The Dissertation Committee for Youngjin Kwon

certifies that this is the approved version of the following dissertation:

Designing Systems for Emerging Memory Technologies

Committee:

Emmett Witchel, Supervisor

Simon Peter, Co-supervisor

Thomas Anderson

Christopher J. Rossbach

Designing Systems for Emerging Memory Technologies

by

Youngjin Kwon

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2018

Designing Systems for Emerging Memory Technologies

Publication No. _____

Youngjin Kwon, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Emmett Witchel
Co-supervisor: Simon Peter

Emerging memory technologies open new challenges in system software: diversity and large capacity.

Non-volatile memory (NVM) technologies will have excellent performance, byte-addressability, and large capacity, blurring the line between traditional volatile DRAM and non-volatile storage. NVM diverges from DRAM in significant ways, like limited write bandwidth. It is likely that future storage market will be diversified, having DRAM, NVM, SSD, and hard disk. Unfortunately, current file systems, built on top of old design ideas, cannot provide an efficient way to take advantage of the different storage media. Strata

is a cross-media file system, fundamentally redesigning file systems to leverage different strengths of storage technologies while compensating their weaknesses.

Modern applications such as large-scale machine learning and graph analytics want to load huge datasets into memory for fast computation. For these workloads, merely adding more RAM to a machine reaches a point of diminishing returns for performance because their poor spatial locality causes them to suffer high virtual to physical memory translation costs. NVM will make this problem worse because it provides cheaper cost-per-capacity than DRAM. Ingens, a efficient memory management system, addresses the shortcomings in modern operating systems and hypervisors that underlies these excessive address translation overheads and redesign huge page memory systems to make huge page widely used in practice.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

System software must change to cope with the diversification of storage hardware and the ever-growing size of memory. From the hardware side, the market for storage devices has fragmented based on a tradeoff between performance and cost-per-capacity. HDD vendors develop shingled magnetic recording [smr17] technology to increase storage capacity with low cost. SSD technologies are fast evolving and outperform traditional HDDs with much wider IO bandwidth and lower latency, but they have lower capacity. In addition, emerging non-volatile memory (NVM) techniques such as Intel's 3D XPoint [int13] and NVDIMM [nvd17] promise revolutionary IO performance, but with high cost-per-capacity. Based on the performance and cost tradeoff, future cloud storage applications will want to leverage multiple storage media such as NVM, SSD, and HDD.

From the software side, applications want both high performance and large capacity from their volatile and non-volatile memories. For example, modern machine learning applications load large datasets into memory for fast computation. This in-memory computing trend is driven by two facts: dataset sizes are growing and memory prices are dropping. Therefore, modern applications extensively use DRAM for fast data access. Emerging memory technologies will accelerate the in-memory computing trend because they provide cheaper cost-per-capacity then DRAM.

1.1 Challenges

To cope with the *diversity* and *large capacity* of modern memory, system software needs major refinement, tweaking will not suffice.

Diversity challenges. Modern applications demand performance and functionality far outside comfort zone of the traditional file systems. Applications require fast, small, scattered updates on vast datasets. To this end, applications want fast NVM to achieve high performance and then rely on SSD and HDD for graceful degradation in performance and cost-effective capacity. However, current file systems are designed for a single type of device because each storage media has idiosyncratic performance characteristics. For example, NVM has low-latency and byte-addressability. To exploit these NVM features, NVM optimized filesystems [VNP$^+$14, XS16, DKK$^+$14, ext14] are necessary because file systems designed for SSD and HDD assume slow, block-granularity devices. As a consequence, to leverage diverse storage media, users must run multiple file systems each optimized for a storage type and then they need an additional mechanism above or below the file system at a high cost to manage data across storage layers.

Large capacity challenges. Large capacity memories are a significant challenge for address translation. All modern processors use page tables for address translation and TLBs to cache virtual-to-physical mappings. Because TLB capacities cannot scale at the same rate as DRAM capacity, TLB misses and address translation can incur crippling performance penalties for large-memory workloads. As a consequence, adding more RAM to a machine reaches a point of diminishing returns for performance because any poor spatial locality causes the machine to suffer high virtual to physical memory translation costs. Hardware manufacturers have addressed increasing DRAM capacity with better support for larger page sizes, or *huge pages*. Huge pages reduce address translation overheads by reducing the frequency of TLB misses. Newer CPUs support thousands of huge page entries in dual-level TLBs (e.g., 1,536 in Intel's Skylake), which is a major change from previous hardware which only supported a handful. To keep up with the great improvements in

memory mapping hardware, the memory system software must improve its support of huge pages.

This thesis discusses how to build storage and memory systems to address the diversity and large capacity challenges. Toward this goal, the thesis introduces two systems.

1.2 Contributions

Strata: A cross-media file system to address diversity challenges. Strata [KFH+17] works as an integrated file system across different storage media, providing low-latency (especially for small writes) and high-throughput IO, transparent data migration across different storage media for low-cost capacity, and a user-friendly crash consistency model. To achieve these goals, Strata has a novel split of user and kernel space responsibilities, storing file-system writes to a write-optimized, user-level log in fast NVM, while asynchronously managing read-optimized, long-term storage in the kernel. Strata continues to provide the POSIX API to run applications without any modification. Closest to the application, Strata employs a user-level library file system (*LibFS*) that uses a fraction of NVM as a private, operational log. The log holds updates to both user data and file system metadata. Logging in NVM has excellent latency and crash consistency properties. For performance, LibFS directly accesses the fast NVM without using the complicated kernel IO path. By leveraging the byte-addressability of NVM, LibFS performs cache-line granularity IO; LibFS blindly appends small writes (e.g., less than a block size) to the log without reading, modifying, and writing a block. To provide user-friendly crash consistency, LibFS *synchronously* updates the log, obviating the need for any `fsync`-related system calls and the attendant bugs of application-level crash consistency protocols [PCA+14]. Upon a crash, the kernel easily restores file system state by replaying log entries.

Strata has an in-kernel file-system module (called *KernelFS*) that organizes multiple storage layers as shared areas and exposes them read-only to applications. For low-cost capacity, KernelFS transparently migrates inactive data to larger, cheaper media via an operation called *digest*. A digest first happens from an application-local log to the system-shared

area, both stored in NVM, and then from one storage-device layer to its next (typically slow and larger) layer.

When digesting data among slower storage layers, KernelFS batches log entries and tailors the data format to the properties of the target medium to improve performance. For example, batching allows kernelFS to accumulate large contiguous file blocks when digesting into the read-optimized shared area. Similarly, batching allows kernelFS to write large amounts of data sequentially to SSDs and HDDs to avoid firmware garbage collection overhead [THL+15].

Ingens: A efficient memory system to address large-capacity challenge. Ingens [KYP+16] is a memory management system for the operating system and hypervisor that replaces the best-effort mechanisms and spot-fixes of the past with a coordinated, unified approach to huge pages; one that is a principled approach to the increased TLB capacity in modern processors.

Most paging hardware uses 4 KB pages, and huge pages are 512× larger at 2 MB, thereby increasing TLB reach and reducing address translation cost. However, despite the common-case performance improvements of huge pages, applications recommend disabling huge page support in Linux due to design flaws in the operating system software [Monb, cou, dok, reda, nuo, sap, spl, vol].

Linux aggressively allocates huge pages in the page fault handler, imposing unacceptable latency for interactive applications. Huge page allocation in the page fault handler requires high-latency operations: zeroing 2 MB memory pages and compacting memory to generate contiguous physical memory when memory is fragmented. To make the page fault handling path fast, Ingens decouples the promotion decision (policy) from huge page allocation (mechanism). The page fault handler decides when to promote a huge page, and a background kernel thread performs huge page allocation when signaled by the page fault handler.

Ingens monitors utilization of huge-page sized regions (space) and how frequently

the huge-page sized regions are accessed (time) to make better decisions about huge page allocation (which requires contiguous physical memory). Ingens distributes the contiguity of physical memory as a resource and does it fairly using information about space and time (the application's access patterns).

Modern hypervisors eliminate duplicate pages across virtual machines to save memory. To save more memory, Linux frequently splits huge pages, incurring a significant performance impact for applications that require huge pages. To balance memory savings and performance, Ingens allows Linux to split huge pages only when they are infrequently accessed.

1.3 Outline

This thesis begins by describing a cross media file system that addresses the diversity challenge (Chapter 2) and introduces a new huge page management system to address the large capacity challenge (Chapter 3). Chapter 4 discusses related work, and Chapter 5 concludes this work.

Chapter 2

Strata: A cross media file system

File systems are being stressed from below and from above. Below the file system, the market for storage devices has fragmented based on a tradeoff between performance and capacity, so that many systems are configured with local solid-state drives (SSD) and conventional hard disk drives (HDD). Non-volatile RAM (NVM) will soon add another device with a third distinct regime of capacity and performance.

Above the file system, modern applications demand performance and functionality far outside the traditional file system comfort zone, e.g., common case kernel bypass [PLZ+14], small scattered updates to enormous datasets [red17, DG11], and programmer-friendly, efficient crash consistency [PCA+14]. It no longer makes sense to engineer file systems on the assumption that each file system is tied to a single type of physical device, that file operations are inherently asynchronous, or that the kernel must intermediate every metadata operation.

To address these issues, we propose Strata, an integrated file system across different storage media. To better leverage the hardware properties of multi-layer storage Strata has a novel split of user and kernel space responsibilities, storing updates to a user-level log in fast NVM, while asynchronously managing longer-term storage in the kernel. In so doing, Strata challenges some longstanding file system design ideas, while simplifying others. For example, Strata has no system-wide file block size and no individual layer implements a

standalone file system. Although aspects of our design can be found in NVM [VNP+14, XS16, DKK+14, ext14] and SSD [LSHC15, JBLF10]-specific file systems, we are the first to design, build, and evaluate a file system that spans NVM, SSD, and HDD layers.

Closest to the application, Strata's user library synchronously logs process-private updates in NVM while reading from shared, read-optimized, kernel-maintained data and metadata. In the common case, the log holds updates to both user data and file system metadata, ensuring correct file system behavior during concurrent access, even across failures. Client code uses the POSIX API, but Strata's synchronous updates obviate the need for any `sync`-related system calls and their attendant bugs [PCA+14]. Fast persistence and simple crash consistency are a perfect match for modern RPC-based systems that must persist data before responding to a network request [zoo17, red17].

Strata tailors the format and management of data to improve the semantics and performance of the file system as data moves through Strata's layers. For example, operation logs are an efficient format for file system updates, but they are not efficient for reads. To reflect the centrality of data movement and reorganization in Strata, we coin the term *digest* for the process by which Strata reads file system data and metadata at one layer and optimizes the representation written to its next (typically slower and larger) layer. Digestion is periodic and asynchronous. Digestion allows a multitude of optimizations: it coalesces temporary durable writes (overwritten data or temporary files, e.g., due to complex application-level commit protocols [PCA+14]), it reorganizes and compacts data for efficient lookup, and it batches data into the large sequential operations needed for efficient writes to firmware-managed SSD and HDD.

Strata's first digest happens between a process-local log and the system-shared area, both stored in NVM. For example, a thread can create a file and a different thread can perform several small, sequential writes to it. The file create and the file data are logged in NVM. Each operation completes synchronously and in order. On a system crash, Strata recovers the latest version of the newly created file and its contents from the log. Eventually,

Strata digests the log into an extent tree (optimized for reads) requiring physical-level file system work like block allocation for the new file. It will also merge the writes, eliminating any redundancy and creating larger data blocks. Strata resembles a log-structured merge (LSM) tree with a separate log at each layer, but a Strata digest changes the format and the invariants of its data much more than an LSM tree compaction. Strata minimizes re-writing overhead (§2.4.1) and increases performance significantly in cases where digestion can reduce the amount of work for the lower layer (e.g., a mail server can avoid copying 86% of its log §2.4.2). Strata currently also has limitations. For example, Strata reduces mean time to data loss (MTTDL) by assuming that all of its storage devices are reliable. It is optimized for applications requiring fast persistence for mostly non-concurrently shared data. Concurrent shared access requires kernel mediation. For more limitations see §2.3.1.

We implemented our Strata prototype within Linux. The Strata user-level library integrates seamlessly with glibc to provide unmodified applications compatibility with the familiar POSIX file IO interface. Our prototype is able to execute a wide range of applications, successfully completing all 201 unit tests of the LevelDB key-value store test suite, as well as all tests in the Filebench package. Microbenchmarks on real and emulated hardware show that, for small ($\leq$ 16 KB) IO sizes, Strata achieves up to 33% better write tail-latency and up to 7.8$\times$ better write throughput relative to the best performing alternative purpose-built NVM, SSD, and HDD file systems. Strata achieves 26% higher throughput than NOVA [XS16] on a mailserver workload in NVM, up to 27% lower latency than PMFS [DKK$^+$14] on LevelDB, and up to 22% higher SET throughput than NOVA on Redis, while providing synchronous and unified access to the entire storage hierarchy. Finally, Strata achieves up to 2.8$\times$ better throughput than a block-based two-layer cache provided by Linux's logical volume manager. These performance wins are achieved without changing applications.

Starting with a discussion of the technical background (§2.1) for Strata's design, we then discuss its contributions.

Memory	Latency	Seq. R/W GB/s	$/GB
DRAM	100 ns	62.8	8.6
NVM	300 ns	7.8	4.0
SSD	10 μs	2.2 / 0.9	0.25
HDD	10 ms	0.1	0.02

Table 2.1: Memory technology latency, bandwidth, and $/GB as of April 2017. NVM numbers are projected (§2.4).

- We present the Strata architecture (§2.2). We show how maintaining a user-level operation log in NVM and asynchronously *digesting* data among storage layers in the kernel leads to fast writes with efficient synchronous behavior, while optimizing for device characteristics and providing a unified interface to the entire underlying storage hierarchy.
- We implement a prototype of Strata on top of Linux that uses emulated NVM and commercially available SSDs and HDDs on a commodity server machine (§2.3).
- We quantify the performance and isolation benefits of providing a unified file system that manages all layers of the modern storage hierarchy simultaneously (§2.4).

2.1 Background

We review current and near-future storage devices and discuss how Strata addresses this diversified market. We then discuss the demands of modern applications on the file system and how current alternatives fall short.

2.1.1 Hardware storage trends

Diversification. Storage technology is evolving from a single viable technology (that of the hard disk drive) into a diversified set of offerings that each fill a niche in the design tradeoff of cost, performance, and capacity. Three storage technologies stand out as stable contenders in the near-future: Non-volatile memory (NVM), solid state drives (SSDs), and high-density hard disk drives (HDDs). While HDDs and SSDs are already a commodity today, NVM is expected to be added in the future (Intel's 3D XPoint memory technology was released in March 2017, initially to accelerate SSDs [opt17b, opt17a]).

Table 2.1 shows each technology and its expected long-term place in the design

space. Latencies are from specifications while sequential read/write bandwidth for 4KB IO sizes are measured (see §2.4 for details). Prices are derived from the lowest device prices found via a Google Shopping search in April 2017. The NVM price is derived from the current price of Intel's 3D XPoint-based Optane SSD DC P4800X. NVM performance is based on a study [ZS15]. Each storage technology offers a unique tradeoff of latency, throughput, and cost, with at least an order of magnitude difference relative to other technologies. This diversity suggests that future systems are likely to require several coexisting storage technologies.

Device management overhead. The physical characteristics of modern storage devices often prevent efficient update in place, even at a block level. SSDs have long needed a multi-block erasure before a physical block can be re-written; typical erasure region size has grown larger over time as vendors optimize for storage density and add lanes for high throughput. Although HDDs traditionally allowed efficient sector overwrites at the cost of a disk head seek, recently disks optimized for storage efficiency have adopted a shingle write pattern, similar to SSDs in that an entire region of disk sectors must be re-written to update any single sector.

To support legacy file systems, SSD and HDD device firmware maintains a persistent virtual to physical block translation; blocks are written sequentially at the physical level regardless of the virtual block write pattern. Depending on the write pattern, this can carry a high cost, where blocks are repeatedly moved and re-written to create empty regions for sequential writes on both SSDs and HDDs. On SSDs, this write amplification wears out the device more rapidly.

Even using a large block size is not enough to avoid the overhead when the disk is in steady-state. For example, using the Tang et al. methodology [THL+15] on our testbed SSD (§2.4 for details), we observe a throughput slow-down of 12.2× in steady-state for 8 MiB blocks written randomly to a full disk. Similarly, write latency is inflated by a factor of 2.8× for 4 KiB random updates in steady state, and 10× for 128 KiB updates. When

writing sequentially within erasure block boundaries, performance does not decline. Write amplification can also negatively affect IO tail latency and throughput isolation among applications, as the overhead is observed due to past use of the device, making it difficult to account performance costs to the originating application.

We leverage the multi-layer nature of Strata to achieve the full performance of the SSD and HDD layers, despite firmware management. Migration of blocks from NVM to SSD are made in full erasure block chunks (512 MiB on our testbed SSD); this is only possible because Strata coalesces data as it moves between persistent layers, with frequently updated data filtered by the NVM layer.

2.1.2 Application demands on the file system

Many modern applications need crash consistency for their files. The performance cost and complexity to achieve user-level crash consistency for files has grown over the past decade, with no relief in sight. Often, files are merely named address spaces that contain many internal objects with frequent, crash-consistent updates. Small, random writes are common on both desktop machines [HDV+11] and in the cloud through the use of key-value stores, data base backends, such as SQLite [sql17] and LevelDB [DG11], revision management systems, and distributed configuration software, such as Zookeeper [zoo17]. On many file systems, efficient crash consistency for these applications is difficult and slow so many applications sacrifice correctness for performance [PCA+14].

Strata provides in-order file system semantics (including writes). This matches developer intuition [NVCF06] and simplifies crash recovery, but is usually considered too slow to be a practical goal for a file system. Given NVM devices, such semantics are now possible to provide efficiently [XS16].

2.1.3 Current alternatives are insufficient

Existing file systems specialize to a storage technology. Existing file systems make tradeoffs that are appropriate for a specific type of storage device; no single file system is appropriate across different storage media. For example, NOVA [XS16] and PMFS [DKK+14]

require byte-addressability, limiting them to NVM; F2FS [LSHC15] uses multi-head logging and a buffer cache that are unnecessary on NVM. Strata is built to leverage the strengths of each storage device and compensate for weaknesses. By contrast, layering independent file systems on different media unnecessarily duplicates mechanisms, such as block and inode allocation, and lacks expressive inter-layer APIs. For example, block usage frequency and fragmentation information are not easily relayed across independent file systems (§2.4.3).

File system write amplification. As shown in §2.4, many file systems pad updates to a uniform block size (e.g., setting a bit in an in-use block bitmap will write an entire block), and file systems often require metadata writes to complete an update (e.g., a data write can update the file size in the inode). As with device-level write amplification, file system write amplification is often a major factor for application performance, especially for NVM devices that support efficient small writes. Using an operation log at the NVM layer that is later digested into block updates, Strata is able to efficiently aggregate repeated data and metadata updates, significantly lowering file system write amplification.

Block stores are not the only answer. Strata provides a file system rather than a block store interface to applications because of the file system's strong combination of backward compatibility, performance and functionality. The file system name space is a powerful persistent data structure with well understood properties (and limitations); its storage costs are moderate in time and space across a wide variety of access patterns; and it is used to share data by millions of applications and system tools. Multi-layer cloud-persistent block stores [s317] face many of the same issues as Strata in managing migration of data across multiple devices, and can be appropriate for standalone applications that do their own block-level operations. We focus our design and evaluation on the unique opportunities provided by having a semantically rich view of application file system behavior.

