CCF Transactions on High Performance Computing (2020) 2:16–35
https://doi.org/10.1007/s42514-020-00019-8

REGULAR PAPER

Virtualize and share non-volatile memories in user space

Chih Chieh Chou[1] · Jaemin Jung[2] · A. L. Narasimha Reddy[1] · Paul V. Gratz[1] · Doug Voigt[3]

Received: 15 June 2019 / Accepted: 22 January 2020 / Published online: 24 February 2020

Abstract
Emerging non-volatile memory (NVM) has attractive characteristics such as DRAM-like low-latency together with the non-volatility of storage devices. Recently, byte-addressable, memory bus-attached NVM has become available. This paper addresses the problem of combining a smaller, faster byte-addressable NVM with a larger, slower storage device, such as SSD, to create the impression of a larger and faster byte-addressable NVM which can be shared across multiple applications concurrently. In this paper, we propose vNVML, a user space library for virtualizing and sharing NVM. vNVML provides for applications transaction-like memory semantics that ensures write ordering, durability, and persistency guarantees across system failures. vNVML exploits DRAM for read caching to improve performance and potentially to reduce the number of writes to NVM, extending the NVM lifetime. vNVML is implemented in C and evaluated with realistic workloads to show that vNVML allows applications to share NVM efficiently, both in a single OS and when docker-like containers are employed. The results from the evaluation show that vNVML incurs less than 10% overhead while providing the benefits of an expanded virtualized NVM space to the applications, and allowing applications to safely share the virtual NVM.

Keywords Non-volatile memory · User space library · Virtualization · Transactional semantics · Concurrent accesses

1 Introduction

Emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM) (Lee et al. 2010), NV-DIMM (Narayanan and Hodson 2012), and 3D-XPoint (Intel 2019), will dramatically shake up future system designs (Dulloor et al. 2014; Kwon et al. 2017; Liang et al. 2016; Yang et al. 2015; Zhang et al. 2015). In particular, not only do these NVM technologies promise much faster access times than existing NAND-based SSDs, within an order of magnitude of DRAM, but they also are "byte" addressable and will be placed directly on the memory buses. Furthermore, these NVM technologies could be used to replace existing permanent storage devices or even volatile memory (i.e. single level system).

To date there have been some significant works in this domain. Some prior works, such as (Condit et al. 2009; Dulloor et al. 2014; Kwon et al. 2017; Qiu and Reddy 2013; Wu and Reddy 2011; Xu and Swanson 2016), engineer novel file systems tailored for exploiting NVM. Other prior works, such as (Venkataraman et al. 2011; Yang et al. 2015), employ NVM as the only media in their (single level) system and carefully design their data store manipulation mechanism to directly access some data structures stored in NVM. Their aim is to maximize performance by eliminating unnecessary data movement between volatile memory and persistent storage devices. These prior schemes, however, currently present no way to virtualize and share persistent NVM among multiple applications and users.

Traditionally, there are two common ways for applications to access data content in storage devices. One is through the file system `read/write` interface, the other is via the memory mapped file (`mmap`) interface. The cost of system

✉ Chih Chieh Chou
ccchou2003@tamu.edu

Jaemin Jung
j.jaemin@samsung.com

A. L. Narasimha Reddy
reddy@tamu.edu

Paul V. Gratz
pgratz@tamu.edu

Doug Voigt
doug.voigt@live.com

[1] Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA

[2] Samsung Semiconductor, San Jose, CA, USA

[3] Boise, ID, USA

calls incurred by accessing through the file system, however, would squander the low-latency as well as performance provided by NVM. Thus, to attain the maximum gain from NVM, in this paper, we focus on memory mapped file access form, which is also the recommended form by SNIA (2017).

Currently, when files on storage devices are `mmapped` to volatile memory (DRAM), the POSIX interface can support *shared* `mmap`; that is, the same region of files can be shared/accessed between multiple processes. Also, thanks to the swapping mechanism provided by virtual memory, which writes dirty pages to the backend storage devices (swap space) and therefore produces clean pages for the future use, the impression of physical available DRAM is extended. These two attractive properties of existing volatile virtual memory, however, are not supported in the prior work for NVM.

