Simurgh: A Fully Decentralized and Secure NVMM User Space File System

Nafiseh Moti
Johannes Gutenberg University Mainz
Mainz, Germany
moti@uni-mainz.de

Frederic Schimmelpfennig
Johannes Gutenberg University Mainz
Mainz, Germany
frschimm@uni-mainz.de

Reza Salkhordeh
Johannes Gutenberg University Mainz
Mainz, Germany
rsalkhor@uni-mainz.de

David Klopp
Johannes Gutenberg University Mainz
Mainz, Germany
dklopp@students.uni-mainz.de

Toni Cortes
Universitat Politècnica de Catalunya
Barcelona, Spain
toni@ac.upc.edu

Ulrich Rückert
Bielefeld University
Bielefeld, Germany
rueckert@techfak.uni-bielefeld.de

André Brinkmann
Johannes Gutenberg University Mainz
Mainz, Germany
brinkman@uni-mainz.de

ABSTRACT

The availability of non-volatile main memory (NVMM) has started a new era for storage systems and NVMM specific file systems can support extremely high data and metadata rates, which are required by many HPC and data-intensive applications. Scaling metadata performance within NVMM file systems is nevertheless often restricted by the Linux kernel storage stack, while simply moving metadata management to the user space can compromise security or flexibility.

This paper introduces Simurgh, a hardware-assisted user space file system with decentralized metadata management that allows secure metadata updates from within user space. Simurgh guarantees consistency, durability, and ordering of updates without sacrificing scalability. Security is enforced by only allowing NVMM access from protected user space functions, which can be implemented through two proposed instructions. Comparisons with other NVMM file systems show that Simurgh improves metadata performance up to 18x and application performance up to 89% compared to the second-fastest file system.

CCS CONCEPTS

• **Information systems** → **Storage class memory**; **Phase change memory**; • **Software and its engineering** → **File systems management**.

KEYWORDS

phase change memory, shared memory, file system

ACM Reference Format:
Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. 2021. Simurgh: A Fully Decentralized and Secure NVMM User Space File System. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476180

1 INTRODUCTION

Non-volatile main memory (NVMM) technologies combine the persistence of traditional storage devices with the byte addressability and performance of dynamic random-access memory (DRAM). This blurs the line between memory and storage and changes how software and storage systems are developed [17, 22, 56, 69]. The recent commercial availability of Intel Optane DC persistent memory now enables the wide-spread adoption of NVMM in different layers of the memory and storage hierarchy [7, 15, 30, 32, 33, 43, 48, 57, 77].

There are several NVMM use cases in HPC. Large capacities and low power consumption of NVMM, e.g., provide the opportunity for HPC applications to store their local datasets and checkpoints at lower cost compared to DRAM and at higher speed compared to SSDs [24, 66, 74]. The persistency of NVMM furthermore allows the fast initialization of in-memory indexing data structures [37]. Additionally, many distributed systems and HPC file systems use local file systems as their storage backend and can therefore directly benefit from the performance of NVMM [1, 9, 10, 14, 54].

File systems have also been one of the first targets to be adapted to NVMM, as enabling file systems to efficiently use NVMM immediately offers performance benefits to many applications. For decades, file systems have been implemented inside the operating system (OS) under the assumption that storage devices are slow compared to DRAM and that it is beneficial to implement complex optimizations inside the storage software stack to minimize accesses to the storage devices. NVMM completely changes this assumption, and optimizations like buffering data in a DRAM page cache are bypassed by NVMM file systems to, e.g., avoid costly extra data copies between NVMM and DRAM [25, 29, 79].

Additionally, the OS and its software stack complexity have become a significant overhead when applied to NVMM. Inefficiencies of kernel file systems in Linux result, e.g., from syscalls and the generality of the virtual file system (VFS) layer. The VFS overhead is mainly due to updating file system structures, copying between user and kernel internal structures, low scalability, locking issues, and namespace management [41, 50]. Poor metadata performance, especially in shared directories and scalability of local file systems, are known to be a major bottleneck in distributed systems [1].

The costly interaction with the kernel can be minimized for the data path by manually or semi-automatically transforming applications to use memory-mapped I/O [21, 49] or by providing preloading libraries to directly access data [19, 39]. However, providing metadata scalability by removing the kernel from a file system's control path is more challenging, as the control path typically relies on the kernel to enforce security. Previous approaches to implement the corresponding metadata management in user space either use a central user space metadata server, including additional serialization efforts [45, 68], or map parts of the NVMM space into the applications' address spaces. The second approach allows to fully scale file system performance in the number of processes, while known implementations restrict security to coarse-grained permission settings [28].

In this paper, we propose Simurgh[1], a user space file system to fully exploit the performance benefits of NVMM while ensuring security and isolation equal to traditional file systems. Simurgh maps NVMM into the address space of each application and does not duplicate data or metadata by caching it in DRAM. All metadata structures can be concurrently accessed and modified by otherwise independent processes. Simurgh is stand-alone and does not depend on another underlying file system.

Security is enforced via a proposed extension to the CPU instruction set architecture (ISA) that enables the secure execution of user space functions in privileged mode. The protected functions are implemented through two new instructions, `jmpp` (jump protected) and `pret` (protected return). Jump protected temporarily changes the processor's privilege level from running in user space to kernel space upon calling the protected functions. In contrast to a syscall, `jmpp` does not induce the overhead of the kernel's dispatching table and successfully works with the CPU's jump predictor.

After its initial loading, Simurgh does not require any further OS involvement. All changes to data, metadata, and file mappings are supervised by Simurgh and require the use of protected functions. As a result, an application cannot read or write data and metadata without proper access permissions.

We emulated the new security mechanisms in the gem5 simulator [6] and show that the time to perform protected functions is close to a standard function call and 6x faster than an empty syscall. We then evaluated Simurgh using synthetic benchmarks and real-world applications on an Intel x86 server equipped with Optane NVMM by adding the measured overhead of protected function calls to file system operations. Simurgh provides up to 18x higher metadata performance compared to previous file systems, while applications can benefit from speedups of up to 89%.

[1]Simurgh is a bird in Persian mythology that represents the union between the earth and the sky and acts as a mediator between the two.

In summary, the paper makes the following contributions:

- We propose a new security mechanism that enables protected functions in user space without any performance overheads compared to system calls. Support for protected functions only requires small changes to the CPU ISA and its page table design. Protected functions are not limited to file systems and can have broad applications, e.g., in microkernel operating systems.
- We present Simurgh, an NVMM file system completely implemented as a preloading library in user space. It avoids OS traps and synchronous interprocess communication (IPC) for data and metadata operations. Kernel involvement is limited to the startup of an application.
- Simurgh forms a decentralized system, as the preloading libraries of concurrently running applications are only coordinated through accesses to NVMM and shared DRAM. We show that by simplifying internal data structures of a file system and by minimizing locks and operations in the critical path, we can gain significant performance improvements compared to previous file systems.

In the remainder of this paper, Section 2 presents the motivation and related work. Protected functions and the Simurgh security architecture are presented in Section 3. The design goals and Simurgh's internal file system architecture are discussed in Section 4. Section 5 presents the experimental results and Section 6 concludes with a summary and an outlook.

2 RELATED WORK AND MOTIVATION

This section provides an overview of NVMM file systems, their design challenges, and their impact on applications' overall execution time. We also discuss the security challenges of managing metadata in user space.

We show that existing solutions cannot offer a scalable and low-overhead user space file system that is able to provide the same granularity of access permissions as existing kernel level file systems. We therefore conclude that it is necessary to have a lightweight security mechanism for user space in-memory file systems for HPC and scientific computing applications.

Kernel level NVMM file systems: General-purpose kernel file systems like EXT4 have been adapted to NVMM by adding direct NVMM access (DAX) to bypass the page cache [25]. Additionally, new kernel file systems like PMFS or NOVA have been specifically designed and optimized for NVMM [23, 29, 51, 79]. PMFS, e.g., uses undo logging for metadata updates [29], while NOVA introduced per inode logs to improve file system concurrency [79].

All NVMM kernel file systems suffer from the scalability and performance limitations of the OS software stack and often induce additional internal overheads. The OS overhead consists of the time for syscalls, VFS, file system code, and data copies. We measured the execution time of three applications using NOVA on top of NVMM to understand the impact of the OS and file system software stack on performance (see Section 5.1 concerning the setup).

Table 1 distinguishes between the time spent inside the application, the time for data copies, and the time required by the file system. Data copies are listed as a separate column as even optimal file systems need to move data between NVMM and the application.

App	Application	Data Copy	File System
YCSB LoadA	27.02%	18.18%	54.62%
Tar Pack	8.29%	35.82%	55.89%
Git Commit	32.81%	0.45%	66.29%

Table 1: Breakdown of execution times for NOVA

The results show that the file system overheads are significant and that it is possible to improve selected applications' performance by a factor of up to two by optimizing the file system software stack. NVMM, therefore, changes performance trade-offs in a way that syscalls, the VFS, and file system inefficiencies can become more costly than the application logic.