2.2 Strata Design

The goal of Strata is to design a new file system that manages data across different storage devices, combining their strengths and compensating for their weaknesses. In particular, we have the following design goals for Strata.

- **Fast writes.** Strata must support fast, random, and small writes. An important motivation for fast small writes is supporting networked server applications which must persist data before issuing a reply. These applications form the backbone of modern cloud applications.
- **Efficient synchronous behavior.** Today's file systems create a usability and performance problem by guaranteeing persistence only in response to explicit programmer action (e.g., `sync`, `fsync`, `fdatasync`). File systems use a variety of complicated mechanisms (e.g., delayed allocation) to provide performance under the assumption of slow device persistence. Strata supports a superior, programmer-friendly model where file system operations persist in order, including synchronous `write`s, without sacrificing performance.
- **Manage write amplification.** Write amplification at the device and file system level have a first-order effect on performance, wear, and QoS. Examples include metadata updates for EXT4 and PMFS or copies introduced by the flash translation layer in SSDs [THL+15]. Managing write amplification allows us to minimize its effect on performance and QoS. Managing write amplification is simpler once it is decoupled from the write fast-path.
- **High concurrency.** Strata supports concurrent logging from multiple threads in a single process using atomic operations. Logs from multiple processes can be digested in parallel within the kernel because logs are guaranteed to be independent (see §2.2.4).
- **Unified interface.** We provide a unified file system interface to all devices in the entire underlying storage hierarchy. Strata is backward compatible with existing POSIX applications but easily customizable since the API is provided entirely in a user-level library [VNP+14, PLZ+14].

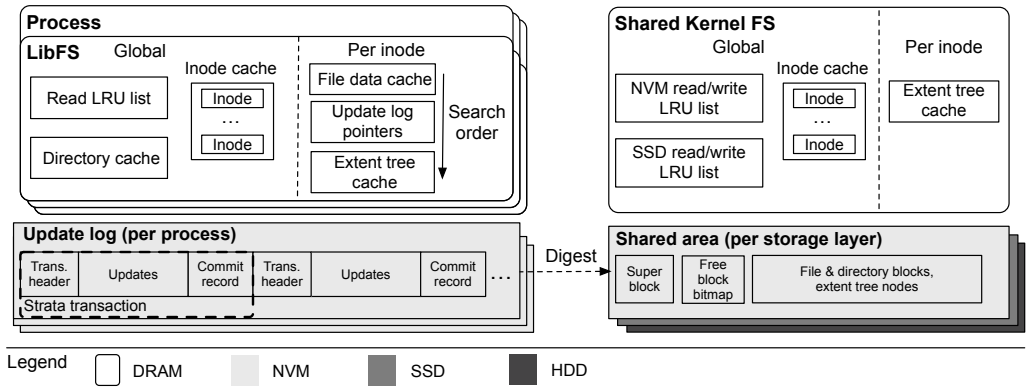


Figure 2.1: Strata design. Writes go to the update log. Reads are served from the shared area. File data cache is a read-only cache, containing data from SSD or HDD.

Concept	**Explanation**
Update log	A per-process record of file system updates.
Shared area	Holds file system data in NVM, SSD, and HDD. Read-only for user code, written by the kernel.
File data cache	Read-only cache; caching data from SSD or HDD.
Update logpointers	An index into the update log; mapping file offsets to log blocks.
Strata transaction	A unit of durability; used for file system changes made by a system call.
Digest	Apply changes from an update log to the shared area.
Lease	Synchronizes updates to files and directories.

Table 2.2: Major concepts in Strata.

Strata's basic architecture resembles a log-structured merge (LSM) tree [OCGO96]. Strata first writes data synchronously to an operation log (*logging*) stored in NVM. Logging provides persistence with low and predictable latency, efficiently represents small updates, serializes operations in order, and supports data consistency, crash recovery and operation atomicity. Logs are highly desirable for writing, but are cumbersome to search and read. Thus, logs are periodically *digested* into a read-optimized tree format. In Strata, this format is based on per-file *extent trees* [MCB+07]. Digests happen asynchronously, and the log is garbage-collected. Table 2.2 summarizes major concepts in Strata, and Figure 2.1 shows a high-level overview of the Strata design which we now discuss.

Log at user-level, digest in the kernel. To attain fast writes, Strata separates the responsibilities of logging and digesting and assigns them to user-level software and the kernel,

respectively. We call the user-level portion of Strata *LibFS*. Using leases to manage concurrent file accesses (explained in §2.2.4), the kernel grants LibFS direct access to a small private space in NVM for efficient logging of file system updates (the *update log*). The kernel also gives LibFS direct read-only access to portions of the shared extent tree space and data blocks (the *shared area*). Hardware, configured by the kernel, enforces access control [PLZ+14].

The kernel-level file system (*KernelFS*) is responsible for digesting. Digesting is done in parallel across multiple threads for high throughput and runs asynchronously in the background. The update log is deep to allow the digest to batch log entries, amortizing and aggregating meta-data updates over an entire sequence of operations. KernelFS checks and enforces metadata integrity when digesting an application's log, such that when a digest completes, the digested data can become visible to other processes. Upon a crash, the kernel can recover file system state simply by re-digesting each application's remaining log. A log replay overwrites data structures with their proper contents even if they were partially written before the crash (log replay is idempotent). The log remains authoritative until garbage collected after a completed digest. Since data is updated in a log-structured way, synchronization of log update and digest are simple. Writers make sure not to overwrite already allocated log blocks, while only allocated blocks are digested (and garbage collected). Write and digest positions are kept in NVM.

Sequential, aligned writes. One benefit of digesting writes in bulk is that, however they are initially written, file data can be coalesced and written sequentially to the shared area, minimizing fragmentation and meta-data overhead. Digestion minimizes device-level write amplification by enabling sequential, aligned writes. Below the NVM layer, all device writes are sequential and aligned to large block boundaries chosen to be efficient for the device, such as erasure blocks for SSDs and write zones for shingled disks. These parameters are determined by Strata for each device [THL+15]. When data is updated, old versions are not immediately overwritten. Instead, Strata periodically garbage collects cold blocks

to reclaim free space. Garbage collection consumes entire erasure/shingle block size units so that the device sees only full block deletes, eliminating collection overhead from the device layer. This process is similar to what would occur within device firmware but takes into account application data access patterns and multiple layers, segregating frequently from infrequently accessed data and moving them to appropriate layers for better device utilization and performance isolation.

Use hardware-assisted protection. To bypass the kernel safely and efficiently, Strata makes use of the hardware virtualization capabilities available in modern server systems. Strata specifies access rights for each application to contiguous subsets of each device's storage space, enforced by hardware. The MMU trivially supports this feature at memory page granularity for NVM, while NVMe provides it via *namespaces* that can be attached to hardware-virtualized SSDs [nvm17]. Strata moves all latency-sensitive mechanisms of the file system into a user-level library. HDDs do not require kernel bypass.

We next describe each component of Strata and their interaction. Since Strata breaks the responsibilities of a traditional file system into LibFS and KernelFS, we organize our description along these lines. We start by describing Strata's principal meta-data structures.

2.2.1 Meta-data Structures

Strata keeps meta-data in superblocks, inodes, and per-layer bitmaps of free blocks. These data structures are similar to structures in other file systems and we only briefly describe them here. Strata caches all recently accessed meta-data structures in DRAM.

Superblock. Strata's superblock is stored in NVM and describes the layout of each storage layer and the locations of all per-application logs. It is updated by KernelFS whenever per-application logs are created or deleted.

Inodes and directories. Inodes store file meta-data, such as access rights, owner, and creation times. As in EXT4, they also store a root of each file's extent tree, though for Strata, an inode has multiple roots, one for each storage device. When unfragmented, extent

tree nodes point directly to a file's data blocks. As the extent tree fragments, nodes point to other internal tree nodes before pointing to data blocks. Strata stores inodes ordered by number in a hidden, sparse *inode file* and manages it like a normal file: Strata accesses the inode file via an extent tree and migrates blocks of the inode file to other layers. An inode for the inode file is located in the superblock.

Strata directories are similar to EXT4, holding a chained array of file names and associated inode numbers in their data blocks. On file reads, LibFS first consults per-inode *update log pointers* to find any updates in the log. The log pointers are invalidated when the local log is digested. We hash entire directory names [TZR+15] to improve our directory cache hit rate, reducing full directory traversals by up to 60%. Strata inodes fill 256 bytes. To efficiently protect inodes under kernel-bypass, inodes with different access permissions have to be stored on different pages or within different NVMe namespaces. POSIX specifies that all inodes stored within a directory have access permissions according to the directory inode. Thus, Strata organizes inodes of the same directory together by reserving consecutive inodes in multiples of 16 for each directory. Unused inodes remain reserved until allocated.

Free block bitmap. Strata has a per-layer persistent bitmap to indicate which of its blocks are allocated and free. For high throughput, KernelFS digest threads allocate and free blocks in large batches. These threads reserve blocks (e.g., the size of an erasure block) by adjusting a unit allocation count in DRAM using compare-and-swap, and then marking specific blocks as in use in the bitmap. Once the allocation count reaches the maximum, digesting moves on to a new erasure/shingle unit. Freed blocks are reset in the bitmap.

Multiple device instances. The Strata prototype supports only a single storage device at each level, but the design would generalize to multiple devices at each level, where devices are logically concatenated. For example, Strata can treat two 8TB SSDs as a single 16TB SSD. This approach allows Strata to add capacity, while redundancy is left as future work.

2.2.2 Library File System (LibFS)

Strata's library file system (LibFS) provides the application-level mechanism to conduct file IO. Its goal is to provide fast, crash-consistent, and synchronous read and write IO to the entire underlying storage hierarchy and a unified API that is fully compatible with existing POSIX applications and can be put underneath an application by re-linking with LibFS.

Fast and synchronous persistence. Synchronous persistence provides clear semantics (e.g., ordering guarantees and crash recovery) to applications [NVCF06], but it is not widely used under the assumption that storage devices are slow. Modern NVM storage technology allows Strata to provide synchronous IO semantics without sacrificing performance. In fact, synchronous semantics can accelerate overall IO performance for NVM. Strata writes data once to NVM and does not copy it to a DRAM buffer cache. Memory copy latencies are comparable to NVM write costs [CNF+09], so eliminating the memory copy approximately halves write latency.

Upon an application write request, LibFS writes directly to a per-application update log in NVM, bypassing the OS kernel. Favoring the byte-addressable feature in NVM, LibFS does *blind write* for small-sized writes (e.g., less than 4 KB). A small write is written sequentially to the log and turned into a block write when KernelFS digests it, maximizing IO throughput and eliminating write amplification. Synchronous semantics allow Strata to provide zero-copy IO—LibFS performs IO directly between a user's DRAM buffer and NVM. Strata does not use a page cache which eliminates cache lookup and data copy in the write path. However, LibFS does maintain caches of the locations of logged file updates, as well as meta-data, such as inodes, file sizes, and modification times (inode and directory tables in Figure 2.1).

LibFS organizes the update log as an operation log. The operation log reduces IO compared to a data log because the data log usually contain blocks, which are the minimum-sized addressable units for the file system. For example, when updating a directory, the data log requires three (block) writes: directory inode, directory block, and log header. The oper-

ation log requires only a record indicating the directory change such as `ADD filename, inode number`. This information is small enough to fit into the log header, resulting in a single write for directory changes.

We arrange the log format so that its effects are idempotent; applying the log multiple times results in the same file system state. For example, log entries use both the inode and offset to refer to locations modified in a file or directory. LibFS allocates inode numbers eagerly to simplify logging. It requests batches of inodes from the kernel, such that inode allocation does not require a system call in the common case.

Crash consistent logging. LibFS logs changes to all file system state, including file and directory meta-data. All data is appended sequentially to the log, naturally capturing the ordering of file system changes. Logging also provides crash consistent updates efficiently. As shown in Figure 2.1, when an application creates a file and then writes data to the file, LibFS first logs a file creation record (with file length of 0) followed by the data write record in causal order.

LibFS has a unit of durability, called *Strata transaction*. Strata transactions provide ACID semantics up to an application's update log size, allowing Strata to atomically persist multiblock write operations up to the size of the log. To do so, LibFS wraps each POSIX system call in a Strata transaction. However, single system calls with more data than the per-application log size (on the order of GBs) cannot be persisted atomically and are instead broken into multiple, smaller Strata transactions. Many applications desire ordered, atomic multiblock writes and can benefit from these semantics [PCA+14].

Each Strata transaction consists of a number of *log entries*: a header, the relevant updates to file (meta-)data, followed by a commit record. The commit record contains a unique and monotonically increasing Strata transaction number and a checksum of the header contents. When a Strata transaction commits, LibFS ensures atomicity and isolation by atomically allocating log space using a compare-and-swap operation and by first writing the header and data, waiting for persistence, and then persisting the commit record. Log

headers contain a pointer to the next log header so the log can be easily replayed.

Digest and garbage collection. The log is a limited resource and needs to be periodically digested into the shared area and garbage collected. Once the log fills beyond a threshold (30% in our prototype), LibFS makes a digest request to KernelFS. KernelFS digests the log asynchronously in the background and replies to LibFS once the digest request is complete. After completion, LibFS can safely reclaim log entries (also in the background) by resetting each log header's valid bit. Strata data structures allow the user to add records to a log that the kernel is concurrently digesting.

If an application completely fills its log, LibFS must wait for an in-progress digest to complete before it can reclaim log space and restart file system activity. Log garbage collection involves trimming log headers (a device-level trim operation zeroes the trimmed data blocks) and invalidating the corresponding entries in the data cache. LibFS garbage collects using a background thread. The application can continue to append log blocks during garbage collection. The log's idempotency ensures crash consistency. If the system crashes during a digest, the log is re-digested on recovery, resulting in the same file system state as a successful digestion without a crash.

Fast reads. LibFS caches data and meta-data in DRAM. However, data is only cached when read from SSD or HDD. NVM does not require caching. The file data cache is managed in 4 KB block units and evicted to the update log in an LRU manner. Meta-data such as file access time and file data locations (in the log and in the shared area) are cached in the inode cache indexed by inode number. LibFS also records update addresses in the log using the update log pointers and it caches extent tree nodes. To optimize performance of sequential reads from SSD or HDD, LibFS uses a read-ahead buffer in DRAM of 256 KB.

To resolve a file location with the most up-to-date data, LibFS searches the file data cache, the update log, and then the (cached) extent trees from highest (NVM) storage layer to lowest (HDD), as shown in Figure 2.1. If the file data is not found in the data cache, but in the update log pointers, then the latest data is read from the log and (depending on

the read size) possibly merged with blocks from the shared area. In that case, both log data and shared blocks are fetched and merged before returning data to the read request. If a lookup misses in the extent tree cache for a layer, then Strata traverses the extent tree stored in that layer's shared area and updates the cache, before advancing to the next layer. Extent trees in multiple layers can be present for a file if subsets of its data blocks have been migrated. Extent trees indicate which of a file's data blocks are present at the tree's layer. Strata's layered data storage is not inclusive and a data block can be simultaneously present in any subset of layers. Strata's migration algorithm ensures that higher layers have the most up-to-date block and thus, higher layers take precedence over lower layers.

2.2.3 Kernel File System (KernelFS)

Strata's kernel file system (KernelFS) is responsible for managing shared data that can be globally visible in the system and may reside in any layer of the storage hierarchy. To do so, it digests application logs and converts them into *per-file extent trees*. Digestion happens asynchronously in the background, allowing KernelFS to batch Strata transactions and to periodically garbage collect and optimize physical layout. LibFS provides least-recently-used (LRU) information to KernelFS to inform its migration policy among layers of the storage hierarchy. KernelFS also arbitrates user-level concurrent file access via *leases* (§2.2.4).

Digest. When the log size grows beyond a threshold, LibFS makes a digest request to KernelFS. Digest latencies have an impact on applications' IO latencies as the log becomes full. To reduce the digest latencies, KernelFS employs a number of optimizations. KernelFS digests large batches of operations from the log, coalescing adjacent writes, as well as identifying and eliminating redundant operations. KernelFS begins digestion by first scanning the submitted log and then computing which operations can be eliminated and which can be coalesced. For example, if KernelFS detects an inode creation followed by deletion of the inode, it skips log entries related to the inode.

These optimizations reduce digest overhead by eliminating work, batching updates

to extent trees, and reducing the number of tree lookups. Coalescing writes increases the average size of write operations, minimizing fragmentation and thus extent tree depth. Optimizing the digest reduces bandwidth contention for the storage device between KernelFS and LibFS, as well as write amplification. Experiments with Filebench [TZS16] show that optimizations reduce digest latency up to 80%, improving application throughput by up to 15%. Scanning the log before digesting allows KernelFS to determine which new data and metadata blocks are required and to allocate them in large, sequential batches. Log scanning also allows KernelFS to determine if two logs contain disjoint updates and thus can be digested in parallel.

For all data updates, Strata writes new data blocks before deleting old blocks. Even metadata structures like extent trees are completely written before the updates are committed when the inode's root pointer is updated.

Data access pattern interface. To take advantage of the entire storage hierarchy, KernelFS transparently migrates data among different storage layers, keeping least recently used blocks in better performing layers. In order to migrate data efficiently, KernelFS requires LRU information for each block. Because reads bypass the kernel, LibFS must collect access information on reads and communicate the information to KernelFS via a kernel interface. LRU information is not persisted and only maintained in DRAM, which conserves NVM log space. Writes are observed by the kernel when digesting update logs, so there is no need for LibFS to provide additional metadata about writes.

The KernelFS maintains LRU lists for each storage layer except for the last one. An LRU list is a sequence of arbitrary length, of logical 4KB block numbers. LibFS can submit access information as frequently as it wishes via a system call. KernelFS transforms the LibFS-provided LRU lists into coarser-grained lists for storage layers that have larger block sizes (e.g., 1MB blocks for NVM and 4MB for SSD).

KernelFS does not trust the LRU information provided by a LibFS and enforces that blocks specified as recently used are actually accessible by the process. Applications can

misuse the interface to get the kernel to place more blocks in NVM, but this is equivalent to current systems where an application can read data to get the kernel to place it in the DRAM page cache. Resource allocation interfaces like Linux's memory cgroups [cgr15] would further limit the impact of API misuse, though integrating with cgroups is left as future work.

Data migration. To take advantage of the storage hierarchy's capacity, the kernel transparently migrates data among different storage layers in the background. To benefit from concurrency and to avoid latency spikes due to blocking on migration, Strata migrates data before a layer becomes full (at 95% utilization in our prototype). Migration is conducted in a block-aligned, log-structured way, similar to digestion. To make migrations efficient and at the same time reduce fragmentation, Strata moves SSD data in units of flash erasure blocks (order of hundreds of megabytes) and HDD data in shingles (order of gigabytes). After migrating a unit, the whole unit is trimmed (via the device TRIM command) to make a large, unfragmented storage area available. When migrating data, KernelFS tries to place hot data in higher layers of the storage hierarchy, while migrating cold data down to slower layers. To maintain a log-structured write pattern, KernelFS always reserves at least one migration unit on each layer and writes blocks retained in that layer to the reserved migration unit sequentially.

