Libpubl: Exploiting Persistent User Buffers as Logs for Write Atomicity

Jaewon Jung
DMC Engineering
Sungkyunkwan University
Mobile Div., Samsung Electronics
jaewon8868@skku.edu

Jungsik Choi
Computer Science and Eng.
HPC Research Center
Sungkyunkwan University
chjs@skku.edu

Hwansoo Han
Computer Science and Eng.
Human-AI Interaction
Sungkyunkwan University
hhan@skku.edu

ABSTRACT

With recent advancement in NVM technologies, non-volatile main memory (NVMM) has been highlighted as promising systems for large scale data centers. By utilizing memory-mapped IO, researchers have attempted to provide fast IOs without kernel mode switches and complex kernel IO layers. Even though they improve the performance by taking advantage of NVMM characteristics, memory-mapped IOs still require additional user-level logging techniques to guarantee write atomicity and file system integrity. We propose a user-level library file system, Libpubl, which is designed to minimize the write amplification overhead of traditional logging by exploiting persistent user buffers as logs. Libpubl ensures atomic updates of data and improves the performance and the scalability. Compared to the state-of-the-art NVM file systems, Libpubl increases the performance by 50~120% in Fio benchmark.

CCS CONCEPTS

• **Software and its engineering → File systems management**; • **Information systems → Storage class memory**; • **Hardware → Memory and dense storage**.

KEYWORDS

Non-Volatile Memory, File System, Direct Access, Write Atomicity, Logging

ACM Reference Format:
Jaewon Jung, Jungsik Choi, and Hwansoo Han. 2021. Libpubl: Exploiting Persistent User Buffers as Logs for Write Atomicity. In *13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21), July 27–28, 2021, Virtual.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465332.3470874

HotStorage '21, July 27–28, 2021, Virtual

ACM ISBN 978-1-4503-8550-3/21/07…$15.00
https://doi.org/10.1145/3465332.3470874

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies such as PCM [21, 29], STTM-RAM [19, 26], NVDIMMs [14], and Intel Optane [37] provide comparable latency to DRAM memory, while providing density comparable to flash memory storage. With the advantages of high scalability, low power consumption, and competitive performance, these memories are expected to complement or even replace DRAM in future systems [2, 20, 21]. Computer systems equipped with non-volatile main memory (NVMM) are even able to store permanent data such as files in the main memory. By adopting NVMM in file systems, we can further reduce the file IO latency, which often involve a significant overhead to access storage devices in traditional systems [1, 4, 7].

Recently, numerous NVM-aware file system researches have attempted to exploit the performance of NVM [5, 8, 10, 33, 36]. Their approaches commonly include bypassing the traditional block layer and the page cache layer to reduce unnecessary software overheads in NVM file systems. Furthermore, researchers have proposed user-level file systems based on NVM to remove the whole IO stack overhead within the OS kernel [6, 15, 18, 31, 34]. Overcoming the limitations of existing file systems, which are mainly designed for slow block devices, these NVM-aware file systems have achieved notable performance improvements.

The NVM file systems that support write atomicity have their own recovery mechanisms in case of system failure. Main techniques in recovery mechanisms are to use copy-on-write (CoW) [8, 23, 30, 35] and logging [6, 10, 12, 24] to guarantee write atomicity. However, CoW will cause a severe write amplification, when the address of the data to update is not page-aligned or the size of the data is not multiple of the page size. Logging also incurs the overhead of writing all updates twice — once to the log and once again to the file. It is crucial to reduce the write amplification in non-volatile main memory, as memory writes take up a large portion of the overhead in NVM file systems. Many sophisticated

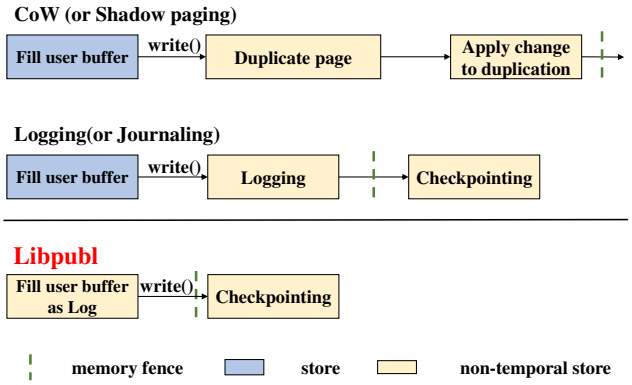

Figure 1: Libpubl— Comparison to CoW and Logging

techniques have been investigated to alleviate the logging overhead in NVMM systems [6, 16, 17]. Nevertheless, the overhead to ensure write atomicity is still quite large [25, 27, 28]. Particularly interesting aspect in NVMM systems is that user buffers can be non-volatile. Since user buffers can be allocated in non-volatile memory regions of NVMM systems, we can innovate write operations by taking advantage of the non-volatility of user buffers.