This paper considers this problem of virtualizing and sharing byte-addressable NVM across multiple applications. Here we introduce "vNVML", an efficient library for virtualizing and sharing NVM in user land. What we mean by "sharing NVM" here is that not only can the same physical NVM pages be reallocated and reused across users, but the data content on NVM pages can also be seen/accessed by applications concurrently, exactly like *shared* `mmap` access form of virtual memory.

One of the main aims of vNVML is to provide the impression of larger NVM availability to applications, much like virtual memory allowing the use of more main memory than the actual physical memory in the machine. In order to virtualize NVM in this manner, this paper examines extending a smaller amount of byte-addressable NVM with larger, traditional storage devices.[1]

Further, we examine mechanisms to safely leverage DRAM as cache to improve the performance of persistent memory access. There are certain advantages in employing DRAM even when the applications access virtual NVM. First, DRAM may have better performance (lower latency) than most types of NVM, except for NV-DIMM. Second, DRAM may alleviate lifetime issues of NVM in read-intensive workloads, since many NVM technologies have write endurance limits. In our design, some reads can be served by reading pages from storage devices to DRAM, bypassing the NVM entirely. This might be a better design choice compared to simply employing NVM as both read and write cache in terms of reducing the number of NVM write accesses.

We design and implement vNVML with the hope that programmers could access (virtual) NVM with a similar interface as for existing memory mapped files. That is, after a file on the storage device is `mmapped` as a virtual NVM region, a pointer to this region is returned. This pointer can be directly used in the programs as a typical `mmapped` pointer to virtual memory. However, when NVM is exploited as permanent storage by applications, the durability and ordering of writes must be assured. Write ordering as required by byte-addressable NVM has been discussed almost in every prior work (Coburn et al. 2011; Condit et al. 2009; Giles et al. 2015; Venkataraman et al. 2011; Volos et al. 2011; Yang et al. 2015). Many approaches have been proposed to address the write ordering problem within persistent memories, including hardware capacitors to ensure eviction order of all data from volatile memory to NVM (Condit et al. 2009), epoch-based writes (Coburn et al. 2011; Condit et al. 2009), transaction-like semantics (Coburn et al. 2011; Giles et al. 2015; Memaripour et al. 2017; Volos et al. 2011), versioning (Venkataraman et al. 2011), and special data stores and algorithms designed for single level NVM (Venkataraman et al. 2011; Yang et al. 2015). Here vNVML proposes to use transaction-like semantics to guarantee the write ordering, atomicity, and durability of NVM accessing.

To sum up, vNVML employs DRAM as cache, NVM as log buffer and write cache, and the backing storage device as the final destination of writes. From our evaluations, vNVML incurs less than 10% throughput overhead, if the cache system of vNVML can absorb the write traffic, compared to directly accessing NVM without the atomicity and write ordering guarantees.

The contributions of this paper are as follows:

- Propose a transactional interface for virtualizing and sharing persistent non-volatile memory.
- An implementation of this interface in our virtualized NVM Library, vNVML.
- This implementation leverages caching in DRAM, coupled with write logging and caching in NVM and lazy writeback to the backing storage to provide a high performance, virtualized, and shareable NVM to applications.
- We evaluate this proposed vNVML under not only synthetic but also realistic (YCSB+MongoDB) workloads and show that vNVML is competitive with prior techniques which do not support virtualization and sharing of NVM.

The remainder of this paper is organized as follows. Section 2 describes background and prior work in this area. Section 3 presents a design overview and discusses the design decisions of vNVML. Section 4 explains the implementation of vNVML in detail. Section 5 presents our results of evaluation of vNVML and Sect. 6 concludes.

[1] Note: while our approach can be applied with magnetic disks as the backing stores, here we limit ourselves to SSDs.