User Space NVMM File Systems: NVMM user space file systems have been designed to further improve performance compared to kernel implementations. Some of the user space optimizations could, in principle, also be implemented in the OS, while their adoption is typically slow, as kernel maintainers have to ensure VFS compatibility with many backend file systems.

Strata [45] is a cross-device file system library that intercepts file system calls and partly handles data accesses in user space while it implements metadata management inside the kernel. SplitFS handles all data accesses in user space [39]. It uses EXT4-DAX for metadata management and minimizes metadata overheads for append operations. Strata and SplitFS can improve the performance of data operations, but they still suffer from most of VFS's scalability restrictions for metadata operations. Aerie moves most metadata operations to a trusted metadata server running in user space [68]. However, communication between Aerie clients and the metadata server requires (costly) remote procedure calls (RPCs). Moving from the kernel to a central process for metadata operations does not necessarily result in a scalable file system since the communication between client processes and the metadata process introduces a new bottleneck.

Moving fine-grained control out of the kernel has also been investigated in Arrakis [52]. Arrakis moves data protection and I/O scheduling to the I/O devices themselves, with the OS only managing the applications' access permissions. DevFS goes one step further by moving even more functionality to an NVMe SSD [40]. Security, integrity, crash consistency, and concurrency are all managed within the NVMe device so that a file system library can bypass the OS. Challenges tackled by DevFS are limited resources within the NVMe device and the lack of visibility of the state of the OS. They are using trusted threads inside the SSD as metadata processes. Since NVMMs do not have processing capabilities like SSDs, implementing hardware threads in their controller is not possible. Therefore, DevFS can only be implemented as a kernel file system and will suffer from the syscall overheads. Additionally, DevFS is based on PMFS, therefore its directory operations' performance is very low. In contrast to Arrakis and DevFS, Simurgh is directly working on top of NVMM attached to the CPU memory bus, whereas security is ensured by a CPU ISA extension. EvFS [83] is a file system that reduces syscall overheads by leveraging SPDK [82] and asynchronous I/O. It supports both NVMe devices and NVMM through SPDK, while performance results have only been reported for NVMe SSDs. EvFS does not support multi-process NVMM region updates and uses a page cache to update NVMMs, which incurs additional overheads due to extra copies.

CrossFS partitions the tasks of a file system between a user space component (LibFS) running on the host CPU and a firmware file system component (FirmFS) running on an intelligent NVMe device [55]. It exploits the availability of many I/O queues in modern NVMe devices by assigning each file descriptor to a dedicated I/O queue. CrossFS therefore requires additional HW and SW support from the storage device and cannot directly run on NVMM, even if the authors have emulated their file system using NVMM storage and dedicated cores borrowed from the host CPU. CrossFS can therefore not be directly compared with Simurgh. Ren et al. have shown that CrossFS can improve data and application performance, while no metadata benchmarks have been performed. We expect that the current emulation of CrossFS would have restricted the metadata performance based on relying on a modified version of PMFS and locks like *inode_table_mutex*.

Designing user space file systems that work directly on NVMM without a central instance introduces the challenge to provide adequate security because each file system client accessing sensitive data might maliciously change it.

Dong et al. propose a container approach where all files in a single *coffer* share the same permissions and belong to the same root page and metadata [28]. A kernel module ensures isolation between containers at the granularity of groups of pages. ZoFS is a user space library on top of coffers that moves most metadata operations to user space but still relies on some serialized functions like *coffer_enlarge*, which induces, according to the authors, scalability limitations for metadata operations like *create*. A direct performance comparison with ZoFS is, due to the non-availability of the ZoFS sources, not possible, while the performance improvements of ZoFS compared to NOVA are significantly lower than the improvements of Simurgh compared to NOVA. For instance, ZoFS offers 4.5x higher file create throughput than NOVA in a shared directory, while we improve file creates by more than 17x compared to NOVA for the same benchmark.

The main drawback of ZoFS is its incompatibility with existing security concepts, which allow distinguishing access permission between files at a very fine granularity. ZoFS additionally induces significant overhead in cross-coffer operations and protects against application errors using MPK, which limits a process to open only 15 coffers. In contrast, Simurgh does not require MPK support, as user code does not have direct access to NVMM.

KucoFS [20] tries to address some of ZoFS shortcomings. It removes the need for MPK and improves the scalability of metadata operations. However, it still traps into the kernel for metadata updates, where a single thread handles all requests. This thread can still become the performance bottleneck in highly concurrent use-cases. For instance, according to Fig. 7 in [20], KucoFS offers 25% higher throughput than NOVA in file create in different directories, while Simurgh provides 2.2x the throughput of NOVA (see Fig. 7a).
User level protection: The idea of isolation and privilege escalation in user space is not new and has been implemented, e.g., based on the now obsolete segmentation model of CPUs [3, 64]. Previous works on hypervisors use privilege separation [11, 60] or escalation [46] to provide higher privilege levels to user space functions using

ISA extensions [60] or hardware support from Intel SGX [26] and ARM TrustZone [75]. These techniques however induce higher overheads than syscalls.

There are several works that offer user space library protection through Intel MPK. Hodor uses Intel protection key hardware to offer low overhead library protection [34]. ERIM offers in-process isolation using MPK and code instrumentation [62]. Similar to ZoFS, these approaches however lack flexibility and the protection granularity of a general purpose file system. A general approach to tackle security is taken by capability-based operating systems like Opal [16] or Eros [59]. Also, new CPU architectures like CHERI provide built-in support for enforcing library-based protection [73], while CODOMs [67] uses code-centric domains and instruction pointers as capabilities. Simurgh combines the idea of privilege escalation and capability-based systems to design a secure user space file system without sacrificing performance and scalability.

Opportunities for HPC environments: I/O performance tuning to provide efficient file systems for next-generation storage systems like NVMM can be exploited by scientific applications in many different ways. The performance benefits of Simurgh, e.g., directly influence the runtime of data-intensive applications from the domains of machine learning and big data processing, which use node-local file systems to buffer intermediate data.

Burst buffer file systems furthermore use node-local file systems to store data which is accessed in the context of a single application run or for longer-running workflows [10, 65, 70, 71]. Also parallel file system like Lustre can directly benefit from performance advantages of local file system, both for object storage devices and for node-local client caching [53].

NVMMs additionally provide an opportunity for HPC applications if they are coupled in the form of disaggregated memory through RDMA. Orion is a distributed file system for NVMM that reduces CPU overheads by leveraging RDMA [80]. Orion uses NOVA's code as a base for a local file system, while adding distributed features. Orion's performance is lower than the performance of NOVA and higher than of other RDMA file systems. FileMR Similarly extends the node-local NOVA to an RDMA environment [81].

Octopus is a distributed persistent memory file system that uses FUSE for file I/O [47]. FUSE uses traditional VFS internal locks, which are the main bottleneck for metadata operations in shared directories. FUSE also imposes additional data copies and syscall overhead to every file system function call [63]. This overhead is non-negligible in NVMM environments.

Simurgh provides a light-weight security mechanism to bypass the kernel that enables a fully decentralized design. The decentralized internal structure of Simurgh, which includes allocators, metadata management, and process recovery, can be used to design distributed file systems which bypass the kernel. The design can be applied to disaggregated and distributed shared memory and to user level burst buffer systems that exploit local file systems as storage backend. The focus of this paper is on the scalability of this decentralized design on the performance on multi-core nodes, while future work will extend Simurgh to directly support disaggregated NVMM.

3 SECURITY

We propose extensions to the CPU instruction set and page table architecture to implement protected functions that enforce fine-grained security in user space. File system security can then be implemented entirely in user space and does not require the kernel as a trusted entity. Protected functions can therefore radically change how user space file systems are implemented.

Protected functions support performance-critical operations and simplify the development process for security-sensitive operating system components. The time for trapping into the kernel for file system operations like `stat` and `open`, e.g., can be more costly than the file system operations themselves [20], while we will show that the overhead of calling protected functions is in the same order as a standard function call.

Furthermore, the limitations of the VFS can induce additional delays and limit the scalability of concurrent and parallel applications [1, 50], while significantly changing the VFS or bypassing it through additional syscalls is discouraged by kernel developers to minimize the maintenance overhead of the kernel code [5].

Protected functions also simplify the implementation and modification of file systems, as software developers can rely on their standard development environment and programming languages. The concept of protected functions can be applied to the protected execution of arbitrary user level services like relational databases that require fine grained permissions or access control or to the design of complete microkernel operating systems.

We are aware that the discussion of possible security implications of a new instruction like `jmpp` cannot be entirely handled within a single paper. Discussions in this section include means to overcome jumps to arbitrary positions inside a protected function or attacks on the return address through multi-threading. We furthermore think that the efforts to securely implement `jmpp` are significantly lower than for ISA extensions like Intel SGX and that these efforts are justified by the resulting performance improvements.

3.1 Protected user space functions

This section presents a CPU extension to run protected functions, including a switch to a higher privilege mode, without OS involvement.

CPU architectures typically enforce security by grouping instructions into different protection levels [36, 38]. The running state of the x86 architecture, e.g., is divided into four protection levels. The running state can be determined using the current protection level (CPL) register, where CPL=3 indicates the least-privileged user mode and CPL<3 the different supervisor modes. For simplicity, we will only distinguish in the following between two protection levels, a kernel mode and a user mode. Memory pages are divided into user pages and kernel pages, which can be marked through page table entries (PTE).