2.2.4 Sharing (leases)

Strata supports POSIX file sharing semantics, while optimizing application access to files and directories that are not concurrently shared. KernelFS supports *leases* on files and sections of the file system namespace. Leases have low execution time overhead for coarse-granularity sequential sharing of file data. We expect that processes that require fine-grained data sharing will use shared memory or pipes—avoiding the file system altogether due to its generally higher overhead.

Similar to their function in distributed file systems [HN15], leases allow a LibFS exclusive write or shared read access to a specific file or to a region of the file system

namespace rooted at a certain directory. For example, a LibFS can lease one or more directories and then create nested files and directories. Multiple LibFS may hold read leases, while only one write lease may exist.

Write leases are strict, they function like an exclusive lock. As long as a write lease is held, a thread in a process may write to the leased namespace (or file) without kernel mediation, while operations from other processes are serialized before or after the lease period. Threads within the same process see each others' updates as soon as operations complete, using fine-grained inode locks to synchronize file system updates. Leases are independent from file system access control checks, which occur when a file or directory is opened.

A process that holds a write lease is notified via an upcall (via a UNIX socket in our prototype) if another process also wants the write lease. Upon revocation of a write lease, applications can insist on the write-back of new data (via a log digest) to the kernel's shared file system area (e.g., to NVM). Waiting for a digest operation will increase the latency of revoking the lease. Leases are also revoked when an application is unresponsive and the lease times out. Because user-level operations are transactional, Strata can abort any in-progress file system operation upon revocation of a lease if necessary. LibFS caches are invalidated upon loss of a lease.

Programs may acquire leases using explicit system calls, which allows user-level control, but is not POSIX compatible. Our prototype lazily acquires an exclusive (shared) lease on the first write (read) to any file or directory (unless the process already has a lease). This policy works for our benchmarks, but other policies are possible. Bad policy choices lead to poor performance, but do not compromise correct sharing semantics because Strata can always fall back to kernel mediation for all file system operations. If a file is opened read/write by multiple processes, the kernel eliminates logging.

To show the worst-case performance overhead of sharing through the file system, we measured update throughput of two processes using a lock file to coordinate small (4KB)

updates to a shared data file. In one iteration, a process tries to create the lock file. Once creation succeeds, the winning process writes a 4KB block to the data file, then it unlinks the lock file. Note that neither file is ever synced. To guarantee strict ordering and synchronous persistence, LibFS must first acquire a lease in order to create the lock file and relinquish the lease and perform a digest after the lock file is unlinked. Strata achieves a throughput of 10,400 updates/s, $4.3\times$ slower than EXT4-DAX and $1.7\times$ slower than NOVA. EXT4-DAX can perform metadata updates in the buffer cache, but unlike Strata and NOVA, it lacks synchronous, ordered file semantics. Both NOVA and EXT4-DAX only write shared data once, while Strata must write it again during digestion to make the data globally visible. We thus view logging in Strata as an optimization to accelerate infrequently shared data. However, in situations with less strict ordering and atomicity guarantees, logging could be used even when sharing frequently.

2.2.5 Protection and performance isolation

Protection with kernel bypass. Strata supports POSIX file access control, enforced by MMU and NVMe namespaces. The MMU provides protection for kernel-bypass LibFS operations and Strata aligns each per-file extent tree on a page boundary (and pads the page) to facilitate MMU protection. The kernel maps all data and meta-data pages of the accessed file read-only into the caller's virtual address space. Extent tree nodes refer to blocks using logical block numbers. An entire device can be mapped contiguously, making the mapping from logical block number to address a simple addition of the base address. However, more parsimonious mappings are possible along with a table to track the mapping between address and logical block number.

For SSD-resident data, Strata uses NVMe namespaces for protected access to file data. File extent trees must be aligned on a NVMe sector (512 bytes or 4KB, depending on how the device is formatted). Upon opening a file on the SSD, the kernel creates a read-only NVMe namespace for the file if the namespace doesn't already exist and attaches it to the application's NVMe virtual function. The NVMe standard supports up to 2^{32} namespaces,

which limits the total number of open files on the SSD to this number. If an SSD does not support virtual functions, namespace management, or a sufficient number of namespaces, this functionality can be efficiently emulated in software, with an overhead of up to 3 μs per system call [PLZ+14]. HDD access is kernel mediated.

Performance isolation. Write amplification has an effect on IO performance isolation by inflating device bandwidth utilization. When device firmware amplifies writes it can throw off the operating system's management algorithms. Firmware-managed devices often have unpredictable and severe write amplification from wear leveling and garbage collection [THL+15]. Since Strata minimizes firmware write amplification via aligned sequential writes, almost all amplification occurs in software. This has the benefit that it can be accurately observed and controlled by Strata. For example, KernelFS can decide to stop digesting from an application if the incurred write amplification would violate the QoS (specified as per-application I/O bandwidth allocations) of another application.

2.2.6 Example

To summarize the design, we walk through an example of overwriting the first 1 KB of data in an existing, non-shared file and then reading the first 4 KB.

Open. The application uses the `open` system call to open the file. Upon this call, LibFS first checks to see whether the file exists and whether it can be accessed, by walking all path components from the root. For each component, it acquires read leases and checks the directory and inode caches for cached entries. If a component is not found in a cache, LibFS finds the inode by number from the inode file located in the shared area. Assuming the data is in NVM, LibFS will map the corresponding inode page read-only. The kernel allows the mapping if the inode is accessible by the user running the application. LibFS first copies the inode's content to the inode cache in DRAM. It then reads the inode (from cache) to determine the location of the directory by walking the attached extent tree, storing extent tree entries in the extent tree cache. Finally, LibFS finds the correct entry within the directory. The directory entry contains the inode number of the file, which LibFS resolves

in the same manner. The file is now open, and LibFS allocates a file descriptor.

Write. The application issues the `write` system call to write 1 KB to the beginning of the file. LibFS wraps the system call in a Strata transaction and requests a write lease for the corresponding inode. No other processes are accessing the file, so the kernel grants the lease. The Strata transaction can commit and LibFS appends the write request, including payload to the update log, checks the file data cache for invalidation, and updates the corresponding block in the update log pointers with addresses of the update log. The write is complete.

Read. The application issues a `pread` system call to read the first 4 KB from the file. Like the write case, LibFS first tries to obtain a read lease, waiting until KernelFS grants the read lease. LibFS first searches the file data cache with offset 0 and finds that the block is not in the cache (invalidated by the write above). Then, it searches the update log pointers with offset 0, finding a block in the update log. However, the update log does not contain the entire 4 KB (it has a 1 KB partial update). In that case, LibFS first finds the 4 KB block of the file by walking the extent tree at each layer from the inode. It finds the block in the SSD. To read it, it requests a new NVMe namespace for the block, which the kernel creates on the fly. This allows LibFS to read the block bypassing the kernel. LibFS allocates a file data cache entry (at the head of LRU list), reads the block into the cache entry, patches it with the update from the update log. LibFS can now return the complete block from the file data cache to the user.

Close. The application closes the file. At this point, LibFS relinquishes the lease to the KernelFS (if it still has it).

Digest. At a later point, the kernel digests the update log contents. It reads the same 4KB block from the SSD, patches the block with the 1 KB update from the log, and writes the complete block to a new location in NVM (the block was recently used). Next, it updates the extent tree nodes to point to the new location by first reading them from the appropriate layers and then writing them to NVM. Finally, it updates the inode containing the extent tree root pointer in NVM. The digest is done and LibFS garbage collects the update log

entry.

2.3 Implementation

We have implemented Strata from scratch, using Intel's Storage Performance Development Kit (SPDK) [Int17] for fast access to NVMe SSDs bypassing the Linux kernel and Intel's libpmem [pme17] to persist data to emulated NVM using non-temporal writes to avoid polluting the processor cache and the appropriate sequence of store fence and cache flush to guarantee persistence [ZS15] (our testbed does not support the optimized `CLWB` instruction, so libpmem uses `CLFLUSH` to flush the cache to NVM). We also use the extent tree implementation of the EXT4 [MCB+07] file system and modified it for log-structured update.

Our prototype of Strata is implemented in 21,255 lines of C code. Shared data structures, such as lists, trees, and hash tables, account for 4,201 lines. LibFS has 10,131 and KernelFS has 6,923 lines of code. The main functionality in LibFS is writing to the update log. In KernelFS it is the extent tree update code and code for data migration. On top of Strata's low-level API, we implement a POSIX system call interposition layer. To do so, we modify glibc to intercept each storage-related system call at user-level and invoke the corresponding LibFS version of the call. The interposition layer is implemented in 1,821 lines of C code.

Our prototype is able to execute a wide range of applications. Strata successfully completes all 201 unit tests of the LevelDB key-value store test suite, as well as all tests in Filebench.

2.3.1 Limitations

Our current prototype has a few limitations, which we describe here. None of them impact our evaluation.

Kernel. Instead of loading our kernel module into the kernel's address space, we have placed it in a separate process and use the sockets interface to communicate "system calls" between LibFS and KernelFS. This results in higher overhead for system calls in Strata due

to the required context switches. However, we believe the impact to be small, as a design goal of Strata is to minimize kernel-level system calls.

Leases. Leases are not fully implemented. We have evaluated their overhead, especially worst case performance (§2.2.4), but the prototype does not implement directory consistency, for example. Our benchmarks do not stress fine-grained concurrent sharing that would make lease performance relevant.

Memory mapped files. We did not implement memory mapped files because they are not used by our target applications. Memory mapped files increase write amplification for applications with small random writes. The hardware memory translation system is responsible for tracking updates to memory mapped files via dirty bits that are available only at a page granularity. A page is thus the smallest write unit. This is a general problem for memory mapped files, in particular as page sizes grow.

The common case of read-only mappings or writable private mappings are easy to accommodate in Strata. NVM pages can be mapped into a process' address space just as current OSes map page cache pages. The difficulty with shared writable mappings is their requirement that writes into memory are visible to other processes mapping the file. If writes must be immediately visible, Strata cannot do any user-level buffering and logging, but if writes can be delayed, Strata can buffer (and log) updates. On `msync`, LibFS writes updates (pages on which the dirty bit is set) to the log, and they are visible to other processes after digesting.

Fault tolerance. Strata currently does not contain any redundancy to compensate for storage device failures. Because it stores data across several devices, its mean time to data loss (MTTDL) will be the minimum of all devices. It remains future work to apply distributed reliability techniques to improve MTTDL in Strata [HSX+12, CLG+94]. With Strata it is also not safe to remove individual storage devices from a powered down machine, without advance warning.

2.4 Evaluation

We evaluate the performance and isolation properties of Strata. To put the performance of Strata into context, we compare it to a variety of purpose-built file systems for each storage layer. For NVM, we compare with the Linux EXT4-DAX [ext14] file system in its default ordered data mode, as well as to PMFS [DKK+14] and NOVA [XS16]. On the SSD, we compare to F2FS [LSHC15]. On the HDD, we compare to EXT4 [MCB+07], also in ordered data mode. Ordered mode is the Linux default for EXT4 because it provides the best tradeoff between performance and crash consistency.

To evaluate the data management and migration capabilities of Strata, we compare it to a user-space framework that migrates files among layers without being integrated into the file system, as well as to a block-level two-layer cache provided by Linux's logical volume manager (LVM) [lvm17]. The user-space management framework uses the NOVA, F2FS, and EXT4 file systems for the NVM, SSD, and HDD layers, respectively.The LVM cache uses the NVM and SSD layers, with a single F2FS file system formatted on top.

We seek to answer the following questions using our experimental evaluation.

- How efficient is Strata when logging to NVM and digesting to a storage layer? How does it compare to file systems designed for and operating on a single layer?
- How do common applications perform using Strata? How does performance compare on other file systems?
- How well does Strata perform when managing data across layers, compared to solutions above (at user-level) and below the file system (at the block layer)?
- What is the multicore scalability of Strata? How does it compare to other file systems?
- How isolated are multiple tenants when sharing Strata, compared to other file systems?

Testbed. Our experimental testbed consists of 2 $\times$ Intel Xeon E5-2640 CPU, 64 GB DDR3 RAM, 400 GB Intel 750 PCIe-SSD, 1 TB Seagate hard-disk, and a 40 GbE Mellanox MT27500 Infiniband network card. All experiments are performed on Ubuntu 16.04 LTS and Linux kernel 4.8.12. We reserve 36 GB of DRAM to emulate NVM and leave

the remaining 28 GB as DRAM. The other devices are used to capacity. Strata reserves 1 GB of write-only log area for each running application within NVM, the rest is dedicated to the shared area. To benefit from overlapping operations, LibFS starts a digest when its update log is 30% full. This value provided a good balance between digest overlap and log coalescing opportunities in a sensitivity study we conducted. All experiments involving network communication bypass the kernel using the rsockets [rso17] library.

NVM emulation. To emulate the performance characteristics of non-volatile memory, we have implemented a software layer that uses DRAM but delays memory accesses and limits its memory bandwidth to that of NVM. The emulation implements an NVM performance model according to a recent study [ZS15] (we could not obtain the PMEP hardware emulator used in the study).

The study predicts that NVM read latencies will be higher than DRAM. As done in NOVA and other studies, our model emulates this latency on all NVM reads by adding the latency differential to a DRAM read. In reality, read latency would be incurred only on a cache miss, but (like other studies) we do not emulate this behavior (making our model conservative). Due to write-back caching, writes do not have a direct latency cost as they reside in the cache. The study investigates the cost of a *write barrier* (e.g., Intel's PCOMMIT instruction) which ensures that flushed data does not remain in volatile buffers within the memory controller. Intel deprecated this write barrier from the x86 architecture [Rud16], instead requiring NVM controllers to be part of the system's power-fail safe persistence domain. Data flushed from the cache are guaranteed to be made persistent on a power fail due to on-chip capacitors. Thus, our model does not require write barriers and their attendant (non-trivial) latency. Strata uses the mandatory fences and cache flush to enforce ordering, incurring that cost. Finally, NVM is bandwidth-limited compared to DRAM by an estimated ratio of $\frac{1}{8}$. Our performance model tracks NVM bandwidth use and if a workload hits the device's bandwidth limit, the model applies a bandwidth-modeling delay $B = \frac{\sigma \times (1 - \text{NVM}_b/\text{DRAM}_b)}{\text{NVM}_b}$, with σ the size of the write IO in bytes, NVM_b the NVM

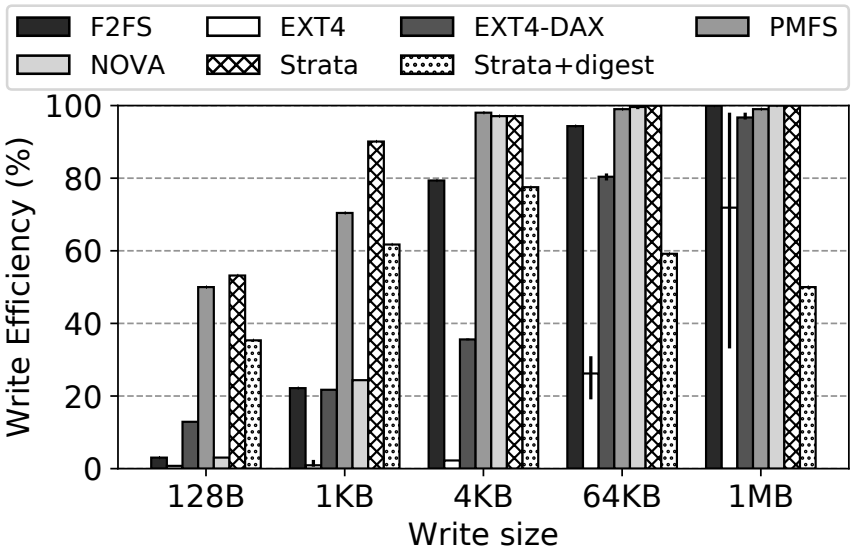


Figure 2.2: File system Zipf IO write efficiency. Error bars show minimum and maximum measurement.

bandwidth, and DRAM_b the DRAM bandwidth. The emulator resets the bandwidth limit every 10ms, which provided stable performance in a sensitivity study. With $\frac{\text{NVM}_b}{\text{DRAM}_b} = \frac{1}{8}$, we measure stable peak NVM bandwidth of 7.8GB/s as shown in Table 2.1.

2.4.1 Microbenchmarks

Hardware IO performance. To ensure that no other resource in our testbed system is a bottleneck, we first measure the achievable IO latency and throughput for each memory technology contained in our testbed server using sequential IO. The measured hardware IO performance matches the hardware specifications of the corresponding device (see Table 2.1). We measure DRAM using a popular memory bandwidth measuring tool [zsm17]. The reported NVM performance is in line with our NVM performance model.

File system write efficiency. Write amplification is a major factor in a file system's common case performance. Most file systems amplify writes by writing meta-data in addition to user data, lowering their *write efficiency* (defined as the inverse of write amplification). For example, if a program writes and syncs 2 KB of data and the file system updates and

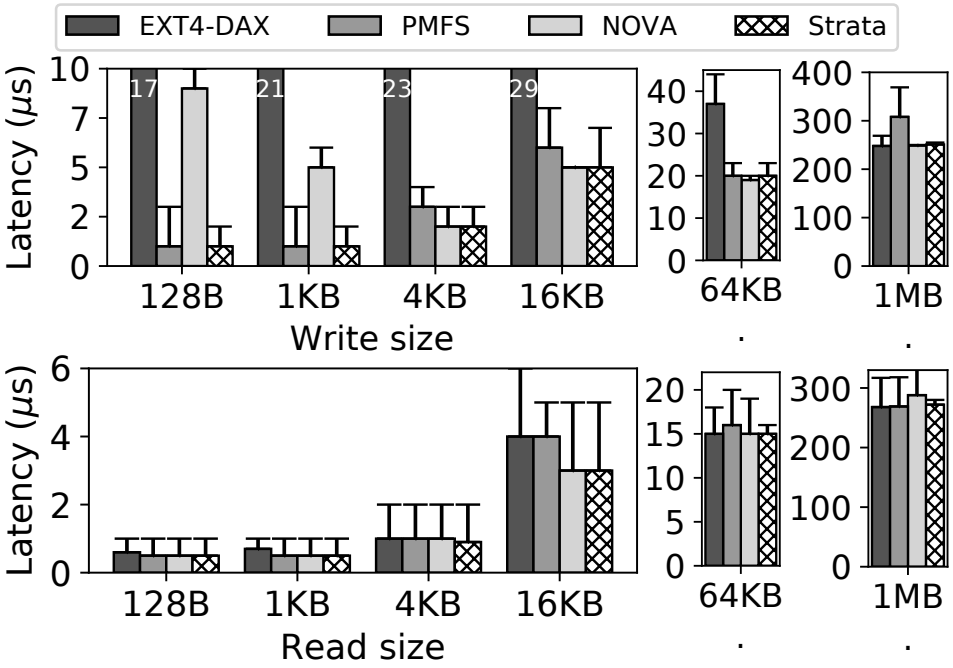


Figure 2.3: Average IO latency to NVM. Error bars show 99th percentile.

writes a 4 KB data block and a 4 KB metadata block, then the write amplification is 4 and the write efficiency is 25%.

