

# NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems

Rémi Dulong\*, Rafael Pires<sup>‡</sup>, Andreia Correia\*, Valerio Schiavoni\*, Pedro Ramalhete<sup>§</sup>, Pascal Felber\*, Gaël Thomas<sup>†</sup>

\*Université de Neuchâtel, Switzerland, first.last@unine.ch

<sup>‡</sup>Swiss Federal Institute of Technology in Lausanne, Switzerland, rafael.pires@epfl.ch

<sup>§</sup>Cisco Systems, pramalhete@gmail.com

<sup>†</sup>Telecom SudParis/Insitut Polytechnique de Paris, gael.thomas@telecom-sudparis.eu

**Abstract**—This paper introduces NVCACHE, an approach that uses a non-volatile main memory (NVMM) as a write cache to improve the write performance of legacy applications. We compare NVCACHE against file systems tailored for NVMM (Ext4-DAX and NOVA) and with I/O-heavy applications (SQLite, RocksDB). Our evaluation shows that NVCACHE reaches the performance level of the existing state-of-the-art systems for NVMM, but without their limitations: NVCACHE does not limit the size of the stored data to the size of the NVMM, and works transparently with unmodified legacy applications, providing additional persistence guarantees even when their source code is not available.

## I. INTRODUCTION

NVMM is a type of memory that preserves its content upon power loss, is byte-addressable and achieves orders of magnitude better performance than flash memory. NVMM essentially provides persistence with the performance of a volatile memory [30]. Examples of NVMM include phase change memory (PCM) [14], [24], [38], [39], [11], resistive RAM (ReRAM) [8], crossbar RAM [32], memristor [58] and, more recently, Intel 3D XPoint [27], [41], [6], [5].

Over the last few years, several systems have started leveraging NVMM to transparently improve input/output (I/O) performance of legacy POSIX applications. As summarized in Table I, these systems follow different approaches and offer various trade-offs, each providing specific advantages and drawbacks. §V details our analysis but, as a first summary, a system that simultaneously offers the following properties does not exist: (i) a large storage space while using NVMM to boost I/O performance; (ii) efficient when they provide useful correctness properties such as synchronous durability (*i.e.*, the data is durable when the write call returns) or durable linearizability (*i.e.*, to simplify, a write is visible only when it is durable) [28]; and (iii) easily maintainable and does not add new kernel code and interfaces, which would increase the attack surface of the kernel.

We propose to rethink the design of I/O stacks in order to bring together all the advantages of the previous systems (large storage space, advanced consistency guarantees, stock kernel), while being as efficient as possible. To achieve this goal, we borrow some ideas from other approaches and reassemble them differently. First, like Strata [37] and SplitFS [33], we

propose to split the implementation of the I/O stack between the kernel and the user space. However, whereas Strata and SplitFS make the user and the kernel space collaborate tightly, we follow the opposite direction to avoid adding new code and interfaces in the kernel. Then, as DM-WriteCache [53] or the hardware-based NVMM write cache used by high-end SSDs, we propose to use NVMM as a write cache to boost I/Os. Yet, unlike DM-WriteCache that provides a write cache implemented behind the volatile page cache of the kernel and therefore cannot efficiently provide synchronous durability without profound modifications to its code, we implement the write cache directly in user space.

Moving the NVMM write cache in user space does, however, raise some major challenges. The kernel page cache may contain stale pages if a write is added to the NVMM write cache in user space and not yet propagated to the kernel. When multiple processes access the same file, we solve the coherence issue by leveraging the `flock` and `close` functions to ensure that all the writes in user space are actually flushed to the kernel when a process unlocks or closes a file. Inside a process, the problem of coherence also exists if an application writes a part of a file and then reads it. In this case, the process will not see its own write since the write is only stored in the log and not in the Linux page cache. We solve this problem by updating the stale pages in case they are read. Since this reconciliation operation is costly, we use a read cache that keeps data up-to-date for reads. As the read cache is redundant with the kernel page cache, we can keep it small because it only improves performance in the rare case when a process writes and quickly reads the same part of a file.

As a result, because it combines all the advantages of state-of-the-art systems, our design becomes remarkably simple to deploy and use. In a nutshell, NVCACHE is a plug-and-play I/O booster implemented only in user space that essentially consists in an NVMM write cache. NVCACHE also implements a small read cache in order to improve the performance when a piece of data in the kernel page cache is stale. Finally, using legacy kernel interfaces, NVCACHE asynchronously propagates writes to the mass storage with a dedicated thread. Table I summarizes the advantages of our system. By adding strong persistence guarantees, NVCACHE

arXiv:2105.10397v2 [cs.DC] 3 Sep 2021

**TABLE I:** Properties of several NVMM systems, all fully compatible with the POSIX API.

	Ext4-DAX [20], [56]	NOVA [57]	Strata [37]	SplitFS [33]	DM-WriteCache [53]	NVCCACHE
Offer a large storage space	–	–	+	–	+	+
Efficient for synchronous durability	+	++	++	++	–	+
Durable linearizability [28]	+	+	+	+	–	+
Reuse legacy file systems	+ (Ext4)	–	–	+ (Ext4)	+ (Any)	+ (Any)
Stock kernel	+	–	–	–	+	+
Legacy kernel API	+	+	–	–	+	+

prevents any rollback effect in case of crash with legacy software, such as DBMS, and obviates the need for a developer to handle data inconsistencies after crashes. Using NVCCACHE reduces code complexity without sacrificing performance, and thanks to persistence, the cache layer becomes transparent and reliable, unlike the Linux default RAM page cache.

Our design provides three main benefits. First, NVCCACHE can easily be ported to any operating system compliant with the POSIX interface: NVCCACHE is only implemented in user space and just assumes that the system library provides basic I/O functions (`open`, `pread`, `pwrite`, `close`, `fsync`). This design choice only adds the cost of system calls on a non-critical path, while drastically simplifying the overall design. Second, by design, NVCCACHE offers a durable write cache that propagates the writes to the volatile write cache of the kernel before propagation to the durable mass storage. While using a volatile write cache behind a durable write cache could seem unexpected, this design has one important advantage: NVCCACHE naturally uses the volatile write cache to decrease I/O pressure on the mass storage without adding a single line of code in the kernel. In details, instead of individually flushing each write that modifies a single page to the mass storage, the kernel naturally combines the writes by updating the modified page in the volatile page cache before flushing the modified page to disk only once. Strata also combines writes in volatile memory but, because it cannot leverage the kernel page cache by design, Strata must implement combining in its own kernel module. Finally, NVCCACHE naturally benefits from the numerous optimizations provided by the modern stock kernels it builds upon, *e.g.*, arm movements optimization for hard drives [47], [9] or minimization of write amplification for solid-state drive (SSD) [46], [44].

This paper presents the design, implementation and evaluation of NVCCACHE. Using I/O-oriented applications (SQLite, RocksDB), we compare the performance of: NVCCACHE with a SATA SSD formatted in Ext4, a RAM disk (`tmpfs`), a direct access (DAX) file system backed by a NVMM (Ext4-DAX), a file system tailored for NVMM (NOVA), an SSD formatted with Ext4 boosted by DM-WriteCache and a classical SSD formatted with Ext4. Our evaluation notably shows that:

- Under synchronous writes, NVCCACHE reduces by up to 10× the disk access latency of the applications as compared to an SSD, even when using DM-WriteCache.
- NVCCACHE is as fast as (or faster than) Ext4-DAX and often as efficient as NOVA, but without limiting the working set of the application to the available NVMM.
- On half of our workloads, NVCCACHE remains less efficient than NOVA (up to 1.8×). We show that this performance limitation results from the reliance on a generic I/O stack

to avoid modifications to the kernel.

- NVCCACHE is implemented in 2585 LoC only in user space, while offering the same advantages as Strata (large storage space and advanced correctness properties) with its 21 255 LoC, 11 333 of which are in the kernel.

This paper is organized as follows. We first describe the design of NVCCACHE in §II and detail its implementation in §III. In §IV, we present our evaluation using legacy applications and real-world workloads. We survey related work in §V, before concluding with some perspectives in §VI.

## II. NVCCACHE

As outlined in introduction, we have designed NVCCACHE with three different goals. First, we want to use NVMM to offer fast *synchronous durability*. With NVCCACHE, instead of routing data writes first to the volatile kernel cache, which is risky as it can lead to data loss upon crash, applications directly and synchronously write to durable storage. Furthermore, write performance is only limited by the achievable throughput of the actual NVMM.