The security extension needs to 1) prevent normal functions from accessing file system data, 2) disallow normal functions to change protected code, 3) provide a mean for transitioning privilege from normal to supervised mode, and 4) restrict the privileged execution to the predefined trusted functions or locations.

We achieve Requirements 1 and 2 by marking the file system data and metadata pages and protected functions as kernel pages.

To support the safe transition to privileged mode to support Requirement 3, we introduce a new security bit *execute protected* `ep` in page table entries that indicates whether the corresponding page can be securely executed or not. The `ep` bit can only be set from within kernel mode, and a page can only be written to from kernel mode if the `ep` bit is set.

The `ep` bit is evaluated if a user space application jumps to a function using the new *jump protected* `jmpp` instruction. If `ep` is set and the jump target is within a protected page, then the CPU can directly switch to kernel mode and execute the following instructions with enhanced privileges.

Nevertheless, using `jmpp` to jump to arbitrary positions within a protected page would immediately lead to security vulnerabilities. Therefore to support Requirement 4, the `jmpp` instruction checks whether the target address is at a fixed offset of the protected page before changing the execution mode. A single `ep` bit supports a fixed number of protected functions within a protected page by requiring that the first instruction at the predefined address is not allowed to be, e.g., a `nop` instruction.

The example in Figure 1 assumes a page size of 4 KB, that each page access is 32-bit aligned, and that the CPU supports four entry points into a protected page. These entry points are set to the offsets `0x000, 0x400, 0x800,` and `0xc00`. We assume that the `read()` and `write()` functions can be encoded in less than 1 kB, while the size of the `open()` function is slightly bigger than 1 kB. Therefore, `open()` has to be encoded in a way that the instruction at position `0xc00` is a `nop`. Code jumping to any offset other than `0x000, 0x400,` or `0x800` inside this page using `jmpp` immediately leads to an exception, even if the `ep` bit is set. These predefined addresses act as entry points to the privilege escalation.

It is necessary that the protected function reduces its privilege level before it returns. Nested protected calls are possible if `jmpp` increments a respective counter at each invocation that is decremented using a *protected return* `pret` instruction. Unauthorized `jmpp` calls have to be detected within the instruction decoding process that starts when converting a virtual address into a physical address through the translation lookaside buffer (TLB) [27].

3.2 Simurgh security architecture

Simurgh enables co-operative changes to the file system by mapping all NVMM data and metadata into the address space of each application that has access to Simurgh. The mapping ensures that these NVMM pages can only be accessed by an application having kernel privileges through protected functions. Consequently, the application cannot bypass the security mechanisms and access a file system page without prior involvement of the file system.

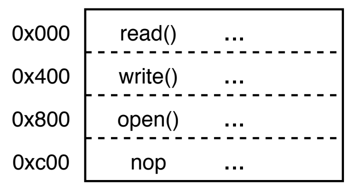


Figure 1: Internal structure of a 4 KB protected page

Figure 2 shows the overall process of moving permissions from the OS to the file system. The file system is linked in Step ① to the application as a preload library. This preload library does not have the permission to set the `ep` bit for its pages. Therefore, all protected functions have to be initialized jointly with the operating system using a bootstrap process in Step ②. In Step ③, the bootstrap within Simurgh calls the new `load_protected()` system call and passes the name of the required functions (here summarized as `simurgh`).

The protected functions have to be known to be secure by the operating system. Any jump from a protected function to a non-protected function immediately compromises security since applications can change such functions.

Also, the control flow of protected functions must not rely on any data outside the protected pages. For example, they should not read the return address from a stack stored in normal user pages. Ensuring this prevents other threads in the same process from changing the control flow of a thread running protected functions [12]. To prevent these stack modification attacks, we change the stack pointer, once entering a protected function, to a location inside the protected pages.

The security module inside the operating system ④ loads the binary of Simurgh into memory pages. It then modifies the page table of the application process and adds mappings for Simurgh pages in Step ⑤. Such pages are flagged as being protected. To protect against modifications of protected functions, relevant system calls, such as `mmap()`, are updated to prevent applications from changing the mapping table entries of protected functions.

Applications then call the standard `libc` functions to access files, and the preloading library redirects the calls to the corresponding Simurgh function using the `jmpp` instruction.

Permission enforcement using `jmpp` and `pret` is performed similar to kernel file systems. The effective user and group ID of the calling process are passed upon preloading the library and placed in the protected pages. Simurgh uses this information and the permission bits in the inodes to check the access permissions. Permission checking is performed during the path look-up process. Unlike kernel file systems, permission bits retrieval in Simurgh does not require a lazy or a second path-walk similar to RCU-walks as Simurgh does not cache metadata on DRAM and access to persistent inodes and attributes are immediate upon path look-up.

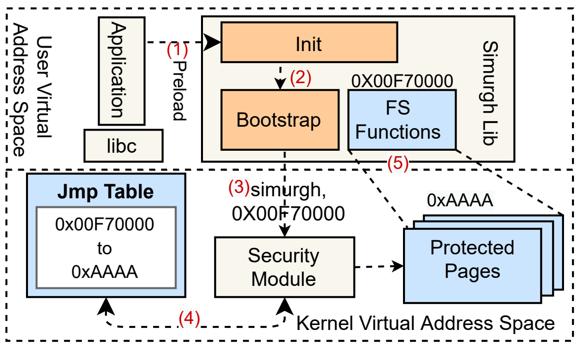


Figure 2: Security bootstrap of Simurgh

3.3 Implementation

x86 CPUs determine the Current Privilege Level (CPL) by checking the last two bits of the code segment register (%CS). Its contents can only be modified through special instructions like syscalls that involve altering the execution flow. Our proposed instruction (`jmpp`) modifies the CPL value when the conditions for jumping to the protected code are met and then performs a normal call routine.

We implemented the proposed ISA changes in the cycle-accurate gem5 [6] simulator, tested its functionality, and measured its performance impact. These simulations aim to show that the `jmpp` overhead is low compared to a standard function call. It would follow that the evaluation of Simurgh on today's HW without support for protected functions delivers comparable results to future evaluations on HW, including the `jmpp` instruction. It would not be possible to derive the same conclusions from a "real" HW implementation inside CPU architectures available to academic customers, as these CPUs are not as optimized as commercial CPUs.

The gem5 overhead of a standard x86 call routine including its return is ~24 cycles. The gem5 implementation `jmpp` checks the `ep` bit in the page table, modifies the CPL value of the CPU, and performs the call routine to the predefined protected function page. The `jmpp` and `pret` combined overhead is ~70 cycles and therefore in the same order as a standard function call. In order to compare the protected functions' overhead with system calls, we measured the latency of the `getuid` syscall as well as an empty syscall, which both took ~1200 cycles on gem5 simulator.

The syscall overhead is of course not only induced by the call itself, but also by setting up the registers and copying parameters to memory, switching to the kernel context, and locating the corresponding function for the syscall through the dispatching table. In contrast, `jmpp` employs the same technique as a normal function call for passing parameters and does not require a context switch. User code simply calls protected functions by their addresses and not by a number, and hence, `jmpp` does not need a dispatching table. Changing the CPL value and writing the return address in the protected stacks are the subset of the syscalls' operations also needed by `jmpp`, and take ~30 cycles. Additionally, checking the `ep` bit and entry points can be done in ~6 cycles.

On modern CPUs, the overhead of syscalls can be lower than indicated by gem5. On our experimental setup described in Section 5.1, `geteuid()` only took ~400 cycles, which is still 6x more cycles than for protected functions. Nevertheless, also `jmpp` can be further improved, e.g., by checking the `ep` in parallel to the instruction.

Kernel Modification: The bootstrap process, which has been explained in Section 3.2, is implemented as a Linux kernel module. It maps the protected functions and changes the `ep` bit. Since there is no remapping of the pages, the TLB flushing penalty for changing the `ep` bit in the page table entry happens only once, and its overhead is negligible. We also modified the CPU scheduler so that upon returning from interrupts and/or preemption, the CPL is set with regards to the running mode. The kernel module is designed in such a way that other user level libraries requiring protection can use it without any modification. Of course, a privileged user needs to allow the kernel module to load such libraries. In the case of the absence of the security implementations or the kernel module, Simurgh is still usable if one can trust the application.

4 SIMURGH FILE SYSTEM LIBRARY

In this section, we present the underlying design goals for the file system architecture. The main design goals are: 1) Simurgh should work as a user space file system that only interacts with the OS kernel during a bootstrap process. 2) It should allow independent processes to share and persist data and metadata directly from user space while scaling performance within the number of applications running on different cores. 3) Simurgh should provide protection and isolation guarantees equal to kernel file systems and a POSIX API to support a broad range of applications. Applications should be able to use Simurgh without source code changes.

Simurgh is built as a user level library. Upon start, it loads the protected functions into the application's address space and calls its initialization function. The protected initialization function maps NVMM into the application's address space and sets up the file system (see Section 3.2).

