Unified Address Translation for Memory-Mapped SSDs with FlashMap

Jian Huang Anirudh Badam[†] Moinuddin K. Qureshi Karsten Schwan

Georgia Institute of Technology [†]Microsoft Research

{jian.huang,moin,karsten.schwan}@gatech.edu {anirudh.badam}@microsoft.com

Abstract

Applications can map data on SSDs into virtual memory to transparently scale beyond DRAM capacity, permitting them to leverage high SSD capacities with few code changes. Obtaining good performance for memory-mapped SSD content, however, is hard because the virtual memory layer, the file system and the flash translation layer (FTL) perform address translations, sanity and permission checks independently from each other. We introduce FlashMap, an SSD interface that is optimized for memory-mapped SSD-files. FlashMap combines all the address translations into page tables that are used to index files and also to store the FTL-level mappings without altering the guarantees of the file system or the FTL. It uses the state in the OS memory manager and the page tables to perform sanity and permission checks respectively. By combining these layers, FlashMap reduces critical-path latency and improves DRAM caching efficiency. We find that this increases performance for applications by up to 3.32x compared to state-of-the-art SSD file-mapping mechanisms. Additionally, latency of SSD accesses reduces by up to 53.2%.

1. Introduction

A growing number of data-intensive applications use solid state disks (SSDs) to bridge the capacity and performance gaps between main memory (DRAM) and magnetic disk drives (disks). SSDs provide up to 5000x more IOPS and up to 100x better latency than disks [19]. SSDs provide up to 20TB capacity per rack-unit (RU) [56], whereas DRAM scales only to a few hundred GBs per RU [6, 8]. SSDs are used today as a fast storage medium to replace or augment disks.

An emerging approach to using SSDs treats them as a slower form of non-volatile memory. For example, NoSQL databases like MongoDB [39, 41], LMDB [35] (backend for OpenLDAP) and others [4, 36, 38] which are widely deployed [40] use SSDs via a memory-mapped file interface. There are three advantages to this approach. First, the virtual memory interface eases development. For example, MongoDB uses TCMalloc [51] to manage SSD-file backed memory to create data structures like its B-tree index, and for using the Boost template library. Second, SSDs are automatically tiered under DRAM by the OS's memory manager and finally, memory that is backed by a file enables durability for data. Such hybrid memory systems have also been proposed in the academic community [5, 44, 45, 55, 57].

Using SSDs in this manner, unfortunately, is inefficient as there are three software layers with redundant functionalities between the application and NAND-Flash. The first of these, memory-level indirection, involves page table translations and sanity checks by the OS memory manager. The second of these, storage-level indirection, involves converting file offsets to blocks on the SSD and permission checks by the file system. The final one, device-level indirection, is for the flash translation layer (FTL) of the SSD. Redundant indirections and checks not only increase the latency of NAND-Flash accesses, but also affect performance by requiring precious DRAM space to cache indirection data across all the three layers.

In this paper, we present FlashMap, a holistic SSD design that combines memory, storage, and device-level indirections and checks into one level. This is a challenging problem because page table pages and OS memory manager state that form the memory-level indirection are process-specific, private, and non-shared resources while direct/indirect blocks (file index) that form the storage-level indirection in a file system, and the FTL that converts the logical block addresses to physical block addresses are shared resources. FlashMap introduces the following three new techniques to combine these layers without losing their functionality:

- FlashMap redesigns a file as a contiguous global virtual memory region accessible to all eligible processes. It uses page table pages as the indirect block index for such files where these page table pages are shared across processes mapping the same file.
- FlashMap enables sharing of page table pages across processes while preserving semantics of virtual memory protections and file system permissions by creating private page table pages only on demand.
- FlashMap introduces a new SSD interface with a sparse address space to enable storing of the FTL's mappings inside page table pages.

More importantly, *FlashMap preserves the guarantees of all the layers in spite of combining them into virtual memory*. We

ISCA'15, June 13-17, 2015, Portland, OR USA

 ISBN 978-1-4503-3402-0/15/06$15.00

DOI: http://dx.doi.org/10.1145/2749469.2750420

implement FlashMap in Linux for EXT4 on top of a functional SSD-emulator with the new interface proposed. Experimental results demonstrate that data intensive applications like NoSQL stores (Redis [46]), SQL databases (Shore-MT [49]) and graph analytic software (GraphChi [22]) obtain up to 3.32x better performance and up to 53.2% less latency.

The rest of this paper is organized as follows: The background and motivation of our work are described in Section 2. The design and implementation of FlashMap are presented in Sections 3, and 4 respectively. Section 5 presents the evaluation results. Section 5.3 analyzes the cost-effectiveness of using SSDs as memory compared to DRAM. Related work is described in Section 6, and our conclusions from this work are given in Section 7.

2. Background & Motivation

To meet the high-capacity needs of data-intensive applications, system-designers typically do one of two things. They either scale up the amount of DRAM in a single system [23, 33] or scale out the application and utilize the collective DRAM of a cluster [2, 42, 60]. When more capacity is needed, SSDs are useful for scale-up scenarios [10, 16, 17, 32, 34] and for scale-out scenarios [3, 7, 12]. Adding SSDs not only improves performance normalized for cost (as shown in Section 5.3), but also improves the absolute capacity.

Today SSDs scale up to 10TB per system slot [19] (PCIe slot) while DRAM scales only to 64GB per slot (DIMM slot). Even though systems have more DIMM slots than PCIe slots, one would require an impractically high (160) number of DIMM slots to match per-slot SSD density. SSDs provide up to a million IOPS at less than 100μsec and lack seek latencies. Such performance characteristics make SSDs more similar to DRAM than to disks. Moreover, SSDs are non-volatile. Therefore, data-intensive applications are using SSDs as slow, but high-capacity non-volatile memory [39, 35, 38, 5, 45] via memory-mapped files. Such an approach has the following three advantages.

First, virtual memory interface helps existing in-memory applications adopt SSDs with only a few code changes. For example, we find that less than 1% of the code has to be changed in Redis (an in-memory NoSQL store built for DRAM) to use SSD-backed memory instead of DRAM. On the other hand, more than 10% of the code has to be modified if SSDs are used via read/write system calls. Further benefits of the interface's ease are shown in Section 5.3, in our previous work [5] and is also evident from the fact that widely deployed databases like MongoDB map SSD-files and build data structures such as BTree indexes using TCMalloc [51] and Boost [9].

Second, memory-mapped access ensures that hot data is automatically cached in DRAM by the OS and is directly available to applications via `load/store` instructions. To get such a benefit without memory-mapping, an application would have to design and implement custom tiering of data between DRAM and SSD which can take months to years depending on application complexity. Finally, file-backed memory as opposed to anonymously-mapped memory (SSD as a swap space) allows applications to store data in the file durably and exploit other file system features such as atomicity (via transactions and journaling), backup, space management, naming and sharing.

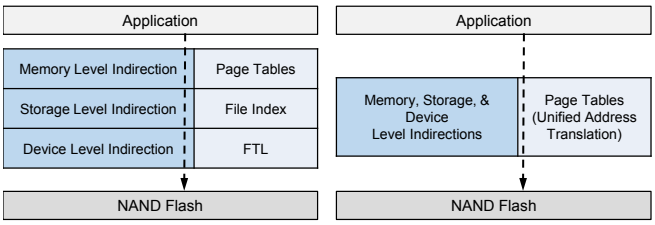

Figure 1: Comparison of (a) conventional memory-mapped SSD-file's IO stack and (b) FlashMap that combines all the address translations for mapping files on SSD into page tables.