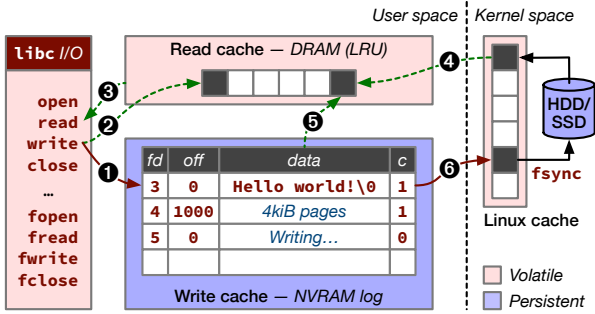
Second, NVCCACHE supports legacy applications. Our design allows applications to persist data at native NVMM speed without the size limitations of specialized file systems for NVMM, and without requiring any modifications to the application code.

Finally, NVCCACHE does not require new kernel interfaces and code, reducing the risk of introducing new attack vectors and simplifying maintainability. Our design also leverages the numerous existing optimizations implemented in the kernel of the operating system for the storage sub-systems. In particular, we take advantage of the kernel page cache to increase read performance and to combine writes in memory before propagating them to the mass storage.

### A. Approach and workflow

NVCCACHE implements a write-back cache in NVMM and executes entirely in user space. This design choice avoids kernel modifications and shortens the path between the application and the I/O stack. Specifically, NVCCACHE intercepts the writes of the applications and caches them in a log, stored in NVMM. Then, it asynchronously propagates them to the disk through the kernel using regular I/O system calls. Because NVCCACHE asynchronously propagates written data to the kernel, the kernel page cache may contain stale data. For a modified data, NVCCACHE has thus to reconstruct a fresh state by applying the last writes recorded in the NVCCACHE log during a read operation. Since this reconciliation operation is costly, NVCCACHE also implements a small read cache in user space in volatile memory to keep data up-to-date for reads.

Figure 1 gives a high-level overview of the architecture and workflow. NVCCACHE intercepts application calls to I/O



**Fig. 1:** Overall architecture of NVCACHE (*fd*: file descriptor, *off*: offset, *c*: committed).

functions of the legacy system library (`libc`). In case of a write, NVCACHE adds the written data in the write cache (Figure 1-①) by appending it to the NVMM log, which makes the write durable.

If the write modifies data in the volatile read cache, NVCACHE also updates the data accordingly (Figure 1-②) so that subsequent reads always see up-to-date data.

In case of a read, NVCACHE retrieves the data from the read cache. The data is either present and up-to-date (cache hit, Figure 1-③) or unavailable (cache miss). Upon cache misses, NVCACHE loads the data from the kernel page cache (Figure 1-④). If the data in the kernel is stale, *i.e.*, the data was already modified in the write cache, NVCACHE also applies the writes saved in the NVMM log (Figure 1-⑤).

In background, a dedicated *cleanup thread* asynchronously propagates writes to the kernel page cache (Figure 1-⑥), which will itself subsequently propagate them to disk. To do so, the cleanup thread uses standard `write` system calls from the legacy system library. In addition, the cleanup thread is also in charge of removing pending writes from the log as soon as the kernel has flushed them to disk.

NVCACHE is optimized for applications opening files in read-only mode. In such instances, NVCACHE entirely bypasses its read caches, hence avoiding the use of dynamic random-access memory (DRAM) altogether, because the kernel page cache already contains fresh data for read-only files.

We first describe the mechanisms underlying the write and read caches assuming a single-threaded application, focusing on multi-threading in §II-D.

### B. Design of the write cache

NVCACHE implements its write cache as a circular log in NVMM. Each entry in the log contains a write operation, *i.e.*, the target file descriptor, the starting offset, data itself and the number of bytes to write.

In addition to the log, NVCACHE stores in NVMM a table that associates the file path to each file descriptor, in order to retrieve the state after a crash. NVCACHE also keeps a *persistent tail index* in NVMM to track the oldest entry of the log. The cleanup thread and the recovery procedure use the tail index to propagate the oldest writes to disk before the newer ones, in order to preserve the write order of the application.

**TABLE II:** Page states.

		Is page in DRAM read cache?	
		Yes	No
Are there corresponding log entries in NVMM write cache?	Yes	Loaded	Unloaded dirty
	No	Loaded	Unloaded clean

NVCACHE uses a *head index* in volatile memory<sup>1</sup> to create new entries in the log. When NVCACHE intercepts a write, it advances the head index and fills the new log entry accordingly. Finally, NVCACHE also maintains a copy of the persistent tail index in volatile memory. NVCACHE uses this volatile tail index to synchronize a writer with the cleanup thread when the log is full (see §III for details).

**Failure management.** When a new entry is created in the log but its fields are not yet completely populated, *i.e.*, it corresponds to a non-committed write, the recovery procedure of NVCACHE ignores the entry. To detect such scenarios, we could use a *committed index* in NVMM to keep track of the last committed entry. While we considered this approach, we decided to avoid it for three reasons. First, maintaining a shared index in NVMM is inefficient in multi-threaded environments because of the synchronization costs for accessing it consistently. Second, non-trivial synchronization mechanisms are also required to handle scenarios when a thread commits a new entry before an older one that is not yet committed. Finally, since data moves from the processor caches to the NVMM at cache-line granularity, updating an index would lead to an additional cache miss to load the index.

Instead, in order to handle non-committed writes left in the log after a crash, we directly embed a commit flag with each entry (column *c* in Figure 1). With this design, a thread can independently commit an entry while bypassing the synchronization costs of a shared index. NVCACHE also avoids an additional cache miss because the commit flag lives inside an entry that already has to be loaded during a write.

### C. Design of the read cache

Since the kernel may contain stale data in its page cache, NVCACHE implements a small read cache in user space. Technically, NVCACHE implements an approximation of an LRU cache in volatile memory. When a file is not opened in write-only mode (in which case the read cache is not required), NVCACHE maintains a radix tree [4] used to quickly retrieve the pages of the file, an approach similar to NOVA [57].<sup>2</sup>

NVCACHE also keeps track (in volatile memory) of the cursor and size of each file as both could be stale inside the kernel, *e.g.*, when a newly appended data block is still in flight in the NVCACHE write cache.

**Page descriptor and page content.** A leaf of the radix tree contains a *page descriptor*, which is created on demand during a read or a write. The page descriptor keeps track of the state of a page (see Table II). When a page is present in the read cache, the page is in the *loaded* state. In this state, a *page content* is associated to the page descriptor. A page content is a

<sup>1</sup>This index is not in NVMM, *i.e.*, not needed for recovery after a crash.

<sup>2</sup>A page size has to be a power of two because we use a radix tree. Nevertheless, pages in NVCACHE are not related to hardware pages.

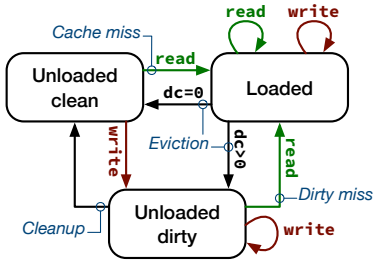


Fig. 2: State machine of pages (*dc*: dirty counter).

piece of cached data, always kept consistent: when NVCACHE intercepts a write, it also updates the content of a loaded page in the read cache.

When a page is not present in the read cache, its page descriptor (when it exists) is not associated to a page content. When unloaded, a page can have two different states. A page is in the *unloaded-dirty* state if the NVCACHE write cache contains entries that modify the page. In this state, the content of the page outside NVCACHE (in the kernel page cache or on disk) is dirty. A page is in the *unloaded-clean* state otherwise. NVCACHE distinguishes the unloaded-dirty from the unloaded-clean state with a counter, called the *dirty counter*, stored in the page descriptor. NVCACHE increments this counter in case of writes, and the cleanup thread decrements this counter when it propagates an entry from the write cache.

**State transitions.** Our design has the goal of avoiding any costly synchronous write system call in the critical path of the application. Technically, NVCACHE avoids write system calls when it evicts a dirty page or when the application modifies an unloaded page. Instead of synchronously writing the page, NVCACHE simply marks the page as unloaded-dirty and lets the cleanup thread asynchronously perform the propagation.

