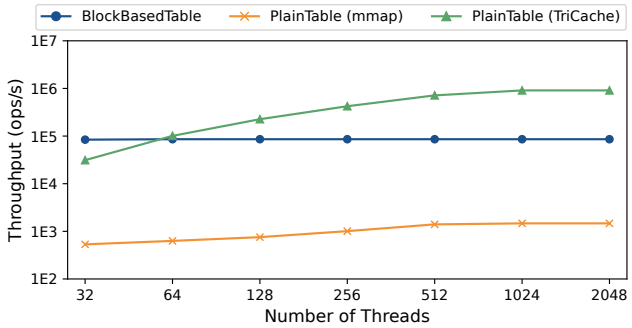
And the hit cycles of Direct SATC and Private SATC in Table 1 show that the software address translation of TriCache is quite efficient. The average costed cycles of Direct SATC hits in PageRank and Shuffle Sort are approximately 50 cycles; Direct SATC hits in GNU Sort and Private SATC hits in Shuffle Sort take about 150 cycles; Private SATC hits in GNU Sort use about 450 cycles. To give an idea of how much time they take, we list some hardware latencies: 50 cycles are close to a NUMA-local L3 cache hit or an L2 cache false sharing within a NUMA node; 150 cycles correspond to about the half of a NUMA-local memory access; 450 cycles are less than a cross-NUMA memory access or a cross-NUMA cache false sharing. Therefore, TriCache with SATC is efficient enough to provide a virtual memory interface and also to deliver memory-comparable performance.

Table 2: Miss rate and average cycles on each cache level

	Direct SATC		Private SATC		Shared Cache	
	MissRate	HitC.	MissRate	HitC.	MissRate	Acc.C.
PageRank	0.003	52.6	0.063	321	0.626	2.36M
ShuffleSort	0.001	63.0	0.001	162	0.969	1.68M
GNUSort	0.045	143	0.007	488	0.926	789K



(a) Computation time of PageRank (lower is better)



(b) Throughput of RocksDB

Figure 14: Performance of TriCache and baselines under different numbers of threads

Performance and Numbers of Threads. We also compare the performance of TriCache and baselines under different numbers of threads for PageRank and RocksDB with 64GB of memory. More precisely, "the performance under a given number of threads" means the maximum performance with less than or equal to this number of threads (only searched over powers of two). Since TriCache uses 16 server threads as the default configuration in the evaluation section, the number of threads starts with 32 threads (including server threads).

As shown in Figure 14, TriCache achieves good scalability for the workloads of both PageRank and RocksDB, which is one of the reasons for why TriCache performs well. For example, from 32 threads to 256 threads (the number of hard-

ware threads), Ligra with TriCache (in Figure 14a) achieves a 4.29× speedup, and RocksDB with TriCache (in Figure 14b) yields a 13.5× performance improvement.

Meanwhile, with a small number of threads, TriCache's performance is worse than that of manually optimized prefetch and asynchronous I/O because of TriCache's synchronous scheme for triggering I/O and its lack of program-specific optimizations (similar to `mmap`). In order to mitigate these limitations, over-subscription can help to utilize the queue depth of SSDs as much as possible. Through over-subscription, the performance of TriCache is improved by 2.58× for PageRank and 2.15× for RocksDB, thus enabling good performance for TriCache even without manual optimizations.

5 Related Work

There is a series of work that tries to improve the page caching performance with customized memory-mapped file I/O paths or swapping approaches [27–30, 39, 41]. Kmmap [28] provides several improvements to reduce the variation in performance owing to the aggressive write-back policy of Linux. FastMap [29] addresses scalability issues by separating clean and dirty pages and using per-core data structures to avoid centralized contentions, with the help of a custom Linux kernel. Still, our evaluation shows that FastMap cannot saturate current high-performance NVMe SSD arrays. Aquila [27] offers a library OS solution that eliminates the need for kernel modifications, relying on hardware support for virtualization, which makes it not easy to deploy on cloud environments. Umap [30] provides an `mmap`-like interface to user-space page fault handlers based on `userfaultfd` [2] in Linux, but is faster than `mmap` only with large page sizes. LightSwap [55] redesigns the swapping system to reduce context switching and page fault overheads, but it requires both kernel and program modifications. TriCache exposes a memory interface like these kernel-involved solutions, but runs completely in the user space to achieve maximal out-of-core performance.

Block caches (or buffer managers) are critical components in data-intensive applications for supporting out-of-core processing [6, 12, 15, 20, 52, 54]. Some attempts try to improve the performance of block caches. SAFS [53], the storage backend for FlashGraph [54], adopts a lightweight cache design based on NUMA-aware message passing. Users need to program with its asynchronous I/O interface to exploit maximal I/O performance on SSD arrays. LeanStore [20] proposes to use pointer swizzling so that pages residing in memory can be directly referenced without page lookups. However, it requires pages to form a tree-like structure, and thus is applicable to limited scenarios. TriCache shares similar goals but provides a memory interface that is user-transparent and more general.

Remote cache systems [14, 24] have been developed upon ideas of the disaggregated architecture [3, 11, 13, 18, 19, 25, 31, 34–36], which utilizes high bandwidth and low latency of modern networks. In this paper, we focus on scaling-up through NVMe SSD arrays. And we intend to consider support for disaggregated architectures in our future work.