Unfortunately, existing OSes and SSDs are not optimized for this style of using the SSD. There are three distinct software layers with separate indirections between the application and NAND-Flash. The separation of the virtual memory, the file system, and the FTL as shown in Figure 1(a) reduces performance and increases latency. This paper presents a holistic SSD designed from scratch for memory-mapped SSD-files that combines these layers.

2.1. Overhead from Redundant Software

To service a page fault in a memory mapped region from a file on an SSD, three types of address translations are required. First, the CPU traverses the page tables (memory-level indirection) to trigger the page fault. Later, the file system uses indirect blocks to translate the faulting file offset to a block on the SSD (storage-level indirection). Finally, the FTL converts this block address to an actual NAND-Flash level address (device-level indirection). Multiple layers of indirection not only increase the latency but also decrease the performance of applications. Each layer wastes precious DRAM space for caching address translation data. The latency increases further if the required address translation is not cached in DRAM. Locking all the address translation data in DRAM is not a feasible option as one would require as much as 60GB of DRAM for a 10TB SSD[1].

Even if the address translation data is cached in DRAM, the software latency is still significant (5–15μsec in each layer) as each layer performs other expensive operations. For example, the memory manager has to check if the faulting address is in an allocated range, the file system has to check if the process has access to the file (checks that can be efficiently enforced using permission bits in page tables that are always enforced by the CPU) and FTLs with sparse address spaces [20] have to check allocation boundaries (checks that can be efficiently performed by the memory manager itself). There is a need for a new SSD design that is optimized for memory-mapped SSD-

[1] A minimum of eight bytes per indirection layer per 4KB page

files, one that uses a single software layer (virtual memory), one address translation, one permission check (in page tables), and one sanity check (in memory manager).

2.2. Challenges for a Combined Indirection Layer

Memory-level indirection SSD-files have to be mapped with small pages (4KB) as large pages are detrimental to the performance of SSDs. An x86_64 CPU provides memory usage (read/fault and write/dirty) information only at the granularity of a page. Therefore, smaller page sizes are better for reducing the read/write traffic to the SSD. For example, our enterprise class SSD provides 700K 4KB random reads per second, while it provides only 1,300 2MB random reads per second. Thus using a 4KB page size means that the size of the page tables would be about 20GB for a 10TB dataset. While keeping page table pages only for the pages in DRAM can reduce the space required, it does not reduce the software-latency of handing a page fault which is dominated by the layers below the memory manager. To reduce this latency, we propose performing all the address translations with *page table pages* as they are an x86_64 necessity for file-mapping and cannot be changed or removed. Moreover, the virtual memory protection bits can be exploited for *all* permission checks and the memory allocation metadata in the OS memory manager can be exploited to perform *all* sanity checks.

Storage-level indirection is the direct/indirect blocks of a file system. For typical block sizes (2–8KB), this indirection layer requires 10–40GB of space for a 10TB SSD. Larger blocks unfortunately decrease DRAM caching efficiency and increase the traffic from/to the SSD. Extent-based file indexes such as the one used in EXT4 [18] can reduce this overhead. However, using such an index does not remove all the file system overhead from the IO-path. It is well known that permission checks of file systems increase latency in the IO-path by 10–15μs [13, 43]. A combined memory and storage-level indirection layer would not only eliminate a level of indirection but would also perform all the necessary checks efficiently by using the protection bits in the page tables.

Device-level indirection. NAND-Flash supports three operations – read, write, and erase. Reads can be performed at a granularity of a page (4KB), which can be written only after they are erased. Unfortunately, erases can be performed only at a large granularity of a block (eight or more pages at a time). Moreover, each block is rated only for a few thousand erases and therefore it is vital for the blocks to age uniformly. SSDs employ a log-structured data store with garbage collection (GC) [1] using indirections in the FTL for out-of-place writes and ensuring uniform wear. To improve performance and lifetime, high-performance SSDs implement such logs by employing a fine-granular and fully-associative page-level index [20, 27]. Such an index at a granularity of a page (4KB) requires more than 20GB of space for a 10TB SSD.

Traditionally, FTLs have cached their mappings in embedded SRAM/DRAM to provide predictable performance.

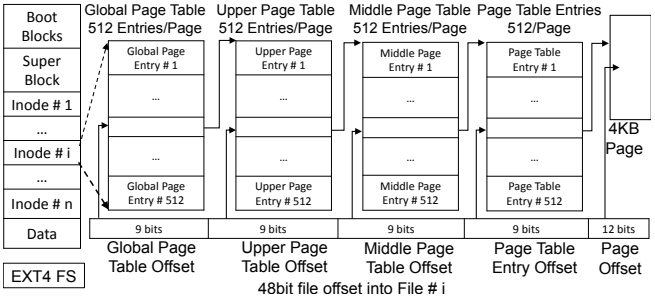

Figure 2: FlashMap uses a page table design for indexing files.

Unfortunately, it is not possible to provision large amounts of RAM inside SSDs. Therefore, high-capacity SSDs store the mappings in the host [20] where DRAM scales better. FlashMap leverages this SSD design pattern to combine the FTL mappings with indirections in the higher layers.

Combining page tables with storage and device-level indirections, however, is challenging because page table *pages* are process specific and private entities while the remaining indirections are system-wide entities that are shared by all processes. Furthermore, page table pages cannot be shared frivolously across processes because it may lead to false sharing of memory and violate permissions and protections. To address these problems, FlashMap introduces a new virtual memory design where the page table pages needed for mapping a file belong to the file system and are system wide resources shared across processes mapping the same file. FlashMap enforces file permissions and virtual memory protections as required at a low-overhead by creating process-specific private page table pages only on demand. This design helps FlashMap unify the memory, storage and device interfaces (Figure 1(b)) without changing *any* guarantees from virtual memory, file system, or FTL.

3. FlashMap Design

To combine the memory and storage-level indirections, we re-imagine a file as a contiguous virtual memory region of an abstract process. The region is indexed by an x86_64 style four-level page table as opposed to the traditional direct/indirect block/extent representations that file systems use. Figure 2 illustrates the indirect blocks in our file design – *the rest of the file system, however, remains unaltered*. The 48-bit physical frame numbers (PFN) in this page table are the block pointers that the file system uses to index the file. Since we treat a file as a region of memory of an abstract process, we will use page and block interchangeably. We will refer to the file index as shared page tables.

Such files can be accessed via the `POSIX` file API without any application changes. Most file systems are designed to abstract away the indexing from the API and the rest of the file system. We leverage this abstraction to preserve the rest of the file system and `POSIX` API. When mapping such files, however, *necessary shared page table pages have to be borrowed by the process* in contrast to traditional file-mapping

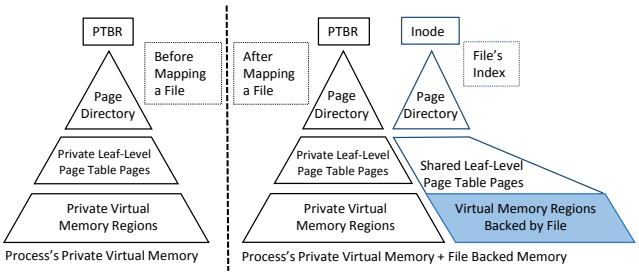

Figure 3: File brings leaf-level page tables with itself to a process that maps it. Higher-level page table pages are not shared, they are created on-demand for mapping the file.

and memory management techniques where private page table pages are created for the process.

At a first glance this problem might seem simple and that it can be solved by grafting the process's page table with as much of the shared page table pages as possible. However, the solution is not so simple. If two processes map the same file with different permissions (READ_ONLY vs. READ_WRITE), then sharing page table pages by frivolous grafting can violate file system permissions. To address these problems in a systematic manner, we introduce the notion of Semi-Virtual Memory which is a mechanism to share some of the shared page table pages across processes that map the same file with different file-level permissions.