2 Background

Much of the early work to-date incorporating NVM in systems assumes basic hardware changes. New memory controllers are proposed (Doshi et al. 2016; Qureshi et al. 2009; Zhou et al. 2009). Kiln (Zhao et al. 2013) proposes a victim cache for buffering and Atom (Joshi et al. 2017) deploys a hardware logging approach to eliminate software logging overhead. BPFS (Condit et al. 2009) develops a new "epoch" for write ordering. While these approaches show promise, they require significant hardware redesign, which may take several years to be reflected in commercial hardware.

In the more near term, we might prefer to the solutions requiring little or no change to the basic processor caching and memory management hardware because memory bus-attached NVM DIMMs have become available. For example, recently Intel has released its Optane DC Persistent Memory DIMM (Intel 2019). Near-term systems will incorporate this type of NVM attached directly on the memory bus, where it will be accessible via the system's physical address space. These system architectures argue for a pure-system software approach to management.

Existing work to-date looking at system software approaches to managing this (memory bus-attached) form of NVM primarily focuses on constructing new file systems to handle the underlying NVM, such as SCMFS (Wu and Reddy 2011), NVMFS (Qiu and Reddy 2013), BPFS (Condit et al. 2009), PMFS (Dulloor et al. 2014), NOVA (Xu and Swanson 2016), STRATA (Kwon et al. 2017), FRASH (Jung et al. 2010), and Aerie (Volos et al. 2014). Some of those works have considered building file systems across multiple types of NVM and storage technologies. These include NVMFS (Qiu and Reddy 2013) (NVM and SSD) and Strata (Kwon et al. 2017) (DRAM, NVM, SSD, and HDD). The design concept of Strata is close to our vNVML; both of them contain DRAM and NVM as caches. However, the DRAM of Strata only caches the pages read from SSDs and HDDs and all updates would go directly to NVM only. Therefore, Strata needs to search for the up-to-date data locations. In addition, Strata does not support memory mapped files access form, which is the primary form used by our target applications.

Accessing NVM through file system APIs has fundamental drawbacks. First, accessing NVM via the file system interface is not suitable for random, small accesses to NVM due to the high (context switch) overheads incurred by system calls compared with the relative low-latency that NVM offers. Next, the file system naturally cannot support concurrent accesses of the same file by different threads or processes, even though those threads or processes access different objects/pages in the same file. For example, to access a certain location of a file, users must acquire a file lock first, and then `seek` to the location and `read/write` from and to that location of file. After `read/write` command is completed, the file lock can be released. Another drawback is that such (file system) software approaches require users and applications to deploy dedicated file systems.

Some works, such as NV-Tree (Yang et al. 2015) and CDDS-Tree (Venkataraman et al. 2011), consider replacing DRAM and storage devices entirely with NVM to construct a single level, NVM-only system; their idea is to manipulate all data structure operations directly on NVM and therefore eliminate all data movements between DRAM and storage devices. Such approaches can improve performance significantly; however, they restrict themselves to only some specific data structures and cannot be easily applied to general memory access. Also, they totally ignore the lower latency offered by DRAM and do to further explore the potential performance improvements. For example, their approaches might be not suitable for the read-intensive, cache friendly workloads.

SPAN (Fedorov et al. 2017) proposes some new swapping enhancements in the operating system (OS) kernel to exploit NVM as extended system memory. Its concept is similar to the memory mode supported by Intel's Optane DC Persistent Memory (Watts 2019). NV-Heaps (Coburn et al. 2011) provides some useful features, such as type-safe pointers and garbage collection, but it requires programmers to use its specific object framework and hardware must support epoch as BPFS (Condit et al. 2009) does.

