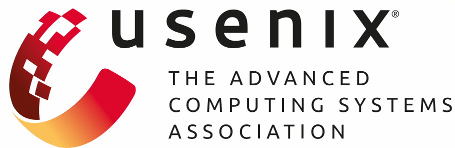


Optimizing Memory-mapped I/O for Fast Storage Devices

Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas, *FORTH-ICS*

https://www.usenix.org/conference/atc20/presentation/papagiannis

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Optimizing Memory-mapped I/O for Fast Storage Devices

Anastasios Papagiannis[1], Giorgos Xanthakis[1], Giorgos Saloustros,
Manolis Marazakis, and Angelos Bilas[1]
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
{apapag, gxanth, gesalous, maraz, bilas}@ics.forth.gr

Abstract

Memory-mapped I/O provides several potential advantages over explicit read/write I/O, especially for low latency devices: (1) It does not require a system call, (2) it incurs almost zero overhead for data in memory (I/O cache hits), and (3) it removes copies between kernel and user space. However, the Linux *memory-mapped I/O* path suffers from several scalability limitations. We show that the performance of Linux *memory-mapped I/O* does not scale beyond 8 threads on a 32-core server. To overcome these limitations, we propose *FastMap*, an alternative design for the *memory-mapped I/O* path in Linux that provides scalable access to fast storage devices in multi-core servers, by reducing synchronization overhead in the common path. *FastMap* also increases device queue depth, an important factor to achieve peak device throughput. Our experimental analysis shows that *FastMap* scales up to 80 cores and provides up to 11.8× more IOPS compared to *mmap* using *null_blk*. Additionally, it provides up to 5.27× higher throughput using an Optane SSD. We also show that *FastMap* is able to saturate state-of-the-art fast storage devices when used by a large number of cores, where Linux *mmap* fails to scale.

1 Introduction

The emergence of fast storage devices, with latencies in the order of a few μs and IOPS rates in the order of millions per device is changing the I/O landscape. The ability of devices to cope well with random accesses leads to new designs for data storage and management that favor generating small and random I/Os to improve other system aspects [2, 35, 42, 43]. Although small and random I/Os create little additional pressure to the storage devices, they result in significantly higher CPU overhead in the kernel I/O path. As a result, the overhead of performing I/O operations to move data between memory and devices is becoming more pronounced, to the point where a large fraction of server CPU cycles are consumed only to serve storage devices [11, 46].

In this landscape, *memory-mapped I/O*, i.e. Linux *mmap*, is gaining more attention [8, 13, 24, 42, 43] for data intensive applications because of its potentially lower overhead compared to read/write system calls. An off-the-shelf NVMe block device [28] has access latency close to 10 μs and is capable of more than 500 KIOPS for reads and writes. Byte-addressable, persistent memory devices exhibit even better performance [29]. The traditional read/write system calls in the I/O path incur overheads of several μs [11, 46] in the best case and typically even higher, when asynchronous operations are involved.

In contrast, when using *memory-mapped I/O* a file is mapped to the process virtual address space where the user can access data with processor *load/store* instructions. The kernel is still responsible for moving data between devices and memory; *mmap* removes the need for an explicit system call per I/O request and incurs the overhead of an implicit page fault only when data does not reside in memory. In the case when data reside in memory, there is no additional overhead due to I/O cache lookups and system calls. Therefore, the overhead for hits is reduced dramatically as compared to both the kernel buffer cache but also to user-space I/O caches used in many applications. In several cases *memory-mapped I/O* removes the need to serialize and deserialize user data, by allowing applications to have the same format for both in-memory and persistent data, and also the need for memory copies between kernel and user space.

A major reason for the limited use of *memory-mapped I/O*, despite its advantages, has been that *mmap* may generate small and random I/Os. With modern storage devices, such as NVMe and persistent memory, this is becoming less of a concern. However, Figure 1 shows that the default *memory-mapped I/O* path (*mmap* backed by a device) for random page faults does not scale well with the number of cores. In this experiment (details in Section 4), we use *null_blk*, a Linux driver that emulates a block device but does not issue I/Os to a real device (we use 4TB dataset and 192GB of

[1] Also with the Department of Computer Science, University of Crete, Greece.

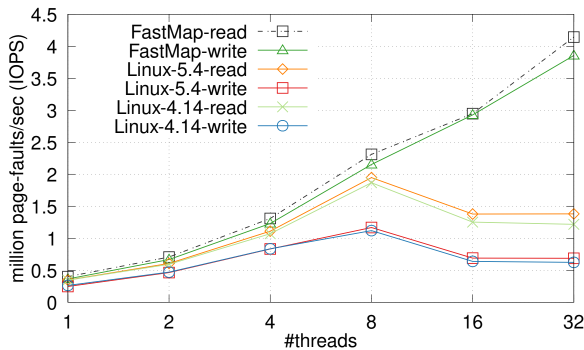


Figure 1: Scalability of random page faults using two versions of Linux *memory-mapped I/O* path (v4.14 & v5.4) and *FastMap*, over the *null_blk* device.

DRAM cache). Using *null_blk* allows us to stress the Linux kernel software stack while emulating a low-latency, next-generation storage device. Linux *mmap* scales up to only 8 cores, achieving 7.6 GB/s (2M random IOPS), which is about 5× less compared to a state-of-the-art device [29]; servers with multiple storage devices need to cope with significantly higher rates. We observe that from Linux kernel 4.14 to 5.4 the performance and the scalability of the *memory-mapped I/O* path has not improved significantly. Limited scalability also results in low device queue depth. Using the same micro-benchmark for random read page faults with 32 threads on an Intel Optane SSD DC P4800X, we see that the average device queue depth is 27.6. A large queue depth is essential for fast storage devices to provide their peak device throughput.

In this paper, we propose *FastMap*, a novel design for the *memory-mapped I/O* path that overcomes these two limitations of *mmap* for data intensive applications on multi-core servers with fast storage devices. *FastMap* (a) separates clean and dirty-trees to avoid all centralized contention points, (b) uses full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduces a scalable DRAM cache with per-core data structures to reduce latency variability. *FastMap* achieves both higher scalability and higher I/O concurrency by (1) avoiding all centralized contention points that limit scalability, (2) reducing the amount of CPU processing in the common path, and (3) using dedicated data-structures to minimize interference among processes, thus improving tail latency. As a further extension to *mmap*, we introduce a user-defined read-ahead parameter to proactively map pages in application address space and reduce the overhead of page faults for large sequential I/Os.

We evaluate *FastMap* using both micro-benchmarks and real workloads. We show that *FastMap* scales up to 80 cores and provides up to 11.8× more random IOPS compared to Linux *mmap* using *null_blk*. *FastMap* achieves 2× higher throughput on average for all YCSB workloads over Kreon [43], a persistent key-value store designed to use *memory-mapped I/O*. Moreover, we use *FastMap* to extend the virtual address space of memory intensive applications beyond the physical memory size over a fast storage device. We achieve up to 75× lower average latency for TPC-C over Silo [54] and 5.27× better performance with the Ligra graph processing framework [50]. Finally, we achieve 6.06% higher throughput on average for all TPC-H queries over MonetDB [8] that mostly issue sequential I/Os.

