SplitFS: Reducing Software Overhead in File Systems for Persistent Memory

Rohan Kadekodi[1] Se Kwon Lee[1] Sanidhya Kashyap[4]
Taesoo Kim[4] Aasheesh Kolli[2,3] Vijay Chidambaram[1,2]

[1]*University of Texas at Austin* [2]*VMware Research* [3]*Pennsylvania State University*
[4]*Georgia Institute of Technology*

1 Introduction

One of the big challenges in building software for Persistent Memory (PM) is software overhead: given the low latencies of PMs, any inefficiencies in the software show up prominently. This is true of PM file systems as well: every file-system operation must be efficient when performed on PM; otherwise, the low latency benefits of PM are squandered.

File systems such as PMFS [2] and NOVA [3] are implemented completely inside the kernel. This approach suffers from two drawbacks: (1) overheads caused due to kernel crossings are left unaddressed and (2) These file systems suffer from high software overhead in the critical path of applications for providing strong consistency guarantees. Aerie [4] and Strata [5] both seek to reduce overhead by not involving the kernel for most file-system operations. The problem with this approach is that building and maintaining a POSIX compliant file system is a difficult task; POSIX has a number of corner cases that are hard to get right. In this work, we try to answer the question, is it possible to minimize file system software overhead without creating an entirely new file system from scratch?

We present SPLITFS [1], a new file system for persistent memory. SPLITFS employs a novel split of responsibilites between user-space and kernel. All the data operations such as file reads and writes are served from user space. All metadata operations (such as file open, close, rename, etc.) are served from the mature ext4-DAX. SPLITFS lowers software overhead by up to 17× as compared to ext4-DAX and improves application performance by up-to 2× as compared to NOVA. Different applications relying on different consistency and durability guarantees can run concurrently on SPLITFS without interfering with each other. SPLITFS is available online at https://github.com/utsaslab/splitfs.

2 Design

SPLITFS uses three techniques in order to achieve both low software overhead and provide strong guarantees, while re-using the maturity and active development of the ext4-DAX file system.

2.1 Split Architecture

SPLITFS employs a novel split architecture. SPLITFS consists of a user-space component, U-Split, and a kernel component, K-Split. SPLITFS translates POSIX data operations (read and in-place write system calls) into user-space loads and stores on a memory-mapped file. As a result, these file-system operations enjoy low overhead and high performance. POSIX metadata operations (like file close, open, renames, etc.) are handled by SPLITFS kernel component.

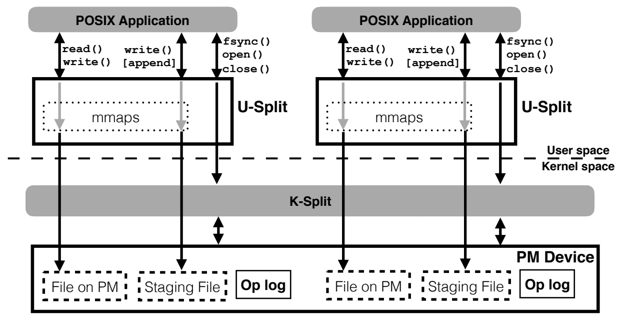

Figure 1: **SPLITFS Overview**. The figure provides an overview of how SPLITFS works. Read and write operations are transformed into loads and stores on the memory-mapped file. Append operations are staged in a staging file and *relinked* on `fsync()`. Other metadata POSIX calls like `open()`, `close()`, etc. are passed through to the in-kernel PM file system.

One of the unique aspects of SPLITFS is that it uses an existing in-kernel file system, Linux ext4 [6] (with DAX mode enabled), as its kernel component. All metadata operations are routed to ext4, relieving SPLITFS of the complexity involved in accurately implementing POSIX. Another unique advantage of the split architecture is that SPLITFS can allow applications running concurrently to obtain different consistency and durability guarantees from the file system without interfering with each other. This is possible because each application is linked to a different instance of U-Split. Tailoring the guarantees for individual applications can significantly improve application performance. To cater to different applications, SPLITFS provides three different modes:

1. POSIX mode: all metadata operations are atomic. This is similar to ext4-DAX.
2. Sync mode: all operations are synchronous, in addition to metadata operations being atomic. This is similar to PMFS.
3. Strict mode: all operations (including data operations) are atomic and synchronous. This is similar to Strata and NOVA with copy-on-write enabled.

