Scale and Performance in a Filesystem Semi-Microkernel

Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye,
Sudarsun Kannan*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison *Rutgers University**

Abstract

We present uFS, a user-level filesystem semi-microkernel. uFS takes advantage of a high-performance storage development kit to realize a fully-functional, crash-consistent, highly-scalable filesystem, with relative developer ease. uFS delivers scalable high performance with a number of novel techniques: careful partitioning of in-memory and on-disk data structures to enable concurrent access without locking, inode migration for balancing load across filesystem threads, and a dynamic scaling algorithm for determining the number of filesystem threads to serve the current workload. Through measurements, we show that uFS has good base performance and excellent scalability; for example, uFS delivers nearly twice the throughput of ext4 for LevelDB on YCSB workloads.

CCS Concepts: • **Software and its engineering → File systems management; Operating systems.**

Keywords: Filesystem, Microkernel, Direct Access

1 Introduction

Recent work in high-performance networking has ushered in a renaissance of microkernel-based approaches [31, 38, 45, 52]. Made possible by the large number of CPU resources available in modern, multicore machines, these systems deliver high performance to end applications via a user-level multi-threaded networking service, thus hoisting most networking functionality out of the kernel, while leaving other OS functionality in the main monolithic OS (i.e., Linux). We call this kernel architecture a *semi-microkernel*.

The advantages of the semi-microkernel approach are manifold, including much faster code velocity for the hoisted subsystem (kernel code being challenging to develop quickly and correctly) and better vertical integration with end applications (kernel code being hard to tailor to specific use cases) [38, 41]. Importantly, these ends are achieved without sacrificing security and other important kernel properties.

SOSP'21, October 26-29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10...$15.00.
http://dx.doi.org/10.1145/3477132.3483581

In this paper, we explore the semi-microkernel approach but for a different and important OS subsystem: the local filesystem. The time is ripe for such an investigation for the following three reasons. First, high performance storage devices, based on non-volatile memory technologies, stress existing kernel protection mechanisms; there is a notable and high overhead to trapping in and out of the kernel [9, 54]. Second, the same code velocity issues that exist in networking stacks exist in kernel filesystems; the development of new features is slow [37]. Finally, the advent of new storage development kits – essentially custom libraries that give direct and powerful device controls to applications – are readily utilized at user-level and can deliver high performance without kernel involvement.

To assess the utility of the filesystem semi-microkernel (or "filesystem as a process" [35]), we design, implement, and evaluate uFS, a crash-consistent user-level filesystem. The uFS system consists of two main components: the uFS server and the uFS library. The uFS server is implemented as a multi-threaded process built atop the Storage Performance Development Kit (SPDK) [55]; applications link with the uFS library to communicate with it and request file service via high-performance interprocess communication channels. The rest of the operating system remains as is, acting as an intermediary on rare events (e.g., process startup) to provide authentication, but (importantly) does not participate in filesystem requests.

Our main focus within this paper is on the absolute performance and scale of uFS. Can such a system deliver high performance under varying application demands, while enabling the benefits of user-level development and deployment? What are the key mechanisms and policies necessary to extract peak performance from underlying high-performance devices?

uFS achieves its performance goals through the following design. The basic architecture of the uFS server contains a single primary and a variable number of worker threads, much like the original Google File System [21]. For simplicity, the primary handles the metadata workload (e.g., file creations and deletions), whereas workers handle the mainline data path (e.g., reads and writes to files). uFS adopts a "shared nothing" data parallel architecture [56] across workers to improve cache locality and increase scalability; important data structures are designed so as not

to require synchronization across workers. The key unit of allocation across workers is the inode; at any moment, a file inode is owned by a single worker which serves all reads and writes to that file. To provide scalable crash consistency, uFS again carefully partitions data structures. Finally, uFS reassigns inodes across workers to balance load and dynamically scales the number of workers to demand, thus utilizing only as many CPU cores as needed.

The uFS library is also carefully designed for performance, including lease-based [23] caching of data and file descriptors to reduce client-server communication. The library also includes various other optimizations, mostly designed to reduce copying while maintaining security.

We evaluate uFS with a series of microbenchmarks and real application workloads. We compare to Linux ext4 [40], a traditional, time-tested and optimized kernel-based filesystem. Via microbenchmarks, we establish the baseline performance of uFS, showing that when utilizing only a single thread, it performs similarly to ext4 (sometimes better, sometimes slightly worse). We also show that the adaptive load management works well, achieving nearly peak performance while minimizing the number of cores used by uFS. Finally, through a series of real application workloads, we demonstrate the most significant benefits of the scalable semi-microkernel approach. Specifically, uFS improves performance of a mail server, a web server, and a database application from 1.2x to 3x.

The rest of this paper is structured as follows. We first cover background on microkernels and storage development kits (§2), and then dive into the design of uFS (§3). We next evaluate uFS (§4), discuss related work (§5), and conclude (§6).

2 Background

Two trends motivate our work. The first is the emergence of the semi-microkernel as a high-performance approach to subsystem design, mostly in the domain of networking. The second is the development of storage development kits, which enable high-performance, user-level direct access to modern devices. Together these trends provide the foundations for the creation of uFS.

2.1 Microkernel-based Approaches

Microkernels have long played a part in the discussion of how to best structure operating system services. One of the earliest microkernels, developed in the late 1960's, was Brinch Hansen's Nucleus [24], argued for a microkernel approach based on modularity. The next generation of microkernels used similar arguments. For example, in the mid-1980s Mach [48, 66] stated: "[Mach] provides a small set of primitive functions designed to allow more complex services and resources to be represented as references to objects." However, performance studies revealed high caching costs [10], perhaps reducing interest in microkernels for general-purpose usage.

In later years, microkernels have seen a resurgence, both in research and in practice, across various domains. For example, some variants of the L3 microkernel provide high performance [34], and L3 is widely used on phones and other embedded devices. Furthermore, the more compact nature of these systems has led to pioneering efforts in OS verification [32].

Most relevant to this work, recent efforts in the networking domain, including IsoStack, Snap, TAS, and Shenango [31, 38, 45, 52], have created a new type of microkernel, which we refer to as a *semi-microkernel*. A semi-microkernel works in tandem with the main (monolithic) operating system, but realizes a partial or even entire OS subsystem (in this case, the networking stack) inside a user-level process. Applications wishing to use its services communicate with the semi-microkernel process via high-performance IPC channels.

The semi-microkernel approach has numerous benefits, as these works have articulated. Likely most important is *code velocity*, i.e., the ability to quickly develop, modify, and deploy system software. Instead of the slow pace common in kernel development, the hoisted subsystem can be developed in a manner more similar to application code. Application-level tools and testing frameworks can also be utilized, further improving developer productivity.

However, other benefits exist as well [31, 38]. Particularly important is the ability to scale system services independently from the applications using them [31]. When a particularly system-intensive workload is running, the semi-microkernel can recruit more resources and handle the load, regardless of the number of application threads. Similarly, when there are a large number of application threads but little system demand, the semi-microkernel can reduce its usage (perhaps down to one core).

While this approach has seen a great deal of activity and success in the networking domain, less work has arisen in the context of filesystems, where high performance approaches have mostly centered around user-level libraries (possibly with some kernel support) [9, 17, 30, 28, 33]. However, the emergence of storage development kits has now made it possible to explore the utility of a filesystem semi-microkernel; we discuss these kits next.

2.2 Storage Performance Development Kit

High-performance devices have led to the rise of user-level development kits. These libraries allow applications to directly access devices and bypass the kernel entirely. For example, the Data Plane Development Kit (DPDK) [19] consists of libraries to accelerate packet processing on a range of CPU architectures, enabling a variety of network-oriented applications to be readily implemented.

The Storage Performance Development Kit (SPDK) [55] is open-source software that enables the creation of high performance user-mode storage applications running on NVMe devices. In our approach, uFS is the "application"

while the user applications link with the uFS library to pass requests to the uFS server, which then directly uses the SPDK library APIs to perform I/O and provide the expected filesystem services.

The SPDK is realized as a user-mode device driver, and, as such, the kernel is not involved in any interactions with the device (indeed, once active, the kernel no longer can access the device at all). SPDK provides an abstraction of a *queue pair* to submit requests and receive responses; within an application (in our case, the uFS server), each thread is assigned its own queue pair, and can submit requests to the device without coordination (i.e., locking).

In this work, we focus on the lowest level SPDK APIs, which enable direct submission of requests to the device. Specifically, reads and writes are submitted via calls to spdk_nvme_ns_cmd_read and spdk_nvme_ns_cmd_write, respectively. Higher level interfaces (such as the block stack) are provided by SPDK but not utilized herein.

SPDK provides a polling-based interface via a non-blocking call to spdk_nvme_qpair_process_completions. This approach works well for high-performance devices [43] and is more natural at user-level (where event handling can be clumsy or inefficient) but requires care, as excessive polling will waste CPU resources.

Memory management is also important when using the SPDK, as memory must be pinned to enable DMA to and from the NVMe device. To facilitate this, two calls (spdk_dma_malloc and spdk_dma_free) are provided. The current SPDK implementation uses Linux huge pages, and thus the calls to allocate memory must be used judiciously.

In general, the growing popularity of these types of toolkits gives rise to the question: can SPDK (or similar libraries) enable the construction of not only a specific high-performance application (which was perhaps the intended use-case), but a high-performance user-space filesystem? We believe the answer is yes, and develop an architecture to investigate this question more deeply.