Figure 2.2 shows write efficiency for Zipfian ($s = 1$) random writes until a total of 1 MB has been written. We can see that for small writes ($\leq$ 1 KB), write efficiency suffers substantially for most file systems. Strata achieves the highest write efficiency among all file systems regardless of write sizes because Strata performs the minimal amount of IO for persisting data and meta-data changes with the operation log. The Strata + digest result includes additional writes to digest data into the shared area in the background. Depending on IO properties, Strata can greatly improve write efficiency by coalescing the digest (§2.4.2). In this case, write efficiency declines as write size approaches total size and coalescing opportunities diminish.

Latency. We compare the read and synchronous (fsync) write latency of Strata and other NVM-optimized file systems using various IO sizes on an otherwise idle system. This

	1 KB	4 KB	64 KB	1 MB	4 MB
EXT4-DAX	35	44	98	812	2947
PMFS	7	10	53	656	2408
NOVA	13	17	54	563	2061
Strata	5	8	49	569	2074
No persist	4	6	30	302	1157

Table 2.3: Latency (μs) of (non-)persistent RPC.

experiment emulates the small, bursty writes often exhibited by cloud application back ends, such as key-value stores. Latency and tail-latency are of primary importance for these applications as it determines the processing latency of a user's web request.

In this experiment, the burst size is 1 GB. For Strata, this case is ideal as it fits into the update log. Hence, no digest occurs during the experiment. Before the read phase, Strata digests the file, hence all reads are served directly from the shared extent tree area. We assume this case to be common, as most key-value stores cache data in DRAM and so it is likely that recently written data can be served from DRAM.

Figure 2.3 shows read and write latencies. We can see that Strata achieves equivalent latency to the best-performing alternative file system, regardless of IO size. In the 99th-percentile tail, Strata achieves up to 33% lower latency for small writes and up to 12% for reads compared to the best-performing alternative file system.

Strata's performance comes from writing an operation log using kernel-bypass. In the other file systems, IO either involves copying data from user to kernel buffers (EXT4-DAX), or various persistent file system structures are modified in-place (PMFS), or they copy-on-write (NOVA), while Strata simply logs write operations using a single log write operation with no DRAM copying.

Persistent RPC. Many cloud services use remote procedure calls (RPCs) that must persist data before returning to the caller. Table 2.3 shows a microbenchmark, where one client sends an RPC to a server that logs it to stable storage. Strata can synchronously persist 1KB of data for each RPC only 1 μs (25%) later than an RPC that persists no data. Strata

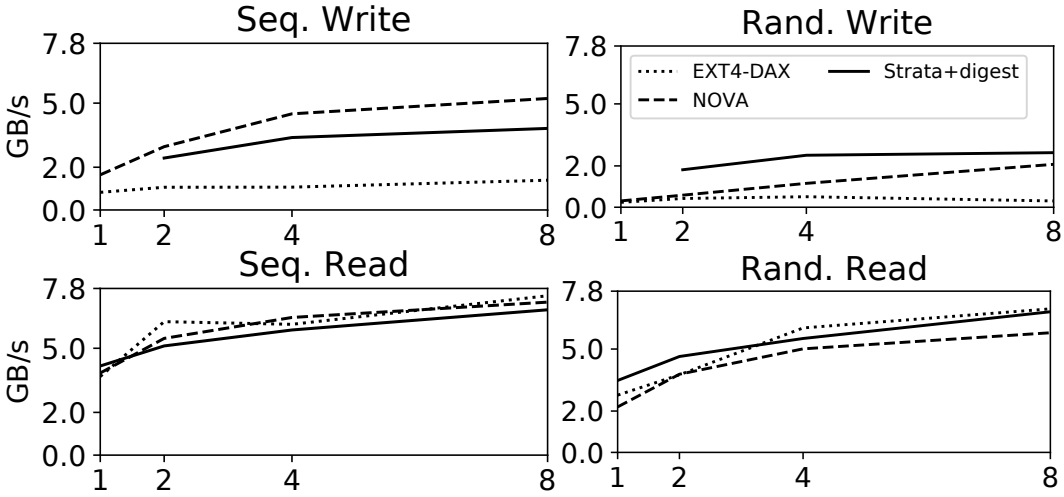


Figure 2.4: NVM throughput scalability (4 KB IO). X-axis = number of threads; Top Y-axis = NVM bandwidth.

makes persistence cheap in terms of latency for the data sizes that are common in RPCs ($\leq$4KB).

Log size sensitivity. Log size is configurable on a per-log basis. LibFS should configure log size according to expected burst behavior of the application. We conduct a sensitivity study to find how performance is affected by log size, analyzing the sequential write microbenchmark and the *varmail* workload from the Filebench suite [TZS16]. We vary the log size between 100 MB and 4 GB. We find that with a log size of 500 MB and larger, both workloads reach maximum performance. A 100 MB log degrades performance by up to 6.4%. This result confirms that Strata's background digest and garbage collection work efficiently even for small log sizes.

Throughput scalability. We compare sustained IO throughput using 4 KB IO size. When reading, update logs are clean. This benchmark emulates common data streaming workloads with sustained busy periods. Strata always logs data to NVM and digests it to the evaluated layer and our results include both operations. The other file systems operate directly on their respective layer, but F2FS and EXT4 use a buffer cache in DRAM. For Strata, we count LibFS (logging and reading) and KernelFS (digesting) threads.

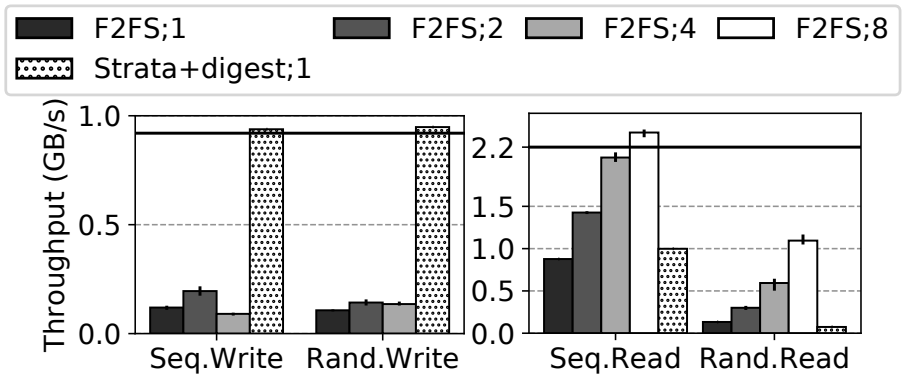


Figure 2.5: Average SSD throughput (4 KB IO) over number of threads. Horizontal lines = SSD bandwidth.

Figure 2.4 shows the result for NVM, varying the number of threads from 1 to 8. Each benchmark run conducts 30 GB of IO, partitioned over the number threads and using a private file for each thread to avoid application-level locking. PMFS crashes when using multiple threads. With 4 threads, Strata approaches NVM bandwidth. Since both workloads have no locality, write efficiency is 50% for Strata, resulting in an application-level throughput that is half the NVM bandwidth. Strata is up to 26% slower than NOVA for sequential writes. This is the worst case for Strata since KernelFS cannot improve write efficiency via digest optimizations. However, for random writes, Strata achieves 28% higher throughput than NOVA with 8 threads and 3$\times$ higher throughput with 2 threads. This is because LibFS blindly writes small, unaligned data to the log, and KernelFS can merge them into block writes when digesting the log, while NOVA has to read and modify blocks. EXT4-DAX is up to 10% faster than Strata on reads, but for a single-threaded workload Strata is 36% faster than EXT4-DAX for random reads.

Figure 2.5 shows SSD throughput of Strata and F2FS. We mount F2FS with the synchronous option to provide the same guarantee as Strata. For writes, Strata achieves 7.8$\times$ better throughput than F2FS by aggregating writes in the update log and batching them to SSD on digest. For sequential reads, both Strata and F2FS read ahead to achieve similar performance (our Strata prototype implementation currently supports only single-

[μs]	EXT4-DAX		PMFS		NOVA		Strata	
Mean	44.97	45.14	4.43	3.72	2.34	1.82	1.58	1.44
99th	74	74	7	6	4	3	2	2
99.9th	84	84	54	45	5	4	3	3
99.99th	95	95	72	50	10	6	6	5

Table 2.4: (Tail-)latencies with two clients (4 KB IO).

threaded SSD access). When accessing the HDD, Strata achieves $\sim$10% better throughput than EXT4 for all operations (full device bandwidth for sequential IO, 10 MB/s for random write and 3 MB/s for random read), but with synchronous write semantics and without any of EXT4's complexity. For example, Strata does not require a journal, delayed block allocation, or locality groups.

Using a portion of NVM as a persistent update log allows Strata to perform similarly or better than file systems purpose-built for each storage layer, while providing synchronous and in-order write semantics.

Data migration. To show Strata's read/write performance on multiple layers, we run a Zipfian ($s = 1$) benchmark with an 80:20 read/write ratio on a 120 GB file that fits in NVM and SSD and compare to F2FS (4 KB IO size). Strata achieves a throughput of 4.4 GB/s, while F2FS attains 1.3 GB/s. The locality of the workload causes most IO to be served from NVM, a benefit for Strata because of NVM's higher capacity compared with the DRAM buffer cache used by F2FS.

Isolation. Clients want the write performance allotted to them regardless of the activities of other clients. We run two processes that write and fsync (and digest using a kernel thread for each process in Strata) a burst of 4KB operations concurrently and observe write (tail-)latency to evaluate how well Strata isolates multiple clients, compared to other file systems. Table 2.4 shows latency experienced by two competing clients. We can see that Strata provides equivalent latencies to the single client case, while other file systems do not isolate clients as well. EXT4-DAX provides equality, but slows down under concurrent access. NOVA and PMFS do not provide equal performance to both clients. Strata allocates

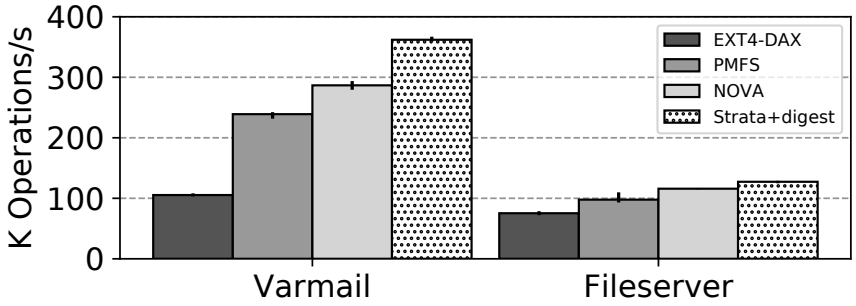


Figure 2.6: Varmail and Fileserver throughput.

fully isolated per-client write logs that can be allocated on client-local NUMA nodes, while other file systems use shared data structures for write operations that cause performance crosstalk because of locks and memory system effects.

2.4.2 Filebench: Mail and Fileserver

Mail servers access and create/delete many small files and are thus a good measure of Strata's meta-data management. File servers are similar, but operate on larger files and have a higher ratio of file IO compared with the number of directory operations. To evaluate these workloads, we use the *Varmail* and *Fileserver* profiles of the Filebench file system benchmark suite that is designed to mimic common mail and file servers. Both benchmarks operate on a working set of 10,000 files. Files are created with average sizes of 32 KB and 128 KB for Varmail and Fileserver, respectively, though files grow via 16 KB appends in both benchmarks. Both workloads read and write data at 1 MB granularity. Varmail and Fileserver have write to read ratios of 1:1 and 2:1, respectively. In this configuration, all workloads fit into NVM and thus we compare performance to NVM file systems.

Varmail uses a write-ahead log for crash consistency (**sync** signifies Varmail waiting for persistence). Its application level crash consistency protocol involves creating and writing the write-ahead log in a separate file, **sync**, appending a mail to the mailbox file, **sync**, and then garbage collecting the now redundant write-ahead log by deleting the separate file. The Fileserver workload is similar to the SPECsfs [spe17] benchmark. It randomly performs file creates, deletes, appends, reads, writes, and attribute operations on a directory

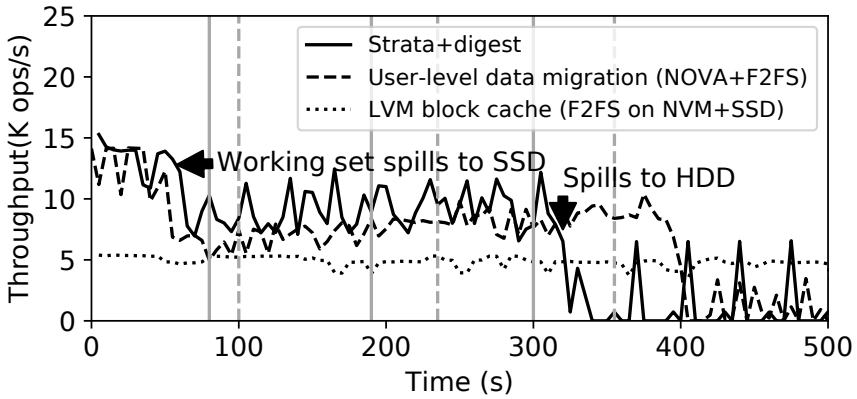


Figure 2.7: Fileserver throughput on multi-layer storage over time. Vertical lines every 1 million operations.

tree.

Figure 2.6 shows the result. Varmail achieves 26% higher write throughput on Strata versus NOVA: 57% of the time is spent in Varmail; 18% is spent in reading application data from NVM and 14% in writing application data to the log area. Another 10% is spent searching directories. Less than 1% of the time is spent in other Strata activities, such as searching extent trees. Fileserver throughput improvements are smaller (7% versus NOVA). This is expected; Fileserver has a larger average write size and no crash consistency protocol.

Strata's log compaction strategy is a good fit for Varmail, which creates and deletes many temporary files and performs many temporary durable writes. Strata digestion skips 86% of the log records because those updates have been superseded by subsequent workload updates: 50% are data writes, 24% are directory updates, and 12% are file creates. For example, if the workload creates a temporary file, it can write and read the file, but if it deletes the file before digestion Strata does not need to copy the file's data and metadata to the shared area. This optimization avoids 14 GB of data copying.

2.4.3 Data Migration

To show performance when Strata uses multiple storage devices, we configure the Fileserver workload to do 1 MB appends with 1000 files. In this case, the working set starts out operating in NVM, but then grows to incorporate the SSD and HDD. Fileserver's workload is uniform random. It has no locality. Thus, all evaluated systems only migrate data down towards slower layers.

We compare Strata to a user-space data migration (UDM) framework using NOVA, F2FS, and EXT4 file systems on the appropriate layers with default mount options. UDM migrates files instead of blocks, which is a best case for UDM because the Fileserver workload performs mostly complete file I/O. We also compare to Linux's logical volume manager (LVM) cache target [lvm17]. LVM caches blocks of a fixed *cache chunk* size underneath the file system. We cache SSD blocks in NVM, running F2FS on top (outperforming EXT4) in synchronous mode (-o sync) for a fair comparison. We cannot use EXT4-DAX or NOVA since LVM requires a block abstraction. To keep crash-consistent block-to-device mappings, it persists cache meta-data to a separate partition. We configure LVM to provide write-back caching with 64 KB cache chunk, reserving 300 MB of NVM space to store the cache meta-data.

Figure 2.7 shows the result. Both Strata and UDM start with full throughput on NVM, but UDM demonstrates more jitter. This is because of log garbage collection in NOVA. After 80 seconds, the working set sizes are large enough that data starts migrating to SSD, and it quickly becomes the bottleneck. In this period, digesting is slower than logging so the application stalls on a full log (between spikes in Strata). UDM's throughput drops below that of Strata causing UDM to fall behind (vertical lines). UDM lags because it migrates entire files, rather than individual blocks. Strata's workload grows to include HDD after 310 seconds (90 seconds later for UDM due to lower throughput) and throughput drops significantly.

Both Strata and UDM start out attaining 2.8$\times$ better throughput than F2FS on top of

	EXT4-DAX	PMFS	NOVA	Strata
Write sync.	**49.2**	**18.9**	**35.2**	**17.1**
Write seq.	8.7	5.4	15.0	4.9
Write rand.	19.5	15.2	25.0	11.1
Overwrite	28.0	20.9	37.7	17.3
Delete rand.	5.6	3.6	12.3	3.3
Read seq.	1.2	1.1	1.1	1.1
Read rand.	6.3	5.8	6.7	5.8
Read hot	1.6	1.5	1.5	1.5

Table 2.5: Latency [μs] for LevelDB benchmarks.

LVM. Once the working set spills to SSD, Strata is 2.0× faster than LVM. Note that LVM does not use the HDD, which is why it maintains its throughput throughout the duration of the experiment. LVM's working set never grows beyond the size of the SSD. Strata's approach to managing multi-layer storage offers higher throughput compared to multi-layer caching at the block layer. Strata can leverage the low latency and byte addressability of NVM directly by providing a user-level update log, while LVM requires a block-level interface in the kernel. LVM also adds additional cache meta-data IO to every file system IO.

Strata's cross-layer approach also performs well compared to a solution operating above the file system, treating each layer as a black box. Strata benefits from combining application access pattern knowledge and cross-layer block allocation. Hence, Strata can maintain more meta-data in faster layers to speed up file system data structure traversal.

2.4.4 Key-value Store: LevelDB

Modern cloud applications use key-value stores like LevelDB [DG11]. LevelDB exports an API allowing applications to process and query structured data, but uses the file system for data storage. We run a number of LevelDB benchmarks, including sequential, random, and random IO with 1% of hot key-value pairs with a key size of 16 bytes and a value size of 1 KB on a working set of 300,000 objects. We measure the average operation latency. This workload fits into NVM and we compare against NVM file systems. To achieve higher

performance, LevelDB does not wait for persistence after each write operation, but Strata guarantees persistence due to its synchronous semantics. We introduce a synchronous random write category (bold font in Table 2.5) to show a case where persistence is requested upon every operation by LevelDB.

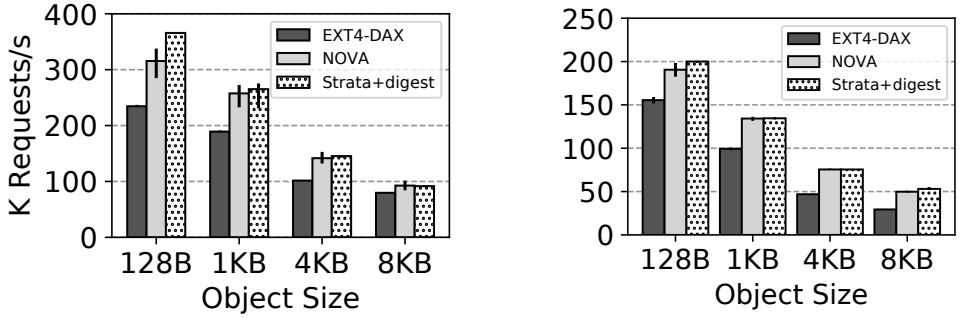
Table 2.5 shows the result. We can see that LevelDB achieves lower latency on Strata than on any of the other NVM file systems, regardless of workload. In particular for random writes and overwrites, Strata performs 27% and 17% better than PMFS, respectively, while providing synchronous write semantics. NOVA does not perform well on this workload, as it uses a copy-on-write approach, which has high latency overhead, while EXT4-DAX incurs kernel overhead.