2.2 Relink

SPLITFS provides strong guarantees to applications in the sync and strict mode with the help of logical logging of operations and a form of copy-on-write for file writes. Writes are not performed in-place, they are redirected to a temporary file. The temporary files are large (160 MB) and memory-mapped using 2 MB huge pages. The temporary files are also pre-faulted so that there is little overhead for

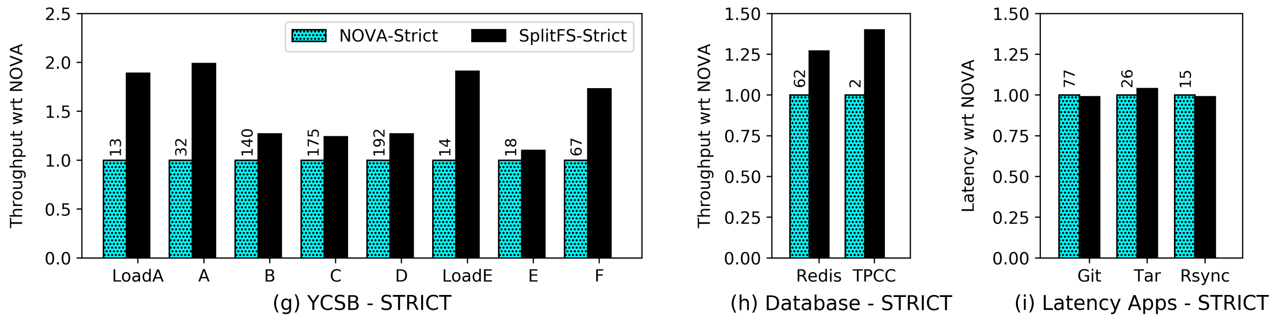

Figure 2: **Real application performance**. This figure shows the performance of both data intensive applications (YCSB, Redis, and TPCC) and metadata instensive utilities (git, tar, and rsync) with NOVA and SPLITFS, providing data atomicity, metadata atomicity and synchronous operations. For throughput workloads, higher is better. For latency workloads, lower is better.

reading a memory-mapped temporary file in the critical path. SPLITFS introduces a new system call called relink that atomically moves an extent of data from one file to another; relink is used to move the data in temporary files back to the original file. Relink logically moves extents without physically copying data: it is a pure metadata operation. Relink is done in a failure-atomic fashion by using the ext4 journal. We found relink to be a versatile and useful primitive: SPLITFS uses it both for file appends and atomic file updates.

2.3 Logging

SPLITFS logs all operations in sync and strict modes to ensure they are atomic, so an efficient logging mechanism is necessary for good performance. To reduce logging overheads, we reduce both the amount of data written to the log and also the number of expensive sfence instructions incurred while updating the logs. SPLITFS uses logical redo logging. In the common case, each operation results in a single cache line (64 bytes) written to the log, followed by a single sfence instruction. The 64 byte log entry contains a 4-byte checksum, used to identify invalid or torn entries. The log entries do not contain data but simply point to a location in a temporary file that contains data for the operation. Each U-Split instance has its own log. Multiple threads coordinate access to the log via atomic compare-and-swap operations on the tail (maintained in DRAM). The tail is not persisted; upon a crash, valid entries are identified using checksums, and all valid entries are replayed. The log entries are idempotent, so replaying them multiple times does not cause incorrect behavior.

3 Evaluation

We evaluate SPLITFS on Intel Optane DC Persistent Memory, on a dual socket server with 750 GB of PM and 375 GB of DRAM. We compare the performance of SPLITFS with NOVA, PMFS and Strata. We report the average of 3 runs for all the experiments, with a variance of <1% across runs. We present the performance of SPLITFS in the strict mode, and compare it with NOVA providing similar guarantees (data atomicity, metadata atomicity and synchronous operations).

3.1 Data Intensive Workloads

Fig 2 shows the performance of SPLITFS and NOVA on data-intensive workloads. SPLITFS outperforms NOVA on all data-intensive workloads. The write-heavy workloads (YCSB Load A, Run A, Run F) show the biggest boost in performance (up-to 2×), because of the relink technique and the split architecture. Read-heavy workloads on the other hand do not show much improvement in performance.

3.2 Metadata Intensive Workloads

Fig 2 compares the performance of SPLITFS with NOVA on metadata-heavy workloads like git, tar and rsync. The metadata-heavy workloads do not present many opportunities for SPLITFS to service system calls in userspace and in turn slow metadata operations down due to the additional bookkeeping performed by SPLITFS. The maximum overhead experienced by SPLITFS is 13%.

References

[1] Kadekodi et. al. SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019. Link to paper: https://www.cs.utexas.edu/~vijay/papers/sosp19-splitfs.pdf.

[2] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[3] Jian Xu and Steven Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

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

[5] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 460–477, 2017.

[6] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.