Figure 3 shows the memory layout of Simurgh. Simurgh performs all data and metadata operations without involving any central instance. Therefore, all processes can directly modify data and metadata without any restrictions. File system operations are performed concurrently by independent processes communicating through shared memory.

4.1 Persistent pointers

Simurgh uses `mmap()` to bring the NVMM region into the application's virtual address space. The start addresses of the memory devices in the virtual address space of the application are not predictable due to address space layout randomization (ASLR) [58]. Therefore, default pointers into NVMM or shared DRAM cannot be used between processes [18].

Others have provided different solutions to manage pointers across applications with different virtual memory layout. These solutions range from pointer swizzling [42], to use fat pointers including their additional pointer de-referencing overhead [44, 76], to sharing the address spaces of the user processes, which suffers from security challenges and the danger of race conditions [13, 16, 35], up to the even more dangerous disabling of ASLR.

An alternative is to replace the default, absolute pointers by universal *relative offsets* from the start of the NVMM or shared DRAM device. Simurgh uses these relative pointers as a replacement for all structures stored on NVMM or shared DRAM. Besides, we also optimized other components like allocators and shared data

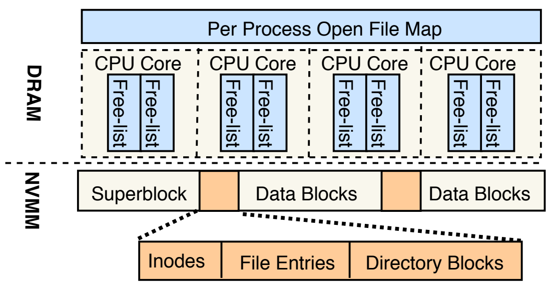


Figure 3: Simurgh Memory Layout

structures to work only with offsets to avoid conversions between offsets and default system pointers.

4.2 Allocation

Simurgh uses two different internal allocators to manage the shared NVMM and DRAM space: 1) A data block allocator which allows the concurrent allocation/deallocation of memory and 2) a metadata object allocator which allocates internal file system data structures.

Block allocation: Simurgh maintains a linked-list of free block ranges to allocate memory blocks in shared DRAM. To improve concurrency, the allocator divides the space into multiple segments, where each segment is responsible for a contiguous block range. Similar to the Hoard [4] allocator, we set the number of segments to twice the number of CPU cores, which reduces the probability of concurrent threads to access the same segment.

An atomic flag per segment is used to provide mutual exclusion to it while a `last_accessed` field stores the timestamp of acquiring this lock. Processes can detect that another process crashed while holding the lock by considering this field, the current time, and the maximum duration that a process is allowed to hold a lock. In order to allocate blocks of the same file closer to each other, threads use a modulo function based on the value of the inode pointer to select a segment. This modulo function also distributes files over segments and therefore reduces contention. If a process selects a busy segment, it simply moves to the next segment. After locking a segment, it updates the segment's timestamp and obtains the blocks from the linked-list according to a first-fit algorithm before unlocking the segment again. Upon deallocating a block, the process finds the corresponding segment, locks it, and adds the block to the list of free blocks before unlocking it.

Data structure allocator: Most metadata operations require at least one allocation/deallocation of a file system data structure. Hence, this allocator needs to be fast and highly scalable. Simurgh uses an allocator for fixed-sized metadata objects such as directory blocks, file entries, and inodes, which works similar to the Linux slab allocator [8]. This allocator creates a pool of metadata objects on preallocated segments acquired from the block allocator. New segments are allocated on demand. Simurgh saves the layout of the preallocated metadata spaces inside the superblock.

Each metadata object contains an atomic `valid` and a `dirty` flag. When the file system requests a metadata object, the allocator assigns an object from the pool, sets the valid and dirty bit, and returns it. A metadata object is ready to be allocated from the pool if both bits are unset. The valid bit is only set by the allocator, while the dirty bit is set for an unprocessed metadata object. Using two bits prevents the loss of an allocated metadata object during a crash. The recovery procedure is done according to the state of these two bits. Upon deallocation, the metadata object needs to be zeroed, therefore the allocator unsets the valid flag, then zeroes the object and unsets the dirty bit.

4.3 Data and Metadata Management

Removing the kernel as the coordinating instance requires user processes to access and modify data and metadata concurrently and consistently while communicating through shared memory. These processes must not depend on each other, and all non-crashing operations need to be completed and persisted by their calling process, while the file system libraries have to provide recovery mechanisms from process-level and system-level crashes.

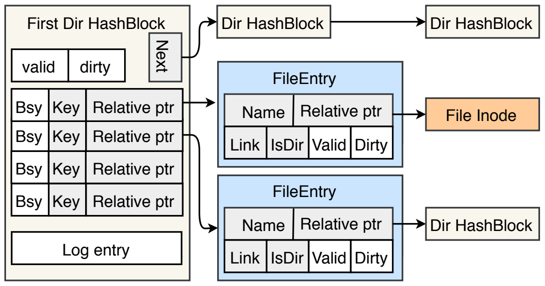


Figure 4: Directory structure

In VFS, all directory operations are sequential, regardless of being reads or writes [50, 78]. In the following, we present our lessons learned from these VFS limitations and how our directory structure shown in Figure 4 enables Simurgh to scale the performance of metadata operations acting on directories.

Inode: Kernel file systems contain two different `inode` structures connected through an inode number. Simurgh does not exchange metadata with the kernel, so we have removed the inode number and the corresponding mappings. Simurgh instead uses the 64-bit persistent pointers as unique inode identifiers, which act as offsets from the beginning of the NVMM space to the inode locations. Simurgh, therefore, does not require costly indexing structures for converting inode numbers to locations and also does not have to copy inode data between the VFS and the file system.

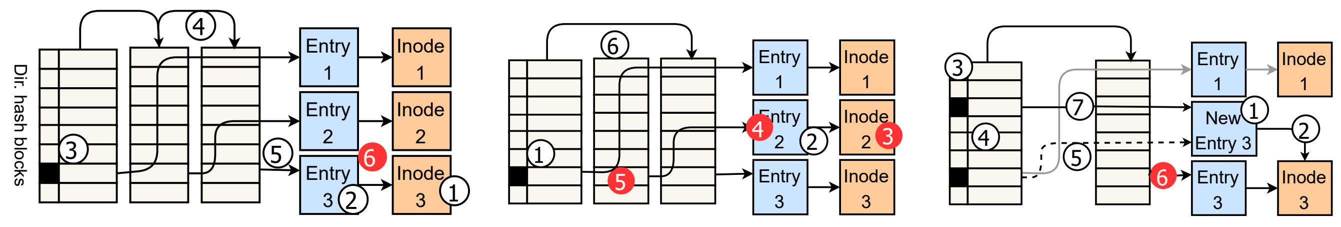
Directory blocks: Simurgh uses linear hash maps as directory blocks. Each directory block maps hashed keys to persistent pointers that link to the physical locations of file entries. File entries can represent files or directories and maintain a name field and the link to their inode or directory block.

Hash maps require no merges or shifts to maintain the ordering of inserts or deletes. Therefore, they allow us to maintain consistency and failure atomicity without performing extra memory copies otherwise required for journaling or logging. Directory blocks can be linearly extended through a `next` field. The first directory block in a directory contains a busy flag per line and a single log entry for cross directory operations.

Open file map: Simurgh uses per-process maps that store file descriptors of open files. Each map entry contains the file's open mode, the current position in the file, the path, and the persistent pointer to the inode. The data structure allocator in Simurgh provides lockless allocation for concurrent multithreaded open/close.

Concurrent file systems operations: In the following, we will present examples of concurrent directory operations and how they benefit from the underlying hash blocks.

Figure 5a shows the workflow for creating the new File 3. We assume that File 1, File 2, and File 3 hash to the same block in the hash table and that there is no empty space in the existing hash blocks. Therefore, a new hash block needs to be created. In Step ①, the inode of File 3 is created and persisted. In Step ②, the corresponding file entry is created and linked to the inode. We have



(a) Creating file 3 (b) Deleting file 2 (c) Renaming file 3

Figure 5: Shared directory metadata operations. The arrows depict the relative pointers to the metadata objects

to set the busy flag of the whole line in the chain of hash blocks in Step ③ before a new hash block is created in Step ④ and is linked to the previous one. The pointer of the new file entry in the hash block is persisted in Step ⑤. Finally, the dirty bits for the newly created data structures are unset in Step ⑥. In case of a crash before Step ⑤, the file is not created and no crash recovery is needed. The allocated objects can be reclaimed by the metadata allocator.

Figure 5b shows the process of deleting File 2. In this example, the three depicted files map to the same row in the hash blocks, and therefore, the directory has three hash blocks. The delete process starts in Step ① by setting the busy flag for the whole line in the chain of hash blocks to prevent concurrent accesses to the row. The valid bit of the file entry of File 2 is unset and the dirty bit is set in Step ②. This indicates that the operation on the file entry metadata object has not been completed. The inode of File 2 is zeroed in Step ③ and its file entry is zeroed in Step ④. The pointer in the middle hash block then points to a zeroed object, which shows that it is invalid and hence, the pointer needs to be zeroed. The offset of the file entry for File 2 in the middle hash block is then zeroed in Step ⑤. Since the middle hash block is empty, it should be freed by unsetting the valid bit and updating the link to the next hash block in Step ⑥. If the process crashes in between Steps ② to ⑤, the next process accessing the same line identifies a null pointer and completes the remaining steps for deletion. The last step is optional, and crashing before that will not impose any inconsistency.