Figure 2 presents the state transitions for the pages handled by NVCACHE. In more detail, in case of cache miss, NVCACHE starts by evicting the least recently used page (loaded to unloaded-clean or to unloaded-dirty according to the dirty counter in Figure 2) to make space for the new page. Then, NVCACHE handles two cases. If the new page comes from the unloaded-clean state (unloaded-clean to loaded in Figure 2), NVCACHE simply loads the page in the read cache by using the *read* system call. If the new page comes from the unloaded-dirty state (unloaded-dirty to loaded in Figure 2), NVCACHE loads the page and additionally executes a custom *dirty-miss* procedure. This procedure reconstructs the state of the page by searching the dirty counter entries that modify the page in the log, starting from the tail index, and applies the modification in the read cache. This procedure is costly but, as shown in our evaluation (see §IV), dirty misses are rare. Thanks to the *dirty-miss* procedure, NVCACHE avoids the synchronous write system calls upon dirty page eviction (loaded to unloaded-dirty in Figure 2) and when writing an unloaded page (unloaded-clean to unloaded-dirty in Figure 2).

#### D. Multi-threading

As required by the POSIX specification, NVCACHE ensures that read and write functions are atomic with respect to each other [1]. NVCACHE ensures thus that concurrent writes to the same pages are executed one after the other while respecting the write order of the application, and ensures that a read cannot see a partial write. Apart from ensuring atomicity for reads and writes on the same pages, the design of NVCACHE also natively supports fully parallel execution of writes to independent pages. We achieve this by leveraging three techniques: fixed-sized entries, page-level locking granularity and scalable data structures, which we describe next.

**Fixed-sized entries.** As a first step to execute independent writes in parallel, NVCACHE uses fixed-size entries. With entries of arbitrary size, one would need to store the size of each entry in the log, which prevents a thread to commit an entry if the previous entry is not yet committed. Indeed, in case of crash, the recovery procedure cannot know if the size of an uncommitted entry is correct, and can thus neither inspect the next entry nor consider it as committed. With fixed-sized entries, a thread can commit an entry, even if a previous entry allocated by another thread is not yet committed. In this case, because the entry size is predefined (a system parameter of NVCACHE), the recovery procedure can ignore an uncommitted entry and continue with the next one.

Using fixed-sized entries, NVCACHE must use multiple entries for large writes not fitting in a single entry. For such large writes, NVCACHE must commit all the entries atomically in order to avoid partial writes. NVCACHE atomically commits the multiple entries by considering the commit flag of the first entry as the commit flag of the group of entries. Technically, NVCACHE maintains a *group* index in each entry. The index is set to -1 for the first entry and, for the following entries, it indicates the position of the first entry. NVCACHE also saves space, and thus cache misses, by packing the commit flag and the group index in the same integer.

**Per-page locking.** When two threads concurrently modify the same pages, NVCACHE ensures that one write is fully executed before the other. Instead of using a single write lock for each file, NVCACHE optimizes parallelism by using a per-page locking scheme, in which each page descriptor contains a lock called the *atomic lock*. In case of a write, a thread starts by acquiring all the atomic locks of the written pages. Then, the thread adds the write to the log by creating, filling and committing one or multiple entries. Finally, before releasing the atomic locks, the thread increments the dirty miss counters and, for modified pages in the loaded state, the thread also updates their content in the read cache.

The atomic lock is also used in the read function in order to ensure atomicity. In case of a read, as for a write, NVCACHE starts by acquiring all the per-page locks, which ensures that a read cannot see a partially updated page.

NVCACHE also uses a second lock per page, called the *cleanup lock*, which is used to synchronize the cleanup thread and the dirty miss procedure. Without this lock, in case of

cache miss, the application may read a stale page from the disk and miss an entry concurrently propagated by the cleanup thread. We avoid this race condition by acquiring the cleanup lock in the cleanup thread before the propagation, and by acquiring the cleanup lock in the application in case of cache miss. In more detail, a reader starts by acquiring the atomic lock in order to ensure atomicity. Then, in case of cache miss, the reader both reads the page from the disk and applies the dirty miss procedure while it owns the cleanup lock, which ensures that the cleanup thread cannot process an entry associated to the page while the application applies the dirty miss procedure.

Finally, NVCACHE synchronizes the access to the dirty miss counter by using atomic instructions and by leveraging the two locks associated to the page. In case of a write, NVCACHE simply increments the dirty miss counter with an atomic instruction while it owns the atomic lock, which ensures that a reader cannot execute the dirty miss procedure at the same time (because the reader has to own the atomic lock to execute a read and thus the dirty miss procedure). The cleanup thread decrements the dirty miss counter with an atomic instruction while it owns the cleanup lock, which ensures that a reader cannot execute the dirty miss procedure at the same time (because the reader has also to own the cleanup lock to execute the dirty miss procedure).

Due to our locking scheme with two locks per page, the cleanup thread never blocks a writer and never blocks a reader when the page is already in the read cache. The cleanup thread can only block a reader in case of cache miss when the cleanup thread propagates an entry that modifies the missed page.

**Scalable data structures.** Most of the internal structures of NVCACHE are designed to scale with the number of threads. First, in order to add an entry in the write log, a thread simply has to increment the head index, which is done with atomic instructions.<sup>3</sup>

Then, the radix tree can operate with a simple lock-free algorithm because NVCACHE never removes elements from the tree, except when it frees the tree upon close. When an internal node or a page descriptor in the radix tree is missing, a thread tries to atomically create it with a compare-and-swap instruction; if this fails, it means that the missing node or page descriptor was concurrently added to the tree by another thread and we can simply use it.

Finally, NVCACHE builds an approximation of an LRU algorithm by storing an *accessed* flag in each page descriptor, which is set during a read or a write. In more detail, NVCACHE maintains a queue of page contents protected by a lock, called the *LRU lock*. Each page content holds a reference to its descriptor and conversely, as already presented, a descriptor links to the associated page content when it is loaded. When NVCACHE has to evict a page, it acquires the LRU lock and dequeues the page content at the head of the queue. Then, NVCACHE acquires the atomic lock of the

<sup>3</sup>Note that we do not need special instructions to handle NVMM since the head lives in volatile memory.

TABLE III: Functions intercepted by NVCACHE.

Function	Action
open, read, write, close	Uses NVCACHE functions
fopen, fread, fwrite, fclose	Uses unbuffered versions
sync, syncfs, fsync	No operation
lseek, ftell, stat, etc.	Uses size/cursor of NVCACHE

page descriptor associated to the head and checks its accessed flag. If the flag is set, NVCACHE considers that the page was recently accessed: it releases the atomic lock of the page descriptor, re-enqueues the page content at the tail, and restarts with the new head. Otherwise, NVCACHE recycles the page content: it nullifies the reference to the page content in the page descriptor, which makes the page descriptor unloaded-clean or unloaded-dirty, and releases the atomic lock of the page descriptor.

**Summary.** To summarize, in case of a write, NVCACHE only acquires the atomic locks to the written pages. In case of read hits, *i.e.*, if the page contents are already in the read cache, NVCACHE also only acquires the atomic locks to the read pages. In case of read misses, NVCACHE has also to acquire the LRU lock and an atomic-lock during eviction, and then the cleanup lock to read the missed page. As a result, except to handle read misses, NVCACHE executes two operations that access different pages in a fully concurrent manner, and without using locks to synchronize with the cleanup thread.

### III. IMPLEMENTATION

We implemented NVCACHE on top of the `musl` libc [26], a lightweight yet complete alternative to `glibc`. `Musl` consists of 85 kLoC and NVCACHE adds 2.6 kLoC. In order to simplify deployment, instead of using `LD_PRELOAD` to wrap I/O-based system calls, we directly replace the I/O-based system calls by ours in `musl`. We also exploit the Linux Alpine distribution [19] using `musl` to easily deploy NVCACHE behind legacy applications (as shown in our evaluation). Moreover, NVCACHE supports Docker containers [23] and our approach allows us to deploy legacy applications with just one minor change to existing manifests, replacing the original libc shared object by ours.

Table III lists the main functions intercepted by NVCACHE. Essentially, NVCACHE wraps `open`, `read`, `write` and `close` to use the read and write caches. Additionally, it replaces the buffered versions of I/O functions (`fread`, `fwrite`, `fopen`, `fclose`) with their unbuffered counterparts, since NVCACHE itself acts as a buffer in user-space with its small read cache in volatile memory.

As for the `fsync` function calls, which force the kernel to synchronously propagate pending writes from the kernel page cache to the disk, NVCACHE simply ignores them. These operations are no longer necessary because NVCACHE already makes the write synchronously durable.

As presented in §II-C, NVCACHE maintains its own versions of file cursors and size information, because the cursor associated to an opened file and the size of a file may be stale in the kernel. Therefore, NVCACHE intercepts the `stat` and `seek` function families in order to return fresh values.