3.1. Preserving File System Permissions

Semi-Virtual Memory is a method to share only the leaf-level page table pages (LL-PTP) that contain page table entries (PTE) across processes that map the same file. When a process maps a file, only the LL-PTPs of the shared page table of the file are borrowed and grafted into the process's page table. The rest of the page table pages (page directory pages) needed for the mapping are created for the process afresh and deleted when the process unmaps the file.

The permissions set in the higher level page table entries (page global, middle, and upper directory entries) override the permissions set at the page table entries in x86_64. Therefore, private copies of higher-level page table pages can be exploited to implement custom file-level permissions during mapping. It helps to think of this memory virtualization at the granularity of a file rather than a page, hence the name Semi-Virtual Memory. Not sharing higher-level page tables would increase the memory overhead of page tables by only a negligible amount. The branching factor of x86_64 page tables is 512 (512 entries per 4KB page), and the overhead is less than 0.5%. Figure 3 shows how only the LL-PTP of the shared page tables are borrowed from the file system when a file is mapped into a process. Rest of the higher-level page table pages are created for the process like the way they are in traditional OSes.

FlashMap avoids false sharing of file permissions by design. Two or more files are mapped to the same process such that the page table entries required for mapping these files are never on

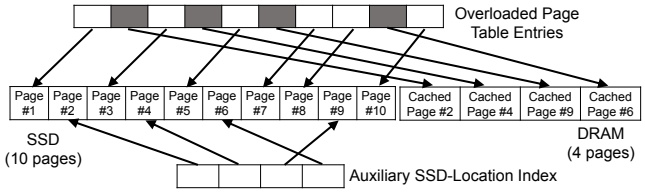

Figure 4: Page table entries are overloaded to store both SSD and DRAM locations of pages. When a page table entry stores the DRAM location of a cached page, the corresponding auxiliary SSD-location entry remembers it's SSD-location.

the same LL-PTP. This requires that file boundaries be aligned to 2MB (512x4KB span) in virtual memory space. FlashMap is designed for x86_64 where there is ample virtual memory available and therefore, we do not see this requirement as a major overhead. This design helps separate permissions for each file mapped to the same process.

This enforcement does not violate the `POSIX` compliance of `mmap`. `POSIX mmap` specifies that the OS may pick an address of its choosing to map the file if the address requested by caller is not available. However, as this scheme itself is deterministic, it is still possible for a process to map a file to the same address across reboots.

In traditional OSes, the separation of the memory and storage-level indexes meant that the memory manager and the file system needed to interact with each other only for data. In a system like FlashMap, they have to additionally collaborate to manage the shared LL-PTPs of the files that are currently mapped. PTE behavior must remain the same for user space in spite of this dual role of LL-PTPs

3.2. Preserving PTE Behavior

FlashMap overloads the PTE. When a page of a file is in DRAM, the corresponding PTE in the LL-PTP is marked as resident, and contains the address of the physical page in DRAM where the page resides. On the other hand, when the page is not cached in DRAM, the PTE in the shared LL-PTP is marked as not-resident, and contains the address of the page on the SSD.

The SSD-location of the page must be stored elsewhere while the page is cached in DRAM. We design an auxiliary index to store the SSD-locations of all the pages cached in DRAM. The auxiliary index is implemented using a simple one-to-one correspondence between DRAM pages and the SSD-Location of the block that the DRAM page may hold – a simple array of 8 byte values. It must be noted that the size of this array is the same as the number of pages in DRAM and therefore is not significant. For example, for a typical server with 64GB DRAM, this auxiliary index would require only 128MB and can be stored in DRAM – an overhead of less than 0.25%. Figure 4 demonstrates how the overloaded page table entries and auxiliary SSD-location index remember the location of all the pages (on SSD or cached in DRAM).

While Semi-Virtual Memory allows processes to adhere

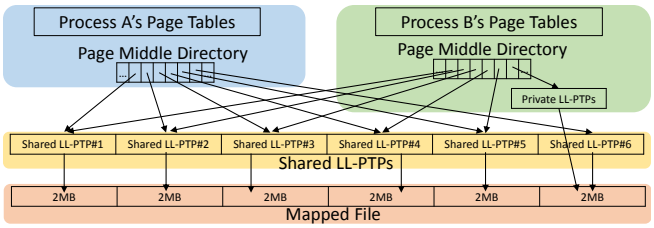

Figure 5: Processes A and B map the same file. However, Process B creates custom memory protections for a small memory region using a private leaf-level page-table page (LL-PTP).

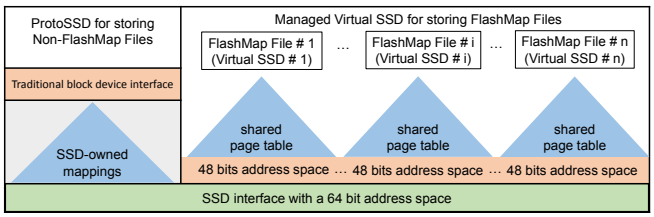

Figure 6: FlashMap uses an SSD with a sparse address space. A portion of this address space is virtualized with a separate device-level indirection. This portion is used for implementing non-performance critical data. Rest of the address space is used for storing the data of FlashMap files. This address space uses shared page tables for indirections

to file-level permissions, it does not provide one of the crucial properties of virtual memory – page-granular memory protection via `mprotect`. The sharing of LL-PTPs between processes means that the protection status of individual pages is also shared. This can violate the semantics of memory protection.

3.3. Preserving Memory Protection Behavior

To preserve the `POSIX` memory protection (`mprotect`) behavior, FlashMap simply disables Semi-Virtual Memory for the LL-PTPs of *only the virtual memory ranges that require custom access permissions only for the requesting process*. If a process requires custom memory protection for a single page, then the OS creates a private LL-PTP on demand for the encompassing virtual memory range that contains this page – the minimum size of such a region would be the span of an LL-PTP in x86_64 (2MB = 512x4KB span). These regions of memory are managed similar to shared memory in operating systems where the memory-level and storage-level indexes are separate. We call these regions "saturated virtual memory".

We believe that saturated virtual memory regions will not increase the memory overhead of address translation significantly. Basu *et al.* [8] report that for many popular memory-intensive applications, less than 1% of memory requires custom per-page protections. Moreover, our focus is on high-performance, in-memory applications where sharing files across processes with differing protections is a rare scenario.

Saturated virtual memory is depicted in Figure 5. Processes A and B map the same file, however Process B requires custom protection status for a page. FlashMap creates a private LL-PTP for the encompassing 2MB region to enforce the protection. The implementation details and the necessary data structures to support this are presented in Section 4.

3.4. Preserving the FTL Properties

FlashMap only changes the interface between the OS and the SSD. The rest of the SSD remains intact. FlashMap requires an SSD that can store only the mappings of the blocks of a file on the host in the shared LL-PTPs. The rest of the mappings – of file system metadata and other metadata – are managed by the SSD itself. Data dominates metadata in our applications which usually map large files to memory. Therefore, having separate translations inside the SSD for metadata of the file system does not add significant overhead.