We propose Libpubl which exploits persistent user buffers as logs to provide a novel recovery mechanism for write transactions. When applications write data to files, general steps consist of filling user buffers with data and invoking write system calls. Assuming we use user buffers in non-volatile memory, it has almost the same effect as redo logging. The only difference between them is that permanent data is written in user buffers, not in separate log data. Libpubl allocates user buffers in non-volatile memory regions and manages them as logs in a sophisticate manner. While previous logging systems intrinsically suffer from the logging overhead, Libpubl can perform logging operations without additional writes to log data.

To minimize software overhead in the OS kernel, we implemented Libpubl as a library file system, performing file IOs as user level operations. Libpubl intercepts file IO requests such as read/write calls and performs memory-mapped IOs along with logging transparently. Since Libpubl runs on any file systems that support DAX-mmap, it is fairly easy to adopt. In addition, Libpubl is implemented to use fine-grained range locks within a file for read operations and log checkpointing operations. Thus, Libpubl shows a scalable IO performance, even when multiple threads access the same single file for IO operations. Compared to the state-of-the-art NVM file systems, Libpubl performs better in Fio [3] benchmark than NOVA [36] and Libnvmmio [6] by 120% and 50%, respectively.

2 LIBPUBL DESIGN

File writes, in essence, are copying data from volatile user buffers to non-volatile storage. While crossing the volatile and non-volatile boundaries, the recovery mechanism ensures safety in case of system failure. With the advent of non-volatile memory, user buffers in the main memory can be non-volatile in NVMM systems. Libpubl proposes a new approach, which takes advantage of non-volatile characteristics in user buffers and copies data already in non-volatile region to the NVMM file system.

2.1 Persistent User Buffers as Logs

Applications fill their user buffers with new data to update and invoke `write()` for the OS kernel to transfer the data in user buffers to the target files in storage. During the data transfer, we may encounter system failures. To maintain the integrity of file systems, file systems typically provide recovery mechanisms. Figure 1 compares the recovery mechanism in Libpubl with two traditional recovery mechanisms — CoW and logging. CoW (or shadow paging) first duplicates the original pages before updates. When the portion of data to update in a page is large, it duplicates too many unnecessary data. Most of them will be updated to a new content soon. After the duplication, CoW copies the data in user buffers to the duplicated page and checkpoints the update by replacing the original page with the updated page. Logging (or journaling) first records the update as a separate log data. Then, it copies again the logged data to the original page of the target file. By writing twice on two permanent locations, we can endure failures while updating original data in the file systems.

Libpubl simplifies the overall process for writing and checkpointing, while these existing techniques inherently amplify the number of writes. Since Libpubl allocates user buffers in non-volatile memory regions, the data in user buffers, which are passed as one of the `write()` parameters, are already logged in permanent states. Inside the `write()` invocation, Libpubl just needs to ensure that all the data is written to the memory system with memory barrier instructions. Once the user buffers are recorded as log data, Libpubl can copy the log data to the original pages of the target file even in the event of a system failure. Since Libpubl exploits user buffers as logs, the contents of the buffers should be preserved as logs. Typical applications have think times between IOs. Reusing the user buffer for the next IO may not be a problem, when checkpointing can be finished during the think time [1]. However, applications with many instantaneous writes may try to modify the contents of the buffers

[1]The time duration between two consecutive IOs to represent the data processing time between the two — Fio also provides *thinktime* option.

```
1  /* allocate a persistent user buffer */
2  PUBL *buf = publ_malloc(len);
3
4  for (off_t i=0; i<filesize; i+=len) {
5      ...
6
7      /* fill the user buffer */
8      publ_puts(buf, data, len);
9
10     /* invoke the write */
11     write(fd, buf, len);
12 }
13
14 /* release the user buffer */
15 publ_free(buf);
```

Figure 2: Example Usage of Libpubl

before their checkpointing processes are done. To accommodate such cases, Libpubl allocates user buffers in PUBL-type objects as shown in Figure 2 — this is called *PUBL object*. A PUBL object contains multiple preliminary buffers inside and they are internally assigned as buffers for an application in a circular fashion. Thus, an immediate use of a buffer before checkpoint completion will make the PUBL object switch to a new preliminary buffer.