Renaming a file in the same directory starts by creating a new file entry which serves as a shadow copy of the old file entry (Step ① in Figure 5c). We update the file entry to point to the same inode in Step ②. In case of a recovery process, we need to identify hash blocks with ongoing renames to scan and fix them. Therefore, we set the busy flag of the whole hash block in Step ③ and the busy flag of the corresponding line for the old name is set in Step ④ to mark the transaction and prevent modifications of lines involved in renames. The pointer to the old file entry is changed to the new file entry in Step ⑤. This makes the line inconsistent since the hash of the new file entry does not match the line that points to it. We use this inconsistency and the dirty and valid bits in the directory block to detect any failed intra-directory rename and the next process accessing the same row in the hash block continues its execution. The old file entry is no longer needed and can be removed in Step ⑥. If the process or the system crash before this step, the old file entry is freed during the next file system maintenance check. The pointer in the new line is updated in Step ⑦ to point to the new file entry and the pointer created in Step ⑤ is removed in Step ⑧.

Cross directory renames i.e., moving a file to a different directory, are special since they require more than two simultaneous updates. Therefore we use one log entry per directory for this purpose. Upon issuing a cross directory rename, the operation will be written in the old directory's log entry in Step ① and its dirty bit is set in Step ②, which specifies a rename operation in the directory. The corresponding old and new rows are being locked in Step ③ and in Step ④ the operation is performed.

Directory operations strongly influence file system performance [50, 78] and Section 5.2 shows that our design can significantly improve the scalability of `create`, `unlink`, and `rename` operations in shared directories.

Symbolic links are supported through the `link` flag in the file entry structure. If this flag is set, then the inode that it points to only stores the destination path. Hard links are supported by different file entry structures being able to point to the same inode and by keeping a reference counter inside the inode. Creating a symbolic link is similar to creating a file entry object with the `link` flag set.

Crash recovery: Simurgh is entirely decentralized, and it is necessary to perform crash recovery after an individual process crash. A process is considered to be failed when another process waits for more than a certain threshold on a busy-wait lock. In this case, the waiting process performs the recovery corresponding to this lock. In case of a crash during a create, delete, or rename, the following process finishes the operation. The recovery decisions are made based on metadata object flags. We ensure that the combination of metadata flags and the operations leads to a unique recovery decision. For example, a dirty directory bit in Figure 5c always indicates a rename process. If a crash happens after step 5, only a single directory scan must be performed to find the mismatched hash and complete the rename operation. Recovery from a whole system crash or improper shut-down is performed by scanning all metadata objects, and the file system's data blocks upon initialization.

Data operations: Simurgh uses non-temporal stores to bypass the CPU caches for writing data to files. We employ a read/write lock per file to ensure exclusive writes while allowing concurrent reads. Simurgh makes sure that the metadata updates occur after the data has been persisted using `sfence`.

5 EXPERIMENTAL RESULTS

This section first presents the evaluation environment and the setup of our experiments. Next, we discuss the impact of our design choices and optimizations on data and metadata performance and scalability. Then, we present the impact of our optimizations on real world applications. We compare Simurgh with EXT4-DAX, PMFS, SplitFS, and NOVA, while a comparison with ZoFS has not been possible, as its sources are not publicly available. We do not compare with overlay approaches like Libnvmmio [21] because they only support a limited number of file system operations directly and dependent on an underlying file system. Finally, we measure the time to recover Simurgh after a system crash.

5.1 Evaluation setup

All experiments have been conducted on a single socket server with a 10 cores Xeon Gold 5212 processor running at 2.5GHz, 192 GByte DRAM, 746 GByte Intel Optane DC persistent memory across 6 DIMMs, running a Linux kernel on CentOS 8.2. We have used kernel version 5.1.0 for running Simurgh, NOVA, and EXT4-DAX. SplitFS and PMFS were compiled against their supported kernel versions (4.13.0 and 4.18.19, respectively). We have chosen the POSIX mode for SplitFS that offers the highest performance. NOVA has been configured with inline writes, which has less strictness and higher performance.

We have used the most recent available and runnable code version for all file systems, and we have seen that the performance of some file systems like NOVA has improved since their initial publication, so that sometimes also the relative performance between them has changed compared to previous publications. Furthermore, the performance of NVMM file systems significantly depends on the number of NVMM DIMMS, so that the absolute performance cannot be directly compared with previous publications.

We zeroed both NVMM and the shared DRAM between experiments and flushed the OS buffer cache after each test. We selected synthetic microbenchmarks from FxMark [50]. We also applied Filebench [61] as macrobenchmark and included YCSB, Git, and Tar as real world applications. All benchmarks have been run 20 times for each data point. To take the overhead of our security mechanism into account, we added 46 cycles (the difference between normal and `jmpp` calls) to each Simurgh call.

5.2 Microbenchmarks

We selected 10 microbenchmarks from FXMark to compare Simurgh with others. FxMark has not been designed for benchmarking NVMM and we have slightly adapted it to minimize the influence of the CPU cache. When reading files, e.g., we do not access the same blocks repeatedly, but select pseudo-random addresses. Fig. 6 shows the bandwidth reported by the original FxMark and the maximum bandwidth of NVMM in our setup, indicating that Simurgh and Nova mostly work on cached data for the original FXMark. It also shows that both file systems' performance is bound by the NVMM bandwidth when using the adapted FXMark.

Simurgh scales significantly better than the competing file systems, while its base performance is also typically the best. The only exceptions are append and fallocate for small numbers of threads. SplitFS offers higher append performance for small thread counts due to its targeted append optimizations append. PMFS has higher throughput for fallocate because of a simpler, but unscalable allocation mechanism. Due to space limitations, we typically

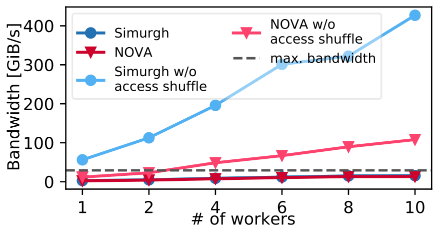


Figure 6: FxMark read (DRBL) performance

only describe for each benchmark the reasons for the performance differences between Simurgh and the best competing file system.

Figure 7a presents the *createfile* throughput in private directories (one directory per thread) when creating 1M files. The benchmark stresses inode allocation and directory management. All file systems are able to scale performance as the running processes do not compete for the same directory blocks. Simurgh creates files 3.4x faster than NOVA for a single thread and 2.2x for 10 threads.

All file systems performing metadata management inside the kernel do not scale for creates inside shared directories (Figure 7b). In their case, the processes compete for a shared VFS inode lock to change directory entries. Many real world applications, e.g., from HPC [2, 31] and mail servers, therefore suffer from performance penalties or have to adapt their code to avoid shared directories. Simurgh scales for both private and shared directories as its directory hash map supports concurrent modifications.

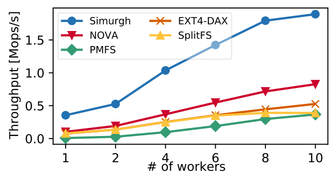
The same behavior can be seen for *deletefile* that removes 1M empty files from private directories (Figure 7c). Constant updates to the directory entry (dentry) cache lead to the poor performance of kernel level file systems [78]. PMFS additionally suffers from its sequential search inside directory blocks. Since deallocating data structures does not require to allocate a metadata object, Simurgh shows even higher performance in deletefile compared to createfile.

renamefile renames empty files in shared directories (Figure 7d). Simurgh is 2.2x times faster than EXT4-DAX for a single thread and 18.8x faster for 10 threads. The fine-grained busy-wait locks on hash table lines allow for fast, concurrent, and consistent updates.

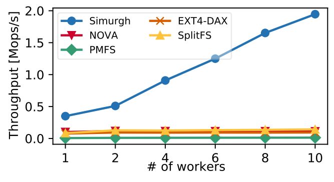
Figure 7e shows the performance for *resolvepath* to open files in private, nested directories of depth five. All kernel file systems show the same performance because of the large dentry cache. Although Simurgh does not benefit from a directory entries cache, it shows superior performance compared to others because of its fast and concurrent lookup in directory hashes. SplitFS performs metadata operations through EXT4-DAX, which seems to induces even higher path resolution overheads.

Contention in the dentry cache can lead to scalability bottlenecks of resolvepath when processes share common paths [50]. Figure 7f confirms this for others, while Simurgh still ensures scalability. This shows that in addition to the overheads of searching for inodes by the inode number, VFS also imposes scalability limitations on path resolution. Simurgh operates directly on pointers and does not induce the overhead of relying on a tree to locate inodes.