At low level, in order to read and write from a cursor maintained by NVCACHE and not the one maintained by the kernel, the cleanup thread uses the `pwrite` function to propagate an entry to the kernel and, upon cache misses, NVCACHE uses `pread` to load a page in the read cache.

NVCACHE does not support asynchronous writes, but they could be implemented. Memory-mapped files, however, are not supported in NVCACHE and their implementation remains an open problem, as loads and stores are not interceptable at the libc level.

In the following, we detail how we have implemented the `open`, `write`, `cleanup` and `recovery` functions because they directly deal with NVMM. The `read` function closely implements the design presented in §II-C and §II-D.

**Open.** The `open` function adds a file to the cache. First, in NVMM, NVCACHE maintains a table that associates the paths of the opened files to their file descriptors. This table is only used during recovery: after a crash, the file descriptors in the log entries are meaningless and the recovery engine therefore needs the information in the table to retrieve the paths of the files associated to these file descriptors.

Then, in volatile memory, NVCACHE keeps track of opened files with two tables in order to handle independent cursors when an application opens the same file twice. The first one, called the file table, associates a *(device, inode number)* pair to a file structure, which contains the size of the file and its radix tree. The second one, called the opened table, associates a file descriptor to an opened-file structure, which contains the cursor and a pointer to the file structure.

In `open`, NVCACHE starts by retrieving the device and the inode number associated to the file path with the `stat` system call. Then, if the file belongs to a block device and if the *(device, inode number)* pair is not yet present in the file table, NVCACHE creates the file structure. Finally, NVCACHE uses the `open` system call to create a file descriptor and creates accordingly an opened-file structure in the opened table.

NVCACHE bypasses its read cache when a file is only opened in read-only mode. For that purpose, NVCACHE only creates a radix tree in the file structure when the file is opened in write mode for the first time and, for a file that does not have a radix tree, NVCACHE bypasses its read cache.

**Write.** Algorithm 1 shows a simplified and unoptimized version of NVCACHE’s write function when the write fits in one page and one entry. Our code uses three NVMM-specific instructions: `pwb(addr)` (e.g., `clwb` on a Pentium) ensures that the cache line that contains `addr` is added in the flush queue of the cache; `pfence` (e.g., `sfence` on a Pentium) acts both as a store barrier and ensures that the `pwb`s that precede are executed before the barrier; and `psync` (e.g. also `sfence` on a Pentium) acts as a `pfence` and furthermore ensures that the cache line is actually drained to the NVMM. With these instructions, the `write(a, v1), pwb(a), pfence, write(b, v2)` sequence ensures that the write to `a` is propagated to NVMM before the write to `b`

**Algorithm 1** — NVCACHE write function.

```

1 struct nvram { // Non-volatile memory
2   struct { char path[PATH_MAX]; } fds[FD_MAX];
3   struct entry entries[NB_ENTRIES];
4   uint64_t persistent_tail;
5 }* nvram;

7 uint64_t head, volatile_tail; // Volatile memory

9 void write(int fd, const char* buf, size_t n) {
10  struct open_file* o = open_files[fd];
11  struct file* f = o->file;
12  struct page_desc* p = get(f->radix, o->offset);

14  uint64_t index = next_entry();
15  struct entry* e = &nvram->entries[index % NB_ENTRIES];

17  acquire(&p->atomic_lock);

19  memcpy(e->data, buf, n); // Write cache
20  e->fd = fd;
21  e->off = o->off;
22  pwb_range(e, sizeof(*e)); // Send the uncommitted entry to NVMM
23  pfence(); // Ensure commit is executed after

25  e->commit = 1;
26  pwb_range(e, CACHE_LINE_SIZE); // Send the commit to NVMM
27  psync(); // Ensure durable linearizability

29  atomic_fetch_add(&p->dirty_counter, 1); // Read cache
30  if(p->content) // Update page if present in the read cache
31    memcpy(p->content->data + o->off %
32    release(&p->atomic_lock);
33 }

35 int next_entry() {
36  int index = atomic_load(&head);
37  while(((index + 1) %
38    latomic_compare_and_swap(&head, index, index + 1))
39    index = atomic_load(&head);
40  return index; // Commit flag at index is 0 (see cleanup thread)
41 }

```

because the cache line of `a` is flushed before the write to `b` is enqueued in the flush queue of the cache.

In our code, after having retrieved (or lazily created) the page descriptor (line 12), the write function finds a new free entry in the log (line 14 and 35–41). In details, `next_entry` first waits if the log is full (line 37) and then advances the head while taking care of concurrent accesses from another threads (line 38). At this step, the commit flag of the entry returned by `next_entry` is necessarily equal to 0 (see the cleanup thread below).

Then, as soon as the write function acquires the lock of the page descriptor (line 17), it adds the write to the log (lines 19 to 27). More precisely, the function fills the entry without committing it (lines 19 to 21), sends the uncommitted entry to the NVMM by flushing the cache lines of the entry (line 22) and executes a `pfence` in order to ensure that the entry is flushed before the commit (line 23). The function then commits the entry (line 25), sends the cache line that holds the commit flag to the NVMM (line 26) and executes a `psync` to ensure durable linearizability (line 27, see the text below for the explanation). At this level, the `atomic_lock` is only taken for atomicity purposes between the writer thread and

the cleanup thread, preventing the cleanup thread to modify a page on the SSD while a cache miss procedure reads it.

Finally, the write function manages the read cache (lines 29 to 31) before releasing the lock at line 32. Specifically, it increments the dirty counter (line 29) because the log contains a new entry that modifies the page. The increment is done with an atomic instruction in order to prevent a conflict with the cleanup thread that may be concurrently decrementing the counter (see §II-D).<sup>4</sup> If the page is in the loaded state (*i.e.*, the page descriptor is associated to a page content, line 30), the function updates the page content in the read cache (line 31).

**Durable linearizability.** Our algorithms ensure durable linearizability [29], which essentially means that if a read sees a write, then the write is committed. For example, this is not the case of Linux when it uses the page cache: indeed, a thread can see a write stored in the page cache but not yet propagated to disk. NVCACHE always ensures durable linearizability because of the `psync` at line 27. This operation ensures that the commit at line 25 is written to NVMM before the lock release at line 32, which is itself executed before the lock acquire of a reader able to see the write.

**Cleanup thread and batching.** The cleanup thread propagates the writes from the NVMM log to the disk. At high level, it consumes the entries one by one, starting at the persistent tail index. The cleanup thread begins by synchronizing with the application through the commit flag: if the entry at the persistent tail is not yet committed, the cleanup thread waits. When the entry at the persistent tail is committed, the cleanup thread consumes the entry while owning the cleanup locks associated to the descriptors of the pages modified by the entry. In more detail, the cleanup thread proceeds in three steps when it consumes an entry. During the first step, the cleanup thread propagates the entry to the mass storage by using `pwrite` to send the write to the kernel page cache and by using `fsync` to synchronously propagate the writes from the kernel page cache to the mass storage. During the second step, the cleanup thread updates both the commit flag of the consumed entry and the persistent tail index, and uses `pwb/pfence` to ensure that the third step can only start after the second step. During the third step, the cleanup thread marks the entry as free for the writers by using the volatile tail index. Because of the use of the two tail indexes, when a writer sees that an entry is free in volatile memory (volatile tail index), we have the guarantee that the entry is also marked as free in NVMM (persistent tail index and commit flag of the entry).

The implementation described above is inefficient because a call to `fsync` is especially costly. The throughput of a random 4kB write on an SSD is at least  $13\times$  faster without `fsync` [35]. To mitigate the negative impact of a slow cleanup thread that continuously calls `fsync`, which would otherwise

<sup>4</sup>Because a writer does not acquire the cleanup lock, the cleanup thread may decrement the dirty counter between the commit at line 27 and the atomic add at line 29, making the counter negative. This temporary negative counter cannot lead to an incorrect behavior because a reader has to take both the atomic and cleanup locks to execute the dirty miss procedure, ensuring that the dirty miss procedure cannot observe an unstable negative counter.

block all the writes of the application when the log is full, the cleanup thread batches the writes. Batching allows us to reduce the frequency of calls to `fsync`, updating the tail index only upon success. The advantages of batching are twofold. First, batching decreases the number of calls to `fsync`, boosting the performance of the cleanup thread and thus decreasing the probability of having a full log. Then, batching allows NVCACHE to leverage kernel optimizations by combining writes or optimizing the sequences of writes for hard drives or SSD. §IV-C presents in detail how batching improves performance.