We propose a virtualized SSD architecture, where the SSD and the file system share indirections *only* for data blocks of files. For each file, FlashMap creates a virtual SSD (48-bit address space) whose blocks have a one-to-one correspondence with the blocks of the file. The mappings of this address space are stored by FlashMap in shared LL-PTPs. Most of the space and IOPS of an SSD, in our scenario, will be spent towards data in large memory-mapped files, and very little on file system metadata (boot blocks, super blocks, inodes and etc.). A one-to-one mapping between FlashMap files and virtual SSDs allows the SSD driver to have a trivial way to store performance-critical mappings in the shared page table *without affecting the metadata design of a file system.*

Virtual SSDs are carved out of a larger virtual address space that the SSD driver implements. Similar to several high-performance SSDs [20], we design an SSD with a 64-bit address space, and carve out smaller virtual SSDs with 48-bit contiguous address spaces from it. The first 48-bits worth of this address space is used for implementing a virtual SSD whose FTL indirections are managed by the SSD driver itself. The file system can use this as a traditional block device for storing non-performance-critical data: including metadata and the on-SSD copy of the shared LL-PTPs. We call this the proto-SSD.

For each 48-bit address range, the file system must remember its allocation status so that they can be recycled and reused as files are created and deleted. The file system maintains a simple one-to-one table to represent this information using some space in the proto-SSD. A 64-bit address space allows 2^{16} 48-bit contiguous address spaces. Therefore, this table requires only tens of megabytes of space. Directories and smaller files are also stored on the proto-SSD.

FTL mappings in LL-PTPs. The rest of the 64-bit address space is used for creating virtual SSDs. Each virtual SSD carves a contiguous 48-bit address space for itself and is used by the file system to implement a logically contiguous file – a one-to-one correspondence between a FlashMap file and a virtual SSD. The SSD driver is designed to use the x86_64 style four-level page table of the file as the data structure for storing the indirections. *The indirections for the proto-*

SSD are implemented and managed by the SSD driver, while the indirections for every virtual SSD are stored inside the shared LL-PTPs of FlashMap. However, the SSD driver can query and update the shared LL-PTPs for each virtual SSD to perform read, write, and garbage-collection operations in an atomic fashion by coordinating with the OS memory manager. Figure 6 illustrates the address space at the SSD level. This design enables the SSD to store its mappings on the host, but not forgo functionality.

FlashMap files are directly mapped to virtual SSDs. New "file system to SSD calls" are required to help the file system leverage virtual SSDs:

- `create_virtual_ssd` allows a file system to create a new virtual SSD from the SSD driver. The file system updates its metadata to remember that this file uses FlashMap. It also remembers the virtual SSD that this file uses.
- `delete_virtual_ssd` is called by the file system when it wishes to delete a virtual SSD because the file is deleted from the file system.

The following operations are supported over a virtual SSD that the memory manager implements atomically:

- `delete_virtual_block` is the TRIM operation.
- `read_virtual_block` reads a block from the SSD.
- `write_virtual_block` writes a block to the SSD.

The log-management, error detection/correction, garbage collection algorithm and other SSD level functionalities are untouched. The actual garbage collection and log management algorithms we use in our implementation are described in more detail in Section 4. The following functions are used by the FTL to access the mappings inside LL-PTPs atomically:

- `update_mapping` allows the SSD driver to change the logical to physical mapping of a particular virtual block of a particular virtual SSD.
- `read_mapping` is used by the SSD to obtain the physical page address of a virtual page of a virtual SSD.
- `lock_mapping` is used by the SSD to lock access to the virtual page. Any accesses will lead to a fault. Faults are not processed until the SSD releases the lock.
- `unlock_mapping` is used by the SSD to unlock access to the virtual page. Fault that occurred since locking are processed.

3.5. Preserving Consistency & Durability Guarantees

The consistency and durability guarantees of a file system that adopts FlashMap style indexing are unaltered. FlashMap leverages in built abstraction of index from data management built in to file systems and preserves the manner in which `fsync()`, journaling or transactions work. However, the shared page tables which are cached in memory need durability and consistency without affecting performance.

LL-PTP management. *The shared pages tables for all the virtual SSDs/FlashMap files are cached on-demand in the host's DRAM*. A process mapping a file has private higher-level page table pages locked into DRAM, but LL-PTPs are fetched only on demand and unused ones are retired back to the SSD. An LL-PTP is marked as used when at least one page indexed by it is detected as recently used by the OS's memory manager (details presented in Section 4).

Durability for LL-PTPs. The LL-PTPs and the auxiliary SSD-index are constantly updated as the GC compacts live data and as new data is written out of place. However, writing them to the SSD everytime they are modified would increase the write traffic. Therefore, a more scalable approach to maintaining the durability of page tables is needed.

We leverage out-of-band space available on SSDs for this purpose. Most SSDs have per-page out-of-band space to store metadata [61]. We expand this space to store reverse-mapping information that can be used to regenerate the indirection table in case of a crash. For each page on the SSD, we store its virtual address in the 64-bit SSD address space where it is mapped. These reverse mappings *only need to be modified when pages are written or moved by the GC*. A sequential scan of the reverse mappings of all the pages in the right order is sufficient to recover the shared page table. The ordering is determined using timestamps that are written along with the reverse-mappings. Note that they are only stored on the SSD, and require as little as 10 bytes (8 for the mapping and 2 for the timestamp) per page. Dirty page table pages are frequently checkpointed in bulk to the SSD so that the amount of data to scan after a crash is no more than a few GBs.

4. FlashMap Implementation

We modify EXT4 and the memory manager in Linux to combine their memory and storage level indexes using Semi-Virtual Memory in combination with overloaded page tables, auxiliary SSD-Location index, and saturated virtual memory. We also implement a wrapper that can be used to convert any legacy SSD into one that provides virtual SSDs and a proto SSD. FlashMap is the combination of these two systems.

Modified EXT4 Index. We implement the shared page tables as the file index in EXT4 by replacing the default indirect (single/double/triple) block representation as depicted in Figure 2. We populate the page table of each file as the file grows contiguously in a virtual SSD's address space. The index implementation is abstracted from the rest of the file system in such a way that traditional POSIX file APIs, page replacement, and other file system entities work seamlessly.

Augmented Memory Manager. We implement a special memory manager helper kernel module that manages physical memory for all the virtual memory regions that map FlashMap files. This contains a page fault handler that brings the relevant data/page tables into DRAM. The module also manages the LL-PTPs. It also maintains the auxiliary SSD-Location index. This module also interacts with Linux's page replacement scheme to identify least recently used pages [21] and LL-PTPs of the file that can be sent back to the SSD.

The memory manager also implements the modified versions of `mmap`, `munmap`, and `mprotect` by intercepting these

system calls. It implements semi-virtual memory and saturated virtual memory. We use a red-black tree to remember the virtual memory regions that require saturated virtual memory and handle the page faults in these regions as if the memory and storage-level indirections are separate. Finally, this module implements the functions required by the SSD for atomically updating and reading the shared LL-PTPs and the auxilliary SSD-index.

SSD Emulator Our physical SSD has four requirements that are different from traditional SSDs. First, it should expose a 64-bit address space like some existing SSDs [20, 28]. Second, it should export a proto-SSD that looks like a block device to the file system like some existing SSDs [61]. Third requirement is the per-page out-of-band space for implementing the reverse-mappings similar to other recent devices [14, 48]. The final requirement is the *only one that we newly propose*: virtual SSDs with the APIs in Section 3.4.

We implement a static software wrapper for any SSD such that it functionally emulates our proposed SSD interface. In the wrapper, we implement a 64-bit sparse address space that is used to implement a log-structured page-store on the unmodified SSD. The wrapper constructs its own x86_64 style page table for indexing the proto-SSD. For the rest of the address space, it relies on the API designed in Section 3.4 to access and manage the shared page tables.