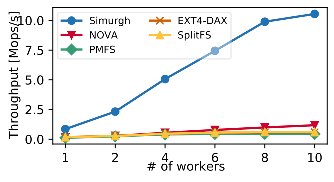
Extremely fast operations like *resolvepath* greatly benefit from our security mechanism, since ~330 cycles saved by removing



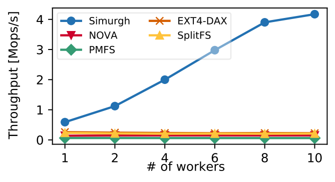
(a) createfile, private dir (MWCL)



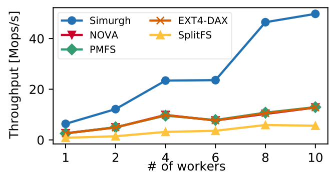
(b) createfile, shared dir (MWCM)



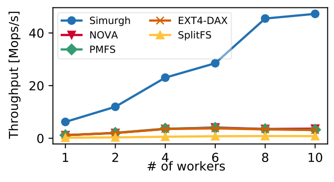
(c) deletefile, private dir (MWUL)



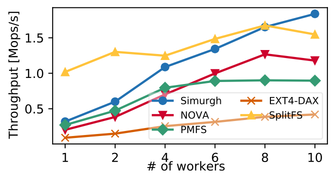
(d) renamefile, shared dir (MWRM)



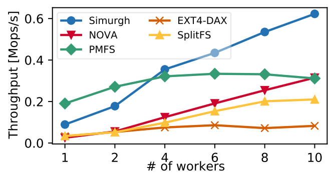
(e) resolvepath, private parents (MRPL)



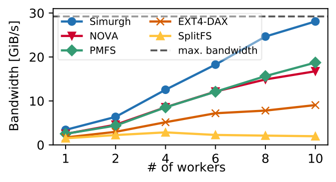
(f) resolvepath, common path (MRPH)



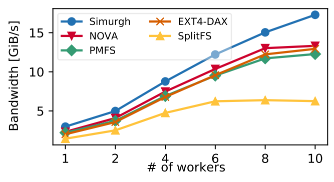
(g) appendfile, private file (DWAL, 4K)



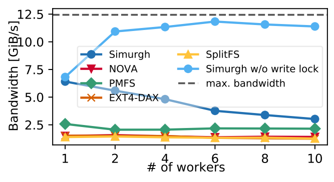
(h) fallocate, private file (4M)



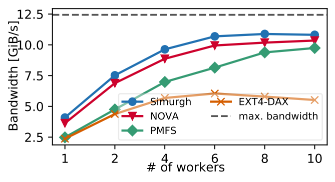
(i) reader, shared file (DRBM, 4K)



(j) reader, private file (DRBL, 4K)



(k) overwrite, shared file (DWOM, 4K)



(l) write, private file (100GB,512MB/op)

Figure 7: Microbenchmark results. The four letter shortcuts are taken from FxMark (if available).

Workload	# Files	Dir Width	File Size	# Threads
Varmail	1,000	1,000,000	128KB	16
Webserver	1,000	20	128KB	100
Webproxy	10,000	1,000,000	16KB	100
Fileserver	10,000	20	128KB	50

Table 2: Filebench Workloads (default settings)

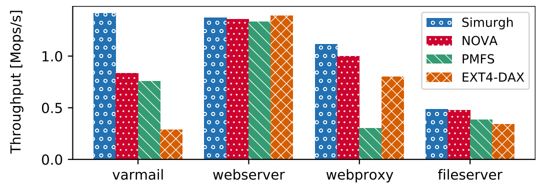


Figure 8: Throughput for Filebench workloads

syscalls can reduce the operation's latency by half. On slower operations, improvements entirely originate from the library design.

appendfile appends 100,000 4 kB data blocks to private files to evaluate the concurrent block allocation (see Figure 7g). Most file systems allocate a 4 kB block for each append and update their data mapping structure for files. SplitFS allocates bigger append regions that are lazily synchronized, reducing the allocation overhead and providing a very high append performance for small thread counts. Simurgh scales to ten threads, while the PMFS' performance remains constant beyond four threads due to its serial block allocator.

fallocate requests 1,000 4 MB data chunks for private files to stress the block allocator. We configured all file systems to not zero the allocated blocks and issued `fsync` to persist the changes. Figure 7h shows that PMFS and EXT4-DAX do not provide scalability due to their sequential allocator. Simurgh scales by distributing the requests across several lists and checking for a free page through bitmaps. NOVA and SplitFS also offer scalability but their base performance is significantly lower than that of Simurgh.

The shared file *read* benchmark (see Figure 7i) investigates the random read performance when accessing a shared file. *Max. bandwidth* denotes the maximum bandwidth of our NVMMs that is measured by issuing sequential read or writes using up to ten threads. We found the NVMM saturation point by increasing the thread count gradually. Simurgh saturates this bandwidth, which shows its efficiency in serving data requests. The private file random read benchmark allocates a separate set of files per thread (see Figure 7j). Simurgh scales similarly for both shared and private files, while the other file systems perform poorly in reading from a shared file. The main reason is Linux's read and write semaphore which is being updated atomically.

A different behavior was observed for overwriting a shared file (see Figure 7k). Simurgh provides file-granular mutually exclusive writes and threads spend more time to acquire these write locks when scaling their number, leading to performance penalties while still being significantly faster than others. The file-granular locks for writes have been implemented to make the Simurgh semantics comparable to other file systems and can be disabled if scalable

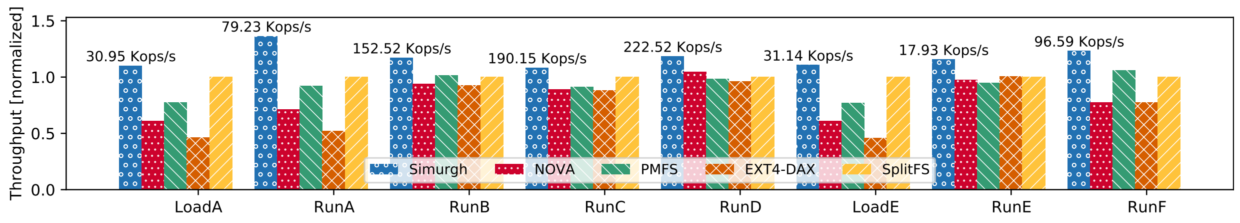


Figure 9: Throughput for YCSB workloads

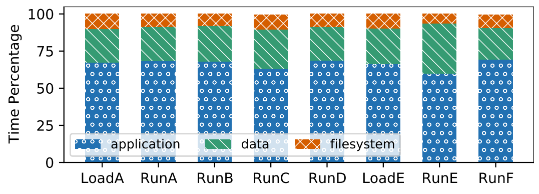


Figure 10: YCSB execution time breakdown for Simurgh

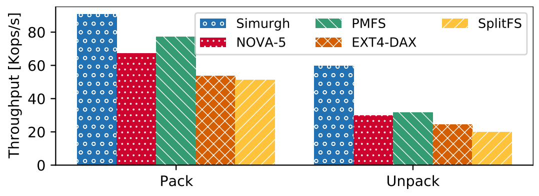


Figure 11: Tar throughput

writes to shared file are required and coordinated by the application itself. Therefore we have also included the results for the *relaxed* write Simurgh. Writes to large private files are slightly slower than overwrites, as they include additional overhead to allocate blocks. Figure 7l shows that Simurgh nevertheless scales well for writes to private files and is again faster than all competing NVMM file systems. We were unable to run SplitFS for this benchmark.

5.3 Macrobenchmarks

We used Filebench to test synthetic workloads (see Table 2 for the settings). Varmail is metadata intensive and Simurgh therefore strongly benefits from its user level optimizations. It outperforms NOVA by a factor of 1.7x (see Figure 8). EXT4-DAX performs poorly in this workload since the files are small and it is optimized towards large files and access sizes.

Webserver concurrently opens and reads files while occasionally performing appends. All file systems perform similarly since this workload mostly consists of reading private files, which confirms the microbenchmark for private reads.

Webproxy consists of concurrent create, read, append and unlink operations. Simurgh achieves an 11% higher performance than NOVA. EXT4-DAX performs better in webproxy, compared to varmail, but worse than webserver, which shows its optimizations towards data operations and its poor performance for metadata. Here PMFS performs poorly as it keeps an unsorted list of directory entries, which hurts the performance of unlinks. This behavior was not seen in varmail, since it contains 10 times fewer files.

Fileserver emulates a file server by performing create, stat, and unlink operations and by reading and writing large files. NOVA and Simurgh offer almost the same performance since reading contributes to most of the program execution time.

5.4 Real-world applications

In this section, we present the performance impact of our optimizations on three real-world applications.

YCSB contains a set of six key-value store workloads. We used LevelDB as the backing database and employed the software provided in [39]. Figure 9 shows the throughput of the examined file systems for this benchmark. All values are normalized to that of SplitFS. Simurgh achieves the highest data throughput in all workloads. Although LevelDB heavily relies on append for maintaining the database and SplitFS is optimized towards append, Simurgh still provides higher performance. This is because of our lower processing overhead needed for servicing data requests and orders of magnitude faster metadata operations, needed for creating and deleting files. The highest improvement of Simurgh compared to SplitFS belongs to RunA (36%) with the highest update ratio.