In summary, our work optimizes the *memory-mapped I/O* path in the Linux kernel with three main contributions:

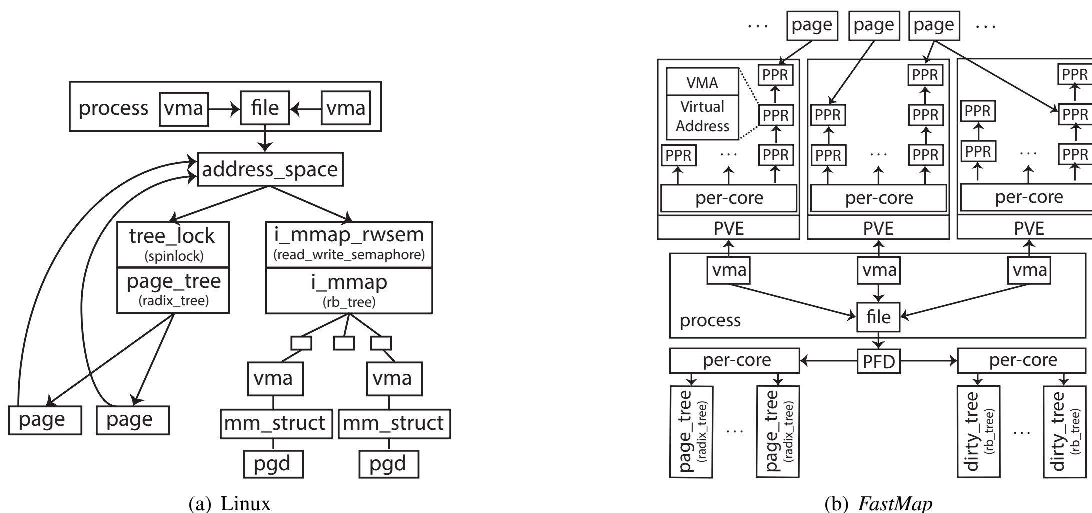
1. We identify severe performance bottlenecks of Linux *memory-mapped I/O* in multi-core servers with fast storage devices.
2. We propose *FastMap*, a new design for the *memory-mapped I/O* path.
3. We provide an experimental evaluation and analysis of *FastMap* compared to Linux *memory-mapped I/O* using both micro-benchmarks and real workloads.

The rest of the paper is organized as follows. §2 provides the motivation behind *FastMap*. §3 presents the design of *FastMap* along with our design decisions. §4 and §5 present our experimental methodology and results, respectively. Finally, §6 reviews related work and §7 concludes the paper.

2 Motivation

With storage devices that exhibit low performance for random I/Os, such as hard disk drives (HDDs), *mmap* results in small (4KB) random I/Os because of the small page size used in most systems today. In addition, *mmap* does not provide a way for users to manage page writebacks in the case of high memory pressure, which leads to unpredictable tail latencies [43]. Therefore, historically the main use of *mmap* has been to load binaries and shared libraries into the process address space; this use-case does not require frequent I/O, uses read-mostly mappings, and exhibits a large number of shared mappings across processes, e.g. libc is shared by almost all processes of the system. Reverse mappings provide all page table translations for a specific page and they are required in order to unmap a page during evictions. Therefore, Linux *mmap* uses object-based reverse mappings [37] to reduce memory consumption and enable fast *fork* system calls, as they do not require copying full reverse mappings.

With the introduction of fast storage devices, where the throughput gap between random and sequential I/O is small, *memory-mapped I/O* has the potential to reduce I/O path overhead in the kernel, which is becoming the main bottleneck for data-intensive applications. However, data intensive applications, such as databases or key-value stores, have different requirements compared to loading binaries: they can be write-intensive, do not require large amount of sharing, and do not use *fork* system calls frequently. These properties make the



(a) Linux

(b) *FastMap*

Figure 2: Linux (left) and *FastMap* (right) high-level architecture for memory-mapped files (acronyms: **PFD**=**P**er-**F**ile-**D**ata, **PVE**=**P**er-**V**ma-**E**ntry, **PPR**=**P**er-**P**ve-**R**map).

use of full reverse mappings a preferred approach. In addition, data intensive applications use datasets that do not fit in main memory and thus, the path of reading and writing a page from the device becomes the common case. Most of these applications are also heavily multithreaded and modern servers have a large number of cores.

3 Design of *FastMap*

The Linux kernel provides the *mmap* and *munmap* system calls to create and destroy memory mappings. Linux distinguishes memory mappings in shared vs. private. Mappings can also be anonymous, i.e. not backed by a file or device. Anonymous mappings are used for memory allocation. In this paper we examine I/O over persistent storage, an inherently shared resource. Therefore, we consider only shared memory mappings backed by a file or block device, as also required by Linux *memory-mapped I/O*.

Figure 2(a) shows the high-level architecture of shared memory mappings in the Linux kernel. Each virtual memory region is represented by a *struct vm_area_struct* (*VMA*). Each *VMA* points to a *struct file* (*file*) that represents the backing file or device and the starting offset of the memory mapping to it. Each *file* points to (a shared between processes) *struct address_space* (*address_space*) which contains information about mapped pages and the backing file or device.

Figure 2(b) illustrates the high-level design of *FastMap*. The most important components in our design are *per_file_data* (*PFD*) and *per_vma_entry* (*PVE*). Combined, these two components provide equivalent functionality as the Linux kernel *address_space* structure. Each *file* points to a *PFD* and each *VMA* points to a *PVE*. The role of a *PFD* is to keep metadata about device blocks that are in the *FastMap* cache and metadata about dirty pages. *PVE* provides full reverse mappings.

3.1 Separate Clean and Dirty Trees in *PFD*

In Linux, one of the most important parts of *address_space* is *page_tree*, a radix tree that keeps track of all pages of a cacheable and mappable file or device, both clean and dirty. This data structure provides an effective way to check if a device block is already in memory when a page fault occurs. Lookups are lock-free (*RCU*) but inserts and deletes require a spinlock (named *tree_lock*). Linux kernel radix trees also provide user-defined *tags* per entry. A *tag* is an integer, where multiple values can be stored using bitwise operations. In this case *tags* are used to mark pages as dirty. Marking a previously read-only page as writable requires holding the *tree_lock* to update the *tag*.

Using the experiments of Figure 1 and *lockstat* we see that *tree_lock* is by far the most contended lock: Using the same multithreaded benchmark as in Figure 1, over a single memory mapped region, *tree_lock* has 126× more contended lock acquisitions, which involve cross-cpu data, and 155× more time waiting to acquire the lock, compared to the second most contended lock. The second more contended lock is a spinlock that protects concurrent modifications in *PTEs* (4th level entries in the page table). This design has remained essentially unchanged from Linux kernel 2.6 up to 5.4 (latest stable version at the time of this writing).

To remove the bottleneck in *tree_lock*, *FastMap* uses a new structure for per-file data, *PFD*. The most important aspects of *PFD* are: (i) a per-core radix tree (*page_tree*) that

keeps all (clean and dirty) pages and (ii) a per-core red-black tree (*dirty_tree*) that keeps only dirty pages. Each of these data structures is protected by a separate (per core) spinlock, different for the radix and red-black trees. We assign pages to cores in a round-robin manner and we use the page offset to identify the per-core structure that holds each page.

We use *page_tree* to provide lock-free lookups (RCU), similar to the Linux kernel. We use per-core data structures to reduce contention in case we need to add or remove a page. On the other hand, we do not use *tags* to mark pages as dirty but we use the *dirty_tree* for this purpose. In the case where we have to mark a previously read-only page as read-write, we only acquire the appropriate lock of *dirty_tree* without performing any additional operations to the *page_tree*. Furthermore, having all dirty pages in a sorted data structure (red-black tree) enables efficient I/O merging for the cases of writebacks and the *msync* system call.