The wrapper implements a read-modify-write style GC to emulate log-structured and page-mapped SSDs. The SSD is organized as a sequence of chunks (128KB/erase granularity). We reserve a page at the head of each chunk to implement the out-of-band space. This header page is never allocated to the file system. We implement a log-structured system of chunks (each with 1 out-of-band data page and 31 data pages) with a read-modify-write style garbage-collector as used by other SSD based systems [5, 7, 44, 61]. In the read phase, the GC reads a chunk into memory. For each page in this chunk, it uses the reverse-mapping in the out-of-band metadata of the page to match against the forward-mapping stored in the shared LL-PTPs. If it is a match, then this page is still valid, if not then the page is garbage.

In the modify phase, the wrapper takes any available dirty pages from the memory manager that have not been recently used (and other synchronously written pages) and overwrites the garbage pages in the chunk's in-memory copy. In the write phase, the wrapper writes the chunk's in-memory copy back to the SSD along with the header page containing the new reverse-mappings for the recently written dirty pages. The header page also contains a timestamp that is used during recovery for determining the order in which pages were written to the SSD. Finally, it modifies the shared page table entries of the dirty pages that were written to new locations and marks them as clean with the help of the augmented memory manager. The GC then advances to the next chunk. Our SSD emulator adequately demonstrates that FTL mappings can be stored inside page tables without compromising on garbage collection and other FTL requirements.

NoSQL Key-Value Store (Section 5.1.1)		
Software		Redis
Workload		Redis benchmark, YCSB (Yahoo Cloud Service Benchmarks)
Graph Computation (Section 5.1.2)		
Software		GraphChi
Workload		PageRank, Connected-component labeling
Graph Dataset	Twitter	social networks, 61.5 million vertices, 1.5 billion edges.
	Friendster	ground-truth communities, 65.6 million vertices, 1.8 billion edges.
	MSD	Million Song Dataset (MSD) has 1 billion records of songs' metadata.
SQL Database (Section 5.2)		
Software		Shore-MT (open-source database manager)
Workload		TPCC, TATP, TPCB

Table 1: Real workloads used in our experiments.

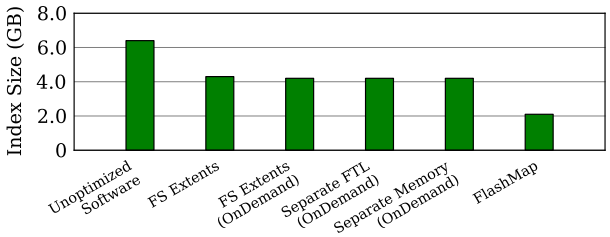

Figure 7: Index size for 1TB SSD.

5. Evaluation

The aim of our evaluation is three fold: (1) To demonstrate that FlashMap improves performance by making efficient usage of DRAM, (2) To demonstrate that FlashMap reduces latency and (3) To demonstrate that using SSDs as memory is a cost-effective method to increase capacity vs. using DRAM. We compare FlashMap with the following systems:

Unoptimized Software: Unmodified `mmap` is used to map a file on unmodified EXT4 file system without extents on the proto-SSD. Private page tables are created by `mmap`. The proto-SSD has its FTL locked in DRAM similar to high-performance FTLs such as Fusion-io's ioMemory [20, 27]. Rest of the indirections are fetched on demand.

FS Extents: This is the same software and hardware stack as above but with file system extents (128MB) enabled in EXT4. An improved version is **FS Extents (OnDemand)** in which the FTL mappings are fetched to DRAM from flash on-demand similar to DFTL [24].

Separate FTL (OnDemand): Page tables and the file system index are combined but the FTL remains separate. All the indirections are, however, fetched on demand.

Separate Memory (OnDemand): The file system and the FTL are combined similar to existing systems such as Nameless Writes [61] and DFS [28] where the memory-level indirections remain separate. However, all the indirections are fetched on demand to DRAM.

The physical machine used for experiments has two Intel

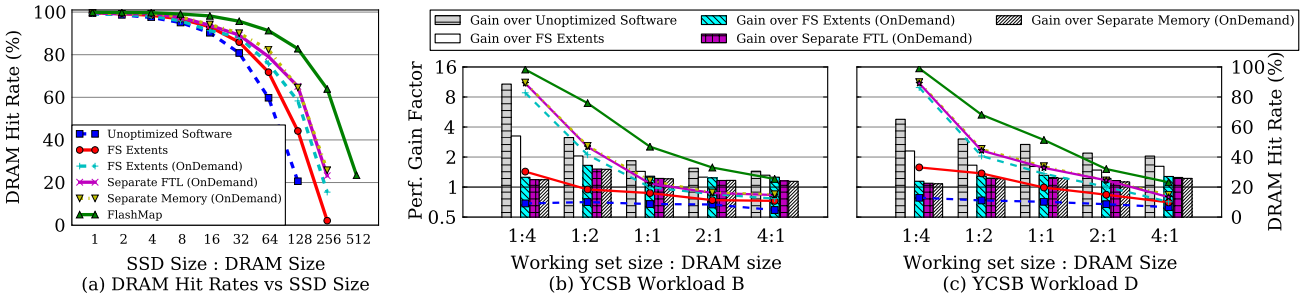

Figure 8: Memory saved by FlashMap helps applications obtain higher hit rates compared to other systems. (a) For DRAM:SSD ratio of 1:64 or more, FlashMap provides up to 4.01x and 1.28x higher hit rate than Unoptimized Software and Separate Memory (OnDemand). For throughput of photo-tagging (b) and status-update (c) workloads with various working set sizes, FlashMap performs up to 12.33x and 1.51x better than Unoptimized Software and Separate Memory (OnDemand) respectively.

Xeon processors each with 6 cores running at 2.1 GHz and 64 GB DRAM. Samsung 840 EVO SSDs are used for experimentation. We use the SSD emulator mentioned in Section 4 to convert the regular SSDs to the ones that support proto SSD. For the experiments in Section 5.2, we emulate high-end SSDs (e.g., PCM-based SSDs) using DRAM with added latency.

5.1. Benefits from Saving DRAM

We first examine the total size of indirections in each system per TB of SSD used. The results are shown in Figure 7. FlashMap requires close to 2GB while unoptimized software requires more than 6GB of indirections per TB of SSD. For 10TB SSDs available today, unmodified software would require more than 60GB of metadata. While more efficient than Unoptimized Software, FS Extents and Separate Memory (OnDemand) are still more than twice as expensive as FlashMap in terms of metadata overhead. This indicates that for a given working set, other methods would require 2-3x more amount of DRAM to cache all the indirections needed for the data in a given working set. For large working sets, the DRAM savings translate to higher performance.

To quantify performance benefits from DRAM savings, a number of experiments are performed. We perform microbenchmark experiments and conduct experiments with key-value stores and graph analytic software to quantify such benefits for real-world workloads. We vary the size of the SSD, the amount of DRAM and/or the working set size to understand the limits of benefits from a technique like FlashMap.

We first begin by examining the performance of each system for a microbenchmark workload as we increase the size of the SSD – DRAM is kept constant. We perform random reads over 0.5 million pages (4KB) selected uniformly at random from a file that spans an entire SSD. The size of the SSD is increased from 2GB to 1TB as the DRAM in the system is kept constant at 2GB (after subtracting the memory needed by the OS and the auxiliary index). The 0.5 million pages (2GB total) that form the workload are first sequentially accessed to warm the DRAM.

Ideally, the cache should hold all the pages and all the read operations should be cache hits in DRAM. However, indirections occupy memory, and therefore cache hits in DRAM decrease. The measured cache hit rates are shown in Figure 8(a). The results demonstrate that the additional DRAM available to FlashMap helps the workload obtain a higher hit rate in DRAM. For a realistic DRAM:SSD ratio between 1:64 and 1:128, FlashMap obtains up to 4.01x higher hit rates. However, real-world workloads are not random. To understand how DRAM savings translate to performance benefits for applications, we conduct the next set of experiments.