Our experiment demonstrates that a file system with a simple, synchronous IO interface can provide low latency IO if the underlying storage device is fast. Modern applications struggle to make logically consistent updates that are crash recoverable [PCA+14] and Strata helps such systems by providing simple recovery semantics. For example, SQLite must call `fsync` repeatedly to persist data to its log, to persist the log's entry in its parent directory, and to persist its data file so it can reclaim the log. All of these `fsync`s are unnecessary when file system operations become persistent synchronously.

2.4.5 Redis

Redis [red17] is an example of a key-value store that is typically used in a replicated, distributed scenario. Redis either logs operations to an append-only-file (AOF) or uses an asynchronous snapshot mechanism. Only the AOF provides persistence guarantees for all operations, as snapshots are only persisted at larger time intervals.

Standalone. We start by benchmarking a single Redis instance. We configure it to use AOF mode and to persist data synchronously, before acknowledging a user's request. Figure 2.8a shows the throughput of SET operations using 12 byte keys and with various value sizes. Redis achieves up to 22% higher throughput on Strata, compared to NOVA for small values, which is the common case for key-value stores [AXF+12].



(a) Standalone

(b) Replicated

Figure 2.8: Redis SET throughput.

Replication. Redis supports replication for fault tolerance. Figure 2.8b shows the throughput of Redis with a single replica, which must persist its record before the master can acknowledge a request. Redis throughput drops by about half relative to the non-replicated case due to the extra network round-trip (see Table 2.3 for round-trip and persistence latencies). Strata improves throughput by up to 29% relative to EXT4-DAX and retains a 5% improvement over NOVA. Persistence in Strata is fast enough for use in high-performance network servers, and Strata's cross-layer management provides more capacity than an NVM-only file system.

2.5 Summary

Trends in storage hardware encourage a multi-layer storage topology spanning multiple orders of magnitude in cost and performance. File systems should manage these multiple storage layers to provide advanced functionality like efficient small writes, synchronous semantics, and strong QoS guarantees.

Chapter 3

Ingens: Coordinated and efficient huge page management system

Modern computing platforms can support terabytes of RAM and workloads able to take advantage of such large memories are now commonplace [FAK+12]. However, increased capacity represents a significant challenge for address translation. All modern processors use page tables for address translation and TLBs to cache virtual-to-physical mappings. Because TLB capacities cannot scale at the same rate as DRAM, TLB misses and address translation can incur crippling performance penalties for large memory workloads [BGC+13, GBHS14] when these workloads use traditional page sizes (i.e., 4KB). Hardware-supported address virtualization (e.g., AMD's nested page tables) increases average-case address translation overhead because multi-dimensional page tables amplify worst-case translation costs by $6\times$ [Int16]. Hardware manufacturers have addressed increasing DRAM capacity with better support for larger page sizes, or *huge pages*, which reduce address translation overheads by reducing the frequency of TLB misses. However, the success of these mechanisms is critically dependent on the ability of the operating systems and hypervisors to manage huge pages.

While huge pages have been commonly supported in hardware since the 90s [Sha96, SW98], until recently, processors have had a very small number of TLB entries reserved for huge pages, limiting their usability. Newer architectures support thousands of huge page

entries in dual-level TLBs (e.g., 1,536 in Intel's Skylake [inta]), which is a major change: the onus of better huge page support has shifted from the hardware to the system software. There is now both an urgent need and an opportunity to modernize memory management.

Operating system memory management has generally responded to huge page hardware with best-effort algorithms and spot fixes, choosing to keep their management algorithms focused on the 4KB page (which we call a *base page*). For example, Linux and KVM (Linux's in-kernel hypervisor) adequately support many large-memory workloads (i.e., ones with simple, static memory allocation behavior), but a variety of common workloads are exposed to unacceptable performance overheads, wasted memory capacity, and unfair performance variability when using huge pages. These problems are common and severe enough that administrators generally disable huge pages (e.g., MongoDB, Couchbase, Redis, SAP, Splunk, etc.) despite their obvious average-case performance advantages [Monb, cou, dok, reda, nuo, sap, spl, vol]. Other operating systems have similar or even more severe problems supporting huge pages (see §3.1.2 and §3.2.4).

Ingens[1] is a memory manager for the operating system and hypervisor that replaces the best-effort mechanisms and spot-fixes of the past with a coordinated, unified approach to huge pages; one that is better targeted to the increased TLB capacity in modern processors. Ingens does not interfere with workloads that perform well with current huge page support: the prototype adds 0.7% overhead on average (Table 3.4). Ingens addresses the following problems endemic to current huge page support, and we quantify the impact of these problems on real workloads using our prototype.

• **Latency.** Huge pages expose applications to high latency variation and increased tail latency (§3.2.1). Ingens improves the Cloudstone benchmark [SSS+08] by 18% and reduces 90th percentile tail-latency by 41%.

• **Bloat.** Huge pages can make a process or virtual machine (VM) occupy a large amount of physical memory while much of that memory remains unusable due to internal

[1] Ingens is Latin for huge.

fragmentation (§3.2.2). For Redis, Linux bloats memory use by 69%, while Ingens bloats by just 0.8%.

• **Unfairness.** Simple, greedy allocation of huge pages is unfair, causing large and persistent performance variation across identical processes or VMs (§3.2.5). Ingens makes huge page allocation fair (e.g., Figure 3.5).

• **High-performance memory savings.** Services that reduce memory consumption, such as kernel same-page merging (KSM), can prevent a VM from using huge pages (§3.2.6). On one workload (Figure 3.11), Linux saves 9.2% of memory but slows down the programs by 6.8–19%. Ingens saves 71.3% of the memory that Linux/KVM can save with only a 1.5–2.6% slowdown.

Ingens is a memory management redesign that brings performance, memory savings and fairness to memory-intensive applications with dynamic memory behavior. It is based on two principles: (1) memory contiguity is an explicit resource to be allocated across processes and (2) good information about spatial and temporal access patterns is essential to managing contiguity; it allows the OS to tell/predict when contiguity is/will be profitably used. The measured performance of the Ingens prototype on realistic workloads validates the approach.

3.1 Background

Current trends in memory management hardware are making it critical that system software support huge pages efficiently and flexibly. This section considers those trends along with the challenges huge page support creates for the OS and hypervisor. We provide an overview of huge page support in modern operating systems and conclude with experiments that show the performance benefits for the state-of-the-art in huge page management.

3.1.1 Virtual memory hardware trends

Virtual memory decouples the address space used by programs from that exported by physical memory (RAM). A page table maps virtual to physical page number, with recently used

page table entries cached in the hardware translation lookaside buffer (TLB). Increasing the page size increases TLB *reach* (the amount of data covered by translations cached in the TLB), but larger pages require larger regions of contiguous physical memory. Large pages can suffer from internal fragmentation (unused portions within the unit of allocation) and can also increase external fragmentation (reducing the remaining supply of contiguous physical memory). Using larger pages requires more active memory management from the system software to increase available contiguity and avoid fragmentation.

Seminal work in huge page management recognized the importance of explicitly managing memory contiguity in the OS [NIDC02] and formed the basis for huge page support in FreeBSD. Innovations of Ingens relative to previous work are considered in detail in Section 3.2.4; here we survey recent hardware trends that make the need for system support of huge pages more urgent.

DRAM Growth. Larger DRAM sizes have led to deeper page tables, increasing the number of memory references needed to look up a virtual page number. x86 uses a 4-level page table with a worst case of four page table memory references to perform a single address translation.

Hardware memory virtualization. Extended page tables (Intel) or nested page tables (AMD) require additional indirection for each stage of memory address translation, making the process of resolving a virtual page number even more complex. With extended page tables, both the guest OS and host hypervisor perform virtual to physical translations to satisfy a single request. During translation, guest physical addresses are treated as host virtual addresses, which use hardware page-table walkers to perform the entire translation. Each layer of lookup in the guest can require a multi-level translation in the host, amplifying the maximum cost to 24 lookups [Int16, AMD10], and increasing average latencies [MT16].

Increased TLB reach. Recently, Intel has moved to a two-level TLB design, and in the past few years has provided a significant number of second-level TLB entries for huge pages, going from zero for Sandy Bridge and Ivy Bridge to 1,024 for Haswell [intb] (2013)

Name	Suite/Application	Description
429.mcf	SPEC CPU 2006 [spe]	Single-threaded scientific computation
Canneal	PARSEC 3.0 [par]	Parallel scientific computation
SVM [LL14]	Liblinear [Liba]	Machine learning, Support vector machine
Tunkrank [tun]	PowerGraph [GLG+12]	Large scale in-memory graph analytics
Nutch [nut]	Hadoop [hada]	Web search indexing using MapReduce
MovieRecmd [mov]	Spark/MLlib [spa]	Machine learning, Movie recommendation
Olio	Cloudstone [tun]	Social-event Web service (ngnix/php/mysql)
Redis	Redis [red17]	In-memory Key-value store
MongoDB	MongoDB [mona]	In-memory NoSQL database

Table 3.1: Summary of memory intensive workloads.

Issue	OS	Hyp
Page fault latency (§3.2.1)	O	
Bloat (§3.2.2)	O	
Fragmentation (§3.2.3)	O	O
Unfair allocation (§3.2.5)	O	O
Memory sharing (§3.2.6)		O

Table 3.2: Summary of issues in Linux as the guest OS and KVM as the host hypervisor.

and 1,536 for Skylake [inta] (2015).

Better hardware support for multiple page sizes creates an opportunity for the OS and the hypervisor, but it puts stress on the current memory management algorithms. In addition to managing the complexity of different page granularities, system software must generate and maintain significant memory contiguity to use larger page sizes.

3.1.2 Operating system support for huge pages

Early operating system support for huge pages provided a separate interface for explicit huge page allocation from a dedicated huge page pool configured by the system administrator. Windows and OS X continue to have this level of support. In Windows, applications must use an explicit memory allocation API for huge page allocation [win] and Windows recommends that applications allocate huge pages all at once when they begin. OS X applications also must set an explicit flag in the memory allocation API to use huge pages [osx].

Initial huge page support in Linux used a similar separate interface for huge page allocation that a developer must invoke explicitly (called `hugetlbfs`). Developers did not like the burden of this alternate API and kernel developers wanted to bring the benefits of huge pages to legacy applications and applications with dynamic memory behavior [hug, thpa]. Hence, the primary way huge pages are allocated in Linux today is *transparently* by the kernel.

Transparent support is vital. Transparent huge page support [thpb, NIDC02] is the only practical way to bring the benefits of huge pages to all applications, which can remain unchanged while the system provides them with the often significant performance advantages of huge pages. With transparent huge page support, the kernel allocates memory to applications using base pages. We say the kernel **promotes** a sequence of 512 properly aligned pages to a huge page (and demotes a huge page into 512 base pages).

Transparent management of huge pages best supports the multi-programmed and dynamic workloads typical of web applications and analytics where memory is contended and access patterns are often unpredictable. To the contrary, when a single big-memory application is the only important program running, the application can simply map a large region and keep it mapped for the duration of execution, for example fast network functions using Intel's Data Plane Development Kit [dpd]. These simple programs are well supported by even the rudimentary huge page support in Windows and OS X. However, multi-programmed workloads and workloads with more complex memory behavior are common in enterprise and cloud computing, so Ingens focuses on OS support for these more challenging cases. While transparent huge page support is far more developer-friendly than explicit allocation, it creates memory management challenges in the operating system that Ingens addresses.

Linux running on Intel processors currently has the best transparent huge page support among commodity OSes so we base our prototype on it and most of our discussion focuses on Linux. We quantify Linux's performance advantages in Section 3.2.4. The

design of Ingens focuses on 4 KB (base) and 2 MB (huge) pages because these are most useful to applications with dynamic memory behavior (1 GB are usually too large for user data structures).

Linux is greedy and aggressive. Linux's huge page management algorithms are greedy: it promotes huge pages in the page fault handler based on local information. Linux is also aggressive: it will always try to allocate a huge page. Huge pages require 2 MB of contiguous free physical memory but sometimes contiguous physical memory is in short supply (e.g., when memory is fragmented). Linux's approach to huge page allocation works well for simple applications that allocate a large memory region and use it uniformly, but we demonstrate many applications that have more complex behavior and are penalized by Linux's greedy and aggressive promotion of huge pages (§3.2). Ingens recognizes that memory contiguity is a valuable resource and explicitly manages it.

3.1.3 Hypervisor support for huge pages

Ingens focuses on the case where Linux is used both as the guest operating system and as the host hypervisor (i.e., KVM [KKL+07]). The Linux/KVM pair is widely used in cloud deployments [ope, ibma, cloa]. In the hypervisor, Ingens supports host huge pages mapped from guest physical memory. When promoting guest physical memory, Ingens modifies the extended page table to use huge pages because it is acting as a hypervisor, not as an operating system.

Because operating system and hypervisor memory management are unified in Linux, Ingens adopts the unified model. Some of the problems with huge pages that we describe in Section 3.2 only apply to the OS and some only to the hypervisor (summarized in Table 3.2). For example, addressing memory sharing vs. performance (§3.2.6) requires only hypervisor modifications and would be as successful for a Windows guest as it is for a Linux guest. We leave for future work determining the most efficient way to implement Ingens for operating systems and hypervisors that do not share memory management code.

Workloads	h_B g_H	h_H g_B	h_H g_H
429.mcf	1.18	1.13	1.43
Canneal	1.11	1.10	1.32
SVM	1.14	1.17	1.53
Tunkrank	1.11	1.11	1.30
Nutch	1.01	1.07	1.12
MovieRecmd	1.03	1.02	1.11
Olio	1.43	1.08	1.46
Redis	1.12	1.04	1.20
MongoDB	1.08	1.22	1.37

Table 3.3: Application speed up for huge page (2 MB) support relative to host (h) and guest (g) using base (4 KB) pages. For example, h_B means the host uses base pages and h_H means the host uses both base and huge pages.

3.1.4 Performance improvement from huge pages

Table 3.1 describes a variety of memory-intensive real-world applications including web infrastructure such as key/value stores and databases, as well as scientific applications, data analytics and recommendation systems. Measurements with hardware performance counters show they all spend a significant portion of their execution time doing page walks. For example, when using base pages for both guest and host, we measure 429.mcf spending 47.5% of its execution time doing page walks (24.2% for the extended page table and 23.3% for the guest page table). On the other hand, 429.mcf spends only 4.2% of its execution time walking page tables when using huge pages for both the guest and host.

We execute all workloads in a KVM virtual machine running Linux with default transparent huge page support [thpb] for both the application (in the guest OS) and the virtual machine (in the host OS). The hardware configuration is detailed in Section 3.5.

Table 3.3 shows the performance improvements gained with transparent huge page support for both the guest and the host operating system. The table shows speedup normalized to the case where both host and guest use only base pages. In every case, huge page support helps performance, often significantly (up to 53%). The largest speedup is always

attained when both host and guest use huge pages.

These results show the value of huge page support and show that Linux's memory manager can obtain that benefit under simple operating conditions. However, a variety of more challenging circumstances expose the limitations of Linux's memory management.

3.2 Current huge page problems

This section quantifies the limitations in performance and fairness for the state-of-the-art in transparent huge page management. We examine virtualized systems with Linux/KVM as the guest OS and hypervisor. The variety and severity of the limitations motivate our redesign of page management. All data is collected using the experimental setup described in Section 3.1.4.

3.2.1 Page fault latency and synchronous promotion

When a process faults on an anonymous memory region, the page fault handler allocates physical memory to back the page. Both base and huge pages share this code path. Linux is greedy and aggressive in its allocation of huge pages, so if an application faults on a base page, Linux will immediately try to upgrade the request and allocate a huge page if it can.

This greedy approach fundamentally increases page fault latency for two reasons. First, Linux must zero pages before returning them to the user. Huge pages are $512\times$ larger than base pages, and thus are much slower to clear. Second, huge page allocation requires 2 MB of physically contiguous memory. When memory is fragmented, the OS often must compact memory to generate that much contiguity. Previous work shows that memory quickly fragments in multi-tenant cloud environments [AMM+11]. When memory is fragmented, Linux will often synchronously compact memory in the page fault handler, increasing average and tail latency.

To measure these effects, we compare page fault latency when huge pages are enabled and disabled, in fragmented and non-fragmented settings. We quantify fragmentation using the *free memory fragmentation index* (FMFI) [GW05], a value between 0 (unfrag-

SVM	Synchronous	Asynchronous
Exec. time (sec)	178 (1.30×)	228 (1.02×)
Huge page	4.8 GB	468 MB
Promotion speed	immediate	1.6 MB/s

Table 3.4: Comparison of synchronous promotion and asynchronous promotion when both host and guest use huge pages. The parenthesis is speedup compared to not using huge pages. We use the default asynchronous promotion speed of Ubuntu 14.04.

mented) and 1 (highly fragmented). A microbenchmark maps 10 GB of anonymous virtual memory and reads it sequentially.

When memory is unfragmented (FMFI < 0.1), page clearing overheads increase average page fault latency from 3.6 μs for base pages only to 378 μs for huge pages (105× slower). When memory is heavily fragmented, (FMFI $= 0.9$), the 3.6 μs average latency for base pages grows to 8.1 μs (2.1× slower) for base and huge pages. Average latency is lower in the fragmented case because 98% of the allocations fall back to base pages (e.g. because memory is too fragmented to allocate a huge page). Compacting and zeroing memory in the page fault handler penalizes applications that are sensitive to average latency and to tail latency, such as Web services.

To avoid this additional page fault latency, Linux can promote huge pages asynchronously, based on a configurable asynchronous promotion speed (in MB/s). Table 3.4 shows performance measurements for asynchronous-only huge page promotion when executing SVM in a virtual machine. Asynchronous-only promotion turns a 30% speedup into a 2% speedup: it does not promote fast enough. Simply increasing the promotion speed does not solve the problem. Earlier implementations of Linux did more aggressive asynchronous promotion, incurring unacceptably high CPU utilization for memory scanning and compaction. The CPU use of aggressive promotion reduced or in some cases erased the performance benefits of huge pages, causing users to disable transparent huge page support in practice [ibmb, mys, hadb, clob].

Workload	Using huge pages	Not using huge pages
Redis	20.7 GB (1.69×)	12.2 GB
MongoDB	12.4 GB (1.23×)	10.1 GB

Table 3.5: Physical memory size of Redis and MongoDB.

3.2.2 Increased memory footprint (bloat)

Huge pages improve performance, but applications do not always fully utilize the huge pages allocated to them. Linux greedily allocates huge pages even though underutilized huge pages create internal fragmentation. A huge page might eliminate TLB misses, but the cost is that a process using less than a full huge page has to reserve the entire region.

Table 3.5 shows memory bloat from huge pages when running Redis and MongoDB, each within their own virtual machine. For Redis, we populate 2 million keys with 8 KB objects and then delete 70% of the keys randomly. Redis frees the memory backing the deleted objects which leaves physical memory sparsely allocated. Linux promotes the sparsely allocated memory to huge pages, creating internal fragmentation and causing Redis to use 69% more memory compared to not using huge pages. We demonstrate the same problem in MongoDB, making 10 million `get` requests for 15 million 1 KB objects which are initially in persistent storage. MongoDB allocates the objects sparsely in a large virtual address space. Linux promotes huge pages including unused memory, and as a result, MongoDB uses 23% more memory relative to running without huge page support.

Greedy and aggressive allocation of huge pages makes it impossible to predict an application's total memory usage in production because memory usage depends on huge page use, which in turn depends on memory fragmentation and the allocation pattern of applications. Table 3.5 shows if an administrator provisions 18 GB memory (1.5× overprovisioning relative to using only base pages), Redis starts swapping when it uses huge pages, negating the benefits of caching objects in memory [redb].