3 uFS Design and Implementation

Designing and implementing a new filesystem using the semi-microkernel approach involves addressing a wide range of issues. To begin, we describe issues necessary for correctness and base-line efficiency given a single-threaded uFS server: managing interactions between the application, uFSserver, and I/O device; scheduling requests; and library-side caching to avoid unnecessary interactions. We then expand to issues related to scalable performance with a multi-threaded server; we efficiently partition on-disk and in-memory structures across threads with an approach where each inode is owned by a specific worker and directory operations are handled by a primary thread. Next, we describe how crash consistency can be added to uFS without harming performance; each thread independently writes to a globally-ordered logical journal. Finally, we discuss how a multi-threaded server can scalably allocate

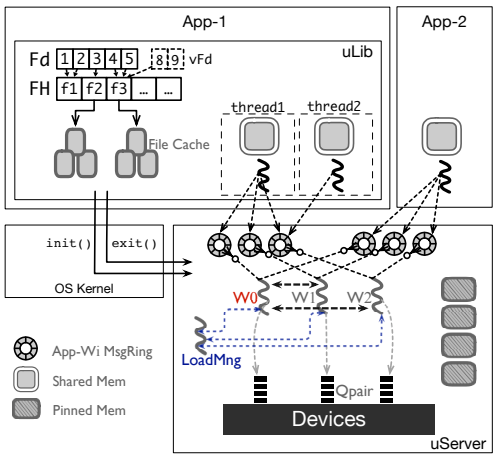

Figure 1: **Architecture of uFS, a Filesystem Semi-Microkernel**. *Multiple applications can share a single uFS. App-1 has a separate ring-buffer to communicate with each uServer worker; to minimize data transfer, App-1 contains fd and data caches and shares memory with uServer. uServer contains multiple workers pinned to cores, of which one acts as a primary; the load manager thread is not pinned. Only initialization must pass through the OS kernel.*

cores and distribute load using an approach where each worker is assigned a goal to accomplish.

3.1 Single-Threaded uServer

We introduce a single-threaded version of uFS to discuss the fundamental issues for a filesystem semi-microkernel: request scheduling, data structures, and caching.

Basic Architecture: As shown in Figure 1, uFS is composed of a filesystem process (uServer) and a library (uLib) linked with each application. The uServer is a user-level process within its own address space; we begin by assuming uServer is composed of a single thread, but this limitation is removed in the next section. We generally assume that each thread of uServer is pinned to a dedicated core. uServer interacts with the storage device with a hardware submission/completion queue pair containing NVMe commands; pinned memory is used to transfer data. uLib, which is dynamically linked into each application, offers POSIX compatibility and coordinates the connection to uServer. Control and data transfers between uLib and uServer are separated. For control, uFS uses a per-application thread-safe lockless ring buffer. For data, each application I/O thread performs allocations from thread-private memory that is shared with uServer.

The only time in which uFS interacts with the OS kernel is for initial authentication: when an application begins, uLib transparently invokes a new system call uFS_init. This system call assigns a key to each application and retrieves the application's credentials (i.e., pid, uid, and gid), which are then stored in uServer. This key is returned to the application and passed to uServer as part of any operation that requires permission checks.

uFS is POSIX-compliant with the exception of support for extended attributes, links, mmap, and chmod/chown.

Many applications run seamlessly with uFS because of such compatibility [28, 60]. Supporting mmap could further improve the usability of uFS. One could leverage *userfaultfd* to dispatch an application's uFS-related page faults for mmap'd files to the uServer; we leave this to future work. uFS is still an initial implementation and is missing some optimizations found in mature filesystems such as read-ahead and delayed allocation.

Scheduling: To provide low latency and high throughput across clients, uFS balances attending to client requests with keeping the device utilized. The single-threaded server iterates through five tasks: receiving requests from clients in the message rings and placing them in a single internal ready queue; processing requests in the ready queue (currently in FIFO order); attending to background activity (e.g., flushing dirty blocks to the device and freeing blocks from deleted inodes); initiating device requests; polling the device for request completion; and notifying the client of results. Processing a client request may generate intermediate operations that are also placed in the ready queue (e.g., pathname lookup creates intermediate operations for reading and checking the permissions of each intermediate inode and directory). The server continues polling and serving requests while other I/O operations are underway.

Data Structures: uFS uses on-disk data structures similar to other UNIX-based filesystems: superblocks, inodes, bitmaps, and directory entries. On-disk inodes are 512 bytes with standard information; in-memory inodes track related FDs and states (dirty, deleted, checkpointed). Bitmaps track blocks of different extent sizes.

Directory entries are simple mappings of name to inode number. The dentry cache is combined with a recursive permission map. For example, for the directory /a/b, the root map stores the pair <a, perms+map for /a>; the map of /a, stores <b, perms+map of /a/b>. As path are visited, they are cached in this permission map, with information obtained from inodes as needed.

Caching and Copy Elimination: uFS reduces data movement across the application, uServer, and device with three techniques: caching, leases, and shared memory. First, to avoid unnecessary I/O between the server and device, uServer contains a pinned user-level block buffer cache for inodes and data blocks. This simple LRU cache is accessed by physical block number; therefore, in-memory data structures contain pointers to the original on-disk representations.

Second, to avoid IPC round trips between the client and server, file descriptors and data are cached on clients with leases. Client caching of file descriptors (FDs) enables a subsequent open, close, or lseek (if it does not depend on current file size) to be handled locally by the client. The FD lease is invalidated if another client renames or unlinks this file, at which point the local objects are flushed to the server. FD caching improves the latency of an open from 5.5us down to 1.5us. The client cache of read data blocks is private to each process (but shared by threads). Multiple client processes can simultaneously hold a read lease; if a write request arrives at the server, the read lease will not be renewed and the writer must wait for two lease terms to expire. When there are no read leases, all reads are sent to the server. Read caching improves the latency of 16KB reads from 10us on the server down to 4.3-8us. We have also implemented a prototype write cache, which is only enabled for the ScaleFS-Bench and LevelDB experiments. When cached, writes to newly created, private files are kept local until fsync is called on that file, at which point the dirty data is flushed to uServer.

Third, to avoid copying data between the application itself and uLib, applications can directly access a memory region shared between uLib and uServer. uFS introduces uFS_malloc to allocate from this shared space; the shared buffer can be exposed to the end application for maximum performance, or hidden within uLib for portability. For example, an application can use a buffer from uFS_malloc and pass it to uFS_allocated_write to avoid any copies between the application, uLib, and uServer. Alternatively, if an application calls uFS_write(buf), uLib calls uFS_malloc, copies the contents of buf to shared memory, and then calls uFS_allocated_write. Avoiding this extra copy significantly improves latency; for a 16KB append, copying data to the server requires 8.5us, sharing an allocated buffer requires 6.5us, and local caching requires only 2.3us.

3.2 Multi-Threaded uServer

A single-threaded semi-microkernel may not be able to deliver the full bandwidth of current I/O devices to applications. Traditional kernel filesystems scale with additional cores with task parallelism: application threads running in privileged mode can concurrently access the same data on different cores. However, kernel filesystems have scalability bottlenecks from data dependencies and synchronization [13, 42].

In contrast, uFS adopts data parallelism for scalability, dividing filesystem data structures across different cores in a shared-nothing architecture [56]. Thus, the server process can be composed of multiple threads. A multi-threaded server must chose the granularity at which to divide filesystem data across threads: the more fine-grained, the more parallelism, but also the higher the complexity and synchronization. In uFS, each server thread holds exclusive ownership of an individual file and each file can be mapped to any thread. Unlike other approaches that have statically partitioned files or data across nodes [27, 29], in uFS the mapping of inodes to threads is dynamic and independent of the directory hierarchy. The drawback of per-inode partitioning is that traffic to a single (or busy) large file cannot be split across server threads.

Basic Architecture: uServer is divided into multiple threads, with each thread pinned to a dedicated core. Each thread accesses the shared storage device directly with its

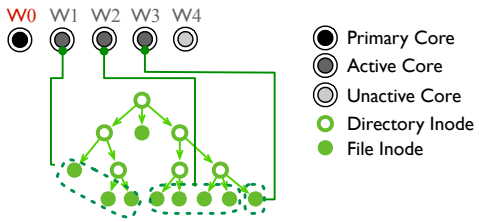

Figure 2: **Dynamic Inode Ownership**. *The dashed line indicates sets of inodes owned by each worker other than the primary; there is no relationship between the directory namespace and ownership. All inodes begin on the primary, but may be reassigned based on load. The primary owns all directory inodes.*

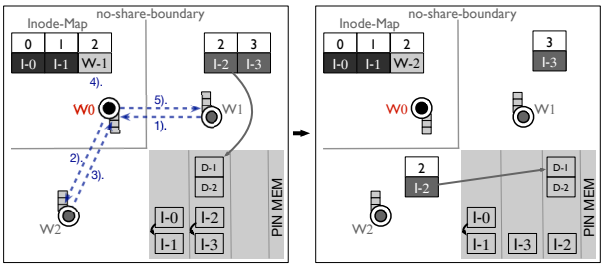

Figure 3: **Inode Reassignment**. *The left side shows initial state and 5 steps for inode 2 to be reassigned from w1 to w2. The right shows final state: the InodeMap on the primary is updated and w2 can use data associated with inode 2 in the buffer cache.*

own qpair; qpairs are not shared across threads, so no locking is needed. Similarly, each server thread has its own ring buffer to communicate with uLib in each application.

uServer is composed of one or more *worker* threads; one of the worker threads also acts as a *primary*. A ring buffer is added between the primary and other workers for communication within uServer. Each worker owns different file inodes and thus handles corresponding file operations. Beyond regular inode operations, the primary has two additional responsibilities. First, the primary owns all directory inodes; as a result, directory operations are serialized in the primary. By locating all directories in the primary, uFS avoids complex coordination for cross-directory operations. Second, the primary tracks the assignment of file inodes to threads. The primary possesses global knowledge of current inode assignments and serves as the central hub for the mechanism of reassigning inodes. All file inodes are initially assigned to the primary, but will be reassigned to other workers depending on load. A division of inodes across server threads is illustrated in Figure 2.