Other approaches try to create user space libraries (Giles et al. 2015; Intel 2015; Memaripour et al. 2017; Volos et al. 2011). PMDK (Intel 2015) and KAMINO (Memaripour et al. 2017) employ only NVM and utilize undo logging to support in-place data updates. SoftWrAP (Giles et al. 2015) combines NVM and DRAM and employs DRAM for write/read accesses and NVM for redo logging. However, during the normal operations, the data content is "retired" from DRAM to the final locations in NVM, and the logs in NVM are referenced only after system failures. The most closely related work to our vNVML presented here is Mnemosyne (Volos et al. 2011), which also uses *redo* logging. While Mnemosyne provides both persistent region and persistent heap allocation methods, vNVML does not support heap style allocation. However, there are some fundamental differences between these two approaches. Mnemosyne achieves NVM virtualization by swapping, which is controlled entirely by the kernel and therefore suffers from context switching overhead in the page fault critical path; vNVML, on the other hand, always writes the dirty pages back to storage devices by a background thread. Further, Mnemosyne does not employ DRAM as read cache, so it requires an extensive search to find up-to-date data. Also, Mnemosyne cannot support true sharing of NVM between

Table 1 Summary of prior software approaches

Categories	Approaches
File system	SCMFS, NVMFS, BPFS, PMFS, NOVA, STRATA, FRASH, Aerie
Single level system	CDDS-Tree, NV-Tree
Persistent object system	NV-heaps
User space library	mnemosyne, PMDK, SoftWrAP, KAMINO

processes. Table 1 summaries above-mentioned software approaches.

In this work, we propose a system software-based, user space management approach, which makes NVM available directly to user applications, without the block-level semantics of traditional file systems. Furthermore, our work also provides write ordering, atomicity, and endurance guarantees while offering a larger than physical available NVM space to the applications, and allows them to safely share the virtual NVM regions while maintaining performance goals by leveraging caching in DRAM.

3 Design overview

In this section we describe our design and provide an overview of the decisions made in the design of the virtual NVM Library (vNVML), a user space library for virtualizing and sharing NVM. Our design decisions are guided by the following four observations.

First, *persistent memory is typically allocated and dedicated to an application*. For example, when file system writes data to a location in persistent memory, that location cannot be reused or reallocated by another application; otherwise, the data content of that location is corrupted. If NVM is similarly allocated and used, then NVM cannot be easily shared[2] across multiple applications if NVM is the only persistent storage device in the systems. In data centers, with dynamic workloads, there is a strong desire to share available resources across many applications. It is essential that we provide mechanisms to share precious resources like NVM across many applications.

Second, *simply replacing traditional storage devices, such as solid-state drives (SSDs) or hard drives (HDDs), with NVM is not good enough*. Although by doing so, all existing applications can benefit immediately from performance improvements provided by NVM without any modifications required; however, this ignores the byte-addressability of NVM, and requires accessing NVM in units of blocks, resulting in a suboptimal approach.

Third, *the access latency of NVM is very close to that of DRAM and is much faster than that of storage devices*. When storage devices are slow (for example magnetic disks), the overheads paid by accessing through system calls (or context switching) from file system APIs may not be a significant part of the entire access latencies. However, as devices get faster, like NVM whose latency is within an order of magnitude of DRAM, these system call overheads become much more significant and hence must be avoided. For example, Intel Storage Performance Development Kit (SPDK) (Intel 2017) implements the whole NVMe device driver in the user space and thus improves the accessing Ultra-Low-Latency (ULL) SSDs, such as Intel NVM-based 3D-Xpoint or Samsung Z-NAND (Samsung 2017), performance significantly.

Finally, while it is possible (and even desirable) to continue running existing or older software on new hardware, *software may have to be redesigned/rewritten to attain the most of the hardware*. This can take the forms of new file systems, new data stores (Venkataraman et al. 2011; Yang et al. 2015) or new libraries (Eisner et al. 2013; Giles et al. 2015; Intel 2015; Memaripour et al. 2017; Volos et al. 2011). In this paper, we take the approach of developing a user space library interface to NVM to achieve our goals.

Based on above four observations, we designed the vNVML user space library, integrating DRAM, NVM, and backend storage devices to construct the abstraction of virtualized NVM. Like virtual memory, adopted almost universally in the modern computer systems, the main idea of our virtual NVM is to provide the impression that applications and users can treat (virtual) NVM contained in their system as large as the capacity of the storage devices in the system and as fast as the speed of NVM (or even DRAM).