While these experiments illustrate the potential impact of bloat for a handful of workloads, it is important to note that the problem is fundamental to Linux's current design.

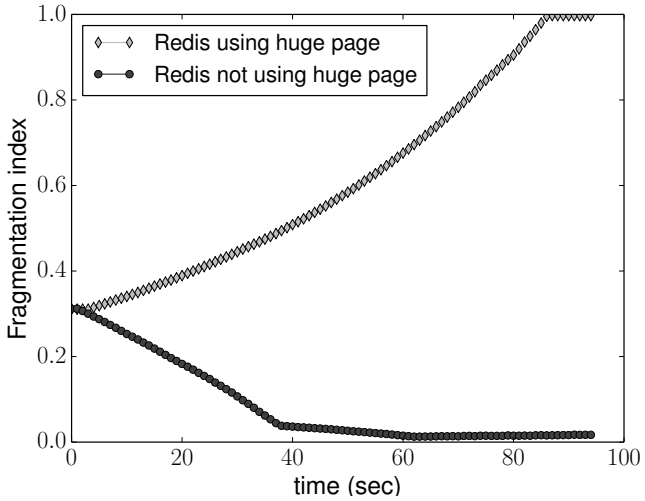


Figure 3.1: Fragmentation index in Linux when running a Redis server, with Linux using (and not using) huge pages. The System has 24 GB memory. Redis uses 13 GB, other processes use 5 GB, and system has 6 GB free memory.

Memory bloating can happen in any working set, memory, and TLB size: application-level memory usage can conspire with aggressive promotion to create internal fragmentation that the OS cannot address. In such situations, such applications will eventually put the system under memory pressure regardless of physical memory size.

3.2.3 Huge pages increase fragmentation

One common theme in analyzing page fault latency (§3.2.1) and memory bloat (§3.2.2) is Linux's greedy allocation and promotion of huge pages. We now measure how aggressive promotion of huge pages quickly consumes available physical memory contiguity, which then increases memory fragmentation for the remaining physical memory. Increasing fragmentation is the precondition for problems with page fault latency and memory bloat, so greedy promotion creates a vicious cycle. We again rely on the free memory fragmentation index, or FMFI to quantify the relationship between huge page allocation and fragmentation.

Figure 3.1 shows the fragmentation index over time when running the popular key-value store application Redis in a virtual machine. Initially, the system is lightly fragmented

OS	SVM	Canneal	Redis
FreeBSD	1.28	1.13	1.02
Linux	1.30	1.21	1.15

Table 3.6: Performance speedup when using huge page in different operating systems.

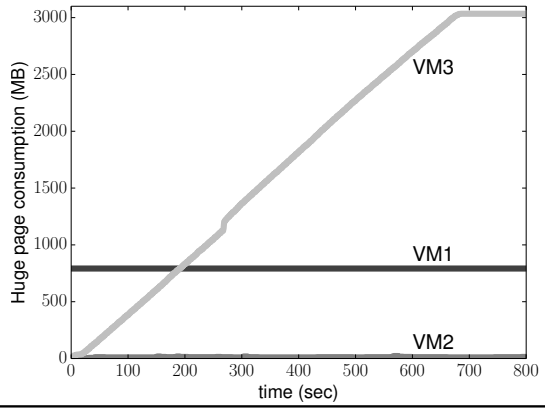
(FMFI = 0.3) by other processes. Through the measurement period, Redis clients populate the server with 13 GB of key/value pairs. Redis rapidly consumes contiguous memory as Linux allocates huge pages to it, increasing the fragmentation index. When the FMFI is equal to 1, the remaining physical memory is so fragmented, Linux starts memory compaction to allocate huge pages.

3.2.4 Comparison with FreeBSD huge page support

FreeBSD supports transparent huge pages using reservation-based huge page allocation [NIDC02]. When applications start accessing a 2 MB virtual address region, the page fault handler reserves contiguous memory, but does not promote the region to a huge page. It allocates base pages from the reserved memory for subsequent page faults in the region. FreeBSD monitors page utilization of the region and promotes it to a huge page only when all base pages of the reserved memory are allocated. FreeBSD is therefore slower to promote huge pages than Linux and promotion requires complete utilization of a 2 MB region.

FreeBSD supports huge pages for file-cached pages. x86 hardware maintains access/dirty bits for entire huge pages—any read or write will set the huge page's access/dirty bit. FreeBSD wants to avoid increasing IO traffic when evicting from the page cache or swapping. Therefore it is conservative about creating writable huge pages. When FreeBSD promotes a huge page, it marks it read-only, with writes demoting the huge page. Only when all pages in the region are modified will FreeBSD then promote the region to a writable huge page. The read-only promotion design does not increase IO traffic from the page cache because huge pages consist of either all clean (read-only) or all modified base pages.

FreeBSD promotion of huge pages is more conservative than in Linux, which reduces memory bloating, but yields slower performance. Table 3.6 compares the perfor-



SVM	VM1	VM2	VM3
Exec. time (sec)	533 (1.12×)	589 (1.24×)	475

Figure 3.2: Unfair allocation of huge pages in KVM. Three virtual machines run concurrently, each executing SVM. The line graph is huge page size (MB) over time and the table shows execution time of SVM for 2 iterations.

mance benefits of huge pages in FreeBSD and Linux. Applications with dense, uniform access memory patterns (e.g., SVM) enjoy similar speedups on Linux and FreeBSD. However, FreeBSD does not support asynchronous promotion, so applications which allocate memory gradually (e.g., Canneal) show less benefit. Redis makes frequent hash table updates and exhibits many read-only huge page demotions in FreeBSD. Consequently, Redis also shows limited speedup compared with Linux.

3.2.5 Unfair performance

All of our measurements are on virtual machines where Linux is the guest operating system, and KVM (Linux's in-kernel hypervisor) is the host hypervisor. Ingens modifies the memory management code of both Linux and KVM. The previous sections focused on problems with operating system memory management, the remaining sections describe problems with KVM memory management.

Unfair huge page allocation can lead to unfair performance differences when huge pages become scarce. Linux does not fairly redistribute contiguity, which can lead to unfair performance imbalance. To demonstrate this problem, we run 4 virtual machines in a setting

Policy	Mem saving	Performance slowdown	H/M
No sharing	–	429.mcf: 278 SVM: 191 Tunkrank: 236	429.mcf: 99% SVM: 99% Tunkrank: 99%
KVM (Linux)	1.19 GB (9.2%)	429.mcf: 331 (19.0%) SVM: 204 (6.8%) Tunkrank: 268 (13.5%)	429.mcf: 66% SVM: 90% Tunkrank: 69%
Huge page sharing	199 MB (1.5%)	429.mcf: 278 (0.0%) SVM: 194 (1.5%) Tunkrank: 238 (0.8%)	429.mcf: 99% SVM: 99% Tunkrank: 99%

Table 3.7: Memory saving and performance trade off for a multi-process workload. Each row is an experiment where all workloads run concurrently in separate virtual machines. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (13 GB) allocated to all three virtual machines.

where memory is initially fragmented (FMFI = 0.85). Each VM uses 8 GB of memory. VM0 starts first and obtains all huge pages that are available (3 GB). Later, VM1 starts and begins allocating memory, during which VM2 and VM3 start. VM0 then terminates, releasing its 3 GB of huge pages. We measure how Linux redistributes that contiguity to the remaining identical VMs.

The graph in Figure 3.2 shows the amount of huge page memory allocated to VM1, VM2, and VM3 (all running SVM) over time, starting 10 seconds before the termination of VM0. When VM1 allocates memory, Linux compacts memory for huge page allocation, but compaction begins to fail at 810 MB. VM2 and VM3 start without huge pages. When VM0 terminates 10 seconds into the experiment, Linux allocates all 3 GB of recently freed huge pages to VM3 through asynchronous promotion. This creates significant and persistent performance inequality among the VMs. The table in Figure 3.2 shows the variation in performance (NB: to avoid IO measurement noise, data loading time is excluded from the measurement). In a cloud provider scenario, with purchased VM instances of the same type, users have good reason to expect similar performance from identical virtual machine instances, but VM2 is 24% slower than VM3.

3.2.6 Memory sharing vs. performance

Modern hypervisors detect and share memory pages from different virtual machines whose contents are identical [Wal02, ksm]. The ability to share identical memory reduces the memory consumed by guest VMs, increasing VM consolidation ratios. In KVM, identical page sharing in the host is done transparently in units of base pages. If the contents of a base page are duplicated in a different VM, but the duplicated base page is contained within a huge page, KVM will split the huge page into base pages to enable sharing. This policy prioritizes reducing memory footprint over preservation of huge pages, so it penalizes performance.

Another possible policy, which we call *huge page sharing*, would not split huge pages. A base page is not allowed to share pages belonging to a huge page to prevent the demotion of the huge page but it can share base pages. In contrast, a huge page is only allowed to share huge pages. We implement huge page sharing to compare with KVM and the result is shown in Table 3.7. We fit the virtual machine memory size to the working set size of each workload to avoid spurious sharing of zeroed pages. KVM saves 9.2% of memory but the workloads show a slowdown of up to 19.0% because TLB misses are increased by splitting huge pages (the percentage of huge pages in use (H/M) goes down to 66%). On the other hand, while huge page sharing preserves good performance, it provides only reduced memory consumption by 1.5%. This tradeoff between performance and memory savings is avoidable. Identical page sharing services can and should be coordinated with huge page management to obtain both performance and memory saving benefits.

3.3 Design

Ingens's goal is to enable transparent huge page support that reduces latency, latency variability and bloat while providing meaningful fairness guarantees and reasonable tradeoffs between high performance and memory savings. Ingens builds on a handful of basic primitives to achieve these goals: utilization tracking, access frequency tracking, and contiguity monitoring.

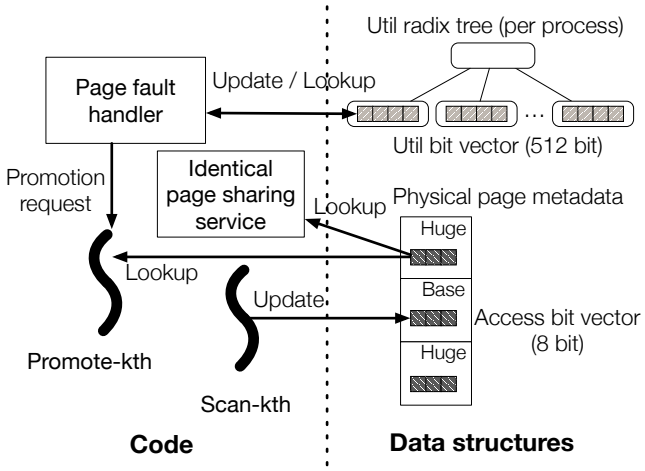


Figure 3.3: Important code and data structures in the Ingens memory manager.

While the discussion in this section is mostly expressed in terms of process behavior, Ingens techniques apply equally to processes and to virtual machines. Figure 3.3 shows the major data structures and code paths of Ingens, which we describe in this section.

3.3.1 Monitoring space and time

Ingens unifies and coordinates huge page management by introducing two efficient mechanisms to measure the utilization of huge-page sized regions (space) and how frequently huge-page sized regions are accessed (time). Ingens collects this information efficiently and then leverages it throughout the kernel to make policy decisions, using two bitvectors. We describe both.

Util bitvector. The util bitvector records which base pages are used within each huge-page sized memory region (an aligned 2 MB region containing 512 base pages). Each bit set in the util bitvector indicates that the corresponding base page is in use. The bitvector is stored in a radix tree and Ingens uses a huge-page number as the key to lookup a bitvector. The page fault handler updates the util bitvector.

Access bitvector. The access bitvector records the recent access history of a process to its pages (base or huge). Scan-kth periodically scans a process' hardware access bits in its

page table to maintain per-page (base or huge) access frequency information, stored as an 8-bit vector within Linux' page metadata. Ingens computes the exponential moving average (EMA) [ema] from the bitvector which we define as follows:

$$F_t = \alpha(\textit{weight(util bitvector)}) + (1 - \alpha)F_{t-1} \tag{3.1}$$

The *weight* is the sum of set bits in the bitvector, F_t is the access frequency value at time t, and α is a parameter. Based on a sensitivity analysis using our workloads, we set α to 0.4, meaning Ingens considers the page "frequently accessed" when $F_t \geq 3 \times bitvector\ size/4$ (i.e., 6 in our case).

We can experimentally verify the accuracy of the frequency information by checking whether pages classified as frequently accessed have their access bit set in the next scan interval: in most workloads we find the misprediction ratio to be under 3%, although random access patterns (e.g. Redis, MongoDB) can yield higher error rates depending on the dynamic request pattern.

3.3.2 Fast page faults

To keep the page fault handling path fast, Ingens decouples promotion decisions (policy) from huge page allocation (mechanism). The page fault handler decides when to promote a huge page and signals a background thread (called `Promote-kth`) to do the promotion (and allocation if necessary) asynchronously (Figure 3.3). Promote-kth compacts memory if necessary and promotes the pages identified by the page fault handler. The Ingens page fault handler never does a high-latency huge page allocation. When Promote-kth starts executing, it has a list of viable candidates for promotion; after promoting them, it resumes its scan of virtual memory to find additional candidates.

3.3.3 Utilization-based promotion (mitigate bloat)

Ingens explicitly and conservatively manages memory contiguity as a resource, allocating contiguous memory only when it decides a process (or VM) will use most of the allocated

region based on utilization. Ingens allocates only base pages in the page fault handler and tracks base page allocations in the util bitvector. If a huge page region accumulates enough allocated base pages (90% in our prototype), the page fault handler wakes up Promote-kth to promote the base pages to a huge page.

Utilization tracking lets Ingens mitigate memory bloating. Because Ingens allocates contiguous resources only for highly utilized virtual address regions, it can control internal fragmentation. The utilization threshold provides an upper bound on memory bloat. For example, if an administrator sets the threshold to 90%, processes can use only 10% more memory in the worst case compared to a system using base pages only. The administrator can simply provision 10% additional memory to avoid unexpected swapping.

Utilization-based demotion (performance). Processes can free a base page, usually by calling `free`. If a freed base page is contained within a huge page, Linux demotes the huge page instantly. For example, Redis frees objects when deleting keys which results in a system call to free the memory. Redis uses jemalloc [jem], whose `free` implementation makes an `madvise` system call with the `MADV_DONTNEED` flag to release the memory[2]. Linux demotes the huge page that contains the freed base page[3].

Demoting in-use huge pages hurts performance. Consequently, Ingens defers the demotion of high utilization huge pages. When a base page is freed within a huge page, Ingens clears the bit for the page in the util bitvector. When utilization drops below a threshold, Ingens demotes the huge page and frees the base pages whose bits are clear in the util bitvector.

3.3.4 Proactive batched compaction (reduce fragmentation)

Maintaining available free contiguous memory is important to satisfy large size allocation requests required when Ingens decides to promote a region to a huge page, or to satisfy other system-level contiguity in service of, for example, device drivers or user-level DMA.

[2]TCMalloc [tcm] also functions this way.

[3]Kernel version 4.5 introduces a new mechanism to free memory efficiently, called `MADV_FREE` but it also demotes huge pages instantly and causes the same memory bloating problem as `MADV_DONTNEED`.

To this end, Ingens monitors the fragmentation state of physical memory and proactively compacts memory to reduce the latency of large contiguous allocations.

Ingens's goal is to control memory fragmentation by keeping FMFI below a threshold (that defaults to 0.8). Proactive compaction happens in Promote-kth after performing periodic scanning. Aggressive proactive compaction causes high CPU utilization, interfering with user applications. Ingens limits the maximum amount of compacted memory to 100 MB for each compaction. Compaction moves pages, which necessitates TLB invalidations. Ingens does not move frequently accessed pages to reduce the performance impact of compaction.

3.3.5 Balance page sharing with performance

Ingens uses access frequency information to balance identical page sharing with application performance. It decides whether or not huge pages should be demoted to enable sharing of identical base pages contained within the huge page. In contrast to KVM, which always prioritizes memory savings over contiguity, Ingens implements a policy that avoids demoting frequently accessed huge pages. When encountering a matching identical base-page sized region within a huge page, Ingens denies sharing if that huge page is frequently accessed, otherwise it allows the huge page to be demoted for sharing.

For page sharing, the kernel marks a shared page read-only. When a process writes the page, the kernel stops sharing the page and allocates a new page to the process (similar to a copy-on-write mechanism). Ingens checks the utilization for the huge page region enclosing the new page and if it is highly utilized, it promotes the page (while Linux would wait for asychronous promotion).

3.3.6 Proportional promotion manages contiguity

Ingens monitors and distributes memory contiguity fairly among processes and VMs, employing techniques for proportional fair sharing of memory with an idleness penalty [Wal02]. Each process has a share priority for memory that begins at an arbitrary but standard value (e.g, 10,000). Ingens allocates huge pages in proportion to the share value. Ingens counts

infrequently accessed pages as idle memory and imposes a penalty for the idle memory. An application that has received many huge pages but is not using them actively does not get more.

We adapt ESX's adjusted shares-per-page ratio [Wal02] to express our per-process memory promotion metric mathematically as follows.

$$\mathcal{M} = \frac{S}{H \cdot (f + \tau(1 - f))} \tag{3.2}$$

where S is a process' (or virtual machine's or container's) huge page share priority and H is the number of bytes backed by huge pages allocated to the process. $(f + \tau(1 - f))$ is a penalty factor for idle huge pages. f is the fraction of idle huge pages relative to the total number of huge pages used by this process ($0 \leq f \leq 1$) and τ, with $0 < \tau \leq 1$, is a parameter to control the idleness penalty. Larger values of $\mathcal{M}$ receive higher priority for huge page promotion.

Intuitively, if two processes' S value are similar and one process has fewer huge pages (H is smaller), then the kernel prioritizes promotion (or allocation and promotion) of huge pages for that process. If S and H values are similar among a group of processes, the process with the largest fraction of idle pages has the smaller $\mathcal{M}$, and hence the lowest priority for obtaining new huge pages. $\tau = 1$ means $\mathcal{M}$ disregards idle memory while τ close to 0 means $\mathcal{M}$'s value is inversely proportional to the amount of idle memory.

A kernel thread (called `Scan-kth`) periodically profiles the idle fraction of huge pages in each process and updates the value of $\mathcal{M}$ for fair promotion.

3.3.7 Fair promotion

Promote-kth performs fair allocation of contiguity using the promotion metric. When contiguity is contended, fairness is achieved when all processes have a priority-proportional share of the available contiguity. Mathematically this is achieved by minimizing $\mathcal{O}$, defined

as follows:

$$\mathcal{O} = \sum_i (\mathcal{M}_i - \bar{\mathcal{M}})^2 \tag{3.3}$$

The $\mathcal{M}_i$ indicates the promotion metric of process/VM i and $\bar{\mathcal{M}}$ is the mean of all process' promotion metrics. Intuitively, the formula characterizes how much process' contiguity allocation ($\mathcal{M}_i$) deviates from a fair state ($\bar{\mathcal{M}}$): in a perfectly fair state, all the $\mathcal{M}_i$ equal $\bar{\mathcal{M}}$, yielding a 0-valued $\mathcal{O}$.