For directory operations, uLib contacts the primary thread. For file operations, uLib can contact any thread; if the contacted thread is not currently the file owner, uLib is notified and redirects requests accordingly.

Scheduling and Caching: With a multi-threaded server, each thread has its own ring buffer per-application, its own ready queue, and its own qpairs with the storage device. In its scheduling loop, each worker now also handles requests sent by the primary. A multi-threaded server changes caching only in that the server user-level buffer cache is now per-worker. In our prototype, each thread is allocated a fixed amount of pinned memory; dynamically sizing the buffer cache per worker remains future work.

Data Structures: Filesystem data structures are dynamically divided across server threads, with the inode as the unit of division. The worker that currently owns an inode is guaranteed to be the sole thread accessing the corresponding data, both in-memory and on-disk; thus, the ownership of an inode grants access to the data structures for handling operations involving only this inode (e.g., reading/writing/allocating data and reading inode metadata). With this separation, there is no lock or data contention for file operations. To enable independent writing of inodes across threads uFS ensures an inode fits in the atomic unit of the storage device (512B).

To manage the dynamic assignment of inodes to workers, the primary contains an inode map mapping inodes to workers; each worker tracks a list of inodes it owns. Given the primary owns all directory inodes, the primary modifies all dentries and performs all directory operations (e.g., creat); as a result, only the primary can allocate and deallocate inodes. Thus, the primary owns the imap and all dentries.

The on-disk representation of data bitmaps are more complex to handle since workers must allocate data blocks without synchronization; although bitmaps provide efficient allocation, they do entangle operations across threads given 512B atomic updates. The in-memory bitmap, in contrast, can be shared by several threads given atomic updates to a single cache line. Thus, the primary contains a *dbmap* block allocation table that maps data bitmap blocks to workers. Once a data bitmap block is used by a worker, that assignment is immutable; each worker allocates many dbmaps at a time to efficiently perform its own block allocations.

uServer also minimizes the sharing of in-memory data structures across cores. The dentry cache plays a critical role in performance, particularly for path resolution and permission checking [61]. The scalable lock-free concurrent dentry cache in uFS is based on an industry-quality hash map [2]. As described previously, each level of a pathname is a key in the map to retrieve the inode and the next level's map; the inode's permission bits are compared with the application's uid and gid. The dentry cache is single-writer (primary) and multi-reader (other workers). For most calls to open or stat, the paths are present in the dentry cache and readable by any worker. When an entry is not present, the primary finishes the lookup and inserts the items into the dentry cache. To guarantee atomicity, the primary handles some operations. For example, for atomic renames, no clients should see both filenames; thus, the primary deletes the relevant items from the dentry cache, forcing workers to redirect lookups to the primary.

Inode assignment mechanism: Figure 3 shows the mechanisms for reassigning a file inode to a worker; the policies for load balancing and determining the number of

cores are described in Section 3.4. The assignment steps are as follows. **1)** The owning thread, w1, initiates the migration of inode I2 by removing I2 from its inode list and completing any related requests. The owner notifies the primary of all state associated with I2 (e.g., opened FDs and entries in the buffer cache). **2)** The primary marks the owner of I2 in its inode map as unknown and forwards this request to the new owner, w2. **3)** The new owner sets up I2's context by linking I2 into its own inode list and extracting the buffer cache entries it can use (no copying is performed); it sends an ack to the primary. **4)** The primary changes I2's owner to w2 in the inode map **5)** The primary notifies w1 that the reassignment is complete. Any requests that arrive at a non-owner are returned to the client to retry at the primary. Once the primary knows the owner, it informs the client to redirect requests to the new owner.

3.3 Crash Consistency

uFS is a crash-consistent filesystem based on ordered metadata journaling [47]. Like other ordered metadata journaling filesystems, uFS first writes user data blocks to their in-place locations on disk; then, within a transaction in the on-disk journal, it logs a description of the metadata changes; after the transaction is marked committed, the in-place metadata can be checkpointed and the transaction marked free. If a crash occurs after the transaction is committed but before it is freed, recovery replays the changes from the transaction. Without journaling, while running, uFS only flushes dirty data blocks for files and directories and on a graceful shutdown writes bitmaps and inodes.

Basic Architecture: uFS achieves highly-scalable performance with crash consistency by allowing each thread to write to a shared journal with minimum coordination. uFS achieves this by leveraging the property that each inode has one owner and, therefore, the owner can perform the transactions involving that inode. However, this ownership is complicated by the fact a migrated inode may contain blocks that were allocated on different workers. Physical journaling at the per-block level, as in ext4, would require writing block bitmaps that are not owned by the inode owner, requiring coordination.

uFS avoids coordination with logical journaling. Each in-memory inode tracks the associated updates to other metadata structures (e.g., the data bitmap) in its ilog, an in-memory per-inode logical log that moves with its inode if reassigned. Thus, when a worker writes a transaction with an inode, it owns everything needed to apply the logical changes. When an inode is reassigned, it leaves no residual state with the previous thread [18]; as a result, an fsync on a reassigned inode requires no coordination with other threads. The primary performs similar operations for all directories with a logical dirlog.

uFS uses a global journal; the global journal simplifies the task of applying transactions in order, while still allowing threads to write concurrently. Because each thread knows the number of journal blocks for a transaction, it can atomically reserve a contiguous range of blocks; threads writing later simply reserve the next range. Journal recovery handles the case where entries that appear earlier in the journal are not committed.

Commits: In the common case of an fsync of a file, the owning thread commits the single ilog. For batched transactions, multiple ilog entries from the same worker can be placed in the same journal entry. For a full system sync, each worker fsyncs its own inodes. Directory operations (such as rename) that require atomicity across inodes imply that those inodes have the same owner; in uFS, all directories are owned by the primary fulfilling this requirement. Like ext4, fsync on a dirty directory will fsync all dirty directories; the primary commits the dirlog and ilogs of all dirty directory inodes. uFS avoids orphaned inodes and correctly handles directories that may be committed before the new inodes they reference.

While most data structures across threads are independent of one another, some exceptions exist. First, an unlink of an inode owned by a worker other than the primary (which owns the directory) requires reassigning the inode to the primary. Second, dependencies can occur across file inodes due to re-allocated data blocks. For example, unlinking an inode X may deallocate a block B which is then allocated to inode Y. To prevent the incorrect ordering of Y, X in the journal, deallocated blocks can be reallocated only after they are committed (similar to reuse after notification [12]).

Checkpoints: A checkpoint, triggered by low free space in the journal, writes committed metadata (i.e., inodes, bitmaps, and directory data) to their in-place locations. Since the current in-memory metadata may be dirty and not yet be committed, uFS maintains a stable in-memory copy of all committed metadata that is used for checkpoints. uFS uses message passing to update stable versions of data block bitmaps on other workers.

Recovery: After a crash, committed transactions are replayed. The primary challenge in uFS is to replay all committed transactions even if some appear after uncommitted entries; this can occur since threads write concurrently to the journal. If recovery were to stop at an incomplete transaction, committed transactions would be lost.

Incomplete transactions can be skipped for the following reasons. First, multiple fsyncs to the same inode are handled serially by the owner (or workers, in the case of inode reassignment); thus, a later fsync to the same inode will not complete if the previous fsync to that inode did not. Similarly, if an incomplete entry contains multiple inodes, it is guaranteed that none of those inodes are in later transactions. Second, similar to other filesystems such as ext4, XFS, and Btrfs [49], on an fsync failure, uFS will accept no more writes; thus, if recovery encounters an incomplete entry, no subsequent journal entries will involve the same uServer thread.

Recovery finds the end of the journal by reading its superblock. Since uFS updates this on-disk superblock only periodically, the contents may be stale by N blocks; thus recovery reads N blocks past the end to find valid entries.

3.4 Load Management

The final feature of uFS, load management, adapts the number of cores dedicated to the server and balances the allocation of inodes across those cores as a function of the current workload. Determining the number of cores is both a challenge and an opportunity that does not exist for traditional kernel filesystems. One option is to statically set the number of uServer threads equal to the number of I/O-intensive application threads; this enables each application thread to send most of its work to a dedicated server thread. However, for many workloads, there is a mismatch between the ideal number of application and server threads: if a few server threads saturate the I/O device, there is no benefit to adding more; if a single client generates significant I/O, additional threads may be useful. Therefore, the option we explore is to dynamically choose the number of server threads to obtain both high I/O throughput and a low core count.

For a given number of cores, uFS determines an assignment of inodes to workers that balances the load with a minimum of inode reassignments. This inode assignment must take into account a number of factors. First, the amount of work for each inode is different, depending on the rate of requests, the types of requests (e.g., reads vs. writes or size), and current system state (e.g., whether data is cached and operations will be in-memory). Second, the amount of work associated with an inode can change substantially over time (e.g., accesses to a particular file can be bursty or only occur in one phase of an application [25, 53]). Finally, co-locating inodes from the same client can improve performance, due to queueing delays.