**Recovery procedure.** When NVCACHE starts, it executes a recovery procedure. It first re-opens the files by using the table that associates file paths to file descriptors stored in NVMM. Then, it propagates all the committed entries of the log by starting at the tail index, invokes the `sync` system call to ensure that the entries are written to disk, close the files and empties the log.

**Multi-application.** The NVMM write log of NVCACHE is either a DAX device, *e.g.*, an entire NVMM module, or a DAX file, *i.e.*, a file in any DAX-capable filesystem. Therefore, in a multi-application context, two instances of NVcache can run simultaneously on the same machine, either with one NVMM module each or sharing the same module split into two DAX files.

## IV. EVALUATION

### A. Hardware and software settings

We evaluate NVCACHE on a Supermicro dual socket machine with two NUMA domains. Each NUMA domain contains an Intel Xeon Gold 5215 CPU at 2.50 GHz with 10 cores (20 hardware threads),  $6\times 32$  GiB of DDR4 and  $2\times 128$  GiB of Optane NVDIMM. In total, the machine contains 20 cores (40 hardware threads), 384 GiB of DDR4 and 512 GiB of Optane NVDIMM. The machine is equipped with two SATA Intel SSD DC S4600 with 480 GB of disk space each. One of them contains the whole system, while the other is dedicated for some of our experiments. The main SSD is formatted with an Ext4 file system. The secondary one is mounted with `lvm2`, linked with a DM-WriteCache stored in one of our Optane NVDIMM. This virtual `lvm2` device is also formatted with Ext4. We deploy Ubuntu 20.04 with Linux version 5.1.0 (NOVA [57] repository version) and `musl v1.1.24`, revision `9b2921be`.

We evaluate NVCACHE with representative benchmarks that heavily rely on persistent storage. RocksDB is a persistent key-value store based on a log-structured merge tree widely used in web applications and production systems, *e.g.*, by Facebook, Yahoo! and LinkedIn [15]. We evaluate RocksDB v6.8 with the `db_bench` workload shipped with LevelDB, an ancestor of RocksDB, which stresses different parts of the database. SQLite [3] is a self-contained SQL database widely used in embedded systems, smartphones and web browsers [31], [36]. We evaluate SQLite v3.25 with a port of the `db_bench` for SQLite. We also use FIO [12] version 3.20, a micro-benchmark designed to control the read and

TABLE IV: Evaluated file systems.

Name	Write cache	Storage space	FS	Synchronous durability	Durable linearizability
NVCache+SSD	NVCache	SSD	Ext4	by default	by default
DM-WriteCache	kernel page cache	SSD	Ext4	O_DIRECT — O_SYNC	no
Ext4-DAX	kernel page cache	NVMM	Ext4	O_DIRECT — O_SYNC	no
NOVA <sup>5</sup>	none	NVMM	NOVA	O_DIRECT — O_SYNC	by default
SSD	kernel page cache	SSD	Ext4	O_DIRECT — O_SYNC	no
tmpfs	kernel page cache	DDR4	none	no	no
NVCache+NOVA	NVCache	NVMM	NOVA	by default	by default

write ratios and frequencies, the number of threads or the read and write patterns (random, sequential).

Unless stated otherwise, we configure NVCache as follows. Each entry in our NVMM log is 4 KiB large. The log itself is constituted of 16 million entries (around 64 GiB). The RAM cache uses 250 thousand pages of 4 KiB each (around 1 GiB). The minimum number of entries before attempting to batch data to the disk is 1 thousand. The maximum number of entries in a batch is 10 thousand.

### B. Comparison with other systems

In this experiment, we compare NVCache with other systems. Table IV summarizes the different file systems evaluated. We compare the normal version of NVCache when it propagates the writes to an SSD formatted in Ext4 (NVCache+SSD) with five other systems and a variant of NVCache.

Specifically, we evaluate: (i) DM-WriteCache, which asynchronously propagates the writes from the volatile kernel page cache to an NVMM write cache, and only later propagates the writes from NVMM to an SSD formatted in Ext4 (DM-WriteCache+SSD); (ii) the Ext4 file system directly stored in NVMM (Ext4-DAX); (iii) the NOVA file system [57], tailored to efficiently use NVMM (NOVA), (iv) an SSD formatted in Ext4 (SSD); (v) a temporary file system, which only stores the data in volatile memory in the kernel page cache (tmpfs).

We also evaluate a variant of NVCache that propagates the writes to the NVMM formatted with the NOVA file system (NVCache+NOVA). This variant does not offer a large storage space like NVCache+SSD but shows the theoretical performance that we could expect from NVCache when using an efficient secondary storage.

NVCache+SSD, NOVA and NVCache+NOVA provide the highest consistency guarantees since they offer both synchronous durability (*i.e.*, the data is durable when the write call returns) and durable linearizability (essentially, a write is only visible when it is durable). In order to make a fair comparison, we also enforce synchronous durability for all the file systems by activating the synchronous mode of our benchmarks. Alternatively, on a non-synchronous benchmark, we could open the files with the `O_SYNC` flag, which guarantees that a write is flushed to disk when the system call returns. We can also optimize these systems by using the `O_DIRECT` flag<sup>6</sup>, which tries to avoid an in-memory copy of the buffer from the user space to the kernel space when possible. DM-WriteCache+SSD, SSD and Ext4-DAX are not designed to

ensure durable linearizability and do not offer the guarantee that a write is visible only when it is durable. The tmpfs file system does not provide durability, and thus no consistency guaranty. Figure 3 presents our results, respectively for write-oriented (left) and read-oriented (right) workloads.

**Read-oriented workloads.** Analyzing the read-oriented workloads, we note how all the systems provide roughly the same performance. This indicates that the different designs do not significantly change the read performance on these benchmarks. Despite NOVA and Ext4 are reading from NVMM while all the others are reading from an SSD, they all benefit from a volatile read cache stored in DDR4 RAM.

**Write-oriented workloads.** When we analyze the write-oriented workloads, we observe that the different designs approaches have a large impact on performance. We first turn our attention to the systems that offer a large storage space: NVCache+SSD, DM-WriteCache+SSD and SSD. We observe that NVCache+SSD is consistently faster than the other systems (at least 1.9 $\times$ ). SSD has the worst performance as it does not leverage NVMM to boost performance. This is not the case of DM-WriteCache+SSD and NVCache+SSD, both using NVMM as a write cache to boost I/O performance. However, NVCache+SSD performs significantly better than DM-WriteCache+SSD. Indeed, the design of NVCache+SSD naturally offers synchronous durability, because the application writes directly in the NVMM. Since, by design, DM-WriteCache+SSD lives behind the volatile kernel page cache, enforcing synchronous durability requires the execution of additional code during a write. This code significantly hampers the performance of DM-WriteCache+SSD. Among the three systems, NVCache+SSD is also the only system that ensures durable linearizability.

We now focus on systems that offer strong correctness guarantees but sacrifice the storage space: Ext4 and NOVA. With RocksDB, NVCache+SSD is 1.4 $\times$  faster than Ext4, and NOVA is 1.6 $\times$  better than NVCache+SSD. With SQLite, NVCache performs better than NOVA (around 1.6 $\times$  better), and NVCache is roughly 3.7 $\times$  better than Ext4. For some workloads, NOVA is more efficient than the other systems because it was specifically tailored for NVMM and bypasses the bottlenecks of Ext4 [57]. With RocksDB, we observe that NVCache also suffers from these bottlenecks. Indeed, when we use NVCache as an I/O booster in front of NOVA instead of SSD (NVCache+NOVA), NVCache can match and even improve performance as compared to NOVA. Overall, these results show that NVCache is able to reach performance comparable to a generic file system on NVMM, while ignoring the limit of NVMM storage space. Our design

<sup>5</sup>Formally, NOVA provides synchronous durability and durable linearizability when mounted with the `cow_data` flag.