5.1.1. Benefits for Key-Value Stores

We evaluate the benefits from using FlashMap for a widely used NoSQL key-value stores named Redis [46, 58]. Redis is an in-memory NoSQL storage engine and in its simplest form is similar to Memcached [37]. We modify Redis (less than 20 lines of code) to use SSD as memory, and run two YCSB [15] workloads.

YCSB [15, 59] is a framework for evaluating the performance of NoSQL stores. It includes six core workloads, workload B and D are used in our experiments to test the performance of key-value stores in the face of skewed workloads. Workload B has 95% read and 5% update with zipfian distribution, which represents photo tagging applications: reading tags happens more frequently than adding a tag; workload D has 95% read and 5% insert with a bias towards records that are created recently. It models status updates on social networks.

The size of key-value pair is 1 KB (by default), and the number of client threads running in YCSB is 50. We conduct 1 million operations against varied amount of "hot" key-value pairs for each workload while keeping the DRAM:SSD ratio at 1:128 (64GB DRAM : 8TB SSD). We leverage the request distribution parameter in YCSB to adjust working set sizes, and plot the throughput for both the workloads in Figure 8.

FlashMap (9.5–62.7K TPS/single thread) provides up to 12.33x, 3.81x and 1.51x better throughput compared to Unoptimized Software (5.8–6.6K TPS/single thread), FS Extents (7.2–19.3K TPS/single thread) and Separate Memory (On-

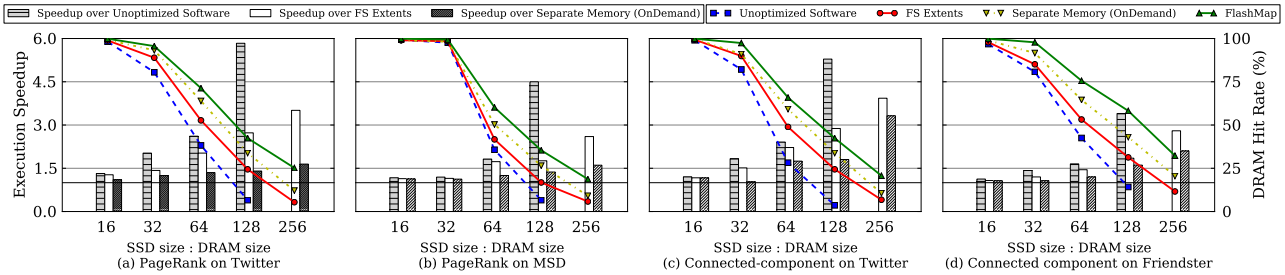

Figure 9: Improvements for analytics on Twitter, Friendster and MSD dataset, with varied DRAM size. Compared to Separate Memory (OnDemand), FlashMap performs 1.15–1.64x better for PageRank, and up to 3.32x better for the connected component labeling.

Demand) (8.3–52.9K TPS/single thread), due to the fact that FlashMap provides up to 7.59x higher hit rate for the YCSB workload B as shown in Figure 8(b). This is because of the fact that the less metadata overhead of FlashMap leads to more effective amount of DRAM being available for caching application data. As the working set size increases, the hit rates increase obtained by using FlashMap decreases. However, FlashMap still provides 14.5% higher throughput than the Separate Memory (OnDemand) even for working sets as large as four times the size of DRAM.

The results of workload D demonstrate the similar trend in Figure 8(c). For such a workload, FlashMap (51.5-19.0K TPS) gains 2.04–4.87x, 1.29–2.08x, 1.08–1.22x higher throughput when compared to Unoptimized Software (10.8-9.3K TPS), FS Extents (22.4-11.8K TPS), Separate Memory (OnDemand) (47.7-15.6K TPS). This demonstrates that by adding SSDs to the memory/storage hierarchy with FlashMap is beneficial for real-world applications because it reduces the programming effort and still provides significant performance benefits.

5.1.2. Benefits for Graph Analytics

We now turn our attention towards more computationally intensive applications like PageRank and connected-component labeling inside graphs. The aim of these experiments is to demonstrate that the additional DRAM provides performance benefits not just for IOPS driven applications, but also for computationally intensive applications. We find that FlashMap reduces the execution time of these memory intensive workloads significantly.

GraphChi [22] is a graph computation toolkit that enables analysis of large graphs from social networks and genomics on a single PC. GraphChi is primarily a disk based graph computation mechanism that makes efficient usage of available memory. It partitions the graph such that each partition can fit in memory while it can work on it at memory speeds. We modify GraphChi to use SSDs as memory and run graph algorithms for various DRAM sizes. We increase the memory budget of GraphChi beyond the amount of DRAM available in the system. This means that the entire SSD is the working set of GraphChi. We run two graph computational algorithms on different graphs (Table 1) with various memory budgets to

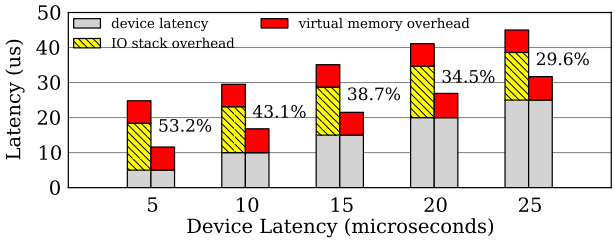

Figure 10: For faster SSDs, FlashMap provides tangible improvements in the latency over Unoptimized Software.

demonstrate the benefits of FlashMap over existing techniques.

First, we use GraphChi to find the PageRank of a real social networking graph. The graphs we use in our experiments are from Twitter [54, 31], Friendster and MSD as shown in Table 1. Figure 9 (a) and (b) demonstrate that FlashMap obtains 1.27–3.51x, 1.10–1.65x, 1.31–5.83x speedup in execution time of the PageRank algorithm than FS Extents, Separate Memory (OnDemand) and Unoptimized Software respectively, across different DRAM sizes.

The execution times of the PageRank algorithm on MSD at 1:64 DRAM:SSD ratio are 589.85 and 471.88 seconds respectively for Separate Memory (OnDemand) and FlashMap. For GraphChi, the working set spans the entire SSD. As the size of the SSD increases, the effective amount of DRAM available for GraphChi runtime decreases. It is natural that this increases the execution time. However, by combining indirections, FlashMap helps speed up the graph computation by freeing up memory. As shown in Figure 9 (b), the DRAM hit rates are 18.7% and 9.1% for FlashMap and Separate Memory (OnDemand) respectively as DRAM:SSD ratio is 1:256. We next evaluate the FlashMap with the `connected-component labeling` algorithm in GraphChi. We run it on the Twitter and Friendster dataset and the results are shown in Figure 9 (c) and (d): FlashMap outperforms FS Extents by up to 3.93x, and Separate Memory (OnDemand) by up to 3.32x.