Basic Architecture: uFS adds a low-overhead *load manager* thread to the server (not pinned to a dedicated core); the manager wakes periodically to gather load statistics from each worker, decide on the number of cores to use in the next window, and to direct the workers to perform load balancing. The manager has minimal responsibilities: it tells each over-loaded worker only the goal it must achieve in terms of how much load to redistribute; the manager does not tell workers how to achieve this goal (e.g., which inodes to redistribute). Thus, the overhead of identifying inodes is distributed across the workers; each worker contains detailed knowledge of the load caused by each inode and can accurately determine which inodes should be moved. The primary, handling all directory operations, has extra work compared to other workers; this load is included naturally in this approach and thus fewer file inodes may be allocated to the primary.

Goal and Statistics: Though different goals are possible, uFS tries to minimize both the number of cores and the queuing time of each request, by keeping each below a configurable threshold. Thus, each worker collects the CPU cycles spent on useful work within its scheduling loop, the CPU cycles spent on work for each client, and *congestion*, the average number of independent requests in the queue ahead of each request.[1] Statistics are smoothed across collection windows; the manager does not need a globally-consistent view and may read worker statistics at slightly different times.

The manager translates between per-client congestion and per-client load; both metrics are needed because clients care about their observed congestion, whereas the system can more easily manage load. The conversion takes into account dependencies across synchronous requests from the same client and non-linear effects at high loads.

Algorithm: Periodically (every 2ms), the manager determines whether cores should be added/removed and/or load should be redistributed. The current N workers are split into source and destination sets based on whether their congestion falls above or below a threshold. If there are no workers with high congestion, the manager predicts that the workers can be reduced to $N-1$ if a set of workers can accept all the load from the least-busy worker while maintaining low congestion. Otherwise, the manager determines if better load balancing on the current N workers would reduce congestion below the threshold. Because keeping requests from one client on the same worker reduces queueing delays for synchronous requests, the manager first attempts to move all load associated with an entire client; if this is not sufficient, the manager determines percentages of client load to move. Finally, if no amount of load can be moved to meet the congestion goals, these steps are repeated with $N+1$ cores. To increase stability, the predicted congestion must match measurements for several windows before the number of cores will be changed or load shifted.

At the end of each balancing window, the manager has determined the amount of per-client load to shift from each over-loaded worker to each under-loaded worker. This goal is shared with each over-loaded worker, which uses per-inode load statistics to determine a set of inodes with appropriate load. Workers avoid reassigning inodes with low (or unknown) activity, since moving those inodes incurs overhead without substantially shifting load. The worker uses the inode reassignment mechanism described previously.

4 Evaluation

We evaluate uFS by answering the following questions: How good is the baseline of single-threaded uFS compared to ext4? Is uFS scalable with multiple server cores? Is client-side caching effective? Does crash consistency add significant overhead? Can uFS adapt to workload changes? And, finally, how well does uFS handle an I/O-intensive application such as LevelDB running the YCSB workloads?

[1] Requests to the same inode are not independent because reassigning that inode will not reduce the waiting time for related requests.

a)	Rand Seq	Mem Disk	Priv Share
read	x	x	x
write	x	x	x
append	-	x	x
stat1	-	-	x
statAll	-	-	x
listdir	-	-	x
creat	-	-	x
unlink	-	-	x
rename	-	-	x

b)	Parameter
read-a	in-mem / on-disk (4KB)
read-b	4KB / 16KB (on-disk)
read-c	hot / cold (4KB, in-mem)
read-abc	2 r-a, 2 r-b, 2 r-c
write-e	4KB / 16KB (fsync)
write-f	overwr / appnd (in-mem)
write-g	hot / cold (overwr, in-mem)
write-efg	2 w-e, 2 w-f, 2 w-g
all-abcefg	read-abc, write-efg

c)	Param	Workload	Range	St
core-a	On-disk / in-mem	N*write(4K) + flush	N: (1, ∞)	19 7
core-b	Think time	In-mem read + think(T)	T (us): (15, 2)	20 6
core-c	# files	In-mem read	clients: (1, 6)	12 4
core-d	Write size	write(N) + flush	N (KB): (64, 4K)	17 5

Figure 4: **Microbenchmarks**. *Three new sets of microbenchmarks.* **a) 32 Single Op Benchmarks.** *An x indicates the specified parameter is varied; - indicates it is not. Data operations are 4KB; writes are non-allocating.* **b) 9 Load-Balancing Benchmarks**. *Each base workload contains 6 clients generating work that varies per inode. The combination workloads contain 6 clients from the base workloads. Each client accesses between 50 and 200 different inodes.* **c) 8 Core Allocation Benchmarks**. *Each workload varies over time a specific parameter: on-disk vs. in-memory, think time, number of files, and the size of operations. One version varies the parameter gradually (e.g., in 19 discrete steps) while a second more abruptly (e.g., in 7 steps). Each workload contains up to 6 clients each accessing 40 files.*

4.1 Platform and Correctness

We run all of our experiments on an Intel Xeon Scalable Gold 6138 SkyLake 2.9GHZ processor with 20 physical cores. We use one machine with 60GB (for microbenchmarks) and another with 120GB of RAM (for Filebench, ScaleFS-Bench, and LevelDB). Each filesystem runs on an Intel Optane 905P Series (960GB) SSD. The OS is Linux 5.4.0 and uFS uses SPDK 18.04.

uFS is implemented in about 35K lines of C++ code and is publicly available [5]. We have also developed tools to simplify development and to check correctness. Our command line tool, cli, supports operations like listdir, stat, and mkdir; cli also transfers files to and from the host filesystem, explicitly manages inode ownership, and dumps metadata for checking.

We ensure uFS passes the test cases in Linux's LTP project [36], adding inode reassignment across workers at controlled points. We use LevelDB extensively to validate data integrity since it stresses the filesystem and checksums all operations.

We have experimentally verified that uFS is crash consistent for a range of scenarios. Using an approach similar to others [44, 46],[2] we emulate crashes by systematically corrupting blocks in the on-disk journal; we recover with those corrupted images and verify that the recovered filesystem matches expectations. We use workloads with multiple applications that perform allocations and commit to the journal. After recovery, all files had the expected size and data, and all bitmaps were consistent. We also test creat, rename, mkdir, and unlink; all directories and files are as expected and uFS is consistent after recovery.

4.2 Single-Threaded uFS

We have two goals in evaluating single-threaded uFS. First, we demonstrate that with a single client, uFS delivers reasonable performance relative to that of a traditional kernel filesystem, ext4. We use ext4 as our standard because it is a widely-used highly-optimized kernel filesystem that scales well under many different workloads[42]; uFS uses similar mechanisms and data structures to those in ext4 (as opposed to the B-Trees in XFS [58] and Btrfs [39]). Second, we show that as the number of clients increases, a single uServer core is a bottleneck for I/O-intensive workloads.

To evaluate base performance and scalability, we have created 32 *single op microbenchmarks* for data and metadata primitives as described in the first table in Figure 4. We present the base performance of single-threaded uFS and ext4 in Figure 5 (data operations) and Figure 6 (metadata operations), both in (a) for later comparison with the scaled uServer in (b) (§4.3).

For base performance, we examine only the left-most point in each graph (1 client). For many in-memory data operations, specifically read (sequential and random) and append, ext4 and uFS perform similarly. The exceptions are that ext4 performs better on in-memory overwrites (sequential and random, shared and private files); one client performs particularly well on ext4 because data is not shared (nor invalidated) across CPU caches.

For on-disk workloads, uFS performs better for append and random reads; with a single client, uFS outperforms ext4 by 1.5x for random reads due to the efficiency of its device-access path. Ext4 performs better for sequential reads because read-ahead is not yet implemented in uFS; disabling read-ahead in ext4 removes this advantage. For on-disk overwrite workloads, we lower the kernel's *dirty_flush_ratio* to ensure that ext4 writes a similar amount of data to disk as uFS; however, overwrites still perform worse on uFS because it does not yet perform sophisticated batching for background flushes. Finally, uFS performs notably better for synchronous journal-intensive workloads (e.g., sequential appends to disk) due to its fast device access.

For metadata operations from a single client (Figure 6 (a)), uFS performs better than ext4 on listdir, create, and rename, and similarly on stat and unlink; uFS performs especially well on listdir by pre-fetching dentries during opendir. Over-

[2] We cannot use CrashMonkey because the tool replays bio_requests not present in SPDK.