<sup>6</sup>[https://ext4.wiki.kernel.org/index.php/Clarifying\\_Direct\\_IO%27s\\_Semantics](https://ext4.wiki.kernel.org/index.php/Clarifying_Direct_IO%27s_Semantics)



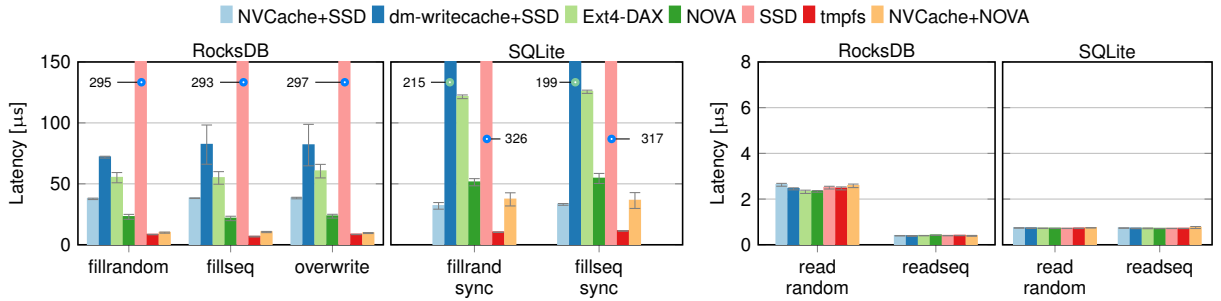


Fig. 3: Performance of NVCACHE for synchronous write-heavy (left) and read-heavy (right) workloads.

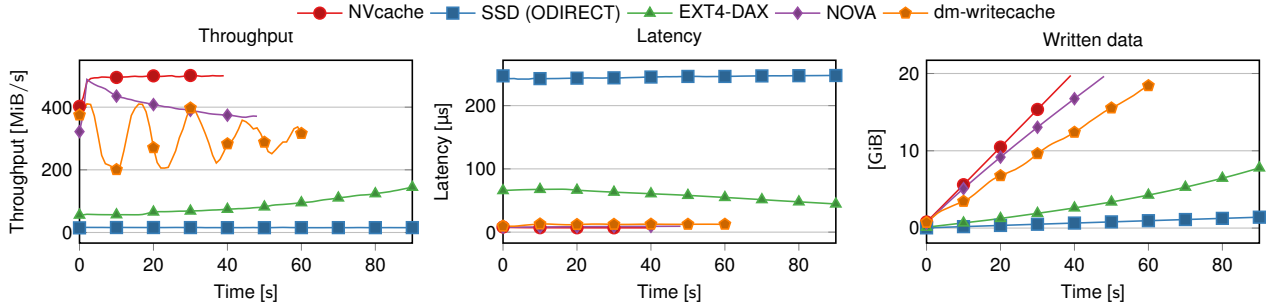


Fig. 4: Performance of NVCACHE under random write intensive loads for 20 GiB.

remains, however, less efficient than a file system specifically tailored for NVMM on some of the workloads because it remains totally independent from the kernel I/O stack.

#### C. Analysis of NVCACHE

In this section, we analyse the behavior of NVCACHE by using the FIO profiling tool. We configured FIO with the `fsync=1` flag to ensure that a write is synchronously durable, and with the `direct=1` flag to open all the files in direct I/O mode. We set the buffer size to 4 KiB with `ioengine=psync`. FIO measures are fetched every second.

**Comparative behavior.** The objective of this experiment is to study the performance of NVCACHE in an ideal case. We configure FIO to generate a random write intensive workload and, in NVCACHE, we use a log of 32 GiB for 20 GiB of written data. As a result, NVCACHE cannot saturate the log and is never slowed down by the cleanup thread.

Figure 4 reports the performance with these settings. The left graph shows the instantaneous throughput as it evolves during the run. The middle graph shows how the average latency evolves during the run. Finally, the right graph shows how the total amount of written data evolves during the run. For these measures, we split the run in small periods and report the average throughput observed during each of them (instantaneous throughput), as well as the average latency and cumulative data written as measured from the beginning of the run to the end of each period.

Figure 4 (left) shows that, in this ideal case, NVCACHE has a better throughput than all the other systems. NVCACHE significantly outperforms SSD, Ext4-DAX and DM-WriteCache (at least  $1.5\times$ ). These systems are designed to leverage the volatile Linux page cache in order to improve read performance, which makes them inefficient for writes when an application requires strong consistency guaranties such

as synchronous durability. We can observe that NVCACHE also outperforms NOVA on this benchmark (493 MiB/s vs. 403 MiB/s on average). NVCACHE is slightly more efficient than NOVA in this ideal case because NVCACHE never calls the system during a write, whereas NOVA has to pay the cost of system calls on the critical path. In Figure 4 (middle) and Figure 4 (right), we can also observe that the average latency and the written data are better in NVCACHE than in other systems. As a result, in this ideal case, NVCACHE writes all the data in 42 s, while it takes 51 s for NOVA, 71 s for DM-WriteCache+SSD, 2 min and 29 s for Ext4 and more than 22 min for SSD.

**Log saturation.** During a long run that intensively writes, NVCACHE may saturate its log, because the cleanup thread can only propagate the writes from the log to the SSD at the speed of the SSD, which is much slower than NVMM. The next experiment highlights this behavior. Using a workload to intensively write 20 GiB at random locations, we measure the performance of NVCACHE with different log sizes.

Figure 5 reports the result of NVCACHE with these settings (instantaneous throughput on the left, average latency in the middle and cumulative data written on the right). With a log of 32 GiB and 20 GiB of written data, the log never saturates. Hence, as in Figure 4 (see §IV-C), the instantaneous throughput and the average latency remain stable.

As shown in Figure 5 (left), we can observe two phases with a log of 8 GiB. During a first phase that starts at 0 and ends at 18 s, the throughput remains stable and high (556 MiB/s on average). At the end of this phase, the throughput suddenly collapses to 78 MiB/s and then remains stable until the end of the run. In this experiment NVCACHE collapses at 18 s because of the log saturation. Still, before saturation, NVCACHE is as fast as with a log of 32 GiB and only limited by the

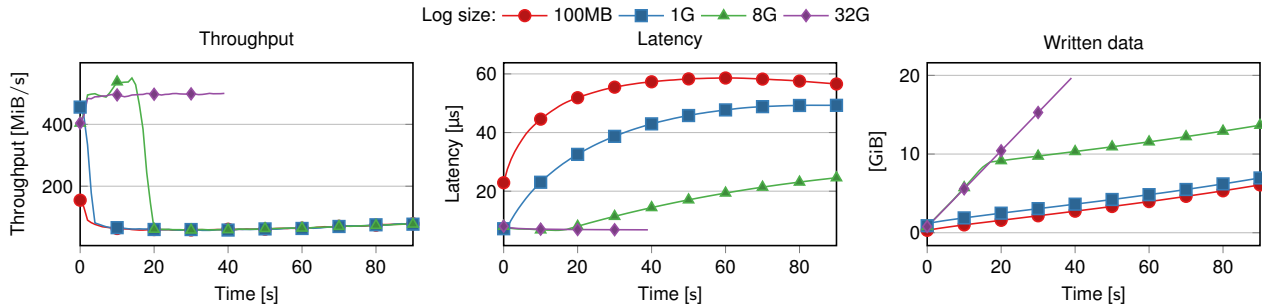


Fig. 5: Performance of NVCACHE under random write intensive loads for 20 GiB, with variable NVMM log size.

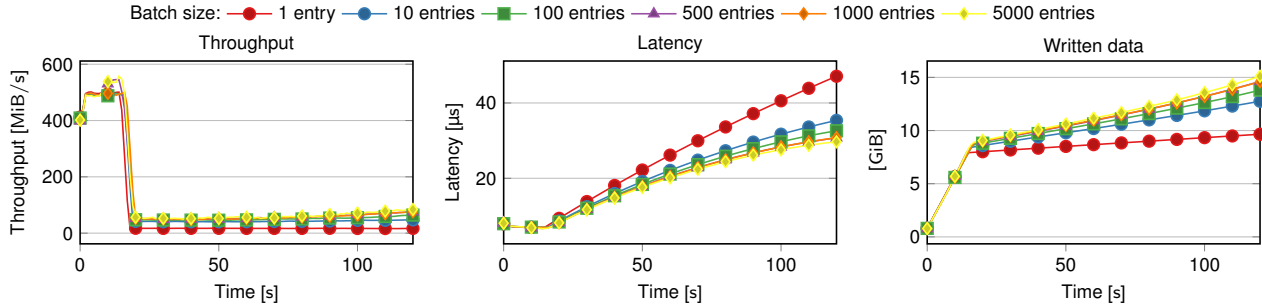


Fig. 6: Influence of batching and batch size parameter.

performance of the NVMM. During the first phase, NVCACHE fills the log but the cleanup thread cannot empty it fast enough because the cleanup thread is limited by the performance of the SSD. After saturation, FIO has to wait for the cleanup thread, which limits performance to the speed of the SSD.