Usually, applications have two means to access the data content stored on storage devices; one is through file system `read/write` commands, the other is through memory mapped files (i.e. `mmap`). Because traditional disks are very slow, accessing data content in disks through file system commands, which in turn trigger some system calls, may not incur too much performance penalty compared to the long access latency of the disks themselves. However, this is not the case when we deal with NVM because of its DRAM-like low latency.

By contrast, when accessing NVM, we must avoid expensive system calls triggered by the file system commands as much as possible. This is especially true for applications requiring abundant random memory accesses, such as tree manipulation operations, which may issue several system calls solely for modifying a few pointers and therefore results in significant system overheads.

[2] Note: the “share” here means the same physical NVM pages can be reallocated/reused by multiple applications.

In order to avoid context switch overheads and to expose the byte-addressability of NVM, in this project we primarily focus on memory mapped file accesses. Here, applications access NVM much like memory, through byte-level load/store interfaces without system calls. However, to utilize the byte-addressability of persistent memory, write ordering issue must be paid special attention (Condit et al. 2009; Swift 2015). Therefore, we design vNVML such that programmers use it in much the same way as they employ existing POSIX `mmap` access for volatile memory, except that they must follow the transaction-like semantics for write ordering. Meanwhile, the benefits of performance improvement, atomicity, and durability are all provided by this transactional interface. Furthermore, we adopt the method of user space library hoping for executing NVM accesses in the user space as much as possible and using system calls only when necessary.

Since vNVML only provides the `mmap`-like transactional interface and only focuses on the file read/write accesses, meaning that vNVML still relies on file system to provide the file system related handlings, such as metadata management, directory management, file permission control, etc. vNVML places no limits on which file system may be used in the system and only requires that file system must support standard POSIX `mmap`. Here we want to highlight that vNVML does not try to place the file systems; instead, vNVML hopes to supplement and augment the current design limitations of file systems, especially when applications require the high performance computing.

From here, we focus on vNVML's *private* `mmap` access mode,[3] meaning that virtual NVM regions can only be accessed by a single process. The mechanism of our *shared*[4] `mmap` mode is slightly different and, to avoid confusion, is explained in Sect. 4.4.

Briefly speaking, vNVML utilizes NVM both as a log buffer and as a write cache and DRAM as a read cache. The reads can only be served by read caches. Modified data are first written to the NVM log buffer only and then copied to DRAM (read cache) when the logged data are committed. NVM (write cache) are updated by committed logs on the background. If pages containing accessed data are not already in NVM or DRAM, they (entire pages) are copied from the storage devices to NVM write cache or DRAM read cache. Data are evicted from the NVM write cache back to storage devices only when the usage of NVM write

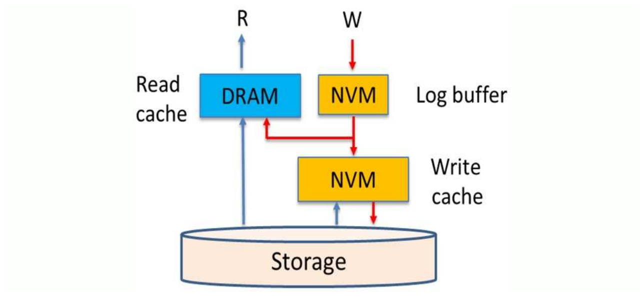


Fig. 1 The read/write data flow of vNVML private mmap mode between DRAM, NVM, and storage device

cache exceeds some threshold (30%) of the physical available NVM. Before programs are terminated safely, all data are completely flushed from NVM (both log buffer and write cache) to files. The interactions between DRAM, NVM, and storage devices are shown in Fig. 1 for both read and write operations.

The key ideas behind our design choices are as follows:

Use NVM as log buffer Like other NVM user space libraries, we also adopt transaction semantics as interface for vNVML to provide write ordering, atomicity and durability. All written data must be immediately stored at some temporary non-volatile storage locations before transactions commit. Since this logging process must be in the write critical path, employing NVM as temporary non-volatile log buffer provides significant performance advantage.