Figure 10 shows that the overhead of Simurgh in YCSB workloads is less than 10% of the overall application runtime and, hence, that additional file system optimizations will not further significantly improve its performance. The rest of file systems spent the same amount of time in the application and data copy process and their lower performance is due to their higher file system overhead.

The *Tar* benchmark packs and unpacks the Linux kernel source code into/from one file. The pack workload measures the performance of locating files while performing data operations. This benchmark does not issue any flushes. Kernel level file systems can benefit from all `libc` and VFS caching mechanisms for both data and metadata. Figure 11 shows that Simurgh can again reduce the execution time of the benchmark compared to others. This shows

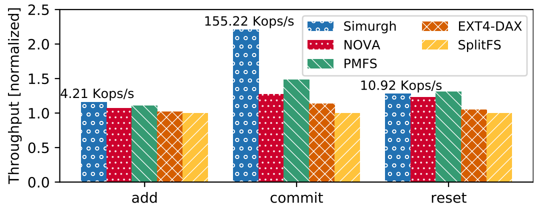


Figure 12: Git throughput

that despite the lack of caching, it offers a higher performance by redesigning the file system architecture. The unpack workload issues several syscalls per file to set, e.g., accessed time or permissions. Since Simurgh does not have to issue syscalls and does not rely on the slow VFS, it offers 2x higher throughput, compared to others. Note that this benchmark employs `mmap` to read the large packed file. Simurgh implements `mmap` similarly to other file systems through the `mmap` syscall by modifying the page table. Therefore, the performance of `mmap` accesses is the same for Simurgh and others.

In the *git* benchmark, we measured the throughput of `add`, `commit` and `reset` using the Linux source code. The git internal automated garbage collection was disabled and between `commit` and `reset`, all files were deleted (Figure 12). In both `add` and `reset`, file system operations contribute to a small percentage of the execution time. Hence, Simurgh does not offer higher performance. In `commit`, git retrieves the metadata of all the files and therefore, Simurgh improves performance by 48% compared to PMFS. Similar to the tar benchmark, git is also single-threaded and PMFS offers higher performance, compared to other previous file systems.

5.5 Recovery Test

Simurgh recovery uses a mark-and-sweep approach; therefore its memory consumption is linear in the number of files and directories. Since most metadata objects are close to each other, we can benefit from the full bandwidth of NVMM when reading them. To measure the recovery time from a full system crash, we crashed a file system with 10 directories each containing the complete Linux source code (672,940 files and 88,780 subdirectories). The recovery process to a healthy state took 4.1 seconds and is therefore slightly faster than results presented in [28, 39]. The time required for a runtime process recovery, e.g., for an unfinished rename operation through scanning hash consistency in one line, is negligible and not measurable. The recovery process and the retrieval of assigned but unused metadata objects can be performed in the background so that recovery start-up times are minimized. Nevertheless, we are aware that file systems can contain billions of files and that the recovery process has to be further improved for production use.

6 CONCLUSION AND OUTLOOK

In this paper, we introduced the concept of protected functions. It enabled us to circumvent the unnecessary software stack of the OS and to offer loosely coupled secure software services to applications. By employing the concept of protected functions, we designed Simurgh, a completely user space NVMM file system in shared memory. Simurgh offers the same protection and access rights as Linux file systems. Simurgh provides scalable data and metadata operations while guaranteeing consistency, durability, and ordering, by simplifying data structures and employing lockless operations. Experimental results using real NVMM show that Simurgh is up to 18x faster than previous file systems for metadata operations and improves real world applications by up to 89%. Future work includes adapting Simurgh to distributed shared memory environments and multi-node HPC applications, the implementation of the proposed instructions inside the RISC-V processor to understand area and performance implications [72] and its application to optimize microkernel OS implementations.

ACKNOWLEDGMENTS

This work has been supported by the European Comission's BigStorage project H2020-MSCA-ITN2014-642963. It is also supported by the Big Data in Atmospheric Physics (BINARY) project, funded by the Carl Zeiss Foundation under Grant No.: P2018-02-003.

REFERENCES

[1] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, ON, Canada, October 27-30*. 353–369.

[2] Ernest Artiaga and Toni Cortes. 2010. Using filesystem virtualization to avoid metadata bottlenecks. In *Design, Automation and Test in Europe (DATE), Dresden, Germany, March 8-12*. 562–567.

[3] Arindam Banerji, John Michael Tracey, and David L. Cohn. 1997. Protected Shared Libraries: A New Approach to Modularity and Sharing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC), Anaheim, California*. USA, 5.

[4] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, USA, November 12-15*. 117–128.

[5] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (ATC), Renton, WA, USA, July 10-12*. 121–134.

[6] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[7] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *USENIX Annual Technical Conference (ATC), July 15-17*. 65–80.

[8] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, June 6-10*. 87–98.

[9] Peter Braam. 2005. The Lustre Storage Architecture. *CoRR* abs/1903.01955 (2005). http://arxiv.org/abs/1903.01955

[10] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. 2020. Ad Hoc File Systems for High-Performance Computing. *Journal of Computer Science and Technology* 35, 1 (2020), 4–26. https://doi.org/10.1007/s11390-020-9801-1

[11] David Brumley and Dawn Xiaodong Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, San Diego, CA, USA*. 57–72.

[12] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, May 19-23*. 985–999.

[13] Rajkumar Buyya, Toni Cortes, and Hai Jin. 2001. Single System Image. *International Journal of High Performance Computing Applications (IJHPCA)* 15, 2 (2001), 124–135.

[14] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In *4th Annual Linux Showcase & Conference, Atlanta, Georgia, USA, October 10-14*.

[15] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), Portland, OR, USA, October 20-24*. 433–452.

[16] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems* 12, 4 (1994), 271–307.

[17] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proceedings of the VLDB Endowment (PVLDB)* 8, 5 (2015), 497–508.

[18] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Cambridge, MA, USA, October 14-18*. 191–203.

[19] Xianzhang Chen, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Ting Wu, Weiwen Jiang, Xiaoping Zeng, and Lin Wu. 2018. UMFS: An efficient user-space file system for non-volatile memory. *Journal of Systems Architecture* 89 (2018), 18–29. https://doi.org/10.1016/j.sysarc.2018.04.004

[20] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST).* 81–95.

[21] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. 2020. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *USENIX Annual Technical Conference (ATC), July 15-17.* 1–16.

[22] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA, USA, March 5-11.* 105–118.

[23] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, Montana, USA, October 11-14.* 133–146.

[24] Giuseppe Congiu, Sai Narasimhamurthy, Tim Süß, and André Brinkmann. 2016. Improving Collective I/O Performance Using Non-volatile Memory Devices. In *2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, September 12-16, 2016.* 120–129.

[25] Jonathan Corbet. 2017. The future of DAX. https://lwn.net/Articles/717953/.

[26] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016), 86.

[27] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Xi'an, China, April 8-12.* 435–448. https://doi.org/10.1145/3037697.3037704

[28] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Ontario, Canada, October 27-30.*

[29] Subramanya R. Dulloor, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Ninth Eurosys Conference (EuroSys), Amsterdam, The Netherlands, April 13-16.* 15:1–15:15.

[30] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI), London, UK, June 15-20.* 377–392. https://doi.org/10.1145/3385412.3386031

[31] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. 2009. Scalable massively parallel I/O to task-local files. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA.* https://doi.org/10.1145/1654059.1654077

[32] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[33] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland, March 16-20.* 775–788. https://doi.org/10.1145/3373376.3378472

[34] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference (ATC), Renton, WA, USA, July 10-12.* 489–504.

[35] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1998. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience* 28, 9 (1998), 901–928.

[36] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.

[37] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proceedings of the VLDB Endowment* 14, 5 (2021), 785–798.

[38] Intel. 2020. Intel®64 and IA-32 Architectures Optimization Reference Manual, Volume 3A: Part 1. https://software.intel.com/en-us/articles/intel-sdm.

[39] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, ON, Canada, October 27-30.* 494–508.

[40] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a True Direct-Access File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST), Oakland, CA, USA, February 12-15.* 241–256.

[41] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys), London, United Kingdom, April 18-21.* 13:1–13:16.

[42] Alfons Kemper and Donald Kossmann. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal* 4, 3 (1995), 519–566.

[43] Nicolas Krauter, Patrick Raaf, Peter Braam, Reza Salkhordeh, Sebastian Erdweg, and Andre Brinkmann. 2021. Persistent Software Transactional Memory in Haskell. *Proc. ACM Program. Lang.* 5, ICFP, Article 63 (Aug. 2021), 29 pages.

[44] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight Jr., and André DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8.* 721–732.

[45] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, October 28-31.* 460–477.

[46] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang. 2018. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19.* 1441–1454. https://doi.org/10.1145/3243734.3243748

[47] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, July 12-14.* 773–785.

[48] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland, March 16-20.* 789–806. https://doi.org/10.1145/3373376.3378456

[49] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *36th IEEE International Conference on Computer Design (ICCD), Orlando, FL, USA, October 7-10.* 413–422.

[50] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (ATC), Denver, CO, USA, June 22-24.* 71–85.