2.2 Libpubl Interface

While minimizing changes to the source code, Libpubl provides existing programs with direct file IOs to NVMM and guarantees write atomicity. It exposes a small set of user interfaces to allocate/deallocate persistent user buffers and get/put contents from/into the persistent user buffers.

- `publ_malloc()`, `publ_free()`
- `publ_gets()`, `publ_puts()`

Libpubl requires no additional interfaces except the above four interfaces. The current version of Libpubl supports ten file IO related POSIX calls — `open()`, `close()`, `read()`, `write()`, `pread()`, `pwrite()`, `lseek()`, `ftruncate()`, `fsync()`, and `fdatasync()`. Libpubl relies on `LD_PRELOAD` environment to intercept the above POSIX calls. Depending on types of file IO operations, they are either served from Libpubl, or routed to the underlying kernel file system — metadata related file IO operations are handled in the kernel.

Figure 2 describes how application programmers can use Libpubl in file IOs. As shown in line 2, `publ_malloc()` allocates a persistent user buffer and returns a reference to PUBL object representing the buffer. In line 8, `publ_puts()` fills the persistent user buffer with the data to write in the file. At this time, it stores the data permanently in the buffer by using non-temporal store instructions. In line 11, `write()` is invoked with the persistent user buffer as one of its parameters. Libpubl intercepts `write()` and issues memory barrier instructions to wait until the data in the buffer is permanently written to the non-volatile memory. On completion of memory barriers, Libpubl records the buffer as a committed log and returns to the program. Background threads handle the remaining checkpoint operations. We will discuss the background checkpointing in detail later in Section 2.3. Finally, `publ_free()` releases the user buffer in line 15.

2.3 Background Checkpointing

To accelerate the IO performance, Libpubl performs checkpointing on background threads away from the critical path of program execution. In Libpubl, `write()` returns immediately after the user buffer is committed via memory fences. After a while, background threads carry out the delayed checkpointing and clear the commit mark for the buffer indicating the completion of checkpointing. While providing the background checkpointing technique, Libpubl also provides an appropriate resolution for applications to access user buffers still marked as commit state logs. When applications access the user buffers in commit state, which will be checkpointed later, Libpubl changes their checkpointing mode to foreground for the buffers in such cases — we call this *early checkpointing*. Buffer accesses are detected easily, as the buffers in Libpubl are designed to access only through the given interfaces: `publ_gets()` and `publ_puts()`.

There are three contexts where early checkpointing is required. First, an application tries to modify the content of the buffer in commit state — this means the buffer is not checkpointed yet. Second, an application calls `write()` with the buffer in commit state — this happens when the same buffer content is written to multiple locations of files. Third, an application tries to read the content of a file which overlaps the ranges of buffers in commit state. Early checkpointing adversely affects the performance, as the intended buffer access should wait until the early checkpointing is completed. When applications try to access buffers that are currently in checkpointing process in the background threads, they also need to wait until the background checkpointing is done.

To reduce the wait times due to early checkpointing and background checkpointing, a PUBL object contains multiple preliminary buffers. Libpubl can configure the number of preliminary buffers at its initialization. We used 8 buffers in our experiments, which was sufficient even for write-intensive workloads. When an application attempts to fill a user buffer which is still in commit state, Libpubl internally switches to another preliminary buffer instead. After checkpointing for that buffer is completed later, Libpubl put the buffer back to the preliminary buffers for later reuse.