Redo logging and DRAM cache Typically, there are two approaches to logging: *undo* and *redo* logging. Both have pros and cons toward different workloads (Wan et al. 2016). *Undo* logging requires that old data are persisted as logs before new data are updated in place. These two actions (both logging and in place update) must be in the write critical path. Alternately, with *redo* logging, all new data are persisted as logs to non-volatile media first, then new data can be updated in place. In-place updates, because they can be executed on the background, do not have to be in the write critical path, but they would affect the read critical path because reads have to be redirected to logs and have to search for the newest data from logs.

In vNVML, we augment *redo* logging by using DRAM as a read cache. Modified data are written to the NVM log buffer, and are also written to DRAM, during the commit command, for reads following this write. With the help of a read cache (DRAM), only persisting writes on the (*redo*) log and updating to DRAM are in the write critical path (from the perspective of the whole transaction), and reads do not need to be redirected to logs as usual (*redo*) log does. Updating data on the storage devices can be executed in the background, without it being in the write critical path.

Although *undo* and our *redo* logging both double the written data in the write critical path (*undo*: 2 NVM versus

[3] Note: our (vNVML) *private* `mmap` mode is different from the standard POSIX *private* `mmap` mode. In our *private* `mmap`, written data can be reflected back to storage devices, but POSIX *private* `mmap` cannot.

[4] Note: the "share" here means that shared regions can be seen and accessed by multiple processes, exactly like the existing POSIX *shared* `mmap` method for volatile memory.

our *redo*: 1 NVM and 1 DRAM), using our *redo* logging still has three advantages. First, even though the access latency of NVM is close to that of DRAM, the write latency of DRAM is still shorter than that of NVM (Chen 2016; Yu and Chen 2016). So, writing to DRAM is still faster than writing to NVM. Second, writing to DRAM does not need ordering constraints, which use `clflush`, `clflushopt`, `clwb`, and `sfence` instructions and therefore are time-consuming (Zhang and Swanson 2015). Third, for read-intensive workloads, read cache can serve some reads without accessing NVM, which might potentially reduce the writes to NVM and alleviate the lifetime issues of NVM.

Only committed data are updated to read cache Before logs of uncommitted transactions can be placed into true destinations, reading the data still in the logs requires parsing the logs to find the newest data, which can be time-consuming. This process is in the read critical path. In most workloads, the frequency of reads is much higher than that of writes. For example, Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al. 2010) framework refers to workload A (50/50 read/write ratio) as update-heavy workload. In terms of the overall performance, shortening the read critical path is more important than write critical path. So, we simply use DRAM as a read cache to serve all read operations, and update the data into the DRAM in the commit command (through parsing the logs belonging to this transaction sequentially). By doing so, the following reads, after transaction commits, could read directly from DRAM. Our design doubles the written data on the write critical path, but it makes the reads faster as data can be directly read from DRAM, without having to search the entire log buffers. The section 4.1 will explain the detailed mappings of read cache, log buffer, and write cache into virtual address space of each process.

Two restrictions are related to this read cache: (1) reads can only be served by the read cache and (2) written data are copied to read cache only when the transaction commits to accomplish the isolation property; that is, only committed data are visible. This is sometimes referred as "read committed" transaction isolation level (Microsoft 2017). However, our transactions are defined differently from transactions of the traditional database systems. In database systems, the focus is on the consistency of transactions to ensure correct data are accessed between multiple concurrent transactions. In vNVML, we emphasize the persistency (Pelley et al. 2014) of transactions. Here we define the *committed* (*uncommitted*, respectively) data are that the written data must be valid (invalid, respectively) after system crashes; meanwhile, the atomicity and durability of data are also guaranteed. We leave the consistency of transactions to the discretion of programmers/applications.

Employ NVM as write cache All written data are at the log buffer when transactions commit. Data need to be gradually moved from the log buffer to their true destinations on storage devices to avoid overflowing the log buffer. We utilize part of NVM as a write cache and committed data are moved to NVM write cache before they are moved to much slower storage devices. This allows us to migrate the logs quickly to more permanent locations and to prevent the log buffer from taking too much space.