We can observe a similar pattern for the average latency in Figure 5 (middle). The latency remains stable and high before saturation and then starts degrading (note that the latency does not collapse because the figure reports the average latency since the beginning of the run). We can also observe that the slope of throughput curves change when the log saturates in Figure 5 (right): when reaching saturation, the number of written data increases much slower. We can observe exactly the same behavior with smaller log sizes, but the log saturates earlier. Interestingly, with a log of 100 MiB, 1 GiB and 8 GiB, the write throughput becomes identical as soon as the log saturates, staying at around 80 MiB/s (which corresponds to the throughput of our SSD performing random writes).

**Batching effect.** Since a call to `fsync` is costly, the cleanup thread uses batching to avoid calling `fsync` for each write in the log. Technically, the cleanup thread consumes entries by batches of a given size. If the log does not contain enough entries, the cleanup thread does not propagate them and waits. In this experiment, we analyze how NVCACHE reacts to different batch sizes by running a workload intensively writing 20 GiB at random locations. We use a log of 8 GiB, in order to observe how NVCACHE reacts to batch sizes before and after the saturation.

Figure 6 reports the instantaneous throughput, the average latency and cumulative data written during the run with these settings. As in §IV-C, we can first observe the log saturation at 18s. We can also observe that, before saturation, changing the batch size does not affect performance (throughput,

latency and written data). After saturation, we can observe the batch size influence on performance. With a very small batch size, the throughput is as low as 21 MiB/s: for each write, NVCACHE triggers a `fsync`, which makes NVCACHE less efficient than the SSD configured with `O_DIRECT` because of the cost of the system call. After the saturation, we observe that NVCACHE becomes more efficient with larger batch sizes, for two reasons: First, NVCACHE does not call `fsync` often, which increases its performance. Then, because NVCACHE first writes a batch in the volatile kernel page cache, writes that modify the same location in the file are combined in volatile memory, decreasing the number of pages Linux has to propagate to the SSD upon `fsync`[43]. We also observe that the difference between a batch size of 100, 1000 and 5000 remains low. This result shows that, as soon as the batch size becomes large enough, the influence of this parameter is low.

**Read cache size effect.** As presented in §II-A, since the Linux page cache becomes stale when a pending write in the log is not propagated, NVCACHE relies on a small read cache to ensure consistency. In this experiment, we show how NVCACHE reacts to different read cache sizes. Figure 7 presents the write (left) and read (right) throughput of NVCACHE with a random read/write workload (50%/50%) and a file of 10 GiB. Since the read cache is required for consistency, we cannot deactivate it. We start thus with a very small cache of 100 entries (400 KiB) for which the probability of cache hit is negligible. The results confirm how the size of the read cache does not affect performance. We can observe that with a large cache of 4 GiB (1M entries) where the probability of cache hit reaches 40%, the performance remains the same. This result is expected since NVCACHE relies on the kernel page cache to improve performance. NVCACHE

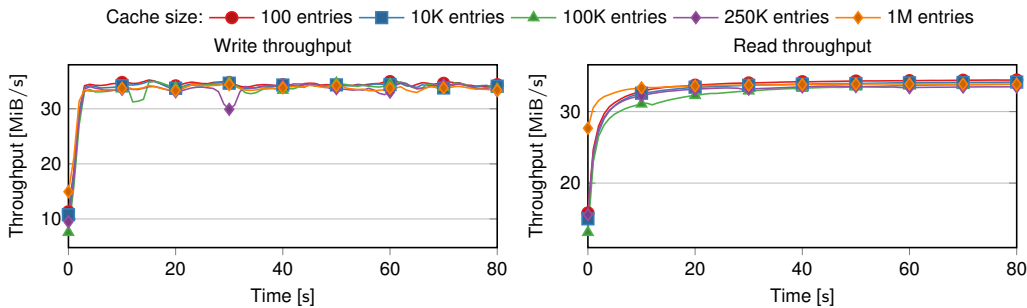


Fig. 7: NVCACHE with different read cache size, under mixed read/write loads.

only uses the read cache to ensure that in case of dirty read (read of a page with a pending write in the log), the read is correct and includes the pending writes. Our evaluation shows that, because of this redundancy, the size of the NVCACHE read cache does not influence performance, which shows that we can keep the read cache small in NVCACHE.

## V. RELATED WORK

Non-volatile memory solutions have been extensively studied. The recent introduction in the market of DRAM-based PMEM (*i.e.*, NVDIMM Optane DC persistent memory modules [7]) initiated a new stream of research exploiting the benefits of these persistent units, as well as novel commercial offerings [22]. Studies highlighted their performance trade-offs [30], [59], some of the compromises for porting legacy applications to the official Intel PMDK [55], as well as the substantial engineering efforts to port complex systems such as Redis and memcached [17] or LevelDB [40].

Interestingly, the impact of non-volatile memories has been evaluated in the context of databases [10], [2], also including approaches with flash memories [34] or as direct accelerators for main memory [45]. As shown in our evaluation, NVCACHE can transparently boost complex DBMS systems, *e.g.*, SQLite or RocksDB, without changes to their source code. To achieve this transparency, NVCACHE intercepts and redirects in user-space I/O function calls, similar to what SplitFS [33] does using Quill [25], an automatic system call wrapper (similar in essence to what we did in our prototype).

A plethora of file systems and I/O boosters have been designed and implemented for non-volatile memory. Table I (see §I) presents an analysis of recent research efforts in the field. To begin, some of the systems implement a file system only tailored for NVMM, either by porting an existing file system to NVMM [52], *e.g.*, Ext4-DAX [20], [56], or by adapting existing file systems to better leverage NVMM, *e.g.*, NOVA [57] or SplitFS [33]. Today, because of high prices,<sup>7</sup> NVMM comes with a much smaller capacity than mass storage devices such as SSD or HDD, making its use for large workloads currently unrealistic. Other systems propose to combine NVMM and mass storage to offer larger storage space. They have either the goal of improve reliability [60], [13] or of boosting I/O performance [37], [53], [18], [50], [61], [16], [42], [48]. These systems require either modifications

in the kernel (hard to maintain) or in the application (hard to apply), or new interfaces between the kernel and the user space (which increase the attack surface of the kernel). With NVCACHE, we show that we can boost I/O performance without modifying the kernel or the applications, and without requiring coordination between the kernel and the user space through new kernel interfaces. DM-WriteCache is a Linux kernel project that boosts I/O performance, but without requiring new kernel interfaces. While most of the other systems can efficiently provide new correctness guarantees such as synchronous durability, our evaluation shows that this is not the case with DM-WriteCache because the write cache is implemented behind the kernel page cache. With NVCACHE we show that, by implementing the write cache on top of the kernel page cache (in user land in our case, but we could also implement the write cache in the upper layers of the kernel), we can both boost write performance and efficiently provide advanced correctness guarantees.

Instead of exploiting NVMM for file systems, persistent memory can be used directly using `load` and `store` instructions. Persistent transactional memory (PTM) libraries use transactions to guarantee a consistent state in the event of a non-corrupting failure [28]. Typically they intercept every `load` and `store` of the application, using one of the following three techniques to guarantee consistency: redo-log, undo-log or shadow data [54], [51], [49], [21]. Most legacy applications that persist data were designed to write to the file system and modifying those applications to use the PTMs described previously would require substantial re-engineering.

## VI. CONCLUSION

NVCACHE is a write cache in NVMM that improves the write performance of legacy applications. Unlike with persistent libraries, NVCACHE allows legacy applications to use NVMM without any redesign or code modifications. Our evaluation shows that NVMM performs as well as Ext4 configured as a direct access file system, and in most cases on par with NOVA, but without restricting the storage space to the NVMM size.

**Experimental reproducibility.** We encourage experimental reproducibility. The code is freely available at <https://github.com/Xarboule/nvcache>.

## ACKNOWLEDGMENTS

This work received funds from the Swiss National Science Foundation (FNS) under project PersiST (no. 178822).

<sup>7</sup>According to Google Shop in September 2020, 1 GB of NVMM remains roughly 100× more expensive than 1 GB of SATA SSD.