3.2 Full Reverse Mappings in *PVE*

Reverse (inverted) mappings are also an important part of *mmap*. They are used in the case of evictions and writebacks and they provide a mechanism to find all existing virtual memory mappings of a physical page. File-backed memory mappings in Linux use object-based reverse mappings [37]. The main data structure for this purpose is a red-black tree, *i_mmap*. It contains all *VMAs* that map at least one page of this *address_space*. A read-write semaphore, *i_mmap_rwsem*, protects concurrent accesses to the *i_mmap* red-black tree. The main function that removes memory mappings for a specific page is *try_to_unmap*. Each page has two fields for this purpose: (i) a pointer to the *address_space* that it belongs to and (ii) an atomic counter (*_mapcount*) that keeps the number of active page mappings. Using the pointer to *address_space*, *try_to_unmap* gets access to *i_mmap* and then iterates over all *VMAs* that belong to this mapping. Through each VMA, it has access to *mm_struct* which contains the root of the process page table (*pgd*). It calculates the virtual address of the mapping based on the *VMA* and the page, which is required for traversing the page table. Then it has to check all active *VMAs* of *i_mmap* if the specific page is mapped, which results in many useless page table traversals. This is the purpose of *_mapcount*, which limits the number of traversals. This strategy is insufficient in some cases but it requires a very small amount of memory for the reverse mappings. More specifically, in the case where *_mapcount* is greater than zero, we may traverse the page table for a *VMA* where the requested page is not mapped. This can happen in the case where a page is mapped in several different *VMAs* in the same process, i.e. with multiple *mmap* calls, or mapped in the address space of multiple different processes. In such a case, we have unnecessary page table traversals that introduce overheads and consume CPU cycles. Furthermore, during this procedure, *i_mmap_rwsem* is held as a read lock and as a write lock only during *mmap* and *munmap* system calls. Previous research shows that even a read lock can limit scalability in multicore servers [15].

The current object-based reverse mappings in Linux have two disadvantages: (1) with high likelihood they result in unnecessary page table traversals, originating from *i_mmap*, and (2) they require a coarse grain read lock to iterate *i_mmap*. Other works have shown that in multi-core servers locks can be expensive, even for read-write locks when acquired as read locks [15]. These overheads are more pronounced in servers with a NUMA memory organization [10].

To overcome these issues *FastMap* provides finer grained locking, as follows: *FastMap* uses a structure with an entry for each VMA, *PVE*. Each *PVE* keeps a per-core list of all pages that belong to this *VMA*. A separate (per core) spinlock protects each of these lists. The lists are append-only as unmapping a page from a different page table does not require any ordering. We choose the appropriate list based on the core that runs the append operation (*smp_processor_id()*). These lists contain *per_pve_rmap* (*PPR*) entries. Each *PPR* contains a tuple $(VMA, virtual_address)$. These metadata are sufficient to allow iterating over all mapped pages of a specific memory mapping in the case of an *munmap* operation. Furthermore, each page contains an append-only list of active *PPRs*, which are shared both for *PVEs* and pages. This list is used when we need to evict a page that is already mapped in one or more address spaces, in the event of memory pressure.

3.3 Dedicated DRAM Cache

An *mmap address_space* contains information about the backing file or device and the required functions to interact with the device in case of page reads and writes. To write back a set of pages of a memory mapping, Linux iterates *page_tree* in a lock-free manner with RCU and writes only the pages that have the dirty tag enabled. Linux also keeps a per-core LRU to find out which pages to evict. In the case of evictions, Linux tries to remove clean pages in order not to wait for dirty pages to do the writeback [37].

The Linux page-cache is tightly coupled with the swapper. For the *memory-mapped I/O* path, this dependency results in unpredictable evictions and bursty I/O to the storage devices [43]. Therefore, *FastMap* implements its own DRAM cache, managing evictions via an approximation of LRU. *FastMap* has two types of LRU lists: one containing only clean pages (*clean_queue*) and one containing only dirty pages (*dirty_queue*). *FastMap* maintains per-core *clean_queues* to reduce lock contention. We identify the appropriate *clean_queue* as clean_queue_id = page_offset % num_cores.

When there are no free pages during a page fault, *FastMap* evicts only clean pages, similar to the Linux kernel [37], from the corresponding *clean_queue*. We evict a batch (with a configurable size, currently set to 512) of clean pages to amor-

tize the cost of page table manipulation and TLB invalidations. Each page eviction requires a TLB invalidation with the *flush_tlb* function, if the page mapping is cached. *flush_tlb* sends an IPI (Inter-Processor-Interrupt) to all cores, incurring significant overheads and limiting scalability [3,4]. We implement a mechanism to reduce the number of calls to *flush_tlb* function, using batching, as follows.

A TLB invalidation requires a pointer to the page table and the *page_offset*. *FastMap* keeps a pointer to the page table and a range of *page_offsets*. Then, we invoke *flush_tlb* for the whole range. This approach may invalidate more pages, but reduces the number of *flush_tlb* calls by a factor of the batch-size of page evictions (currently 512). As the file mappings are usually contiguous in the address space in data intensive applications, in the common case false TLB invalidations are infrequent. Thus, *FastMap* manages to maintain a high number of concurrent I/Os to devices and increase device throughput. *LATR* [33] proposes the use of an asynchronous TLB invalidation mechanism based on message passing. In our case, we cannot delay TLB invalidations as the pages should be used immediately for page fault handling.

FastMap uses multiple threads to write dirty pages to the underlying storage device (writeback). Each of these manages its own *dirty_queue*. This design removes the need of synchronization when we remove dirty pages from a *dirty_queue*. During writebacks, *FastMap* merges consecutive I/O requests to generate large I/O operations to the underlying device. To achieve this, we use *dirty_trees* that keep dirty pages sorted based on the device offset. As we have multiple *dirty_trees*, we initialize an iterator for each tree and we combine the iterator results using a min-max heap. When a writeback occurs, we also move the page to the appropriate *clean_queue* to make it available for eviction. As page writeback also requires a TLB invalidation, we use the same mechanism as in the eviction path to reduce the number of calls to the kernel *flush_tlb* function. Each writeback thread checks the ratio of dirty to clean pages and starts the writeback when the percentage is higher than 75% of the total cache pages. The cache in *FastMap* currently uses a static memory buffer, allocated upon module initialization and does not create any further memory pressure to the Linux page cache. We also provide a way to grow and shrink this cache at runtime, but we have not yet evaluated alternative sizing policies.

To keep track of free pages *FastMap* uses a per-core free list with a dedicated spinlock. During a major page fault i.e., when the page does not reside in the cache, the faulting thread first tries to get a page from its local free list. If this fails, it tries to steal a page from another core's free list (randomly selected). After *num_cores* unsuccessful tries, *FastMap* forces page evictions to cleanup some pages. To maintain all free lists balanced, each evicted page is added to the free list from which we originally obtained the page.

Overall, *FastMap* with per-core data structures requires more memory compared to the native Linux *mmap*. *FastMap*

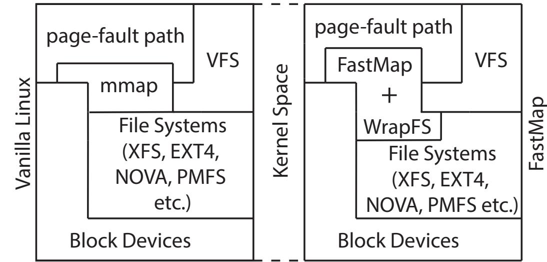


Figure 3: FastMap I/O path.

requires a single *PFD*, which is 1120 bytes, per file for all memory mappings. A single *PVE* is about 512 bytes and a single *PPR* is 24 bytes. We require a single *PVE* for each mmap call, i.e. $1:1$ with the Linux VMA struct. *FastMap* requires a single *PPR* entry per *PVE* for each mapped page, independently of how many threads access the same page. In the setups we target, there is little sharing of files across processes and we can therefore, assume that we only need one *PPR* entry for each page in our DRAM cache. For instance, assume that a single application maps 1000 files and uses $8GB$ of DRAM cache. This results in $1.64MB$ of additional memory, independently of the size of files and the number of threads. *FastMap* targets storage servers with large memory spaces and can be applied selectively for the specific mount points that hold the files of data-intensive applications. While it is, in principle, possible to allow more fine-grain uses of *FastMap* in Linux, we leave this possibility for future work.