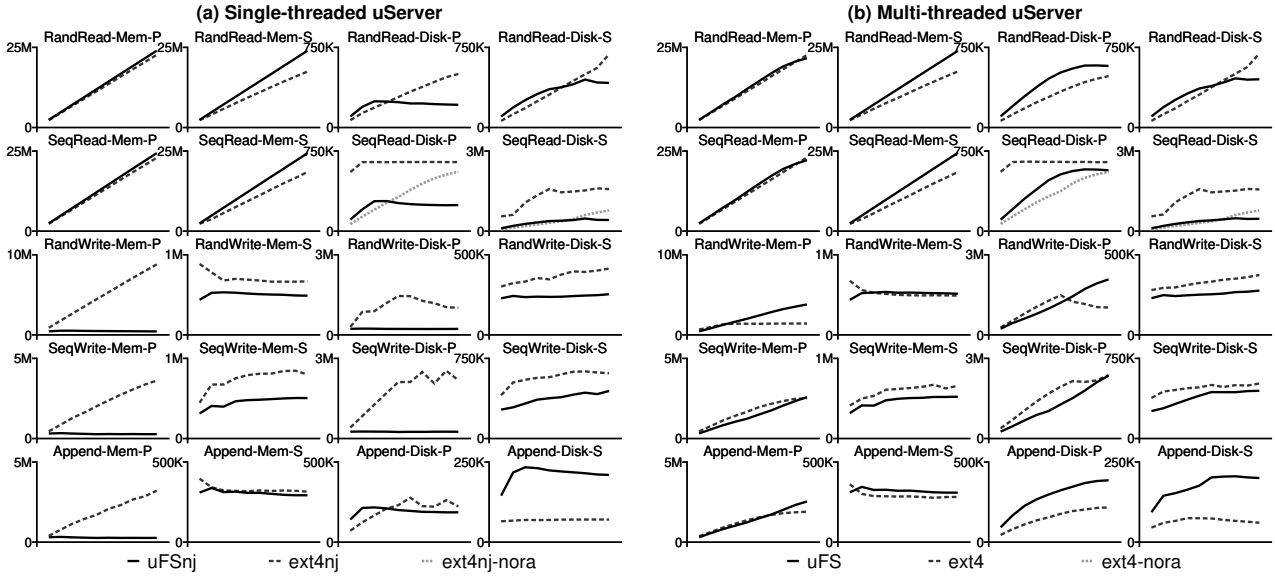

Figure 5: **Data Operation Performance: Single-Threaded vs. Multi-Threaded**. *In (a), the number of uServer cores is fixed at 1; in (b), the number of uServer cores is scaled to match the number of clients (up to 10). In "*-Mem-*" workloads, client read-caches and the server cache are warmed for uFS; the buffer cache is warmed for ext4; writing cache in uFS is not enabled and we ensure no disk access happen in "*Write-Mem-*" cases. Results with ext4 no-readahead (i.e., nora) are shown for sequential reads from disk. "nj" indicates the journaling is disabled and both ext4 and uFS use journaling in (b).*

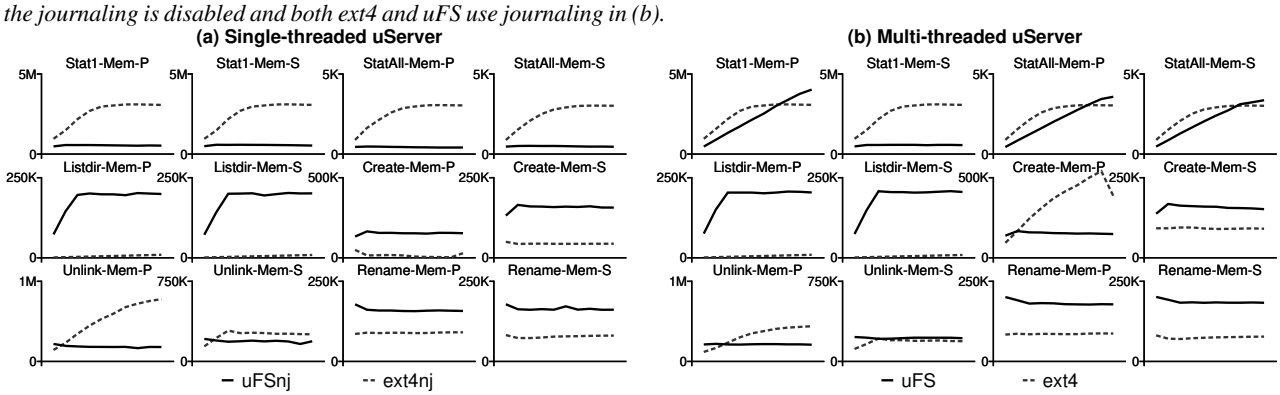

Figure 6: **Metadata Operation Performance: Single-Threaded vs. Multi-Threaded**. *In (a), the number of uServer cores is fixed at 1; in (b), the number of uServer cores is scaled to match the number of clients (up to 10). In all the experiments, the benchmark suite performs warmup round for both systems. "nj" indicates the journaling is disabled and both ext4 and uFS use journaling in (b).*

all, uFS is sufficiently well-optimized relative to ext4 for uFS to be a reasonable semi-microkernel building block.

As illustrated by the full set of data points in Figure 5 (a) and Figure 6 (a), the scalability of uFS and ext4 with additional clients is dramatically different. Although many write and append workloads had comparable performance on a single core, ext4 scales with the number of clients, whereas uFS does not. For many of the metadata operations, scalability is flat for both systems; one exception is stat, for which ext4 scales but uFS does not. For many read workloads, while both ext4 and uFS scale somewhat, ext4 scales better. We explore uFS for random on-disk reads in more detail. Figure 7 shows the CPU utilization of the uServer as a function of the bandwidth it is able to deliver; although increasing the size of reads (4KB-64KB, across lines) and the number of clients (within a line) improves bandwidth, the single core is 100% utilized with just 2 or 3 clients and thus never obtains the peak device bandwidth of 2.5GB/s. These results show that multiple server cores are required for scalable performance.

4.3 Multi-Threaded uFS

Given a more intense I/O workload, the multi-threaded uServer can effectively utilize additional cores. We demonstrate this scalability for the single op microbenchmarks, for Filebench's Varmail and Webserver [59], and for ScaleFS-Bench [7]. We show that journaling does not harm the scalability of uFS, and uFS benefits significantly from client-side caching.

Single Operations: Figure 5 (b) and Figure 6 (b) show the performance of uFS and ext4, both with ordered metadata journaling, on the single op microbenchmarks. The server is allocated as many cores as there are clients; this represents the best-case performance for uFS when no sharing of workers or load balancing is needed.

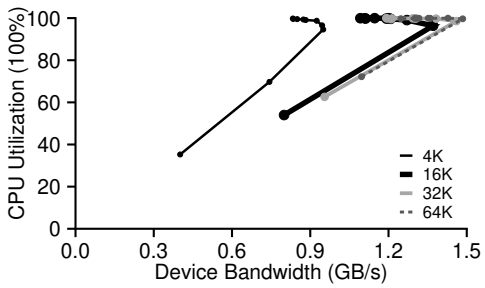

Figure 7: **Single-threaded Server Bottlenecks**. *CPU utilization as a function of delivered bandwidth for different random read sizes and numbers of clients with 1 uServer core.*

Comparing uFS's performance to that in Figure 5 (a), we see that many operations benefit significantly from additional server cores. In particular, the throughput of reads to private on-disk files increases significantly since each worker can perform independent I/O and quickly reach the device throughput limit; uFS now scales slightly better than ext4 with read-ahead disabled. Similarly, writes (both random and sequential) and appends to private files, whether in-memory or on-disk, scale since each worker can be used effectively. The scalability of writes and appends to shared files does not improve because the load is directed to a single worker. uFS makes this trade-off based on the assumption that even though file sharing is common, intensive shared-file access within a small time interval is rare.

The scalability of reads to memory remains similar to that with a single core since client caching was effective to begin with. Finally, metadata operations involving directories are still handled by the primary, and thus do not scale; stat on private files and statall on private and shared directories all scale well since groups of files can be handled by different workers.

Comparing ext4 with journaling in Figure 5 (b) to ext4 without journaling in Figure 5 (a), we see that random writes to private in-memory files perform much worse with ext4 journaling, because ext4 starts a journal transaction (and suffers from spinlock contention) even though an overwrite operation doesn't require a new journal transaction [4]. The improved performance of ext4 with journaling on create is a known anomaly [42].

Journaling in uFS does not have as strong of an impact because each worker thread participates in writing to the journal. With more detailed experiments of uFS journaling (not shown), we have verified that as the frequency of fsync increases, performance of journaling decreases, as expected, and this decrease is due to writing more data to the device and not synchronization. Writing to the global journal involves a small critical section to reserve the contiguous blocks, but eliminating this synchronization does not improve performance (validated by writing to per-worker journals). We have also verified that journaling in uFS does not impose overheads on write-intensive in-memory workloads which must track logical changes to in-memory inodes in ilogs. Journaling and no-journaling uFS obtain equivalent throughput (graph not shown): about 900kops/s with 64 byte writes and 350kops/s with 4K writes. All of our subsequent experiments use journaling in both ext4 and uFS.

Varmail: uFS obtains good base performance and scalability on I/O-intensive workloads beyond single operations. The Varmail benchmark in Filebench [59] performs reads and writes to many 16 KB files; we modify Varmail to perform periodic fsyncs so that data is written to disk during the benchmark. Varmail stresses file allocation and deletion, and is characterized by many small writes to separate files followed by fsyncs. In uFS, the file creates are all performed on the primary.

We compare uFS and ext4 scaling the number of clients, closely examining the benefits of additional workers.[3] As shown in the first graph of Figure 8, uFS is much more scalable than ext4 on Varmail. Ext4 does not scale well with additional clients because the one jbd2 journaling thread becomes a bottleneck performing the many fsync operations. uFS is well-suited to the Varmail workload because each client reads and writes independent files, which can be distributed efficiently across workers.

Even for the base case of a single client and worker, uFS performs better than ext4 due largely to the difference in fsync time (30us vs. 100us). When the number of clients is scaled but uFS is limited to a single worker, uFS performs better than ext4 up through 7 clients. Increasing the number of uServer workers to 2, 3, and 4 continues to increase throughput as each worker initiates more I/O requests; increasing beyond 4 workers does not improve performance because the primary is the bottleneck (CPU $>$ 75%). These results motivate the need to dynamically choose the number of workers to not waste resources.