## REFERENCES

- [1] [https://pubs.opengroup.org/onlinepubs/9699919799.2016edition/functions/V2\\_chap02.html](https://pubs.opengroup.org/onlinepubs/9699919799.2016edition/functions/V2_chap02.html), section 2.9.7. Accessed: 2021-03-26.
- [2] “Non-volatile memory SQL server tail of log caching on NVDIMM,” <https://docs.microsoft.com/en-us/archive/blogs/bobsq/just-runs-faster-non-volatile-memory-sql-server-tail-of-log-caching-on-nvdimm>.
- [3] “SQLite,” <https://www.sqlite.org>, accessed: 2020-05-25.
- [4] “Trees I: Radix trees,” <https://lwn.net/Articles/175432/>, 2006.
- [5] “Intel 3D XPoint™ Technology,” <https://intel.ly/2XBUk4M>, 2019.
- [6] “Micron 3D XPoint™ Technology,” <https://www.micron.com/products/advanced-solutions/3d-xpoint-technology>, 2019.
- [7] “Intel® Optane™ DC Persistent Memory,” <https://intel.ly/2WFisT8>, 2020.
- [8] H. Akinaga and H. Shima, “Resistive random access memory (ReRAM) based on metal oxides,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.
- [9] S. Akyürek and K. Salem, “Adaptive Block Rearrangement,” *ACM Trans. Comput. Syst.*, vol. 13, no. 2, p. 89–121, May 1995.
- [10] J. Arulraj, A. Pavlo, and S. R. Dulloor, “Let’s talk about storage & recovery methods for non-volatile memory database systems,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 707–722.
- [11] A. Athmanathan, M. Stanisavljevic, N. Papandreou, H. Pozidis, and E. Eleftheriou, “Multilevel-cell phase-change memory: A viable technology,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 1, pp. 87–100, 2016.
- [12] J. Axboe, “Fio-flexible I/O tester synthetic benchmark,” <https://github.com/axboe/fio>, accessed: 2020-05-25.
- [13] M. Baker, S. Asami, E. Deprit, J. Ousetterhout, and M. Seltzer, “Non-Volatile Memory for Fast, Reliable File Systems,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, p. 10–22.
- [14] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla *et al.*, “Phase change memory technology,” *Journal of Vacuum Science & Technology B: Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.
- [15] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, “Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook,” in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 209–223. [Online]. Available: <https://www.usenix.org/conference/fast20/presentation/cao-zhichao>
- [16] J. Chen, Q. Wei, C. Chen, and L. Wu, “FSMAC: A file system metadata accelerator with non-volatile memory,” in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2013, pp. 1–11.
- [17] B. Choi, P. Saxena, R. Huang, and R. Burns, “Observations on porting in-memory kv stores to persistent memory,” 2020.
- [18] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O through Byte-Addressable, Persistent Memory,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 133–146.
- [19] N. Copa *et al.*, “Alpine Linux,” accessed on March, 2020. [Online]. Available: <https://alpinelinux.org/>
- [20] J. Corbet, “Supporting filesystems in persistent memory,” *Linux Weekly News*, 2014.
- [21] A. Correia, P. Felber, and P. Ramalheite, “Romulus: Efficient Algorithms for Persistent Transactional Memory,” in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 271–282.
- [22] G. Crump, “Enabling intel optane for modern applications – memverge briefing note,” 2019. [Online]. Available: <https://storageswiss.com/2019/04/10/enabling-intel-optane-for-modern-applications-memverge/>
- [23] Docker, Inc., “Docker,” accessed on March, 2020. [Online]. Available: <https://www.docker.com/>
- [24] E. Doller, “Phase change memory and its impacts on memory hierarchy,” 2009.
- [25] L. A. Eisner, T. Mollov, and S. J. Swanson, “Quill: Exploiting fast non-volatile memory by transparently bypassing the file system,” Department of Computer Science and Engineering, University of California, San Diego, Tech. Rep., 2013.
- [26] R. Felker *et al.*, “musl libc,” accessed on March, 2020. [Online]. Available: <https://musl.libc.org/>
- [27] F. T. Hady, A. Foong, B. Veal, and D. Williams, “Platform storage performance with 3D XPoint technology,” *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.
- [28] J. Izraelevitz, T. Kelly, and A. Kolli, “Failure-atomic persistent memory updates via justdo logging,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 427–442.
- [29] J. Izraelevitz, H. Mendes, and M. L. Scott, “Linearizability of persistent memory objects under a full-system-crash failure model,” in *International Symposium on Distributed Computing*. Springer, 2016, pp. 313–327.
- [30] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module,” 2019.
- [31] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, “I/O stack optimization for smartphones,” in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC ’13. USA: USENIX Association, 2013, p. 309–320.
- [32] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, “3D-stackable crossbar resistive memory based on field assisted superlinear threshold (FAST) selector,” in *2014 IEEE international electron devices meeting*. IEEE, 2014, pp. 6–7.
- [33] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, “SplitFS: Reducing software overhead in file systems for persistent memory,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. New York, NY, USA: ACM, 2019, p. 494–508.
- [34] W.-H. Kang, S.-W. Lee, and B. Moon, “Flash-Based Extended Cache for Higher Throughput and Faster Recovery,” *Proceedings of the VLDB Endowment*, vol. 5, no. 11, 2012.
- [35] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, “Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD 14. New York, NY, USA: Association for Computing Machinery, 2014, p. 529540.
- [36] H. Kim, N. Agrawal, and C. Ungureanu, “Revisiting storage for smartphones,” *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, pp. 1–25, 2012.
- [37] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, “Strata: A Cross Media File System,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: ACM, 2017, p. 460–477.
- [38] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 2–13. [Online]. Available: <https://doi.org/10.1145/1555754.1555758>
- [39] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-change technology and the future of main memory,” *IEEE MICRO*, vol. 30, no. 1, pp. 143–143, 2010.
- [40] L. Lersch, I. Oukid, I. Schreter, and W. Lehner, “Rethinking DRAM caching for LSMs in an NVRAM environment,” in *Advances in Databases and Information Systems*. Springer, 2017, pp. 326–340.
- [41] J. Liu and S. Chen, “Initial experience with 3D XPoint main memory,” in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019, pp. 300–305.
- [42] Y. Liu, H. Li, Y. Lu, Z. Chen, N. Xiao, and M. Zhao, “HasFS: optimizing file system consistency mechanism on NVM-based hybrid storage architecture,” *Cluster Computing*, 2019.
- [43] L. Love, “Linux System Programming, 2nd Edition.” O’Reilly Media, Inc., 2013. [Online]. Available: <https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch04.html>
- [44] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “WiscKey: Separating Keys from Values in SSD-Conscious Storage,” *ACM Trans. Storage*, vol. 13, no. 1, Mar. 2017.

- [45] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *ACM Trans. Storage*, vol. 4, no. 2, May 2008.
- [46] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. USA: USENIX Association, 2012, p. 12.
- [47] E. Mumolo and M. Savron, "Reducing disk I/O times using anticipatory movements of the disk head," *Journal of systems architecture*, vol. 50, no. 1, pp. 17–33, 2004.
- [48] D. Niu, Q. He, T. Cai, B. Chen, Y. Zhan, and J. Liang, "XPMFS: A new NVM file system for vehicle big data," *IEEE Access*, vol. 6, pp. 34 863–34 873, 2018.
- [49] PMDK team, "Persistent Memory Development Kit," <https://pmdk.io/pmdk/>, 2018.
- [50] S. Qiu and A. L. Narasimha Reddy, "NVMFS: A hybrid file system for improving random write in nand-flash SSD," in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2013, pp. 1–5.
- [51] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, "OneFile: A Wait-Free Persistent Transactional Memory," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 151–163.
- [52] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–14.
- [53] R. Tadakamadla, M. Patocka, T. Kani, and S. J. Norton, "Accelerating Database Workloads with DM-WriteCache and Persistent Memory," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 255–263. [Online]. Available: <https://doi.org/10.1145/3297663.3309669>
- [54] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.
- [55] W. Wang and S. Diestelhorst, "Quantify the Performance Overheads of PMDK," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: ACM, 2018, p. 50–52.
- [56] M. Wilcox, "Add support for NV-DIMMs to ext4," <https://lwn.net/Articles/613384>, 2014.
- [57] J. Xu and S. Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. USA: USENIX Association, 2016, p. 323–338.
- [58] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, pp. 1–20, 2013.
- [59] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX, Feb. 2020, pp. 169–182. [Online]. Available: <https://www.usenix.org/conference/fast20/presentation/yang>
- [60] Yiming Hu, Qing Yang, and T. Nightingale, "RAPID-cache-a reliable and inexpensive write cache for disk I/O systems," in *Proceedings 5th International Symposium on High-Performance Computer Architecture*, Jan 1999, pp. 204–213.
- [61] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX, Feb. 2019, pp. 207–219. [Online]. Available: <https://www.usenix.org/conference/fast19/presentation/zheng>