Finally, the Linux kernel also provides private, file-backed memory mappings. These are Copy-On-Write mappings and writes to them do not reach the underlying file/device. Such mappings are outside the scope of this paper, but they share the same path in the Linux kernel to a large extent. Our proposed techniques also apply to private file-backed mappings. However, these mappings are commonly used in Linux kernel to load binaries and shared libraries, resulting in a large degree of sharing. We believe that it is not beneficial to use the increased amount of memory required by *FastMap* to optimize this relatively uncommon path.

3.4 Implementation

Figure 3 shows the I/O path in the Linux kernel and indicates where *FastMap* is placed. *FastMap* is above *VFS* and thus is independent of the underlying file system. This means that common file systems such as *XFS*, *EXT4*, and *BTRFS* [1] can benefit from our work.

FastMap provides a user interface for accessing both a block device but also a file system through a user-defined mount point. For the block device case, we implement a virtual block device that uses our custom *mmap* function. All other block device requests (e.g. *read*/*write*) are forwarded to

[1] *FastMap* has been successfully tested with all of these file systems.

the underlying device. Requests for fetching or evicting pages from *FastMap* are issued directly to the underlying device.

For the file system implementation we use WrapFS [59], a stackable file system that intercepts all *mmap* calls to a specific mount point so that *FastMap* is used instead of the native Linux *mmap* implementation. For fetching or evicting pages from within *FastMap* we use direct I/O to the underlying file system, bypassing the Linux page cache. All other file system calls are forwarded to the underlying file system.

4 Experimental Methodology

In this section, we present the experimental methodology we use to answer the following questions:

1. How does *FastMap* perform compared to Linux *mmap*?
2. How much does *FastMap* improve storage I/O?
3. How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

Our main testbed consists of a dual-socket server that is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, each with 8 physical cores and 16 hyper-threads for a total of 32 hyper-threads. The primary storage device is a PCIe-attached Intel Optane SSD DC P4800X series [28] with 375 GB capacity. For the purposes of evaluating scalability, we use an additional four-socket server. This four-socket server is equipped with four Intel(R) Xeon(R) CPU E5-4610 v3 CPUs running at 1.7 GHz, each with 10 physical cores and 20 hyper-threads for a total of 80 hyper-threads. Both servers are equipped with 256 GB of DDR4 DRAM at 2400 MHz and run CentOS v7.3, with kernel 4.14.72.

During our evaluation we limit the available capacity of DRAM (using a kernel boot parameter) as required by different experiments. In our evaluation we use datasets that both fit and do not fit in main memory. This allows us to provide a more targeted evaluation and separate the costs of the page-fault path and the eviction path. To reduce variability in our experiments, we disable swap and Transparent Huge Pages (THP), and we set the CPU scaling governor to "performance". In experiments where we want to stress the software path of the Linux kernel we also use the *null_blk* [40] and *pmem* [47] block devices. *null_blk* emulates a block device but ignores the I/O requests issued to it. For *null_blk* we use the bio-based configuration. *pmem* emulates a fast block device that is backed by DRAM.

In our evaluation we first use a custom multithreaded microbenchmark. It uses a configurable number of threads that issue *load/store* instructions at randomly generated offsets within the memory mapped region. We ensure that each *load/store* results in a page fault.

Second, we use a persistent key-value store. We choose Kreon [43], a state-of-the-art persistent key-value store that is designed to work with *memory-mapped I/O*. The design of Kreon is similar to the LSM-tree, but it maintains a separate B-Tree index per-level to reduce I/O amplification. Kreon uses a log to keep user data. It uses *memory-mapped I/O* to perform all I/O between memory and (raw) devices. Furthermore, it uses Copy-On-Write (COW) for persistence, instead of Write-Ahead-Logging. Kreon follows a single-writer, multiple-reader concurrency model. Readers operate concurrently with writers using Lamport counters [34] per node for synchronization to ensure correctness. For inserts and updates, it uses a single lock per database; however, by using multiple databases Kreon can support concurrent updates.

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 1: Standard YCSB Workloads.

To improve single-database scalability in Kreon and make it more suitable for evaluating *FastMap*, we implement the second protocol that Bayer et al. propose [7]. This protocol requires a read-write lock per node. It acquires the lock as a read lock in every traversal from the root node to a leaf. In the case of inserts or rebalance operations it acquires the corresponding lock as a write lock. As every operation has to acquire the root node read lock, this limits scalability [15]. To overcome this limitation, we replace the read/write lock of the root node with a Lamport counter and a lock. Every operation that modifies the root node acquires the lock, changes the Lamport counter, performs a COW operation, and then writes the update in the COW node.

For Kreon we use the YCSB [18] workloads and more specifically a C++ implementation [48] to remove overheads caused by the JNI framework, as *Kreon* is highly efficient and is designed to take advantage of fast storage devices. Table 1 summarizes the YCSB workloads we use. These are the proposed workloads, and we execute them in the author's proposed sequence [18], as follows: LoadA, RunA, RunB, RunC, RunF, RunD, clear the database, and then LoadE, RunE.

Furthermore, we use Silo [54], an in-memory database that also provides scalable transactions for modern multicore machines. In this case, we use TPC-C [52], a transactional benchmark, which models a retail operation and is a common benchmark for OLTP workloads. We also use Ligra [50], a lightweight graph processing framework for shared memory with OpenMP-based parallelization. Specifically, we use the Breadth First Search (BFS) algorithm. We use Silo and Ligra to evaluate *FastMap*'s effectiveness in extending the virtual address space of an application beyond physical memory over fast storage devices. For this reason we convert all *malloc/free* calls of Ligra and Silo to allocate space over a memory-mapped file on a fast storage device. We use the *libvmmalloc* library from the Persistent Memory Development Kit (PMDK) [45]. *libvmmalloc* transparently converts all dy-

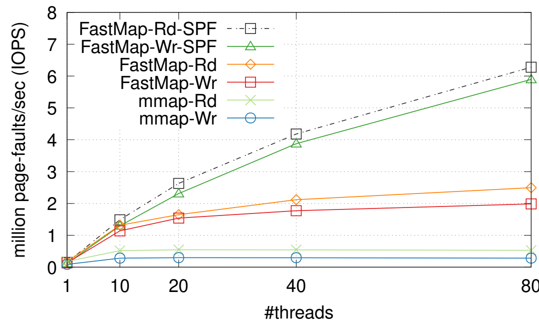


Figure 4: Scalability of random page faults for Linux and *FastMap*, with up to 80 threads, using the *null_blk* device.

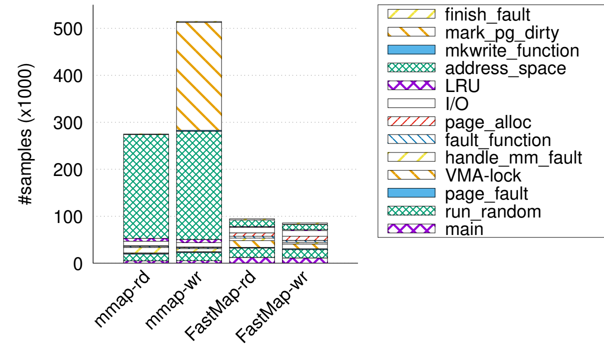


Figure 5: *FastMap* and Linux *mmap* breakdown for read and write page faults, with *null_blk* and 32 cores.

namic memory allocations to persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application. The memory allocator of *libvmmalloc* is based on *jemalloc* [30].

Finally, we evaluate *FastMap* using MonetDB-11.31.7 [8, 39], a column-oriented DBMS that is designed to use *mmap* to access files instead of using the read/write API. We use the TPC-H [53] benchmark, a warehouse read-mostly workload.