2.4 Fine-Grained Range Lock

Libpubl provides scalability through a fine-grained range lock by using radix tree in multi-threaded environments. Libpubl protects files with reader-writer locks based on the address ranges of files to access in read and checkpoint operations. To avoid too many lock acquisitions for a large sized buffer access, Libpubl provides various sized locks ranging from 4KB to 2MB, doubling the lock ranges. Multi-sized fine grained range lock allows file IOs on non-overlapping pages to scale well. In Libpubl, write operations are not required to obtain locks for the ranges of files to access. Since they only turn the user buffers into commit states and store the log information in the radix tree, the contents in the buffers are not available for read operations yet. Later, writer locks are required when Libpubl checkpoints the user buffers. Early checkpointing may cause the performance degradation, but IOs on random addresses or sequential addresses rarely require early checkpointing.

2.5 Recovery

Libpubl manages non-volatile memory pools, where PUBL objects are allocated and passed as non-volatile user buffers. Libpubl maintains one memory pool per process, the top area of which is dedicated to hold the list of the metadata of all Libpubl IO logs for the process. By using these metadata, Libpubl can access all PUBL objects, which are actually IO logs in commit states. Since Libpubl checks whether the recovery operation is needed at every library initialization time, the application, which uses Libpubl for the first time after crash, is responsible for the log recovery from PUBL objects. When Libpubl performs the recovery process, all the memory pools are checked to look up the logs in commit states. The list of the metadata at the top of each memory pool are used to access the logs via their relative offsets within the memory pool. For the logs in commit states, Libpubl proceeds to checkpoint them and finalize the IOs.

3 EVALUATION

We implemented Libpubl from scratch. For allocator and hash table, we extended our code from open source software, jemalloc [11] and uthash [13]. Our prototype of Libpubl has a total 2,936 LOC in C code. To copy data to NVM with non-temporal stores, Libpubl employs the PMDK library [9].

3.1 Experimental Setup

We experimented two file systems such as Ext4-DAX and NOVA, and two library file systems such as Libnvmmio and Libpubl. In our experiments, Libpubl and Libnvmmio used Ext4-DAX [32] on Linux kernel 5.1 as an underlying file system. In terms of write atomicity, Ext4-DAX logs only metadata, it does not guarantee write atomicity. On the other hands NOVA [36] used CoW and Libnvmmio [6] and Libpubl uses logging to guarantees write atomicity in each operation. To evaluate performance, we used two types of NVM: NVDIMM-N [7] and Intel Optane DC Persistent Memory Module [37]. The NVDIMM-N system has 20 cores, 96GB DRAM and 96GB NVDIMM-N. The Optane system has 72 cores (18 cores on each of four CPUs), 768GB DRAM and 3024GB Optanes. The Optane system is set to *App Direct* mode with interleaving enabled. The Optane system is equipped with the second generation Optane memory. Since it is the latest generation of Optane, only Ubuntu 20.10 or higher version supports its driver. Ubuntu 20.10, which is required for the Optane, is packaged with the Linux kernel 5.8. Since NOVA supports up to the Linux kernel 5.1, our experiments on the Optane system were performed on Ext4-DAX, Libnvmmio, and Libpubl, but not on NOVA.

Table 1: Effects of numbers of preliminary buffers

# Preliminary buffers	1	2	4	8	16
early checkpoint (%)	0.02	0.01	0.00	0.00	0.00
background checkpoint (%)	99.94	6.54	0.04	0.02	0.01
TOTAL wait (%)	99.96	6.55	0.04	0.02	0.01
Throughput(MB/s)*	1,872	3,337	3,348	3,399	3,394

* Fio performance with sequential write to NVDIMM-N

Table 2: Effects of logging and checkpointing

	logging		checkpointing (Libpubl)	
	no	yes	foreground-only	background
Throughput(MB/s)*	2,755	1,749	2,754	3,363

* Fio performance with sequential write to NVDIMM-N

3.2 Effects of Libpubl configurations

Number of preliminary buffers. If applications access the buffers in the logs, they should wait for completion of either early checkpointing or background checkpointing. To improve the performance, we need to minimize the waiting situations. To this end, PUBL object employs multiple preliminary buffers. Table 1 shows the percentages of waits due to early checkpointing and background checkpointing among all PUBL object accesses. It also shows the corresponding IO performance according to the number of preliminary buffers configured in Libpubl. The performance results was obtained by performing sequential write with Fio on the NVDIMM-N system. With four or more preliminary buffers, application IOs scarcely wait for checkpoint completion. According to our experiment, the IO performance saturates around 8 preliminary buffers. Thus, we allocate 8 preliminary buffers in PUBL objects.