In practice, to optimize $\mathcal{O}$, it suffices to iteratively select the process with the biggest $\mathcal{M}_i$, scan its address space to promote huge pages, and update $\mathcal{M}_i$ and $\mathcal{O}$. Iteration stops when $\mathcal{O}$ is close to 0 or when Promote-kth cannot generate any additional huge pages (e.g., all process are completely backed by huge pages).

An important benefit of this approach is that it does not require a performance model and it applies equally well to processes and virtual machines.

3.4 Implementation

Ingens is implemented in Linux 4.3.0 and contains new mechanisms to support page utilization and access frequency tracking. It also uses Linux infrastructure for huge page page table mappings and memory compaction.

3.4.1 Huge page promotion

Promote-kth runs as a background kernel thread and schedules huge page promotions (replacing Linux's `khugepaged`). Promote-kth maintains two priority lists: `high` and `normal`. The high priority list is a global list containing promotion requests from the page fault handler and the normal priority list is a per-application list filled in as Promote-kth periodically scans the address space. The page fault handler or a periodic timer wakes Promote-kth, which then examines the two lists and promotes in priority order.

Ingens does not reserve contiguous memory in the page fault handler. When the page fault handler requests a huge page promotion, the physical memory backing the base pages might not be contiguous. In this case, Promote-kth allocates a new 2 MB contigu-

ous physical memory region, copies the data from the discontiguous physical memory, and maps the contiguous physical memory into the process' virtual address space. After promotion, Promote-kth frees the original discontiguous physical memory.

An application's virtual address space can grow, shrink, or be merged with other virtual address regions. These changes make new opportunities for huge page promotion which both Linux and Ingens detect by periodically scanning address spaces in the normal priority list (Linux in `khugepaged`, Ingens in Promote-kth). For example, a virtual address region that is smaller than the size of a huge page might merge with another region, allowing it to be part of a huge page.

Promote-kth compares the promotion metric (§3.3.6) of each application and selects the process with the highest deviation from a fair state (§3.3.7). It scans 16 MB of pages and sleeps for 10 seconds which is also Linux's default settings (i.e., the 1.6 MB/s in Table 3.4). After scanning a process' entire address space, Promote-kth records the number of promoted huge pages and if an application has too few promotions (zero in the prototype), Promote-kth excludes the application from the normal priority list for 120 seconds. This mechanism prevents an adversarial application that can monopolize Promote-kth. Such an application would have a small number of huge pages and would appear to be a good candidate to scan to increase fairness (§3.3.7)).

3.4.2 Access frequency tracking

In 2015, Linux added an access bit tracking framework [pgi] for version 4.3. The kernel adds an idle flag for each physical page and uses hardware access bits to track when a page remains unused. If the hardware sets an access bit, the kernel clears the idle bit. The framework provides APIs to query the idle flags and clear the access bit. Scan-kth uses this framework to find idle memory during a periodic scan of application memory. The default period is 2 seconds. Scan-kth clears the access bits at the beginning of the profiling period and queries the idle flag at the end.

In the x86 architecture, clearing the access bit causes a TLB invalidation for the

corresponding page. Consequently, frequent periodic scanning can have a negative performance impact. To ameliorate this problem, Ingens supports frequency-aware profiling and sampling. When Scan-kth needs to clear the access bit of a page, it checks whether the page is frequently accessed or not. If it is not frequently accessed, Scan-kth clears the access bit, otherwise it clears it with 20% probability. Ingens uses an efficient hardware-based random number generator [drn].

To verify that sampling reduces worst case overheads, we run a synthetic benchmark which reads 10 GB memory randomly without any computation, and measure the execution time for one million iterations. When Ingens resets all access bits, the execution time of the workload is degraded by 29%. Sampling-based scanning reduces the overhead to 8%. In contrast to this worst-case microbenchmark, Section 3.5 shows that slowdowns of Ingens on real workloads average 1%.

3.4.3 Limitations and future work

Linux supports transparent huge pages only for anonymous memory because huge page support for page cache pages can significantly increase I/O traffic, potentially offsetting the benefits of huge pages. If Linux adds huge pages to the page cache, it will make sense to extend Ingens to manage them with the goal of improving the read-only page cache support (implemented in FreeBSD [NIDC02]), while avoiding significant increases in I/O traffic for write-back of huge pages which are sparsely modified.

Hardware support for finer-grain tracking of access and dirty bits for huge pages would benefit Ingens. Hardware-managed access and dirty bits for all base pages within a huge page region could avoid wasted I/O on write-back of dirty pages, and enable much better informed decisions about when to demote a huge page or when huge pages can be reclaimed fairly under memory pressure.

NUMA considerations. Ingens maintains Linux's NUMA heuristics, preferring pages from a node's local NUMA region, and refusing to allocate a huge page from a different NUMA domain. All of our measurements are within a single NUMA region.

Previous work has shown that if memory is shared across NUMA nodes, huge pages may contribute to memory request imbalance across different memory controllers and reduced locality of accesses, decreasing their performance benefit [GLD+14]. This happens due to page-level false sharing, where unrelated data is accessed on the same page, and the hot page effect, which is exacerbated by the large page size. The authors propose extensions to Linux' huge page allocation mechanism to balance huge pages among NUMA domains and to split huge pages if false sharing is detected or if they become too hot. These extensions integrate nicely with Ingens. Scan-kth can already measure page access frequencies and Promote-kth can check whether huge pages need to be demoted.

3.5 Evaluation

We evaluate Ingens using the applications in Table 3.1, comparing against the performance of Linux's huge page support which is state-of-the-art. Experiments are performed on two Intel Xeon E5-2640 v3 2.60GHz CPUs (Haswell) with 64 GB memory and two 256 MB SSDs. We use Linux 4.3 and Ubuntu 14.04 for both the guest and host system. Intel supports multiple hardware page sizes of 4 KB, 2 MB and 1 GB; our experiments use only 4 KB and 2 MB huge pages. We set the number of vCPUs equal to the number of application threads.

We characterize the overheads of Ingens's basic mechanisms such as access tracking and utilization-based huge page promotion. We evaluate the performance of utilization-based promotion and demotion and Ingens ability to provide fairness across applications using huge pages. Finally, we show that Ingens's access frequency-based same page merging achieves good memory savings while preserving most of the performance benefit of huge pages. We use a single configuration to evaluate Ingens which is consistent with our examples in Sections 3.3 and 3.4: utilization threshold is 90%, Scan-kth period is 10s, access frequency tracking interval is 2 sec, and sampling ratio is 20%. Proactive batched compaction happens when FMFI is below 0.8, with an interval of 5 seconds; the maximum amount of compacted memory is 100MB; and a page is frequently accessed if $F_t \geq 6$.

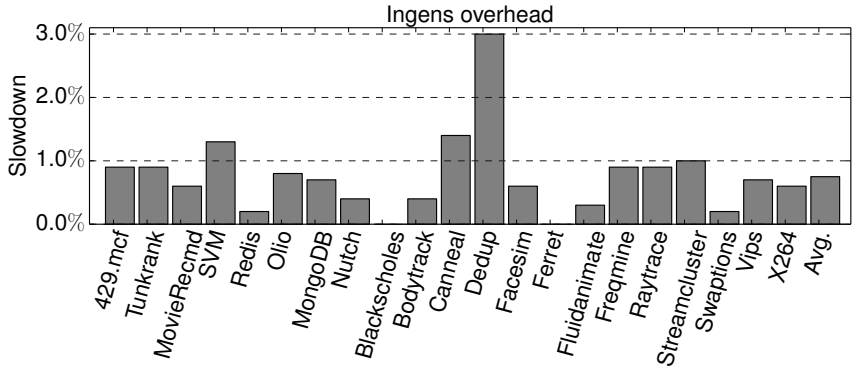


Figure 3.4: Performance slowdown of utilization-based promotion relative to Linux when memory is not fragmented.

Background task	CPU utilization
Proactive compaction	1.3%
Access bit tracking	11.4%

Table 3.8: CPU utilization of background tasks in Ingens. For access bit tracking, Scan-kth scans memory of MongoDB that uses 10.7GB memory.

3.5.1 Ingens overhead

Figure 3.4 shows the overheads introduced by Ingens for memory intensive workloads. To evaluate the performance of utilization-based huge page promotion in the unfragmented case, we run a number of benchmarks and compare their run time with Linux. Ingens's utilization-based huge page promotion slows applications down 3.0% in the worst case and 0.7% on average. The slowdowns stem primarily from Ingens not promoting huge pages as aggressively as Linux, so the workload executes with slower base pages for a short time until Ingens promotes huge pages. A secondary overhead stems from the computation of huge page utilization.

To verify that Ingens does not interfere with the performance of "normal" workloads, we measure an average performance penalty of 0.8% across the entire PARSEC 3.0 benchmark suite.

Table 3.8 shows the CPU utilization of background tasks in Ingens. We measure the

CPU utilization across 1 second intervals and take the average. For proactive compaction, we set Ingens to compact 100 MB of memory every 2 seconds (which is more aggressive than the default of 5 seconds). CPU overhead of access bit tracking depends on how many pages are scanned, so we measure the CPU utilization of Scan-kth while running MongoDB using 10.7 GB of memory.

3.5.2 Utilization-based promotion

To evaluate Ingens's utilization-based huge page promotion, we compare a mix of operations from the Cloudstone WEB 2.0 benchmark, which simulates a social event website. Cloudstone models a LAMP stack, consisting of a web server (nginx), PHP, and MySQL. We run Cloudstone in a KVM virtual machine and use the Rain workload generator [BLY+10] for load.

A study of the top million websites showed that in 2015 the average size exceeded 2 MB [Eve]. In light of this, we modify Cloudstone to serve some web pages that use about 2 MB of memory, enabling the benchmark to make better use of huge pages. The Cloudstone benchmark consists of 7 web pages, and we only modify the homepage and a page that displays social event details to use 2 MB memory. The other pages remain unchanged.

We compare throughput and latency for Cloudstone on Linux and Ingens when memory is fragmented from prior activity (FMFI = 0.9). To cause fragmentation, we run a program that allocates a large region of memory and then partially frees it.

We use Cloudstone's default operation mix: 85% read (viewing events, visiting homepage, and searching event by tag), 10% login, and 5% write (adding new events and inviting people). Our test database has 7,000 events, 2,000 people, and 900 tags. Table 3.9 (a) shows the throughput attained by the benchmark running on Linux and Ingens. Ingens's utilization-based promotion achieves a speedup of 1.18× over Linux. Table 3.9 (b) shows average and tail latency of the read operations in the benchmark. Ingens reduces an average latency up to 29.2% over Linux. In the tail, the reduction improves further, up to

Linux	Ingens
922.3	1091.9 (1.18×)

(a) Throughput of full operation mix (requests/sec and speedup normalized to Linux).

	Event view		Homepage visit		Tag search	
	Linux	Ingens	Linux	Ingens	Linux	Ingens
Average	478	338	236	207	289	240
90th	605	354	372	226	417	299
MAX	694	649	379	385	518	507

(b) Latency (millisecond) of read-dominant operations.

Table 3.9: Performance result of Cloudstone WEB 2.0 Benchmark (Olio) when memory is fragmented.

41.4% at the 90th percentile.

Performance for Ingens improves because it reduces the average page-fault latency by not compacting memory synchronously in the page fault handler. We measure 461,383 page compactions throughout the run time of the benchmark in Linux when memory is fragmented.

When memory is not fragmented, Ingens reduces throughput by 13.4% and increases latency up to 18.1% compared with Linux. The benchmark contains many short-lived requests and Linux's greedy huge page allocation pays off by drastically reducing the total number of page faults. Ingens is less aggressive about huge page allocation to avoid memory bloat, so it incurs many more page faults.

Ingens copes with this performance problem with an adaptive policy. When memory fragmentation is below 0.5 Ingens mimics Linux's aggressive huge page allocation. This policy restores Ingens's performance to Linux's levels. However, while bloat (§3.2.2) is not a problem for this workload, the adaptive policy increases risk of bloat in the general case. Like any management problem, it might not be possible to find a single policy that has every desirable property for a given workload. We verified that this policy performs similarly to the default policy used in Table 3.4, but it is most appropriate for workloads with many short-lived processes.

Linux-nohuge	Linux	Ingens-90%	Ingens-70%	Ingens-50%
12.2 GB	20.7 GB	12.3 GB	12.9 GB	17.8 GB

(a) Redis memory consumption in different configurations. The percentage in the label is a utilization threshold.

	Throughput	90th lat.	99th lat.	99.9th lat.
Linux-nohuge	19.0K	4	5	109
Linux	21.7K	3	4	8
Ingens-90%	20.9K	3	4	64
Ingens-70%	21.1K	3	4	55
Ingens-50%	21.6K	3	4	23

(b) Redis GET Performance: Throughput (operations/sec) and latency (millisecond).

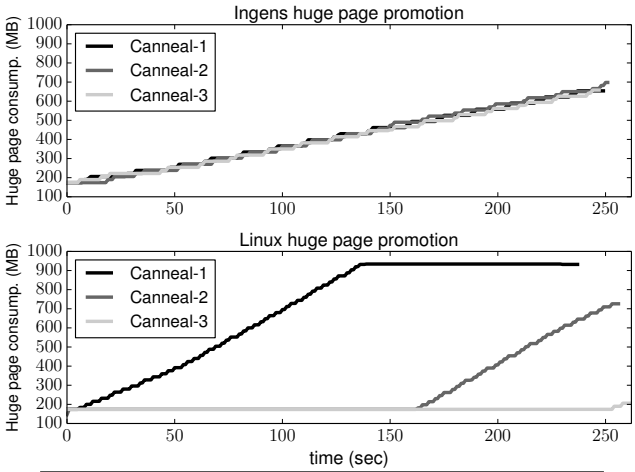
Table 3.10: Redis memory use and performance.

3.5.3 Memory bloating evalution

To evaluate Ingens's ability to minimize memory bloating without impacting performance, we evaluate the memory use and throughput of a benchmark using the Redis key-value store. Redis is known to be susceptible to memory bloat, as its memory allocations are often sparse. To create a sparse address space in our benchmark, we first populate Redis with 2 million keys, each with 8 KB objects and then delete 70% of the key space using a random pattern. We then measure the GET performance using the benchmark tool shipped with Redis. For Ingens, we evaluate different utilization thresholds for huge page promotion.

Table 3.10 shows that memory use for the 90% and 70% utilization-based configurations is very close to the case where only base pages are used. Only at 50% utilization does Ingens approach the memory use of Linux's aggressive huge page promotion.

The throughput and latency of the utilization-based approach is very close to using only huge pages. Only in the 99.9th percentile does Ingens deviate from Linux using huge pages only, while still delivering much better tail latency than Linux using base pages only.



	Canneal-1	Canneal-2	Canneal-3
Linux	181	192	195
Ingens	186	186	187

Figure 3.5: Huge page consumption (MB) and execution time (second). 3 instances of canneal (Parsec 3.0 benchmark) run concurrently and Promote-kth promotes huge pages. Execution time in the table excludes data loading time.

3.5.4 Fair huge page promotion

Ingens guarantees a fair distribution of huge pages. If applications have the same share priority (§3.3.6), Ingens provides the same amount of huge pages. To evaluate fairness, we run a set of three identical applications concurrently with the same share priority and idleness parameter, and measure the amount of huge pages each one holds at any point in time.

Figure 3.5 shows that Linux does not allocate huge pages fairly, it simply allocates huge pages to the first application that can use them (Canneal-1). In fact, Linux asynchronously promotes huge pages by scanning linearly through each application's address space, only considering the next application when it is finished with the current application. Time 160 is when Linux has promoted almost all of Canneal-1's address space to huge pages so only then does it begin to allocate huge pages to Canneal-2.

In contrast, Ingens promotes huge pages based on the fairness objective described

in Section 3.3.7 and thus equally distributes the available huge pages to each application. Fair distribution of huge pages translates to fair end-to-end execution time as well. All applications finish at the same time in Ingens, while Canneal-1 finishes well before 2 and 3 on Linux.

3.5.5 Trade off of memory saving and performance

Finally, we evaluate the memory and performance tradeoffs of identical page sharing. We run a workload mix of three different applications, each in its own virtual machine. We measure their memory use and performance slowdown under three different OS configurations: (1) KVM with aggressive page sharing, where huge pages are demoted if underlying base pages can be shared. (2) KVM where only pages of the same type may be shared and huge pages are never broken up (huge page sharing). (3) Ingens, where only infrequently used huge pages are demoted for page sharing. To avoid unused memory saving, we intentionally fit guest physical memory size to memory usages of the workloads.

Table 3.11 shows that KVM's aggressive page sharing saves the most memory (9.6%), but also cedes the most performance (between 6.5% and 20.2% slowdown) when compared to huge page sharing. When sharing only pages of the same type, it saves memory only 2.1%. Finally, Ingens allows us to save 6.8% of memory, while only slowing down the application up to 2.5%. The main reason for the low performance degradation is that the ratio of huge pages to total pages remains high in Ingens, due to its access frequency-based approach to huge page demotion and instant promotion when Ingens stops page sharing.

3.6 Summary

Physical memory sizes are exploding and popular modern virtualized and big data workloads such as machine learning, data analytics, and graph algorithms easily consume all available memory. For these workloads to be efficient, hardware must map physical memory in performance-critical on-chip structures called translation lookaside buffers (TLBs). Memory not mapped by the TLB forces the hardware to perform high-latency page table

Policy	Mem saving	Performance slowdown	H/M
KVM (Linux)	1438 MB (9.6%)	Tunkrank: 274 (12.7%) MovieRecmd: 210 (6.5%) SVM: 232 (20.2%)	Tunkrank: 66% MovieRecmd: 10% SVM: 72%
Huge page sharing	317 MB (2.1%)	Tunkrank: 243 MovieRecmd: 197 SVM: 193	Tunkrank: 99% MovieRecmd: 99% SVM: 99%
Ingens	1026 MB (6.8%)	Tunkrank: 247 (1.6%) MovieRecmd: 200 (1.5%) SVM: 198 (2.5%)	Tunkrank: 90% MovieRecmd: 79% SVM: 94%

Table 3.11: Memory saving (MB) and performance (second) trade off. H/M - huge page ratio out of total memory used. Parentheses in the Mem saving column expresses the memory saved as a percentage of the total memory (15 GB) allocated to all three virtual machines.

walks; hardware support for memory virtualization (e.g., Intel's extended page tables) exacerbate the problem by amplifying these latencies. Consequently, many workloads pay a high performance price for address translation.

For the TLB to map more physical memory, modern TLBs support page sizes larger than the historical standard size of 4 KB. We reached an inflection point in late 2013 when hardware increased its support for large pages, exposing flaws in the operating system's support. Modern OSes do not support large page sizes as well as they should. With hardware's weak support for large pages, OSes have developed disparate and poorly coordinated mechanisms for managing large pages including policies that harm performance. Large page support is challenging because to map a large page, the OS needs a large region of physically contiguous, aligned physical memory, which can be difficult to find or create as memory becomes fragmented over time. The proposed work brings a coherent set of metrics and algorithms to enable efficient OS management of large pages, providing good performance for memory-intensive workloads while preserving fairness.

By recasting most memory management policy into relatively simple scalar metrics e.g., a priority value, the OS can coordinate its currently disparate mechanisms and avoid performance pathologies. A key challenge is to coordinate global and local memory management policies. Global policies manage system-wide resources like the contiguity of free memory, while local policies manage per-process resources, like how many large pages a

process already uses and how frequently it has accessed the pages it has allocated. By coordinating global and local memory managment policies, the operating system and hypervisor can provide high performance without undue implementation complexity.