We run all experiments three times and we report the averages. In all cases the variation observed across runs is small.

5 Experimental Results

5.1 How does *FastMap* perform compared to Linux *mmap*?

Microbenchmarks: Figure 1 shows that Linux *mmap* fails to scale beyond eight threads on our 32-core server. *FastMap* provides 3.7× and 6.6× more random read and write IOPS, respectively, with 32 threads compared to Linux *mmap*. Furthermore, both versions 4.14 and 5.4 of the Linux kernel achieve similar performance. To further stress *FastMap*, we use our 80-core server and the *null_blk* device. Figure 4 shows that with 80 threads, *FastMap* provides 4.7× and 7× higher random read and write IOPS respectively, compared to Linux *mmap*. Furthermore, in both cases *FastMap* performs up to 38% better even in the case where there is little or no concurrency, when using a single thread.

Figure 4 shows that not only *FastMap* scales better compared to Linux *mmap*, but also that *FastMap* sustains more page faults per second. On the other hand *FastMap* does not achieve perfect scalability. For this reason, we profile *FastMap* using the random read page faults microbenchmark. We find that the bottleneck is the read-write lock (*mmap_sem*) that protects the red-black tree of active *VMAs*. This is the problem that *Bonsai* [15] tackles. Specifically, with 10 cores the cost of *read_lock* and *read_unlock* is 7.6% of the total execution time, with 20 cores it becomes 25.4%, with 40 cores 32%, and with 80 cores 37.4%. To confirm this intuition, we apply Speculative Page Faults (SPF) [20], and attempt to use SRCU (Sleepable RCU) instead of the read-write lock to protect the red-black tree of active *VMAs*, an approach similar to *Bonsai*. We use the Linux kernel patches from [19] as, at the time of writing, they have not been merged in the Linux mainline. As SPF works only for anonymous mappings, we modify it to use *FastMap* for block-device backed memory-mappings. Figure 4 shows that *FastMap* with *SPF* provides even better scalability: 2.51× and 11.89× higher read IOPS compared to *FastMap* without SPF and to Linux kernel, respectively. We do not provide an evaluation of *SPF* without *FastMap* as it (1) works only for anonymous mappings and (2) it could use the same Linux kernel path that has scalability bottlenecks, as we show in Section 3.1.

Figure 5 shows the breakdown of the execution time for both random reads and writes. We profile these runs using *perf* at 999Hz and plot the number of samples (y axis) that *perf* reports. First, we see that for random reads Linux *mmap* spends almost 80% of the time in manipulating the *address_space* structure, specifically in the contented *tree_lock* that protects the *radix_tree* which keeps all the pages of the mapping (see Section 3). In *FastMap* we do not observe a single high source of overhead. In the case of writes the overhead of this lock is even more pronounced in Linux *mmap*. For each page that is converted from read-only to read-write, Linux has to acquire this lock again to set the *tag*. *FastMap* removes this contention point as we keep metadata about dirty pages only in the per-core red-black trees (Section 3.3). Therefore, we do not modify the *radix_tree* upon the conversion of a read-only page to a read-write page.

Figure 6 shows how each optimization in *FastMap* affects I/O performance. Vanilla is the Linux *mmap* and *basic* is *FastMap* with all the optimizations disabled, except the per-core red-black tree. The *per-core radix-tree* optimization is important, because with increasing core counts on modern

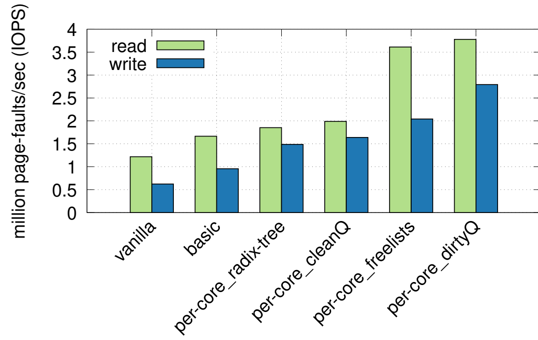


Figure 6: Performance gains from different optimizations in *FastMap*, as compared to "vanilla" Linux using *null_blk* and 32 cores.

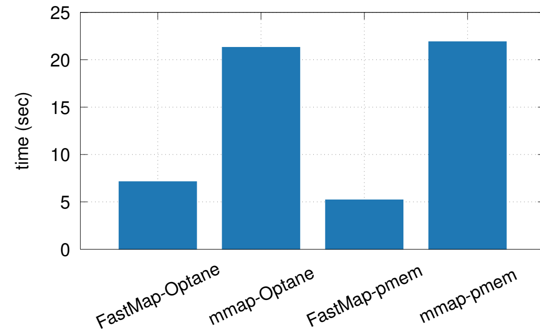


Figure 7: Execution time for Ligra running BFS with 32 threads and using an Optane SSD and a *pmem* device.

servers (Section 3.1) the single *radix tree* lock is by far the most contended lock. *per-core cleanQ* enables the per-core LRU list for clean pages. The *per-core freelists* optimization allows for scalable page allocation, resulting in significant performance gains. Finally, the main purpose of *per-core dirtyQ* is to enable higher concurrency when we convert a page from read-only to read-write and allow for multiple eviction threads with minimal synchronization. This optimization mainly improves the write path, as is shown in Figure 6.

In-memory Graph Processing: We evaluate *FastMap* as a mechanism to extend the virtual address space of an application beyond the physical memory and over fast storage devices. Using *mmap* (and *FastMap*) a user can easily map a file over fast storage and provide an extended address space, limited only by device capacity. We use Ligra [50], a graph processing framework, a demanding workload in terms of memory accesses and commonly operating on large datasets. Ligra assumes that the dataset (and metadata) fit in main memory. For our evaluation we generate a R-Mat [12] graph of 100M vertices, with the number of directed edges is set to 10× the number of vertices. We run BFS on the resulting 18GB graph, thus generating a read-mostly random I/O pattern. Ligra requires about 64GB of DRAM throughout execution. To evaluate *FastMap* and Linux *mmap*, we limit the main memory of our 32-core server to 8 GB and we use the Optane SSD device.

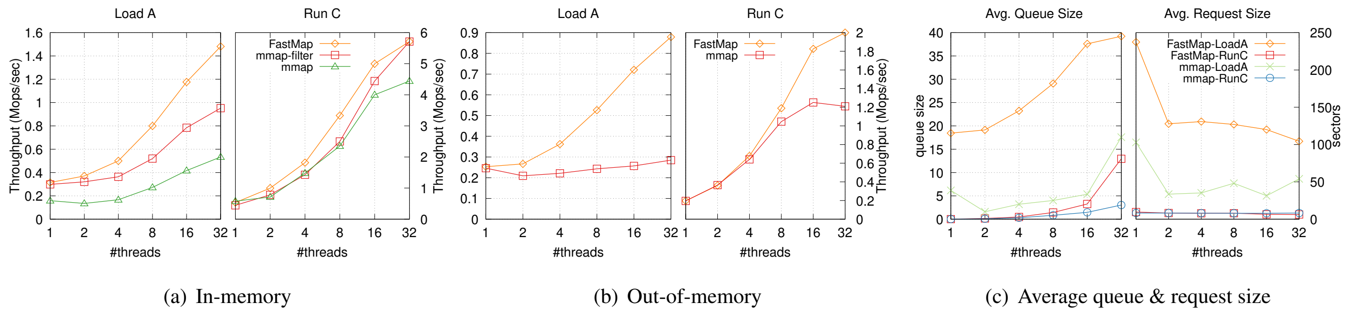
Figure 7 shows that BFS completes in 6.42s with *FastMap* compared to 21.3s with default *mmap* and achieves a 3.31× improvement. *FastMap* requires less than half the system time (10.3% for *FastMap* vs. 27.38% for Linux) and stresses more the underlying storage device as seen in iowait time (19.31% for *FastMap* vs. 9.5% for Linux). This leaves 2.11× more user-time available for the Ligra workload execution. Using a *pmem* device the benefits of *FastMap* are even higher. Linux *mmap* requires 21.9s for BFS, while *FastMap* requires only 4.15s, i.e. a 5.27× improvement. Overall, Ligra induces a highly concurrent I/O pattern that stresses the default *mmap*, resulting in lock contention as described in Section 3.1 and as evidenced by the increased system time. The default *mmap* results in a substantial slowdown, even with a *pmem* device that has throughput comparable to DRAM.