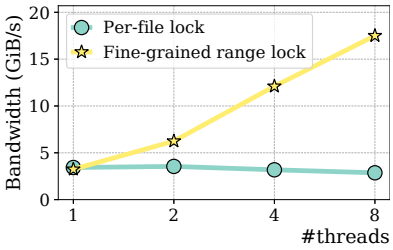

Figure 3: Effects of fine grained range lock

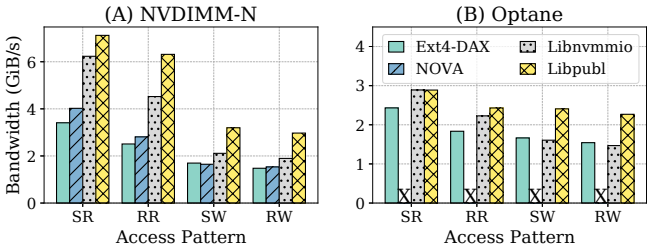

Figure 4: Fio benchmark for various access patterns (with `refill_buffers`[2])

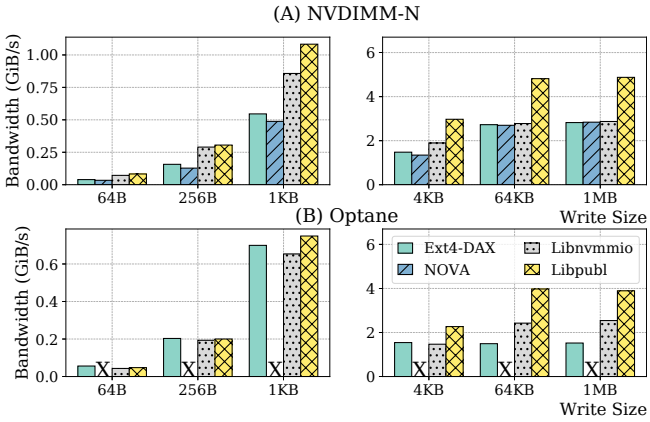

Figure 5: Fio benchmark for various write sizes (with `refill_buffers`[2])

Background checkpointing. Table 2 shows the performance according to the various types of logging. The performance results are obtained by performing sequential write with Fio on the NVDIMM-N system. Memory mapped IO does not guarantee write atomicity without logging. If applications use a logging method to guarantee write atomicity, it should write twice in non-volatile area. This results in the performance reduction as shown in *logging* section of Table 2. Libpubl checkpoints logs by using non-volatile user buffers and the performance results are shown in *checkpointing* section of Table 2. Even if only foreground checkpointing is used, Libpubl shows the same performance as the one without logging. Libpubl can further improve the performance by using background checkpointing.

Fine-grained range lock. Figure 3 shows the performance comparison between per-file lock and fine-grained range lock. The performance results are obtained from sequential write with Fio on the NVDIMM-N system. When per-file locks are used in a single threaded run, the overhead of radix tree search can be eliminated. Thus, they result in better performance than fine-grained range locks in a single thread. On the other hand, the performance on multi-threaded runs shows that fine-grained range locks are far better than per-file locks. Moreover, only the fine-grained range locks show a scalable performance on multi-threaded runs.

3.3 Microbenchmark

Throughput. We measured bandwidth performance using Fio [3]. It iteratively accesses for 15 seconds in 4KB units to 1GB file. The two graphs in Figure 4 show experimental results for NVDIMM-N and Optane, respectively. Four types of file access patterns are used for our experiment: sequential read (SR), random read (RR), sequential write (SW), and random write (RW). As shown in Figure 4, Libpubl achieves the highest throughput for all accesses. It shows 1.14~2.08× performance on the NVDIMM-N system and 1~1.54× performance on the Optane system. Without `refill_buffers` option, Fio fills the data in a buffer once and reuses it repeatedly. Since the behaviors of typical common workloads are often different, we enable `refill_buffers` option. We changed about 10 lines of codes in Fio. A PUBL object, which is a persistent user buffer allocated with `publ_malloc()`, is used for `write()`. When filling the buffer with random values, Fio uses `publ_puts()` to make the data persistent in the buffer. Since data is already persistent after `publ_puts()`, `write()` in Fio logs only the metadata additionally. The log data are later checkpointed asynchronously on background threads. Libpubl shows the best performance. It only copies the data of the memory-mapped file in `read()` and exploits user buffers as log in `write()`. On the other hand, NOVA is slower than Libpubl, as it requires frequent kernel mode switches and goes through the kernel IO stack in both `read()` and `write()`. Libnvmmio is also slower than Libpubl. It needs to search the radix tree to read the updated data in `read()` and needs one more additional memory copy for logging than Libpubl in `write()`.