Non-volatile memory (NVM) enables larger capacity compared with DRAM, and researches have been devoted to memory management instead of paging strategies to render memory access efficient on hybrid NVM and DRAM architectures [17, 32, 42, 56]. Nevertheless, block caches such as TriCache are still better suited for NVMe SSDs due to their higher latencies than NVM or DRAM.

6 Discussion

In TriCache, SATC does not need to be notified by its Shared Cache when a block is swapped out. In contrast, hardware TLB in processors, which also accelerates address translation as SATC, requires OS page cache to explicitly invalidate evicted pages through *TLB shootdown*, incurring considerable overhead [4, 5] due to inter-processor interrupts (IPIs). A comparison of the mechanisms of SATC and TLB shows that SATC utilizes reference counting to prevent evicting blocks currently being used by clients, while the OS is not directly aware of how many TLB entries are still referring to the pages to be evicted. It is possible to extend the design of SATC to TLB. Processors could mark the reference counts for page tables entries (PTEs), e.g., recording the number of TLB entries that currently hold a specific PTE. The OS can then adapt its page swapping and evict policies to avoid evicting pages currently present in TLBs, thus mitigating the performance issue brought by TLB shootdown.

7 Conclusion

In this paper, we explore a new user-space approach to achieving efficient out-of-core processing with in-memory programs, by providing a virtual memory interface on top of a block cache. We implement TriCache based on a novel multi-level design and applies it to various in-memory or `mmap`-based programs without manual code modification. TriCache achieves out-of-core performance that is orders of magnitude higher than that of the Linux OS page cache, and is often comparable to or even faster than specialized out-of-core solutions.

The open-source implementation of TriCache and instructions to reproduce the main experimental results are accessible from: `https://github.com/thu-pacman/TriCache`.

Acknowledgments

We sincerely thank Liuba Shrira (our shepherd) and all the anonymous OSDI and OSDI AE reviewers for their insightful comments and suggestions. This work was partially supported by National Key Research & Development Plan of China under grant 2017YFA0604500 and NSFC U20B2044. The corresponding author is Wenguang Chen.

References

[1] Sort Benchmark Home Page. http://sortbenchmark.org. [Online; accessed 31-May-2022].

[2] Userfaultfd — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html. [Online; accessed 31-May-2022].

[3] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATc} 18)*, pages 775–787, 2018.

[4] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, Santa Clara, CA, July 2017. USENIX Association.

[5] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, pages 1–14, New York, NY, USA, April 2020. Association for Computing Machinery.

[6] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++, 2016.

[7] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, May 2004. Association for Computing Machinery.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 587–596, New York, NY, USA, March 2011. Association for Computing Machinery.

[9] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, dec 2008.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[11] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. *ACM SIGCOMM Computer Communication Review*, 45(4):551–564, 2015.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[13] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 249–264, 2016.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 649–667, 2017.

[15] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system*. Now Publishers Inc, 2007.

[16] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[17] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706, 2015.

[18] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1–15, New York, NY, USA, April 2016. Association for Computing Machinery.

[19] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[21] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 447–461, Huntsville, Ontario, Canada, October 2019. Association for Computing Machinery.

[22] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[23] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, March 2020.

[24] Shuang Liang, Ranjit Noronha, and Dhabaleswar K Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2005.

[25] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[26] Zhiyuan Lin, Minsuk Kahng, Kaeser Md Sabrin, Duen Horng Polo Chau, Ho Lee, and U Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 159–164. IEEE, 2014.

[27] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293. Association for Computing Machinery, New York, NY, USA, April 2021.

[28] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 490–502, 2018.

[29] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827. USENIX Association, July 2020.

[30] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78. IEEE, 2019.

[31] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[32] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[33] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, October 2017. Association for Computing Machinery.

[34] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}: High-performance, application-integrated far memory. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 315–332, 2020.

[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[38] Johannes Singler and Benjamin Konsik. The gnu libstdc++ parallel mode: Software engineering considerations, 2008.

[39] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Transactions on Storage (TOS)*, 12(4):1–27, 2016.

[40] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[41] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[42] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555, 2018.

[43] Wikipedia contributors. Memory-mapped file — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Memory-mapped%20file&oldid=1089594834, 2022. [Online; accessed 31-May-2022].

[44] Wikipedia contributors. Memory paging — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Memory%20paging&oldid=1068326108, 2022. [Online; accessed 31-May-2022].

[45] Wikipedia contributors. NVM Express — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=NVM%20Express&oldid=1090339430, 2022. [Online; accessed 31-May-2022].

[46] Wikipedia contributors. Page cache — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Page%20cache&oldid=1068818367, 2022. [Online; accessed 31-May-2022].

[47] Wikipedia contributors. Pci express — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=1090153203, 2022. [Online; accessed 31-May-2022].

[48] Wikipedia contributors. U.2 — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=U.2&oldid=1066844795, 2022. [Online; accessed 31-May-2022].

[49] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All replacement for a multilevel cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association.

[50] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications, 2017.

[51] Idan Yaniv and Dan Tsafrir. Hash, Don't Cache (the Page Table). *ACM SIGMETRICS Performance Evaluation Review*, 44(1):337–350, June 2016.

[52] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[53] Da Zheng, Randal Burns, and Alexander S. Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[55] Kan Zhong, Wenlin Cui, Youyou Lu, Quanzhang Liu, Xiaodan Yan, Qizhao Yuan, Siwei Luo, and Keji Huang. Revisiting swapping in user-space with lightweight threading, 2021.

[56] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.

[57] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7):1020–1034, March 2020.