5.2 How much does *FastMap* improve storage I/O?

Kreon Persistent Key-value Store: In this section we evaluate *FastMap* using Kreon, a persistent key-value store that uses *memory-mapped I/O* and a dataset of 80M records. The keys are 30 bytes long, with 1000 byte values. This results in a total footprint of about 76GB. We issue 80M operations for each of the YCSB workloads. For the in-memory experiment, we use the entire DRAM space (256GB) of the testbed, whereas for the out-of-memory experiment we limit available memory to 16GB. In all cases we use the Optane SSD device.

Figure 8(a) illustrates the scalability of Kreon, using *FastMap*, Linux *mmap*, and *mmap-filter*, with a dataset that fits in main memory. The *mmap-filter* configuration is the default Linux *mmap* implementation augmented with a custom kernel module we have created to remove the unnecessary read I/O from the block device for newly allocated pages. Using 32 threads (on the 32-core server), *FastMap* achieves 1.55× and 2.77× higher throughput compared to *mmap-filter* and *mmap* respectively, using the LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 9% and 28% higher throughput compared to *mmap-filter* and *mmap* respectively. As we see *mmap-filter* performs always better, therefore, for the rest of the Kreon evaluation we use this configuration as our baseline.

Figure 8(b) shows the scalability of Kreon with *FastMap* and *mmap-filter* (denoted as *mmap*) using a dataset that does not fit in main memory. Using 32 threads (on the 32-core



(a) In-memory (b) Out-of-memory (c) Average queue & request size

Figure 8: Kreon scalability with increasing the number of threads (*(a)* and *(b)*). Average queue size and average request size for an out-of-memory experiment *(c)*. In all cases we use the Optane SSD.

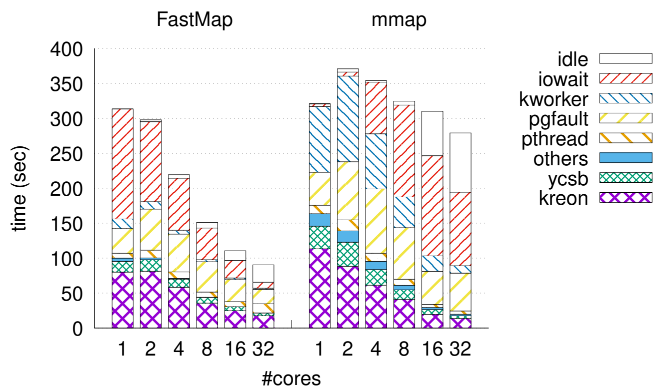


Figure 9: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment for LoadA YCSB workload, with an increasing number of cores, an equal number of YCSB threads, and the Optane SSD.

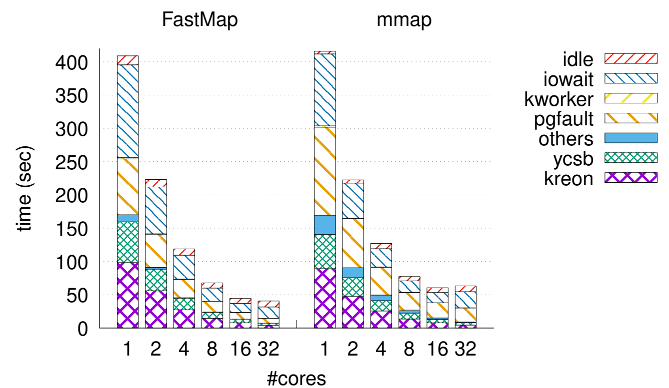


Figure 10: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment with the RunC YCSB workload, with increasing number of cores (and equal number of YCSB threads) and the Optane SSD.

server) *FastMap* achieves 3.08× higher throughput compared to *mmap* using LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 1.65× higher throughput compared to *mmap*. We see that even for the lower core counts, *FastMap* outperforms *mmap* significantly. Next, we provide an analysis on what affects scalability in *mmap* and how *FastMap* behaves with an increasing number of cores.

Figure 9 shows the execution time breakdown for the out-of-memory experiment with an increasing number of threads for LoadA. *kworker* denotes the time spent in the eviction threads both for Linux *mmap* and *FastMap*. *pthread* refers to pthread locks, both mutexes and read-write locks as described in Section 4. First, we observe here that in the case of Linux *mmap* both *iowait* and *idle* time increases. For *iowait* time, the small queue depth that *mmap* generates (discussed in detail later) leads to sub-optimal utilization of the storage device. Furthermore, the idle time comes from sleeping in mutexes in the Linux kernel. We also observe that the *pgfault* time is lower in *FastMap* and this is more pronounced with 32 threads. In summary, the optimized page-fault path results in 2.64× lower *pgfault* time and 12.3× lower *iowait* time due to higher concurrency and larger average request size. In addition, the optimized page-fault path results in 3.39× lower idle time due to spinning instead of sleeping in the common path. This is made possible as we apply per-core locks to protect our data structures, which are less contended in the common case. Similar to the previous figure, Figure 10 shows the same metrics for RunC. In this case the breakdown is similar both for *FastMap* and Linux *mmap*. With 32 threads the notable differences are in *pgfault* and *iowait*. Linux *mmap* spends 2.88× and 1.41× more time for *pgfault* and *iowait*, respectively. The difference in *pgfault* comes from our scalable design for the *memory-mapped I/O* path. As in this case both systems always issue 4KB requests (page size), the difference in *iowait* comes from the higher queue depth achieved on

average by *FastMap*.

Figure 8(c) shows the average queue depth and average request size for both *FastMap* and Linux *mmap*. Using 32 threads, *FastMap* produces higher queue depths for both LoadA and RunC, which is an essential aspect for high throughput with fast storage devices. With 32 threads in LoadA *FastMap* results in an average queue size of 39.2, while Linux *mmap* results in an average queue size of 17.5. Furthermore, *FastMap* also achieves a larger request size of 100.2 sectors (51.2KB) compared to 51.8 sectors (26.5KB) for Linux *mmap*. For RunC, the average request size is 8 sectors (4KB) for both *FastMap* and Linux *mmap*. However, *FastMap* achieves (with 32 threads) an average queue size of 13 compared to 3 for Linux *mmap*.

For all YCSB workloads, Kreon with *FastMap* outperforms Linux *mmap* by $2.48\times$ on average (between $1.25-3.65\times$).

MonetDB: In this section we use TPC-H over MonetDB, a column oriented DBMS that uses *memory-mapped I/O* instead of *read/write* system calls. We focus on out-of-memory workloads, using a TPC-H dataset with a scale factor $SF = 50$ (around 50GB in size). We limit available server memory to 16GB and we use the Optane SSD device. In all 22 queries of TPC-H, system-time is below 10%. The use of *FastMap* further decreases the system time (between 5.4% and 48.6%) leaving more CPU cycles for user-space processing. In all queries, the improvement on average is 6.06% (between -7.2% and 45.7%). There are 4 queries where we have a small decrease in performance. Using profiling we see that this comes from the *map_pages* function that is responsible for the fault-around page mappings, and which is not as optimized in the current prototype. In some cases we see greater performance improvements compared to the reduction in system time. This comes from higher concurrency to the devices (I/O depth) which also results in higher read throughput. Overall, our experiments with MonetDB show that a complex real-life memory-based DBMS can benefit from *FastMap*. The queries produce a sequential access pattern to the underlying files which shows the effectiveness of *FastMap* also for this case.