Figure 5 shows the performance of random writes with various write sizes in Fio. Libpubl shows the best performance over all sizes. Nova, which uses CoW to guarantee write atomicity, suffers from write amplification for IOs with smaller than the page size. In addition, the smaller sized IO has the greater overhead due to frequent system calls.

[2]Fio fills random values in the buffer before every `write()`.

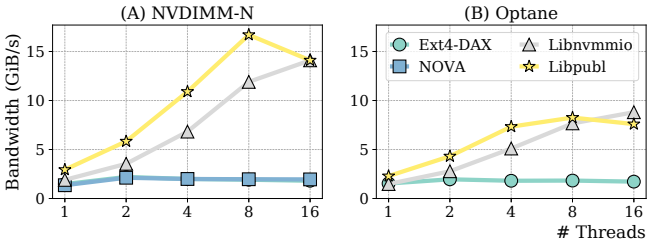

Figure 6: Fio benchmark for multi-threaded IOs (with `refill_buffers`[2])

Thus, NOVA, the kernel file system, becomes slower than Libnvmmio and Libpubl for the IOs on smaller than page size data. By providing a persistent buffer as a log to the application, Libpubl reduces write amplification further than Libnvmmio, which uses logging. For these reasons, Libpubl shows the best performance. As for the IOs with larger size than the page size, memory copying dominates as the access size grows [16]. As the IO size increases, Libnvmmio and NOVA perform similarly. Meanwhile, Libpubl, which has fewer memory copies, performs faster than other file systems.

Scalability. Figure 6 shows the performance of Fio in random write for a shared file in a multi-thread environment. Libpubl begins checkpointing on background threads immediately, whenever logs are available. Libpubl needs a background thread for each IO thread, if Libpubl opens a file with write permission. For this reason, the number of cores in the NVDIMM-N system is insufficient in 16 thread experiments and the performance degrades slightly. Likewise, when experimenting in 16 threads for the Optane system, some threads may run on the remote node in our NUMA configuration. Thus, the performance again degrades slightly. Applications with write-only IOs as in Fio, which requires a large number of cores in turn, are uncommon. Libpubl will scale well in an environment with enough cores for writes. Compare to other file systems, the performance of up to 8 threads, where a sufficient number of cores are provided, shows 1.5~8.7× performance on the NVDIMM-N system and 1.49~4.5× performance on the Optane system. While NOVA uses a single lock for an entire file, Libnvmmio uses per-block locks. Offset-based range locks in Libpubl are similar to per-block locks in Libnvmmio. However, Libpubl requires fewer number of writes than Libnvmmio, as Libpubl does not have separate logging phase. This makes Libpubl faster than Libnvmmio.

4 RELATED WORKS

State-of-the-art NVM-aware file systems have improved file IO performance by using the characteristics of persistent memory. They presented novel techniques to ensure write atomicity in persistent memory systems. Libpubl was designed based on those studies but has the distinct feature of using persistent user buffer as logs to reduce write amplification.

SplitFS [15] writes updated data to staging files rather than the original file. When the application calls `fsync()`, SplitFS performs `relink()` to link the pages in the staging file to the original file. Similarly, SubZero [16] updates files using `patch()` system call. The `patch()` makes user buffers part of the file rather than being copied into it. Both techniques allow file updates without additional memory copies. However, in order to perform the zero-copy update, the entire file block must be updated even when only a few bytes are updated.

NVWAL [17] uses byte-granularity differential logging to reduce write amplification. Libnvmmio [6] uses hybrid logging that dynamically changes the redo/undo logging policy according to the file access type. It can achieve the best performance of both logging policies. However, both NVWAL and Libnvmmio require writing twice for logging.

UBJ [22] allocates the buffer cache in non-volatile memory. The buffer cache, which resides in the OS kernel as page cache for files, replaces the role of file logs for kernel-level file systems. They still need to copy the data in user buffer to the buffer cache. On the other hand, Libpubl allocates user buffers in non-volatile memory and reduces the copy overhead further.