[51] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys), London, United Kingdom, April 18-21.* 12:1–12:16.

[52] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, October 6-8.* 1–16.

[53] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, Dan Feng, Tim Süß, and André Brinkmann. 2019. LPCC: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19.* 88:1–88:14. https://doi.org/10.1145/3295500.3356139

[54] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 2017. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, USA, November 12 - 17, 2017.*

[55] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual Event, November 4-6.* 137–154.

[56] Andy Rudoff. 2017. Persistent Memory Programming. *;login* 42, 2 (2017), 34–40.

[57] Reza Salkhordeh and André Brinkmann. 2019. Online Management of Hybrid DRAM-NVMM Memory for HPC. In *26th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, December 17-20, 2019.* 277–289.

[58] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 26 - March 1.*

[59] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP), December 12-15.* 170–185.

[60] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, May 22-26.* 1–17.

[61] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *;login:* 41, 1 (2016).

[62] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security, Santa Clara, CA, USA, August 14-16.* 1221–1238.

[63] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 27 - March 2.* 59–72.

[64] Pavel Vasek and Kanad Ghose. 1997. A comparison of two context allocation approaches for fast protected calls. In *Proceedings Fourth International Conference on High-Performance Computing.* 16–21.

[65] Marc-Andre Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. GekkoFS - A Temporary Distributed File System for HPC Applications. In *IEEE International Conference on Cluster Computing (CLUSTER), Belfast, UK, September 10-13.* 319–324.

[66] Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science & Engineering* 17, 2 (2015), 73–82.

[67] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with Code-centric memory Domains. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, June 14-18.* 469–480.

[68] Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: flexible file-system interfaces to storage-class memory. In *Ninth Eurosys Conference (EuroSys), Amsterdam, The Netherlands, April 13-16.* 14:1–14:14.

[69] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: light-weight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA, USA, March 5-11, 2011.* 91–104.

[70] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Salt Lake City, UT, USA, November 13-18.* 807–818. https://doi.org/10.1109/SC.2016.68

[71] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. 2017. MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, May 29 - June 2.* 1174–1183. https://doi.org/10.1109/IPDPS.2017.39

[72] Andrew Waterman, Yunsup Lee, Rimas Avizienis, Henry Cook, David A. Patterson, and Krste Asanovic. 2013. The RISC-V instruction set. In *IEEE Hot Chips 25 Symposium (HCS), Stanford University, CA, USA, August 25-27.*

[73] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, May 17-21.* 20–37.

[74] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon D. Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19.* 76:1–76:19.

[75] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing (STC), Alexandria, VA, USA, October 31.* 21–30.

[76] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, June 14-18.* 457–468.

[77] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, UK.* 623–637. https://doi.org/10.1145/3385412.3386000

[78] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Providence, RI, USA, April 13-17.* 427–439.

[79] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-25.* 323–338.

[80] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST), Boston, MA, February 25-28.* 221–234.

[81] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, USA, February 25-27.* 111–125.

[82] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, December 11-14.* 154–161.

[83] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. 2019. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Renton, WA, USA, July 8-9.*

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

The experiments are divided into Simurgh file system library and the modified gem5 simulator containing the ISA extensions for the proposed hardware instructions.

The repository contains the setups, configuration files, and scripts to run the benchmarks and to produce results presented in our submission. A readme file contains detailed steps to prepare the environment and run the scripts. Dependencies are either included in the repo or the links are provided to obtain them.

Simurgh is provided as pre-compiled binaries found at the top level of the repository.

To run the gem5 benchmarks, the source of the extended gem5 version is provided, together with ready to use kernel images containing our modifications.

All file system experiments are performed on CentOS 8.2 running Linux kernel `5.1.0`, unless stated otherwise. Some benchmarks are compiled on the fly by their corresponding scripts. The required `gcc` and `g++` version is `8.3.1`. Some benchmarks use python (version `3.6.8`) to be scheduled and `matplotlib` for their output.

To run benchmarks, persistent name spaces need to be created using the `ndctl` tool. The details and steps for creating and managing the namespaces are available in the repository. Our benchmarks use version `67` of `ndctl`.

Applications can use Simurgh by preloading the provided Simurgh file system library using `LD_PRELOAD`. The repository contains the required binaries and scripts for configuring and formatting the memory regions.

Our benchmarks use the following versions of the other file systems:

- `NOVA 5.1`
- `EXT4` in `DAX` mode, kernel `5.1.0`
- `PMFS` from master branch of `PMFS-new`, kernel `4.18.19`
- `SPLITFS`, from master branch, kernel `4.13` (as suggested)

The links and configuration scripts for setting up the running the environment of the above file systems is provided and explained in detail in the repository.

Applications. Our benchmarks use `git` version `2.28.0`. Due to some custom libc functions getting inlined, git has to be compiled using `no-inline` flags with the commands in the repository.

Further applications, that need to be installed:

- `tar` version `1.3`
- `filebench` version `1.5-alpha3`
- `YCSB`, as described in the repository of `SPLITFS`

System. We ran the benchmarks on a single socket server equipped with a 10 cores Xeon Gold 5212 processor running at 2.5GHz, 192 GB DRAM, and 746 GB Optane DC persistent memory across 6 DIMMs. The benchmarks require at least 64GB DRAM and 256GB NVMM. To clear caches, mount, or unmount file systems, provided scripts have to be run with root privilege.

Running Benchmarks. To run the benchmark scripts inside the repository, an argument specifying the target file system (`SIMURGH`, `NOVA`, `EXT4DAX`, `PMFS`, `SPLITFS`) has to be passed to the provided benchmark script. The repository will provide more details.

For some benchmarks, no single running script could be provided. In this case, the repository contains easy-to-follow steps.

Between different measurements, file system caches are cleared. In case of independent iterations, file systems are formatted between runs.

The microbenchmarks presented in the paper are executed after each other. For default, each benchmark is performed 20 times with different thread counts (up to 10). Iteration times and thread counts can be adjusted.

For `tar`, `git` and `recovery` benchmarks the full source code of Linux kernel is used. `tar` benchmark packs and unpacks it inside the file system. `git` benchmark copies it into a empty repository and measures `add`, `commit` and `reset` times.

Within the `recovery` benchmark, a full recovery procedure is performed. The Simurgh library includes a dedicated entry point for this. The recovery split into two parts: scanning and repairing the persistent data, and rebuilding the shared memory data structures. The time reported in the paper consists of the sum of those two parts. The repository will provide more details.

For Filebench, we used `varmail`, `webserver`, `webproxy`, and `fileserver` workloads. The default configuration for all workloads was used and only the directory for creating the files was modified. The workload configurations, that are used by the provided running script, are also contained in the repository.

To run YCSB, we used the same source code and workflow proposed by SplitFS, in order to have a fair comparison. The details of running the benchmark and obtaining the results can be found in the SplitFS repository. We then used `perf` tool to obtain the execution time breakdown.

Gem5. The benchmarks and the pipeline analysis were conducted using gem5 version `20.0.0.2`. The repository contains the configuration script used to boot a Gentoo RootFS on top of the provided kernel images. The provided kernel images were built to include the PTEditor module. A dummy syscall was added to the kernel which executes no code, to allow measuring the overhead of the Linux system call mechanism. The DerivO3CPU CPU type with caches and L2 caches enabled was specified to be used with the configuration script to boot gem5.

The repository contains the detailed steps to compile the modified gem5 and build the gem5 binary to run the benchmarks.

We ran the benchmarks on gem5 full system mode (FS-mode). FS-Mode simulates a complete environment for running an operating system including support for interrupts, exceptions, privilege levels, etc.

The benchmarks measures the cycle count required for the `jmpp` / `pret`, `SYSCALL_64` / `SYSRET_TO_64` instructions and a complete Linux system call from user space. All measurements are divided into related execution blocks, to show the cycle count required for each relevant part of an instruction. The `m5_reset_stats` and `m5_dump_stats` pseudo instructions allow measuring the cycle count. To achieve minimal measurement overhead, the parameters required to be passed to these instructions in the `rdi` and `rsi` registers were hardcoded. That way, a single macro-op is enough to reset or dump the statistics. Each measurement was performed 100 times to account for caching or speculative execution effects.

The benchmarks need to be compiled using the m5 utilities to be able to run using the gem5 simulator. The folder repository contains the tests to measure the overhead of the new instructions and also an empty system call on the modified gem5. The benchmark performs 100 iterations of the `jmpp` and `pret` instructions. The steps to compile the benchmarks and run them inside the gem5 shell are explained in the repository. Another provided benchmark measures the overhead of an empty syscall.

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.5163624
Artifact name: Simurgh\_SC21\_AD
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel(R) Xeon(R) Gold 5215 CPU @ 2.50GHz, 192 GB DRAM, 746 GB Optane DC persistent memory across 6 DIMMs

Operating systems and versions: CentOS 8.2 Linux kernel 5.1.0

Compilers and versions: gcc g++ 8.3.1, python 3.6.8

Applications and versions: NDCTL, NOVA 5.1, PMFS, SplitFS, git 2.28.0, Filebench 1.5-alpha3, Tar 1.3, YCSB

Input datasets and versions: linux-5.6.14 source code