Webserver: We evaluate how well uFS handles read-intensive in-memory workloads using the Webserver in Filebench [59]. Each client opens, reads, and closes 10,000 16KB private files; a small write is performed to a log file after 10 reads. Both ext4 and the uFS server easily cache the working sets for all clients in main memory and thus neither triggers substantial I/O read traffic. The Webserver stresses the ability of a single worker to handle many appends to a single file and for clients to efficiently cache recently-accessed in-memory data.

We isolate uFS client cache performance by configuring each client's read cache to contain from 0 to 100% of the client's working set; each client's FD cache fits all its opened files (requiring only 64B/FD). The second graph of Figure 8 shows uFS outperforms ext4 when the client cache hit rate is above 25%: handling reads within the uLib client is extremely efficient. Furthermore, uFS outperforms ext4

[3] Since Varmail is relatively static, uFS performs static inode balancing such that the primary handles no file inodes given many other workers ($\geq$3), and only a percentage of file inodes with 1 or 2 others.

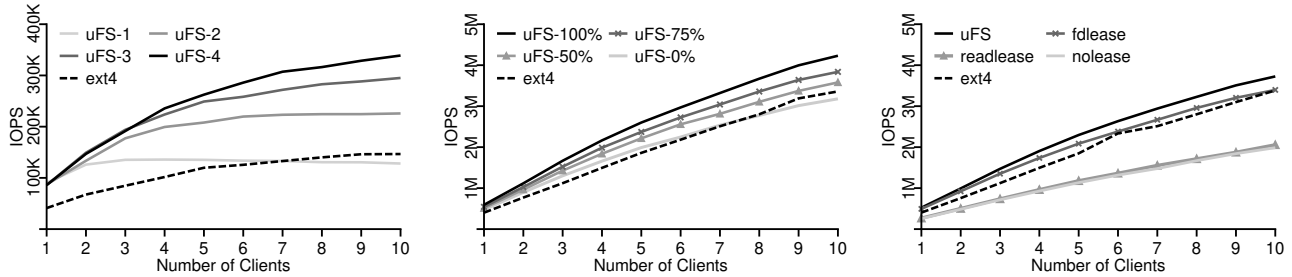

Figure 8: **Multi-threaded Varmail and Webserver**. *The first graph shows Varmail performance; different lines represent different numbers of uServer threads. The second graph shows Webserver with different percentages of the workload fitting in the client cache. The third graph shows the impact of leases in uFS for a 50% client cache hit rate.*

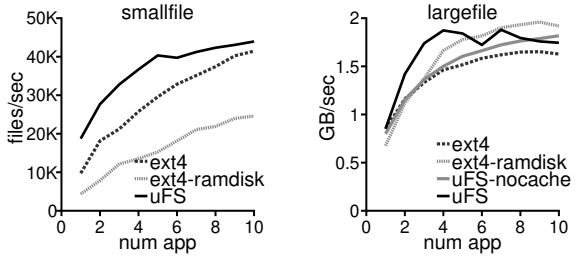

Figure 9: **ScaleFS-Bench Performance**. *The throughput of smallfile and largefile workloads. ext4-ramdisk indicates ext4 is using ramdisk. In the second graph, uFS enables the write cache to handle 256K continuous 4KB append before fsync().*

with only a few clients when their append rate to a single file can be handled by a single worker.

The third graph of Figure 8 shows the effectiveness of FD and read leases for a 50% read-cache hit rate (patterns for other hit rates, not shown, are similar). In this workload, read leases without FD leases are not beneficial because every read is preceded by an open. FD leases on their own are effective since the benefit of an FD lease is much higher than a read lease (open: 5.5us on server vs. 1.5us local; 16KB read: 10us on server vs. 4.3-8us local). As shown by the final uFS performance, given an FD lease, a read lease provides additional benefits.

Since the cost of a client opening a file and reading from the buffer cache in ext4 (2.5us and 6.5us) is less than the cost of a client transferring data from the server, uFS performs client caching with read and FD leases. While both leases are needed for uFS to outperform ext4 for read-intensive in-memory workloads, FD leases are especially valuable given their low memory overhead.

ScaleFS-Bench: We evaluate uFS with two more workloads to better understand how uFS compares to ScaleFS, a scalable kernel filesystem developed in xv6 [7]. We port their smallfile and largefile benchmarks (with minimal modifications) and follow their methodology of using ext4 on ramdisk as the baseline. We cannot compare ScaleFS directly to uFS due to a lack of hardware support for NVMe in xv6.

In the first graph of Figure 9, each application creates 10,000 1KB files, calls sync once, reads each file, and unlinks each file. uFS performs better than ext4 at each data point in the graph, yet ext4 scales better due to the burst unlink phase that stresses uFS's primary worker. If we eliminate the unlink phase, uFS has 1.4x performance on the right-most data point, indicating an optimization for bulk primary-only operations (such as unlink) would be useful.

In the second graph, each benchmark instance creates one private file, issues 100MB of writes (4KB at a time), and finally calls fsync. uFS achieves device bandwidth (2GB/sec) much faster than any others, yet shows some fluctuation when increasing the number of applications. We believe that more careful scheduling of device IO is required to regulate bursts from multiple concurrent uServer threads and provide consistently high performance. Enabling the write cache avoids unnecessary IPC and thus better utilizes the device.

One surprising finding is that comparisons should be performed on actual devices – not ramdisk – even when focusing on CPU scalability. As seen from the graphs, ext4 on the fast SSD has similar or better performance than ext4 on ramdisk. Upon further investigation (with the *RandRead-Disk-P* workload in Figure 5), we found that the kernel spends a large amount of time waiting on ramdisk IO after yielding at *io_schedule*. Thus, the performance of a filesystem run on ramdisk may be limited by the less-optimized block layer.

4.4 Load Management

uFS balances load across available workers and adjusts the number of workers to achieve low client congestion (performance) and a reasonable core count (CPU efficiency). We evaluate the load balancing and core allocation strategies using well-controlled, dynamic workloads.

Load Balancing: We first demonstrate that uFS can balance inodes with different costs across a *fixed* number of cores. We compare uFS to two alternatives: uFS_RR (round-robin inode allocation on the same number of cores as uFS) and uFS_max (each client is matched with a dedicated worker). We stress different costs per inode by constructing 9 *load balancing microbenchmarks* that each vary one parameter as shown in the second table of Figure 4. For six clients, uFS and uFS_RR are allocated only four workers whereas uFS_max uses six.

Figure 10 compares the throughput of uFS and uFS_RR, scaled to uFS_max. For all workloads, uFS achieves

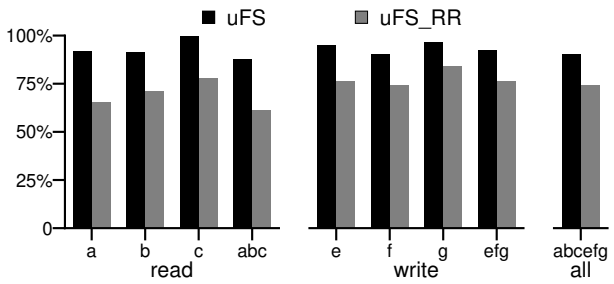

Figure 10: **Load Balancing Performance with Fewer Cores**. *The throughput of uFS and uFS_RR running on only 4 workers are each normalized to the throughput where each of the 6 clients has its own dedicated worker. Each experiment is repeated 5 times.*

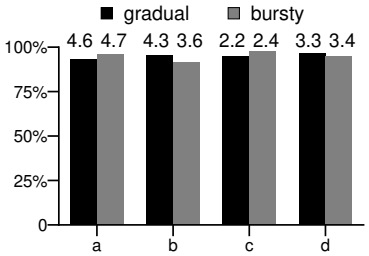

Figure 11: **Core Allocation Performance**. *Each bar shows the performance of uFS normalized to that of uFS_max, where each of the 6 clients has its own dedicated worker. The numbers on top of each bar are the average number of cores used by uFS. Each epxeriment is repeated 5 times.*

between 88% and 100% of the uFS_max's throughput, but on 4 cores as instead of 6; uFS_RR achieves throughput only between 61% and 84%. Across workloads, the more significant the difference across operation costs (e.g., workloads read-abc and all-abcdef), the more important it is to quickly find a suitable placement of inodes. For all workloads, the median time for uFS to find a stable placement is low, between 25 and 75ms.

Core Allocation: To show that uFS dynamically adjusts the number of cores, we create *core allocation microbenchmarks* that vary the load placed on the filesystem in a single dimension as shown in Figure 4. We again consider a maximum of 6 client threads. In these experiments, uFS determines a minimal number of cores that provides sufficiently low congestion and then balances inodes across them. We compare to uFS_max where each client has a dedicated core. Figure 11 shows that uFS delivers between 91% and 98% of the throughput of uFS_max with only 60% of the cores.

We illustrate the adjustments of uFS over time with a challenging workload: 8 different I/O-intensive clients enter and exit the system and change their offered load (described in the caption of Figure 12). The first graph of Figure 12 shows the CPU utilization on uFS_max given 8 dedicated cores. Even with 8 I/O-bound clients, 8 server cores leads to many wasted cycles: each core is below 50% utilization and some are below 20%.[4] Due to polling by the server thread, the OS scheduler believes each thread is using 100% of the CPU once the worker is active (i.e., has non-zero utilization in the graph); for utilization, we show the percentage of CPU cycles effectively performing uFS work. Using an average of 4.73 active (up to 8) cores, clients on uFS_max achieve 695Kops/s (not shown),

The second two graphs show the throughput and CPU utilization of uFS; uFS is configured to start on 1 core but can grow to 8 cores. The CPU graph shows that one core handles the load of the first two clients; as more clients join through time 8s, uFS activates new cores when congestion is high and rebalances inodes. At time 8, uFS observes that core 0 is congested, so adds another core and shifts work from both core 0 and 4 to core 5. When the workload decreases after time 9, uFS removes cores and rebalances inodes. Due to its rebalancing and core allocation policies, uFS achieves similar throughput with a smaller number of cores; uFS delivers 609Kops/sec on an average of 3.4 (up to 6) cores, or 88% of the throughput with 72% of the CPU resources.