5.3 How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

In this section we show how underlying file system affects *FastMap* performance. Furthermore, we also evaluate the impact of batched TLB invalidations. For these purposes we use Silo [54], an in-memory key-value store that provides scalable transactions for multicores. We modify Silo to use a memory-mapped heap over both *mmap* and *FastMap*.

File system choice: Table 2 shows the throughput and average latency of TPC-C over Silo. We use both EXT4 and NOVA. We also use XFS and BTRFS but we do not include these as they exhibit lower performance. We see that *FastMap* with EXT4 provides $52.5\times$ and $75.9\times$ higher throughput using an NVMe and a pmem device respectively, compared to *mmap*. We also see similar improvement in the average latency of TPC-C queries. With NOVA and a pmem device, *FastMap* achieves $1.07\times$ higher throughput compared to EXT4. In all cases we do not use DAX *mmap*, as we have to provide DRAM caching over the persistent device. Therefore, *FastMap* improves performance of memory-mapped files over all file systems, although the choice of a specific file system does affect performance. In this case we see even larger performance improvements compared to Ligra and Kreon. Silo requires more page faults and it accesses a smaller portion of each page. Therefore, Silo is closer to a scenario with a single large file/device and a large number of threads generating page faults at random offsets. Consequently, it exhibits more of the issues we identify with Linux *mmap* compared to the other benchmarks: Kreon performs mostly sequential I/O for writes and a large part of a page is indeed needed when we do reads. From our evaluation we see that Ligra has better spatial locality compared to Silo and this explains the larger improvements we observe in Silo.

Figure 11 shows the breakdown of execution time for the previous experiments. In the case of Linux *mmap* with EXT4, most of the system time goes to buffer management: allocation of pages, LRUs, evictions, etc. In *FastMap*, this percentage is reduced from 74.2% to 10.3%, for both NOVA and EXT4. This results in more user-time available to TPC-C and increased performance. Finally, NOVA reduces system time

Table 2: Throughput and average latency for TPC-C.

	xput (kops/sec)	latency (ms)
mmap-EXT4-Optane SSD	4.3	7.43
mmap-EXT4-pmem	4.2	7.62
FastMap-*EXT4-Optane SSD*	226	0.141
FastMap-*EXT4-pmem*	319	0.101
FastMap-*NOVA-pmem*	344	0.009

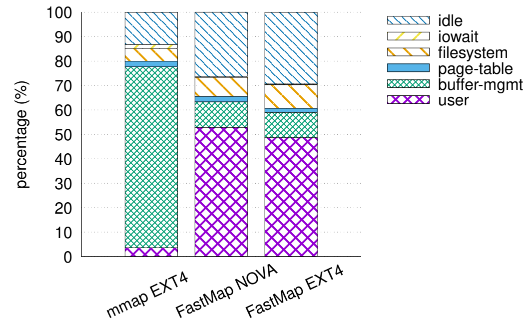


Figure 11: Execution time breakdown for Silo running TPC-C using different file systems and the *pmem* device.

compared to EXT4 and results in the best performance for TPC-C over Silo.

False TLB invalidations: *FastMap* uses batched TLB invalidations to provide better scalability and thus increased performance. Our approach reduces the number of calls to *flush_tlb_mm_range()*. This function uses Interprocessor Interrupts (IPI) to invalidate TLB entries in all cores and can result in scalability bottlenecks [3, 4, 16]. Batching of TLB invalidations can potentially result in increased TLB misses. In TPC-C over Silo, batching for TLB invalidations results in 25.5% more TLB misses (22.6% more load and 50.5% more store TLB misses). On the other hand, we have 24% higher throughput (ops/s) and 23.8% lower latency (ms). Using profiling, we see that without batching of TLB invalidations the system time spent in *flush_tlb_mm_range()* increases from 0.1% to 20.3%. We choose to increase the number of TLB misses in order to provide better scalability and performance. Other works [3,4,16] present alternative techniques to provide scalable TLB shootdown without increasing the number of TLB invalidations and can be potentially applied in *FastMap* for further performance improvements.

6 Related Work

We categorize related work in three areas: (a) replacing *read*/*write* system calls with *mmap* for persistence, (b) providing scalable address spaces, and (c) extending virtual address spaces beyond physical memory limits.

Using *memory-mapped I/O* in data-intensive applications: Both MonetDB [8] and MongoDB [13] (with *MMAP_v1* storage engine) are popular databases that use *mmap* to access data. When data fits in memory, *mmap* performs very well. It allows the application to access data at memory speed and removes the need for user-space cache lookups. Facebook's RocksDB [24], a persistent key-value store, provides both *read*/*write* and *mmap* APIs to access files. The developers of RocksDB state [26] that using *mmap* for an *in-memory* database with a *read-intensive* workload increases performance. However, they also state [25] that *mmap* sometimes causes problems when data does not fit in memory and is managed by a file system over a block device.

Tucana [42] and Kreon [43] are write-optimized persistent key-value stores that are designed to use *memory-mapped I/O* for persistence. The authors in [42] show that for a write-intensive workload the *memory-mapped I/O* results in excessive and unpredictable traffic to the devices, which results in freezes and increases tail-latency. Kreon [43] provides a custom *memory-mapped I/O* path inside the Linux kernel that improves write-intensive workloads and reduces the latency variability of Linux *mmap*. In this work, we address scalability issues and also present results for *memory-mapped I/O* with workloads beyond key-value stores.

DI-MMAP [22, 23], removes the swapper from the critical path and implements a custom (FIFO based) eviction policy using a fixed-size memory buffer for all *mmap* calls. This approach provides significant improvement compared to the default Linux *mmap* for HPC applications. We evaluate *FastMap* using more data-intensive applications, representative of data analytics and data serving workloads. In particular, our work assumes higher levels of I/O concurrency, and addresses scalability concerns with higher core counts.

FlashMap [27] combines memory (page tables), storage (file system), and device-level (FTL) indirections and checks in a single layer. *FastMap* provides specific optimizations only for the memory level and results in significant improvements in a file system and device agnostic manner.

2B-SSD [6] leverages SSD internal DRAM and the byte addressability of the PCIe interconnect to provide a dual, byte and block-addressable SSD device. It provides optimized write-ahead logging (WAL) over 2B-SSD for popular databases and results in significant improvements. FlatFlash [1] moves this approach further and provides a unified memory-storage hierarchy that results in even larger performance improvements. Both of these works move a large part of their design inside the device. *FastMap* works in a device-agnostic manner and provides specific optimizations in the operating system layer.

UMap [44] is a user-space memory-mapped I/O framework which adapts different policies to application characteristics and storage features. Handling page faults in user-space (using *userfaultfd* [31]) introduces additional overheads that are not acceptable in the case of fast storage devices. On the other hand, techniques proposed by *FastMap* can also be used in user-space memory-mapped I/O frameworks and provide better scalability in the page-fault path.

Similar to [14], *FastMap* introduces a read-ahead mechanism to amortize the cost of pre-faulting and improve sequential I/O accesses. However, our main focus is to reduce synchronization overheads in the common *memory-mapped I/O* path and enhance scalability on multicore servers. A scalable I/O path enables us to maintain high device queue depth, an essential property for efficient use of fast storage devices.