Chapter 4

Related Work

Logging and coherence in file systems. WAFL [HLM94] and ZFS [Wat09] both have the capability to use logging to a low-latency medium (non-volatile RAM in WAFL's case) to reduce the latency of file system writes. Similarly, Frangipani [TML97] uses logging to support synchronous persistence of file metadata, but not data, in a distributed file system. xFS [ADN+95] uses an invalidation-based write-back cache coherence protocol to minimize communication overhead among applications and a centralized network server. Strata extends these ideas to provide low-latency kernel-bypass for applications on the local machine, synchronous persistence for data and metadata, and a lease mechanism to provide coherence among applications managing data at user-level. McoreFS [BEC+17] uses per-core logs and operation commutativity properties to improve multicore file system scalability. Strata can leverage these same techniques to improve scalability if needed.

Multi-layer block stores. Various efforts have studied the use of caching among different storage technologies. Strata leverages similar ideas, in the context of a read-write file system. Operating with a file system API allows us to support, and requires us to handle, a broader class of application access patterns. For example, RIPQ [THL+15] is a novel caching layer that minimizes write amplification when using local SSD as a read-only cache for remote photo storage. FlashStore [DSL10] is a key-value store designed to use SSD as a fast cache between DRAM and HDD, similarly minimizing the number of reads/writes

done to SSD. Nitro [LSD+14] is an SSD caching system that uses data deduplication and compression to increase capacity. Dropbox built a general-purpose file system that uses Amazon S3 for data blocks, but keeps metadata in SSD/DRAM [Met16]; technical details on its operation are not public. RAMcloud [OGG+15] uses disk as a back up for data in replicated DRAM. It applies log structure to both DRAM and disk [RKO14], achieving higher DRAM utilization.

NVM/SSD optimized block storage/file systems. Much recent work proposes specialized storage solutions for emerging non-volatile memory technologies. BPFS [CNF+09] is a file system for non-volatile memory that uses an optimized shadow-paging technique for crash consistency. PMFS [DKK+14] explores how to best exploit existing memory hardware to make efficient use of persistent byte-addressable memory. EXT4-DAX [ext14] extends the Linux EXT4 file system to allow direct mapping of NVM, bypassing the buffer cache. Aerie [VNP+14] is an NVM file system that also provides direct access for file data IO, using a user-level lease for NVM updates. Unlike Strata, none of these file systems provide synchronous persistence semantics, as they require system calls for metadata operations. Only NOVA [XS16] goes one step further and uses a novel per-inode log-structured file system to provide synchronous file system semantics on NVM, but requires system calls for every operation. F2FS [LSHC15] is a SSD-optimized log-structured file system that sorts data to reduce file system write amplification; lacking NVM, it does not provide efficient synchronous semantics. Decibel [NWW17] is a block-level virtualization layer that isolates tenants accessing shared SSDs by observing and controlling their device request queues. Strata generalizes these ideas to provide direct and performance-isolated access to NVM for both meta-data and data IO using a per-application update log, along with providing efficient support for much larger SSD and HDD storage regions. Strata also coalesces logs to minimize write amplification, which is new compared to these existing systems.

Managed storage designs. All storage hardware technologies require a certain level of software management to achieve good performance. Classic examples include elevator

scheduling [ele17] and log-structured file systems [spr17]. Modern examples include log-structured merge trees [OCGO96] (LSM-trees) and B^ϵ-trees, used by various storage systems [MHK09, RG13, YZJ+16]. All of these systems rely on a particular layout of the stored data to optimize read or write performance or (in the case of LSM-trees) both. Unlike all of these systems, Strata specializes its data representation to different storage layers, changing the correctness and performance properties on a per-device basis.

Strong consistency. A number of approaches propose to redesign the file system interface to provide stronger consistency guarantees for slow devices. Rethink the sync [NVCF06] proposes the concept of *external synchrony*, whereby all file system operations are internally (to the application) asynchronous. The OS tracks when file system operations become externally visible (to the user) and synchronizes operations at this point, allowing it to batch them. Optimistic crash consistency [CPADAD13] introduces a new API to separate ordering of file system operations from their persistence, enabling file system crash consistency with asynchronous operations. Strata instead leverages fast persistence in NVM to provide ordered and atomic operations.

Virtual memory is an active research area. Our evidence of performance degradation from address translation overheads is well-corroborated [BGC+13, GBHS14, BLM11, MT16].

Operating system support for huge pages. Navarro et al. [NIDC02] implement OS support for multiple page sizes with contiguity-awareness and fragmentation reduction as primary concerns. They propose reservation-based allocation, allocating contiguous ranges of pages in advance, and deferring promotion. Many of their ideas are widely used [thpb], and it forms the basis of FreeBSD's huge page support. Ingens's utilization-based promotion uses a util bitvector that is similar to the population map [NIDC02]. In contrast to that work, Ingens does not use reservation-based allocation, decouples huge page allocation from promotion decisions, and redistributes contiguity fairly when it becomes available (e.g., after process termination). Ingens has higher performance because it promotes more huge pages;

it does not require promoted pages to be read-only or completely modified (§3.2.4). Features in modern systems such as memory compaction and same-page merging [ksm] pose new challenges not addressed by this previous work.

Gorman et al. [GH08] propose a placement policy for an OS's physical page allocator that mitigates fragmentation and promotes contiguity by grouping pages according to relocatability. Subsequent work [GH10] proposes a software-exposed interface for applications to explicitly request huge pages like `libhugetlbfs` [libb]. The foci of Ingens, including trade-offs between memory sharing and performance, and unfair allocation of huge pages are unaddressed by previous work.

Hardware support for huge pages. TLB miss overheads can be reduced by accelerating page table walks [BCR10, Bha13] or reducing their frequency [GHS16]; by reducing the number of TLB misses (e.g. through prefetching [BM09, KS02, SDS00], prediction [PTSM15], or structural change to the TLB [TH94, PVJB12, PBEL14] or TLB hierarchy [BLM11, LBM13, SK10, AJH15, AJH12, KGA+15, BGC+13, GBHS14]). Multipage mapping techniques [TH94, PVJB12, PBEL14] map multiple pages with a single TLB entry, improving TLB reach by a small factor (e.g. to 8 or 16); much greater improvements to TLB reach are needed to deal with modern memory sizes. Direct segments [BGC+13, GBHS14] extend standard paging with a large segment to map the majority of an address space to a contiguous physical memory region, but require application modifications and are limited to workloads able to a single large segment. Redundant memory mappings (RMM) [KGA+15] extend TLB reach by mapping *ranges* of virtually and physically contiguous pages in a range TLB. The level of additional architectural support is significant, while Ingens works on current hardware.

A number of related works propose hardware support to recover and expose contiguity. GLUE [PVLB15] groups contiguous, aligned small page translations under a single speculative huge page translation in the TLB. Speculative translations, (similar to SpecTLB [BCR11]) can be verified by off-critical-path page-table walks, reducing effective page-table walk

latency. GTSM [DZC+15] provides hardware support to leverage contiguity of physical memory extents even when pages have been retired due to bit errors. Were such features to become available, hardware mechanisms for preserving contiguity could reduce overheads induced by proactive compaction in Ingens.

Architectural assists are ultimately complementary to our own work. Hardware support can help, but higher-level coordination of hardware mechanisms by software is a fundamental necessity. Additionally, as none of these assists are likely to be realized in imminently available hardware, using techniques such as those we propose in Ingens are a *de facto* necessity.

Chapter 5

Conclusion

Emerging non-volatile memory has both storage and memory characteristics. As a storage perspective, the market for storage devices has fragmented based on a tradeoff between performance and cost-per-capacity. Applications want the fastest NVM for performance along with low-cost traditional storage devices such as SSD and HDD to store the large volume of data (diversity challenge). As a memory perspective, current TLB reach is insufficient to cover the large address space of DRAM and NVM, causing high address translation costs (large capacity challenge).

This thesis makes the following contributions to address the diversity and large capacity challenges:

Strata is an integrated file system across different storage media. To leverage different hardware properties, Strata splits responsibilities of a file system into two pieces: LibFS and KernelFS. LibFS is linked to applications and provides direct accesses to NVM with a private log. KernelFS runs in the background and manages data across different storage layers. Using low latency NVM, Strata provides synchronous IO, simplifying the POSIX crash consistency programming model.

Ingens is an efficient huge page management system. Ingens rethinks the design flaws and spot fixes of Linux and makes huge page management principled and coordinated way. Ingens proposes the asynchronous and utilization-based huge page allocation, avoiding ex-

tra page fault latency and memory bloating problems. Also, Ingens monitors hot and cold pages and uses the information to allocation huge pages fairly.

Bibliography

[ADN+95] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 109–126, New York, NY, USA, 1995. ACM.

[AJH12] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *ISCA*, 2012.

[AJH15] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Fast two-level address translation for virtualized systems. In *IEEE Transactions on Computers*, 2015.

[AMD10] AMD. *AMD-V Nested Paging*, 2010. `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`.

[AMM+11] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Middleware Industry Track Workshop*, 2011.

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international*

conference on Measurement and Modeling of Computer Systems, pages 53–64, London, England, UK, 2012.

[BCR10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *ISCA*, 2010.

[BCR11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *ISCA*, 2011.

[BEC+17] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[BGC+13] Arkapravu Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, 2013.

[Bha13] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *MICRO*, 2013.

[BLM11] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *HPCA*, 2011.

[BLY+10] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David Patterson. Rain: A workload generation toolkit for cloud computing applications. In *U.C. Berkeley Technical Publications (UCB/EECS-2010-14)*, 2010.

[BM09] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[cgr15] Linux control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, October 2015.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2), June 1994.

[cloa] Apache Cloudstack. https://en.wikipedia.org/wiki/Apache_CloudStack. [Accessed April, 2016].

[clob] Cloudera recommends turning off memory compaction due to high CPU utilization. http://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html. [Accessed April, 2016].

[CNF+09] 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, pages 133–146, New York, NY, USA, 2009. ACM.

[cou] CouchBase recommends disabling huge pages. http://blog.couchbase.com/often-overlooked-linux-os-tweaks. [March, 2014].

[CPADAD13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[DG11] J. Dean and S. Ghemawat. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, 2011.

[DKK+14] 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.

[dok] DokuDB recommends disabling huge pages. https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/. [July, 2014].

[dpd] Data Plane Development Kit. http://www.dpdk.org/. [Accessed April-2016].

[drn] Intel hardware random number generator. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementatio [May, 2014].

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[DZC+15] Yu Du, Miao Zhou, B.R. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *HPCA*, 2015.

[ele17] Elevator algorithm. https://en.wikipedia.org/wiki/Elevator_algorithm, August 2017.

[ema] Exponential moving average. https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average. [Accessed April, 2016].

[Eve] Tammy Everts. The average web page is more than 2 MB size. https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/. [June, 2015].

[ext14] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, September 2014.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[GBHS14] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization. In *MICRO*, 2014.

[GH08] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, 2008.

[GH10] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2010.

[GHS16] Jayneel Gandhi, , Mark D. Hill, and Michael M. Swift. Exceeding the best of nested and shadow paging. In *ISCA*, 2016.

[GLD+14] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[GW05] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, 2005.

[hada] Apache Hadoop. `http://hadoop.apache.org/`. [Accessed April, 2016].

[hadb] High CPU utilization in Hadoop due to transparent huge pages. `https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad/`. [February, 2015].

[HDV+11] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Techni-*

cal Conference on USENIX Winter 1994 Technical Conference, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[HN15] T. Haynes and D. Noveck. Network file system (nfs) version 4 protocol, March 2015. https://tools.ietf.org/html/rfc7530.

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[hug] Application-friendly kernel interfaces. https://lwn.net/Articles/227818/. [March, 2007].

[ibma] IBM cloud with KVM hypervisor. http://www.networkworld.com/article/2230172/opensource-subnet/red-hat-s-kvm-virtualization-proves-itself-in-ibm-s-cloud.html. [March, 2010].

[ibmb] IBM recommends turning off huge pages due to high CPU utilization. http://www-01.ibm.com/support/docview.wss?uid=swg21677458. [July, 2014].

[inta] http://www.7-cpu.com/cpu/Skylake.html. [Accessed April, 2016].

[intb] http://www.7-cpu.com/cpu/Haswell.html. [Accessed April, 2016].

[int13] Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products, 2013. https://newsroom.intel.com/press-kits/

introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-and-memory-products/.

[Int16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*, 2016. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[Int17] Intel Corporation. Storage performance development kit, August 2017. http://www.spdk.io.

[JBLF10] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[jem] Jemalloc. http://www.canonware.com/jemalloc/. [Accessed April-2016].

[KFH+17] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[KGA+15] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrin Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman nsal. Redundant memory mappings for fast access to large memories. In *ISCA*, 2015.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: The linux virtual machine monitor. In *Linux Symposium*, 2007.

[KS02] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *ISCA*, 2002.

[ksm] Kernel Same-page Merging. https://en.wikipedia.org/wiki/Kernel_same-page_merging. [Accessed April, 2016].

[KYP+16] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *OSDI*, 2016.

[LBM13] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[Liba] Liblinear. https://www.csie.ntu.edu.tw/~cjlin/liblinear/. [Accessed April, 2016].

[libb] Huge Pages Part 2 (Interfaces). https://lwn.net/Articles/375096/. [February, 2010].

[LL14] Ching-Pei Lee and Chih-Jen Lin. Large-scale linear RankSVM. *Neural Comput.*, 26(4):781–817, April 2014.

[LSD+14] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 501–512, Berkeley, CA, USA, 2014. USENIX Association.

[LSHC15] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[lvm17] lvmcache – lvm caching. http://man7.org/linux/man-pages/man7/lvmcache.7.html, August 2017.

[MCB+07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, Ottawa, ON, Canada, June 2007.

[Met16] Cade Metz. The epic story of Dropbox's exodus from the Amazon cloud empire, March 2016. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[MHK09] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with anvil. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 147–160, New York, NY, USA, 2009. ACM.

[mona] MongoDB. https://www.mongodb.com/. [Accessed April, 2016].

[Monb] MongoDB recommends disabling huge pages. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/. [Accessed April, 2016].

[mov] Movie recommendation with Spark. http://ampcamp.berkeley.edu/big-data-mini-course/movie-recommendation-with-mllib.html. [Accessed April, 2016].

[MT16] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM.

[mys] High CPU utilization in Mysql due to transparent huge pages. http://developer.okta.com/blog/2015/05/22/tcmalloc. [May, 2015].

[NIDC02] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *OSDI*, 2002.

[nuo] NuoDB recommends disabling huge pages. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb. [May, 2014].

[nut] Intel HiBench. https://github.com/intel-hadoop/HiBench/tree/master/workloads. [Accessed April, 2016].

[NVCF06] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[nvd17] Micron, Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility, 2017. https://www.micron.com/products/dram-modules/nvdimm/.

[nvm17] NVM Express 1.2.1. http://www.nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf, August 2017.

[NWW17] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 17–33, Boston, MA, 2017. USENIX Association.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). In *Acta Informatica*, 1996.

[OGG+15] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[ope] OpenStack. `https://openvirtualizationalliance.org/what-kvm/openstack`. [Accessed April-2016].

[opt17a] Intel Optane memory, August 2017. `http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html`.

[opt17b] Product brief: Intel Optane SSD DC P4800X series, August 2017. `http://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html`.

[osx] Huge page support in Mac OS X. `https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man2/mmap.2.html`. [Accessed April-2016].

[par] PARSEC 3.0 benchmark suite. `http://parsec.cs.princeton.edu/`. [Accessed April, 2016].

[PBEL14] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *HPCA*, 2014.

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, Berkeley, CA, USA, 2014. USENIX Association.

[pgi] Idle Page Tracking. `http://lxr.free-electrons.com/source/Documentation/vm/idle_page_tracking.txt`. [November, 2015].

[PLZ+14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[pme17] Persistent memory programming, August 2017. `http://pmem.io/`.

[PTSM15] M.-M. Papadopoulou, Xin Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly TLB designs. In *HPCA*, 2015.

[PVJB12] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *MICRO*, 2012.

[PVLB15] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized systems: Can you have it both ways? In *MICRO*, 2015.

[reda] Redis recommends disabling huge pages. `http://redis.io/topics/latency`. [Accessed April, 2016].

[redb] Redis SSD swap discussion. http://antirez.com/news/52. [March, 2013].

[red17] Redis. http://redis.io, August 2017.

[RG13] Kai Ren and Garth Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 145–156, Berkeley, CA, USA, 2013. USENIX Association.

[RKO14] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[rso17] rsockets library, August 2017. https://github.com/ofiwg/librdmacm.

[Rud16] Andy M Rudoff. Deprecating the PCOMMIT instruction, September 2016. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[s317] Amazon S3, August 2017. https://aws.amazon.com/s3/.

[sap] SAP IQ recommends disabling huge pages. http://scn.sap.com/people/markmumy/blog/2014/05/22/sap-iq-and-linux-hugepagestransparent-hugepages. [May, 2014].

[SDS00] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *ISCA*, 2000.

[Sha96] Tom Shanley. *Pentium Pro Processor System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[SK10] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *MICRO*, 2010.

[smr17] Shingled magnetic recording, 2017. `https://en.wikipedia.org/wiki/Shingled_magnetic_recording`.

[spa] Apache Spark. `http://spark.apache.org/docs/latest/index.html`. [Accessed April, 2016].

[spe] SPEC CPU 2006. `https://www.spec.org/cpu2006/`. [Accessed April, 2016].

[spe17] Specsfs2014, August 2017. `https://www.spec.org/sfs2014/`.

[spl] Splunk recommends disabling huge pages. `http://docs.splunk.com/Documentation/Splunk/6.1.3/ReleaseNotes/SplunkandTHP`. [December, 2013].

[spr17] The Sprite Operating System. `https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html`, August 2017.

[sql17] SQLite. `https://sqlite.org`, August 2017.

[SSS+08] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[SW98] Richard L. Sites and Richard T. Witek. *ALPHA architecture reference manual*. Digital Press, Boston, Oxford, Melbourne, 1998.

[tcm] Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. [Accessed April-2016].

[TH94] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *ASPLOS*, 1994.

[THL+15] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[thpa] Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/. [January, 2011].

[thpb] Transparent Hugepages. https://lwn.net/Articles/359158/. [October, 2009].

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 224–237, New York, NY, USA, 1997. ACM.

[tun] Cloudsuite. http://parsa.epfl.ch/cloudsuite/graph.html. [Accessed April, 2016].

[TZR+15] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, 2015. ACM.

[TZS16] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*, 41(1), 2016.

[VNP+14] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[vol] VoltDB recommends disabling huge pages. `https://docs.voltdb.com/AdminGuide/adminmemmgt.php`. [Accessed April, 2016].

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[Wat09] Scott Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall, 2009.

[win] Large-page support in Windows. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx`. [Accessed April-2016].

[XS16] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[YZJ+16] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 1–14, Santa Clara, CA, 2016. USENIX Association.

[zoo17] Apache ZooKeeper. `https://zookeeper.apache.org`, August 2017.

[ZS15] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015.

[zsm17] Bandwidth: a memory bandwidth benchmark, August 2017. `http://zsmith.co/bandwidth.html`.