4.5 LevelDB

We lastly show that uFS performs and scales well for LevelDB [51] with YCSB workloads. We measure LevelDB with two ways to load the database and six YCSB workloads [14]. Figure 13 shows that on all workloads, uFS has better base performance than ext4, and much better scalability. For I/O-intensive workloads, ext4 becomes a bottleneck due to its single-threaded journaling, and adding more load to the system does not lead to an increase in ext4 throughput. uFS scales very well with increasing load. Due to the many private writes performed by LevelDB clients, the write cache is especially beneficial in uFS. With additional load, the uFS load manager determines that additional cores are beneficial and thus allocates an average of 6 server cores for the 10 clients across the eight workloads. Thus, the throughput of uFS scales well with the number of clients; for example, on YCSB-F with 10 clients, uFS delivers 1.88x the throughput of ext4.

In Figure 13, the system with uFS (uServer) uses between 4 and 8 more cores than that with ext4. We conduct another experiment (not shown) in which the number of LevelDB clients running on ext4 is increased to use the same number of cores as uFS. However, since the additional cores cannot be used effectively by ext4, more clients and cores do not result in any significant performance gain (a maximum of 7% improvement on ycsb-e and some performance degration on other workloads).

Experimental Summary: Microbenchmarks and real application workloads demonstrate that filesystem semi-microkernels, such as uFS, can have competitive base performance to traditional kernel filesystems, such as ext4; furthermore, because semi-microkernels scale independently of applications, they can benefit from additional cores and provide scalable performance.

[4] The periodic CPU spikes are due to long tail latencies when polling the device and occur on cores with more on-disk work.

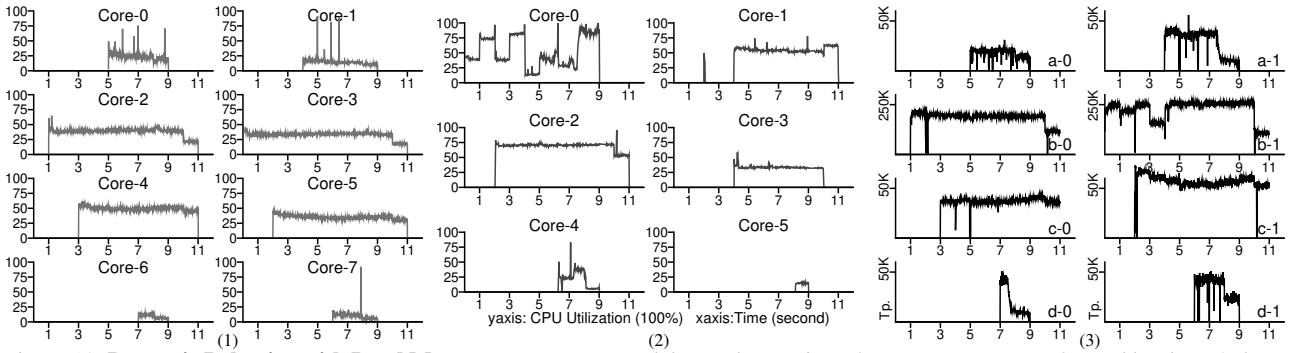

Figure 12: **Dynamic Behavior with Load Management**. *First graph keeps the number of uServer cores set at 8. Workloads: a-0: large on-disk read, a-1: small on-disk read, b-0: cold in-memory read, b-1: hot in-memory read, c-0: write+sync large, c-1: write+sync small, d-0: append, d-1: overwrite. Seconds 0-7: one app joins each second (b,c,a,d); sec 8: a,d increase thinktime; 9: a,d exit; 10: b,c increase thinktime; 11: b,c exit.*

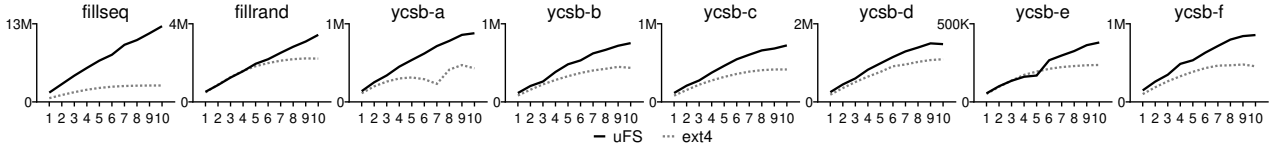

Figure 13: **Performance of LevelDB on YCSB.** *The number of clients is increased along the x-axis. The workloads are: Sequential Load, Random Load, A (write-heavy, w:50%, r:50%), B (read-heavy, w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), E (range-heavy, w:5%, range:95%), and F (read-modify-write:50%, r:50%). We use 16B keys and 80B values with 10M entries, for 1GB per client. YCSB runs 100K operations. Across the 8 workloads uFS allocates 4, 7, 4, 8, 7, 6, 5, and 5 cores for 10 clients.*

5 Related Work

uFS draws on a broad range of recent work in filesystems. We first discuss systems that explore new filesystem architectures; then we present systems that address scalability; finally, we examine related work on user-level filesystem development.

New Filesystem Architectures: Emergent devices (such as NVM and SSDs [65, 1]) have placed a spotlight on kernel overheads and have motivated researchers to revisit filesystem architecture. One approach is to enable applications to directly access the device via user-level libraries, sometimes bypassing a centralized and trusted entity. Because library-based solutions avoid the high cost of trapping into and out of the kernel [62, 33, 30, 28, 17, 67, 50], they generally provide high performance as compared to traditional kernel filesystems.

However, there are challenges with the library-based approach [30]. For example, to maintain filesystem integrity, the manipulation of metadata requires the involvement of a trusted entity, either to update the metadata or to validate the updates done by the library. Thus, maintaining metadata integrity not only slows down metadata-intensive operations, but also complicates the write path, as metadata updates are intertwined with data operations in the traditional filesystem interface (e.g., an append to a file also changes its size).

One early example of this approach is found in Aerie [62], whose library can directly access filesystem data but is read-only for metadata; a separate trusted user-level process takes care of metadata updates and inter-process sharing via a distributed locking mechanism. Strata [33] decouples layout and access methods of different devices via data migration between media. The Strata library accelerates performance by appending to a per-process private NVM log. The library also maintains a DRAM cache for structures (such as inodes) to improve read performance and acquires leases from the trusted entity for shared-file access. ZoFS [17] offloads filesystem functionality into the user's address space, where the library can directly update any data or metadata. To enforce security and permission, ZoFS includes a cooperative protocol between trusted library instances based on Intel MPK [3], but assumes the library is trusted. Similarly, KucoFS [11] equips the library with a per-file range lock to accelerate intra-process concurrent writing to a file. The library directly translates naming into device location, such that the trusted kernel only needs validation instead of costly look-ups for metadata updates.

SplitFS [28] proposes another approach where the library handles data operations and a kernel NVM filesystem (ext4-DAX) processes metadata operations. The SplitFS library improves performance by replacing data copying with linking pages and avoiding page faults on the write path. However, it has the same performance problem for metadata operations as a kernel filesystem. NOVA [64], a DAX kernel filesystem, provides atomic filesystem operations for NVM via an atomic mmap; it optimizes device access performance through per-core structures but still suffers from kernel overhead above the VFS layer.

Finally, a different approach is to push filesystem functionality further down into the devices themselves. For example, DevFS [30] pushes the filesystem entirely

into the device, thus providing direct access and serving as a centralized, trusted entity, but at a cost: the device must provide the full filesystem API – a large change from today's devices – and also be able to serve filesystem needs with limited resources (device CPU and memory). Follow-on work on CrossFS [50] distributes filesystem functionality across hardware, software, and firmware, but requires significant changes to device firmware.

A filesystem semi-microkernel differs from these approaches in that it retains the same key property of kernel-based filesystems: trust is centralized (in server software) instead of being distributed (across library, trusted process, OS, and hardware) and no special hardware is required. As such, it is relatively straightforward to implement, and can deliver scalable high performance across the entire filesystem API.

Filesystems and Multicore Scalability: Researchers have been studying the limitations of OS scalability [8, 13, 16, 22]; most of them find that the poor scalability of applications is primarily attributed to the OS. The kernel scalability bottleneck usually stems from some highly contended lock, leading to significant effort to introduce fine-grained locks and resolve the subsequent concurrency bugs [37]. The most recent Linux kernel filesystem scalability study [42] explores how the design of each kernel filesystem and the VFS layer affects application scalability, which leads to a conclusion of "speculating scalability is precarious." It is thus natural for a semi-microkernel like uFS to consider a scalable-by-design approach.

Clements *et al.* [13] take a principled approach by using the *scalable commutativity rule* to reason about system scalability. ScaleFS [7] follows these scalability guidelines, implementing concurrency-optimized data structures and a per-core private operation log for durability. Scalability is also a critical design point in recent library-heavy filesystems [63, 50, 11], commonly introducing fine-grained concurrency control into the libraries. Unlike ScaleFS, uFS generally uses per-core partitioned structures (no locking needed) and a global journal (with a small critical section).