Byte-addressable persistent memory DIMMs, attached in memory slots, can be accessed similarly to DRAM with the processor *load*/*store* instructions. Linux provides Direct Access (DAX), a mechanism that supports direct mapping of persistent memory to user address space. File systems that provide a DAX mmap [17, 21, 56–58] bypass I/O caching in DRAM. On the other hand, other works [29] have shown that DRAM caching benefits applications when the working set fits in DRAM and can hide higher persistent memory latency compared to DRAM (by up to $\sim 3\times$). Accordingly, *FastMap* uses DRAM caching and supports both block-based flash storage and byte-addressable persistent memory. *FastMap* will benefit all DAX mmap file systems that need to provide

DRAM caching for *memory-mapped I/O*, as *FastMap* is file system agnostic.

Providing a scalable virtual address space: *Bonsai* [15] shows that anonymous memory mappings, i.e. not backed by a file or device, suffer from scalability issues. This type of memory mapping is mainly used for user memory allocations, e.g. *malloc*. The scalability bottleneck in this case is due to a contended read-write lock, named *mmap_sem*, that protects access to a red-black tree that keeps VMAs (valid virtual address spaces ranges). In the case of page faults, this lock is acquired as read lock. In the case of *mmap*/*munmap* this lock is acquired as write lock. Even in the read lock case, NUMA traffic in multicores limits scalability. *Bonsai* proposes the use of *RCU*-based binary tree for lock-free lookups, resulting in a system scaling up to 80 cores. *Bonsai* removes the bottleneck from concurrent page faults, but still serializes *mmap*/*munmap* operations even in non-overlapping address ranges.

In Linux, shared mappings backed by a file or device have a different path in the kernel, thus requiring a different design to achieve scalability. There are other locks (see Section 3.1) that cause scalability issues and *mmap_sem* does not result in any performance degradation. As we see from our evaluation of *FastMap*, using 80 cores the time spent in *mmap_sem* is 37.4% of the total execution time; therefore, *Bonsai* is complementary to our work and will also benefit our approach. Furthermore, authors in [32] propose an alternative approach to provide scalable address space operations, by introducing scalable range locks to accelerate non-conflicting virtual address space operations in Linux.

RadixVM [16] addresses the problem of serialization of *mmap*/*munmap* in non-overlapping address space ranges. This work is done in the *SV6* kernel and can also benefit from *FastMap* in a similar way to *Bonsai*.

The authors in [9] propose techniques to scale Linux for a set of kernel-intensive applications, but do not tackle the scalability limitations of *memory-mapped I/O*. In *pedsort* authors modify the application to use one process per core for concurrency and avoid the contention over the shared address space. In this paper we solve this issue at the kernel level, thus providing benefits to all user applications.

Extending the virtual address space over storage: The authors in [51] claim that by using *mmap* a user can effectively extend the main memory with fast storage devices. They propose a page reclamation procedure with a new page recycling method to reduce context switches. This makes it possible to use extended vector I/O – a parallel page I/O method. In our work, we implement a custom per-core mechanism for managing free pages. We also preallocate a memory pool to remove the performance bottlenecks identified in [51]. Additionally, we address scalability issues with *memory-mapped I/O*, whereas the work in [51] examines setups with up to 8 cores, where Linux kernel scales well.

FlashVM [49] uses a dedicated flash device for swapping virtual memory pages and provides flash-specific optimizations for this purpose. SSDAlloc [5] implements a hybrid DRAM/flash memory manager and a runtime library that allows applications to use flash for memory allocations in a transparent manner. SSDAlloc proposes the use of $16-32\times$ more flash than DRAM compared to FlashVM and to handle this increase they introduce a log-structured object store. Instead, *FastMap* targets the storage I/O path and reduces the overhead of *memory-mapped I/O*. *FastMap* is not a replacement for swap nor does it provide specific optimizations to extend the process address space over SSDs.

NVMalloc [55] enables client applications in supercomputers to allocate and manipulate memory regions from a distributed block-addressable SSD store (over FUSE [36]). It exploits the *memory-mapped I/O* interface to access local or remote NVM resources in a seamless fashion for volatile memory allocations. NVMalloc uses Linux *mmap*. Consequently, it can also benefit from *FastMap* at large thread counts combined with fast storage devices.

SSD-Assisted Hybrid Memory [41] augments DRAM with SSD storage as an efficient cache in object granularity for Memcached [38]. Authors claim that managing a cache at a page granularity incurs significant overhead. In our work, we provide an application agnostic approach at page granularity and we optimize scalability in the common path.

7 Conclusions

In this paper we propose *FastMap*, an optimized *memory-mapped I/O* path in the Linux kernel that provides a low-overhead and scalable way to access fast storage devices in multi-core servers. Our design enables high device concurrency, which is essential for achieving high throughput in modern servers. We show that *FastMap* scales up to 80 cores and provides up to $11.8\times$ more random IOPS compared to *mmap*. Overall, *FastMap* addresses important limitations of Linux *mmap* and makes it appropriate for data-intensive applications in multi-core servers over fast storage devices.

Acknowledgements

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the projects EVOLVE (Grant Agreement ID: 825061) and ExaNeSt (Grant Agreement ID: 671553). Anastasios Papagiannis is also supported by the Facebook Graduate Fellowship. Finally, we thank the anonymous reviewers for their insightful comments and our shepherd Sudarsun Kannan for his help with preparing the final version of the paper.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[3] 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.

[4] 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, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 211–224, USA, 2011. USENIX Association.

[6] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 425–438. IEEE Press, 2018.

[7] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

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

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 1–16, USA, 2010. USENIX Association.

[10] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[13] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[14] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[15] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[16] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[17] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud

serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] Laurent Dufour. Speculative page faults (Linux 4.14 patch). https://lkml.org/lkml/2017/10/9/180, 2017.

[20] Laurent Dufour. Speculative page faults. https://lwn.net/Articles/786105/, 2019.

[21] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[22] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 731–735, Nov 2012.

[23] Brian 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, March 2015.

[24] Facebook. RocksDB. https://rocksdb.org/. Accessed: June 4, 2020.

[25] Facebook. RocksDB IO. https://github.com/facebook/rocksdb/wiki/IO. Accessed: June 4, 2020.

[26] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed: June 4, 2020.

[27] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 580–591, New York, NY, USA, 2015. Association for Computing Machinery.

[28] INTEL. OPTANE SSD DC P4800X SERIES. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html. Accessed: June 4, 2020.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[30] jemalloc. http://jemalloc.net/. Accessed: June 4, 2020.

[31] Linux kernel. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt. Accessed: June 4, 2020.

[32] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 651–664, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.

[35] 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, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Linux FUSE (Filesystem in Userspace). https://github.com/libfuse/libfuse. Accessed: June 4, 2020.

[37] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox professional guides. Wiley, 2008.

[38] Memcached. https://memcached.org/. Accessed: June 4, 2020.

[39] MonetDB. https://www.monetdb.org/Home. Accessed: June 4, 2020.

[40] Null block device driver. https://www.kernel.org/doc/Documentation/block/null_blk.txt. Accessed: June 4, 2020.

[41] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. In *2012 41st International Conference on Parallel Processing*, pages 470–479, Sep. 2012.

[42] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[43] 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*, SoCC '18, pages 490–502, New York, NY, USA, 2018. ACM.

[44] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78, Nov 2019.

[45] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/. Accessed: June 4, 2020.

[46] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[47] pmem.io: Persistent Memory Programming. http://pmem.io/. Accessed: June 4, 2020.

[48] Jinglei Ren. YCSB-C. https://github.com/basicthinker/YCSB-C, 2016.

[49] Mohit Saxena and Michael M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 14, USA, 2010. USENIX Association.

[50] 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*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[51] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4):19:1–19:27, May 2016.

[52] TPC-C. http://www.tpc.org/tpcc/. Accessed: June 4, 2020.

[53] TPC-H. http://www.tpc.org/tpch/. Accessed: June 4, 2020.

[54] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[55] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 957–968, May 2012.

[56] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory and its extensions. *ACM Trans. Storage*, 9(3), August 2013.

[57] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[58] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.