5.2. Latency Benefits

The aim of the experiments in this section is to demonstrate that FlashMap also brings benefits for high-end SSDs with much lower device latencies, as it performs single address translation, single sanity and permission check in the critical

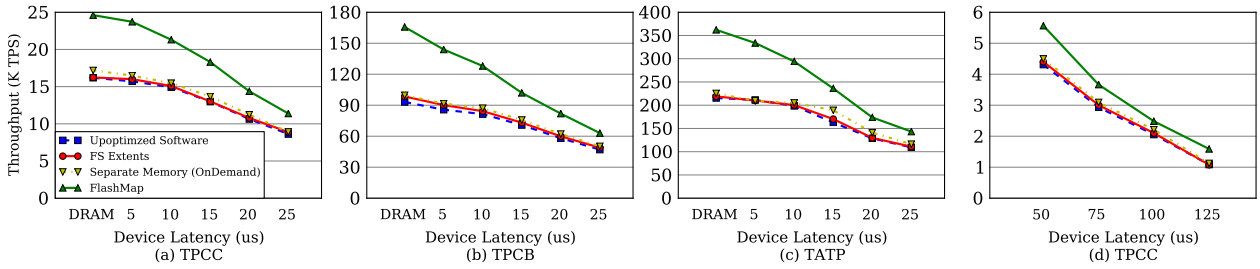

Figure 11: For faster SSDs, FlashMap provides up to 1.78x improvements on throughput over Unoptimized Software with TPCC, TPCB and TATP benchmarks. As for Flash with 100 μs device latency, FlashMap still performs 1.21x more TPS than others.

Overhead Source	Avg. Latency (μsec)
Walking the page table	0.85
Sanity checks	2.49
Updating memory manager state	1.66
Context switches	0.75

Table 2: FlashMap's Overhead

path to reduce latency.

PCM-based SSDs and PCM-based caches on NAND-Flash based SSDs are on the horizon [13, 11, 30]. With much smaller PCM access latencies, FlashMap can provide tangible latency improvements. We emulate such SSDs with various latencies using DRAM and show how the software overhead decreases because of FlashMap. 4KB pages from the SSD are accessed at random in a memory-mapped file and the average latencies are reported. The results are presented in Figure 10 and it shows how FlashMap can improve latency by up to 53.2% for faster SSDs. Note that these SSDs do not have a device-level indirection layer, therefore these benefits are purely from combining the translations and checks in file systems with those in virtual memory.

We further break down the software overheads of FlashMap by intercepting and timing each indirection layer. Table 2 lists the overhead of the key operations in FlashMap and we find that these are comparable to the latencies for each component in unmodified Linux.

Furthermore, we investigate how the latency benefits of FlashMap improves the performance of applications with concurrency. Faster access to data often translates to locks being released faster in transactional applications and this translates to higher application-level throughput.

We modify a widely used database manager (Shore-MT [49]) to `mmap` its database and log files with various techniques (i.e., FlashMap, FS Extents, Separate Memory (On-Demand) and Unoptimized Software). We use TPC-C [53], TPC-B [52] and TATP [50] benchmarks (as shown in Table 1) in our experiments. Their dataset sizes are 32-48 GB and the footprint of address translation data is small. The memory configured for the database manager is 6 GB. As shown in Figure 11, for SSDs with low latency, FlashMap provides up to 1.78x more throughput because of its latency reductions. For SSDs with higher hardware latency, FlashMap provides more than 1.21x improvement on throughput over Separate Memory (OnDemand), as the latency reduction (even small) can relieve the lock contentions in software significantly [25, 26]. We find similar trends for TPC-B and TATP workloads.

5.3. DRAM vs. SSD-memory

In this section, we analyze the cost effectiveness of using SSD as slow non-volatile memory compared to using DRAM with the aim of demonstrating FlashMap's practical impact on data-intensive applications. We survey three large-scale memory intensive applications (as shown in Table 3) to conduct the cost-effectiveness analysis. For this evaluation, we ignore the benefits of non-volatility that SSDs have and purely analyze from the perspective of cost vs performance for workloads that can fit in DRAM today. Additionally, we analyze how real-world workloads affect the wear of SSDs used as memory.

We use three systems for the analysis: Redis which is an in-memory NoSQL database, MySQL with "MEMORY" engine to run the entire DB in memory and graph processing using the GraphChi library. We use YCSB for evaluating Redis, TPC-C [53] for evaluating MySQL, and page-rank and connected-component labeling on a Twitter social graph dataset for evaluating GraphChi. We modify these systems to use SSDs as memory in less than 50 lines of code each. The results are shown in Table 3. The expected life is calculated assuming 3,000 P/E and 10,000 P/E cycles respectively for the SATA and PCIe SSDs, and a write-amplification factor of two. The results show that write traffic from real-world workloads is not a problem with respect to wear of the SSD.

SSDs match DRAM performance for NoSQL stores. We find that the bottleneck to performance for NoSQL stores like Redis is the wide-area network latency and the router throughput. Redis with SATA SSD is able to saturate a 1GigE network router and match the performance of Redis with DRAM. Redis with PCIe SSD is able to saturate a 10GigE router and match the performance of Redis with DRAM. The added latency from the SSDs was negligible compared to the wide-area latency.

SSD-memory is cost-competitive when normalized for performance of key-value stores. For a 1TB workload, the SATA setup and PCIe setup cost 26.3x and 11.1x less compared to the DRAM setup ($30/GB for 32GB DIMMs, $2/GB for

Application	Comparison	Settings	Bottleneck	Slowdown	Cost-Savings	Cost-Effectiveness	Expected Life
NoSQL Store	YCSB	DRAM vs SATA SSD (1GigE switch)	Wide-area latency & router throughput	1.0x	26.6x	26.6x	33.2 years
NoSQL Store	YCSB	DRAM vs PCIe SSD (10GigE switch)	Wide-area latency & router throughput	1.0x	11.1x	11.1x	10.8 years
SQL Database	TPCC	DRAM vs PCIe SSD	Concurrency	8.7x	11.1x	1.27x	3.8 years
Graph Engine	PageRank & others	DRAM vs SATA SSD	Memory Bandwidth	14.1x	26.6x	1.89x	2.9 years

Table 3: Cost-effectiveness of SSD-mapping vs DRAM-only systems for 1TB workload sizes.

PCIe SSDs, $0.5/GB for SATA SSDs). The base cost of the DRAM setup is $1,500 higher as the server needs 32 DIMM slots and such servers are usually expensive because of specialized logic boards designed to accommodate a high density of DIMM slots.

SSDs provide competitive advantage for SQL stores and graph workloads. We find that the bottleneck of performance for MySQL is concurrency. MySQL on PCIe SSD's Tpm-C was 8.7x lower compared to MySQL on DRAM for a 480GB TPCC database. However, the SSD setup cost 11.1x less compared to the DRAM setup that makes the SSD setup 1.27x better when performance is normalized by cost. Processing graphs in DRAM is up to 14.1x faster than processing them on the SSD while the SSD setup used is 26.3x cheaper than DRAM system. However, the advantage of SSDs is not based on cost alone. The ability to use large SSDs as slow-memory allows such applications to handle workloads (up to 20TB/RU) beyond DRAM's capacity limitations (1TB/RU) with very few code modifications.

6. Related Work

Memory-mapped SSDs. Several systems have proposed using SSDs as memory [5, 6, 29, 44, 45, 47, 55, 57]. However, these systems do not present optimizations for reducing the address translation overhead. FlashMap is the first system to provide the benefits of filesystems, exploit the persistence of SSDs and provide the ability to map data on SSDs into virtual memory with low-overhead address translation.

Combining indirection layers. Nameless writes [61] and DFS [28] combine the FTL with the file system's index. However, when mapping a file that uses nameless writes or DFS into virtual memory, page tables are created separately on top which increases the address translation and other software overhead. FlashMap shifts all the address translation into page tables since they are a hardware requirement and cannot be changed easily. However, FlashMap retains all the benefits of a system like Nameless Writes or DFS where the SSD manages the log, and the mappings are maintained on the host. Moreover, FlashMap does not change the semantics of virtual memory or the file system interfaces for achieving this goal.