Several kernel filesystems [29, 27, 15] exploit data partitioning for better scalability. SpanFS [29] and Hare [27] partition both files and directories into cores in a static manner. uFS, instead, dynamically change the mapping of files (excluding directories) into cores. Recent work in WAFL [15] incrementally re-architects a kernel filesystem for scalability by sharding stripes of files to cores and multi-granularity partitioning of directory tree based on the request type. Like uFS, message passing is used for users to submit requests and communication between filesystem threads. WAFL incorporates a scheduling policy that chooses a filesystem thread with more requests into the kernel CPU scheduler, which shares the same purpose as uServer's load balancing and core allocation. The data mapping in WAFL remains static and exploits NetApp's enterprise data to accelerate the common workload scalability. uFS's dynamic data mapping mechanism relies on runtime performance monitoring, and similar optimization based on offline workload characteristics could also apply.

User-level Filesystems: Lastly, we discuss related work on user-level filesystem development. FUSE [20, 26, 57] has been the *de facto* framework for user-level filesystem development. However, FUSE-based filesystems focus on functionality (e.g., ssh-based remote file access, encryption, etc.). Performance is a well-known weakness for FUSE filesystems, arising from its design.

Recently, Bento [41] provides user-space development and debugging without performance cost, by downloading the memory-safe filesystem directly into the kernel. Despite matching kernel filesystem performance (i.e., ext4), Bento still suffers from the performance overhead in VFS and other kernel subsystems, whereas uFS outperforms ext4 along numerous axes. Furthermore, Bento restricts the choice of language (to Rust) whereas uFS could be developed in any language framework.

6 Conclusion

We presented uFS, a filesystem semi-microkernel designed to extract scalable, high performance from modern devices. uFS demonstrates that the semi-microkernel approach works well for filesystems, enabling flexible scaling to match workload needs. Across a range of microbenchmarks and application workloads, uFS meets or exceeds Linux ext4 performance, in some cases by a large margin under high application demand.

Acknowledgments

We thank Simon Peter (our shepherd), the anonymous reviewers and the members of ADSL for their valuable feedback. We are grateful to Mike Swift for his suggestions and Shawn Zhong for his help on testing our artifact. We also thank the anonymous artifact evaluators for their effort. This material was supported by funding from NSF grants CNS-1838733 and CNS-1763810 and funding from Intel, Samsung, Seagate, and VMware. Sudarsun Kannan was partially supported by NSF CNS-1910593. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] Intel Optane SSD 905P Series Specification. https://ark.intel.com/content/www/us/en/ark/products/series/129835/intel-optane-ssd-905p-series.html, 2018.

[2] Folly: Facebook Open-source Library. https://github.com/facebook/folly.git, 2020.

[3] Intel® 64 and IA-32 Architectures Software Developers Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, 2020.

[4] Optimize ext4 file overwrites - perf improvement. https://lore.kernel.org/linux-ext4/cover.1600401668.git.riteshh@linux.ibm.com/, 2020.

[5] uFS Code and Benchmarks. https://research.cs.wisc.edu/adsl/Software/uFS/, 2021.

[6] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*, 1.00 ed. Arpaci-Dusseau Books, August 2018.

[7] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[8] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)* (Vancouver, Canada, December 2010).

[9] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO' 10)* (Atlanta, Georgia, December 2010).

[10] CHEN, J. B., AND BERSHAD, B. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)* (Asheville, North Carolina, December 1993).

[11] CHEN, Y., LU, Y., ZHU, B., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SHU, J. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)* (Virtual conference, February 2021).

[12] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013).

[13] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013).

[14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)* (Indianapolis, Indiana, June 2010).

[15] CURTIS-MAURY, M., DEVADAS, V., FANG, V., AND KULKARNI, A. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)* (Savannah, GA, November 2016).

[16] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013).

[17] DONG, M., BU, H., YI, J., DONG, B., AND CHEN, H. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)* (Ontario, Canada, October 2019).

[18] DOUGLIS, F., AND OUSTERHOUT, J. K. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Berlin, West Germany, September 1987), IEEE, pp. 18–25.

[19] DPDK DEVELOPMENT TEAM. Data Plane Development Kit. https://www.dpdk.org/, 2021.

[20] FUSE. Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.

[21] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, New York, October 2003), pp. 29–43.

[22] GOLESTANI, H., MIRHOSSEINI, A., AND WENISCH, T. F. Software Data Planes: You Can't Always Spin to Win. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '19)* (Santa Cruz, California, November 2019).

[23] GRAY, C. G., AND CHERITON, D. R. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)* (Litchfield Park, Arizona, December 1989).

[24] HANSEN, P. B. The Nucleus of a Multiprogramming System. *Communications of the ACM 13*, 4 (April 1970), 238–241.

[25] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)* (Cascais, Portugal, October 2011).

[26] HUAI, Q., HUAI, W., LU, J., XU, H., AND CHEN, W. XFUSE: An Infrastructure for Running Filesystem Services in User Space. In *Proceedings of the USENIX Annual Technical Conference (USENIX '21)* (Virtual Conference, July 2021).

[27] III, C. G., SIRONI, F., KAASHOEK, M. F., AND ZELDOVICH, N. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the EuroSys Conference (EuroSys '15)* (Bordeaux, France, April 2015).

[28] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)* (Ontario, Canada, October 2019).

[29] KANG, J., ZHANG, B., WO, T., YU, W., DU, L., MA, S., AND HUAI, J. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)* (Santa Clara, California, June 2015).

[30] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a True Direct-Access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)* (Oakland, CA, February 2018).

[31] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. TAS: TCP Acceleration as an OS Service. In *Proceedings of the EuroSys Conference (EuroSys '19)* (Dresden, Germany, March 2019).

[32] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., NORRISH, M., KOLANSKI, R., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)* (Big Sky, Montana, October 2009).

[33] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[34] LIEDTKE, J. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain Resort, CO, December 1995), pp. 237–250.

[35] LIU, J., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND KANNAN, S. File Systems as Processes. In *HotStorage'19* (July 2019).

[36] LTP TEAM. The Linux LTP Project. http://linux-test-project.github.io, 2021.

[37] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)* (San Jose, CA, February 2013).

[38] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RYAN, M., RUBOW, E., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '19)* (Ontario, Canada, October 2019).

[39] MASON, C. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[40] MATHUR, A., CAO, M., BHATTACHARYA, S., ANDREAS DILGE AND, A. T., AND VIVIER, L. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)* (Ottawa, Canada, July 2007).

[41] MILLER, S., ZHANG, K., CHEN, M., JENNINGS, R., CHEN, A., ZHUO, D., AND ANDERSON, T. E. High Velocity Kernel File Systems with Bento. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)* (Virtual conference, February 2021).

[42] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding Manycore Scalability of File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '16)* (Denver, CO, June 2016).

[43] MOGUL, J., AND RAMAKRISHNAN, K. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)* (Seattle, WA, October 1996).

[44] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)* (Carlsbad, CA, October 2018).

[45] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)* (Boston, MA, February 2019).

[46] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, October 2014).

[47] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)* (Anaheim, CA, April 2005), pp. 105–120.

[48] RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)* (Palo Alto, CA, 1991), pp. 31–39.

[49] REBELLO, A., PATEL, Y., ALAGAPPAN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Can Applications Recover from fsync Failures? In *Proceedings of the USENIX Annual Technical Conference (USENIX '20)* (Virtual Conference, June 2020).

[50] REN, Y., MIN, C., AND KANNAN, S. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)* (Virtual conference, November 2020).

[51] SANJAY GHEMAWAT AND JEFFREY DEAN. LevelDB. https://github.com/google/leveldb, 2014.

[52] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack – Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)* (Boston, MA, June 2010).

[53] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (San Jose, CA, October 2002).

[54] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)* (Vancouver, Canada, December 2010).

[55] SPDK OPEN-SOURCE TEAM. The Storage Performance Development Kit. https://spdk.io/doc, 2021.

[56] STONEBRAKER, M. *Readings in Database Systems*. Morgan-Kaufmann, 1994.

[57] SUNDARARAMAN, S., VISAMPALLI, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Refuse to Crash with Re-FUSE. In *Proceedings of the EuroSys Conference (EuroSys '11)* (Salzburg, Austria, April 2011).

[58] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)* (San Diego, CA, January 1996).

[59] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login 41*, 1 (2016), 6–12.

[60] TSAI, C.-C., JAIN, B., ABDUL, N. A., AND PORTER, D. E. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *Proceedings of the EuroSys Conference (EuroSys '16)* (London, United Kingdom, April 2016).

[61] TSAI, C.-C., ZHAN, Y., REDDY, J., JIAO, Y., ZHANG, T., AND PORTER, D. E. How to Get More Value From Your File System Directory Cache. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)* (Monterey, California, October 2015).

[62] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the EuroSys Conference (EuroSys '14)* (Amsterdam, The Netherlands, April 2014).

[63] XU, J., KIM, J., MEMARIPOUR, A., AND SWANSON, S. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (Providence, RI, USA, April 2019).

[64] XU, J., AND SWANSON, S. 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)* (Santa Clara, CA, February 2016).

[65] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)* (Virtual conference, February 2020).

[66] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., AND BARON, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)* (Austin, Texas, November 1987), pp. 63–76.

[67] ZHENG, S., HOSEINZADEH, M., AND SWANSON, S. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)* (Boston, Massachusetts, February 2019).