5 CONCLUSION

We present Libpubl, a library file system that exploits a persistent user buffer as a log to provides low-latency, scalable IOs while guaranteeing write atomicity. Libpubl performs major IO operations at user level without system calls. Libpubl manages persistent user buffers to reduce write amplification. Libpubl immediately returns from write() and checkpoints on background threads to hide checkpointing time. In addition, Libpubl provides scalable IOs with fine-grained range locks on radix tree. In our evaluation, we show that Libpubl outperforms state-of-the-art NVM file systems such as NOVA and Libnvmmio by 50~120% in Fio benchmark.

ACKNOWLEDGMENTS

This research was supported by the National Research Foundation in Korea under PF Class Heterogeneous High Performance Computer Development (NRF-2016M3C4A7952587); and Samsung Electronics.

REFERENCES

[1] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. 2015. DCS: A Fast and Scalable Device-Centric Server Architecture. In *Int'l Symposium on Microarchitecture (MICRO-48).*

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD '15)*.

[3] Jens Axboe. 2005. Flexible I/O tester. https://https://github.com/axboe/fio/

[4] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

[5] Jungsik Choi, Joonwook Ahn, Jiwon Kim, Sungtae Ryu, and Hwansoo Han. 2016. In-memory file system with efficient swap support for mobile smart devices. *IEEE Trans. Consum. Electron.* 62, 3 (2016), 275–282.

[6] 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 (USENIX ATC '20)*.

[7] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Symp. on Operating Systems Principles (SOSP '09)*.

[9] Intel Corporation. 2014. PMDK, Persistent Memory Development Kit. https://pmem.io/pmdk/

[10] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *European Conf. on Computer Systems (EuroSys '14)*.

[11] Jason Evans. 2005. jemalloc, memory allocator. http://jemalloc.net/

[12] R. Hagmann. 1987. Reimplementing the Cedar File System Using Logging and Group Commit. In *Symp. on Operating Systems Principles (SOSP '87)*.

[13] Troy D. Hanson. 2004. uthash - a hash table for C structures. https://troydhanson.github.io/uthash/

[14] Intel. 2015. NVDIMM Namespace Specification. https://pmem.io/documents/NVDIMM_Namespace_Spec.pdf

[15] 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 *Symp. on Operating Systems Principles (SOSP '19)*.

[16] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: Zero-Copy IO for Persistent Main Memory File Systems. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*.

[17] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[18] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Symp. on Operating Systems Principles (SOSP '17)*.

[19] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS '13)*.

[20] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable Dram Alternative. In *Int'l Symp. on Computer Architecture (ISCA '09)*.

[21] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010), 131–143.

[22] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *USENIX Conf. on File and Storage Technologies (FAST '13)*.

[23] C. Mohan. 1999. Repeating History Beyond ARIES. In *Int'l Conf. on Very Large Data Bases (VLDB '99)*.

[24] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *USENIX Annual Technical Conference (USENIX ATC '17)*.

[25] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-Atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *European Conf. on Computer Systems (EuroSys '13)*.

[26] N. Perrissin, S. Lequeux, N. Strelkov, L. Vila, L. Buda-Prejbeanu, S. Auffret, R. C. Sousa, I. L. Prejbeanu, and B. Dieny. 2018. Spin transfer torque magnetic random-access memory: Towards sub-10 nm devices. In *Int'l Conf. on IC Design Technology (ICICDT '18)*.

[27] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application Crash Consistency and Performance with CCFS. In *USENIX Conf. on File and Storage Technologies (FAST '17)*.

[28] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI '14)*.

[29] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-Change Random Access Memory: A Scalable Technology. *IBM J. Res. Dev.* 52, 4 (July 2008), 465–479.

[30] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3 (Aug. 2013), 9:1–9:32.

[31] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *European Conf. on Computer Systems (EuroSys '14)*.

[32] Matthew Wilcox. 2014. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/

[33] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11)*.

[34] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[35] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *USENIX Conf. on File and Storage Technologies (FAST '16)*.

[36] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Symp. on Operating Systems Principles (SOSP '17)*.

[37] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX Conf. on File and Storage Technologies (FAST '20)*.