On demand FTL. DFTL [24] proposes caching only the "hot" mappings in memory while other mappings can be stored on the SSD. While such techniques reduce the space requirement of RAM at the FTL they do not reduce the indirection overheads in higher software layers. FlashMap uses a similar mechanism to manage mappings on-demand in a single indirection layer across all the required software layers while preserving their functionalities.

Reducing SSD latency. The Moneta PCM SSD [11, 13] enabled fast and safe userspace accesses to SSDs via `read`/`write` system calls by performing the critical path OS and file system tasks on the SSD itself. FlashMap reduces the latency of page fault handling for accesses to SSDs via virtual memory by taking a complimentary approach of performing all the translations inside page tables that are mandatory for mapping SSDs to memory.

7. Conclusions and Future Work

Using SSDs as memory helps applications leverage the large capacity of SSDs with minimal code modifications. However, redundant address translations and checks in virtual memory, file system and flash translation layer reduce performance and increase latency. FlashMap consolidates all the necessary address translation functionalities and checks required for memory-mapping of files on SSDs into page tables and the memory manager. FlashMap's design combines these layers but does not lose their guarantees. Experiments show that with FlashMap the performance of applications increases by up to 3.32x, and the latency of SSD-accesses reduces by up to 53.2% compared to other SSD-file mapping mechanisms.

In the future, we are taking FlashMap in two directions. First, we are investigating how to provide transactional and consistency guarantees by leveraging the proto-SSD for storing a transactional log. For example, we could leverage a log/journal on the proto-SSD to implement atomic modifications to the memory-mapped file. Second, we are investigating the benefits of combining the memory and file system layers for byte-addressable persistent memories. In particular, we are evaluating the benefits of a combined indirection layer for leveraging existing file system code as a control plane to manage persistent memory while leveraging virtual memory as a high-performance data plane to access persistent memory.

Acknowledgments

We would like to thank our shepherd Parthasarathy Ranganathan as well as the anonymous reviewers. This research was performed when the lead author was an intern at Microsoft. It was also supported in part by the Intel URO program on software for persistent memories, and by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] Nitin Agarwal, Vijayan Prabhakaran, Tedd Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX ATC*, Boston, MA, June 2008.

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikath Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. 9th USENIX NSDI*, 2012.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. 22nd ACM SOSP*, October 2009.

[4] ArangoDB. `https://www.arangodb.com/`.

[5] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. 8th USENIX NSDI*, 2011.

[6] Anirudh Badam, Vivek S. Pai, and David W. Nellans. Better Flash Access via Shapeshifting Virtual Memory Pages. In *Proc. ACM TRIOS*, November 2013.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proc. 9th USENIX NSDI*, 2012.

[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proc. ISCA*, Tel-Aviv, Israel, 2013.

[9] Boost Template Library. `http://www.boost.org/`.

[10] Mustafa Canim, Georgia A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 3(1-2):1435–1446, 2010.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Micro'10*, Atlanta, GA, 2010.

[12] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *Proc. ACM ASPLOS*, March 2009.

[13] Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS*, March 2012.

[14] Vijay Chidambaram, Tushar Sharma, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Consistency Without Ordering. In *Proc. 10th USENIX FAST*, February 2012.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SoCC'12*, Indianapolis, Indiana, June 2010.

[16] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *PVLDB*, 3(1-2), 2010.

[17] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging DBMS Buffer Pool Using SSDs. In *Proc 30th ACM SIGMOD*, June 2011.

[18] EXT4 File Index. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Extent_Tree`.

[19] Fusion-io: ioDrive Octal. `http://www.fusionio.com/products/iodrive-octal/`.

[20] Fusion-io: ioMemory Virtual Storage Layer. `http://www.fusionio.com/overviews/vsl-technical-overview`.

[21] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[22] GraphChi. `http://graphlab.org/graphchi/`.

[23] Martin Grund, Jens Krueger, Hasso Plattner, Alexander Zeier, and Philippe Cudre-Mauroux Samual Madden. HYRISE–A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[24] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proc. ACM ASPLOS*, March 2009.

[25] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. In *VLDB'15*, 2015.

[26] Ryan Johnson, Ippokratis Pandis, Radu Stoica, and Manos Athanassoulis. Aether: A scalable approach to logging. In *VLDB'10*, 2010.

[27] Jonathan Thatcher and David Flynn. US Patent # 8,578,127, `http://www.faqs.org/patents/app/20110060887`.

[28] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Trans. on Storage*, 6(3):14:1–14:25, 2010.

[29] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *Proc. ACM SIGMETRICS*, Pittsburgh, PA, June 2013.

[30] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *FAST'14*, 2014.

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proc. WWW'10*, Raleigh, NC, April 2010.

[32] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proc. 10th USENIX OSDI*, Hollywood, CA, October 2012.

[33] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4), 2012.

[34] Hyeontaek Lim, Bin Fan, David Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. 23rd ACM SOSP*, Cascais, Portugal, October 2011.

[35] LMDB. `http://symas.com/mdb/`.

[36] MapDB. `http://www.mapdb.org/`.

[37] Memcache. `http://memcached.org/`.

[38] MonetDB. `http://www.monetdb.org`.

[39] MongoDB. `http://mongodb.org`.

[40] MongoDB Deployment. `http://lineofthought.com/tools/mongodb`.

[41] MongoDB: Memory Mapped File Usage. `http://docs.mongodb.org/manual/faq/storage/`.

[42] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[43] Jian Ouyang, Shiding Lin, Song Jiang, Yong Wang, Wei Qi, Jason Cong, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proc. ACM ASPLOS*, 2014.

[44] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleshwar K. Panda. SSD-Assited Hybrid Memory to Accelerate Memcached over High Performance Networks. In *Proc. 41st ICPP*, September 2012.

[45] Roger Pearce, Maya Ghokale, and Nancy M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proc. SC*, New Orleands, LA, November 2010.

[46] Redis. `http://redis.io`.

[47] Mohit Saxena and Michael M .Swift. FlashVM: Virtual Memory Management on Flash. In *Proc. USENIX ATC*, June 2010.

[48] Mohit Saxena, Yiying Zhang, Michael Swift, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Getting Real: Leassons in Transitioning Research Simulations into Hardware Systems. In *Proc. FAST*, 2013.

[49] Shore-MT. `https://sites.google.com/site/shoremt/`.

[50] TATP Benchmark. `http://tatpbenchmark.sourceforge.net/`.

[51] TCMalloc Memory Allocator. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[52] TPCB Benchmark. `http://www.tpc.org/tpcb/`.

[53] TPCC Benchmark. `http://www.tpc.org/tpcc/`.

[54] Twitter: A Real Time Information Network. `https://twitter.com/about`.

[55] Brian Van Essen, Roger Pearce, Sasha Ames, and Maya Gokhale. On the Role of NVRAM in Data-Intensive Architectures: An Evaluation. In *Proc. 26th IEEE IPDPS*, Shanghai, China, May 2012.

[56] Violin Memory 6000 Series Flash Memory Arrays. `http://violin-memory.com/products/6000-flash-memory-array`.

[57] Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Fei Mang, Youngjae kim, and Christian Engelmann. NVMalloc: Exposing an Aggregate SSD Store as a Memory Parition in Extreme-Scale Machines. In *Proc. 26th IEEE IPDPS*, Shanghai, China, May 2012.

[58] Who's using Redis? `http://redis.io/topics/whos-using-redis`.

[59] Yahoo! Cloud Serving Benchmark. `https://github.com/brianfrankcooper/YCSB/wiki`.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstractions for In-Memory Cluster Computing. In *Proc. NSDI*, 2012.

[61] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proc. 10th USENIX FAST*, February 2012.