A background worker (thread) is responsible for copying data from (NVM) log buffer to (NVM) write cache to avoid extra overhead in the write critical path. This design is also suitable for the cache-friendly workloads (or the write working set of workloads is smaller than write cache size) because logs could always be directly copied to NVM cache (where data are moved from NVM to NVM). Writing data to NVM allows us to maintain data safety, providing a better performance if future writes hit in the write cache.

*Update to storage devices from NVM write cache (**private** `mmap` mode) or DRAM cache (shared `mmap` mode)* Data are written to the log buffer using the write commands, and then moved to NVM write cache in the background. These data are also written to the DRAM read cache upon the commit command being issued. Therefore, the data can have two paths, from DRAM or from NVM, to reach the storage devices. Depending on whether the regions (or modes) of virtual NVM are to be shared across applications or not, we adopt different strategies.

For private virtual NVM regions (*private* `mmap` mode), We construct DRAM (read-only) cache by leveraging the existing POSIX *private* `mmap`, which adopts Copy-on-Write mechanism and all written data can only remain at DRAM. Therefore, the data can only be written back to storage devices from NVM cache. We choose this design choice with the hope that NVM write cache can absorb all write traffic (if cache size is larger than the write working set) and avoid all storage device accesses to gain the maximum performance.

On the other hand, when shared NVM regions (*shared* `mmap` mode) are required, the existing POSIX *shared* `mmap` is adopted to construct the DRAM (read/write) cache, and data are written from DRAM to storage devices. Here NVM cache is not used and logs in NVM log buffer are referenced only when recovering from system failures is needed. This design concept is similar to SoftWrAP (Giles et al. 2015). Further details are provided in Sect. 4.4.

Figure 2 illustrates the read/write flow of accessing private virtual NVM region and the page movement between DRAM, NVM, and storage device in detail. (a) A file on the storage with page A and B initially. (b) A read from page A lets page A is copied from the storage device to the memory and then the application reads page A directly from the memory. (c) A write to page A results in a log ΔA is appended to the log buffer. (d) Another write to page B also results in a log ΔB is appended to the log buffer. (e)

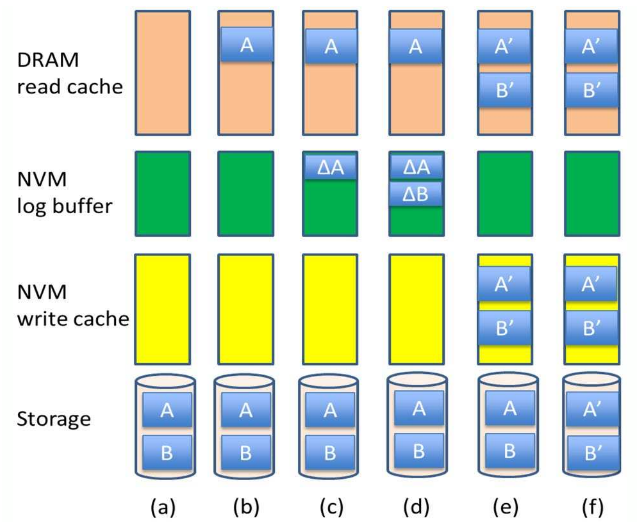


Fig. 2 The read/write data flow and page movement of vNVML private mmap mode

The transaction commits. The page A in memory is updated with ΔA to page A', and page B is copied from storage to the memory by Copy-on-Write mechanism and is also updated with ΔB to page B'. Page A and B are read from storage device to NVM cache and are applied the logs ΔA and ΔB to be page A' and B' by a *redo* background thread. (f) Another writeback background thread writes the page A' and B' from NVM cache back to the storage device.

4 vNVML API and implementation

In this section, we explain the implementation of vNVML in detail. We start from the introduction of the APIs that vNVML provides and describe their functions. Next, we describe the data structures that vNVML manages in the user space of applications, and then introduce two background workers (per process) for parallel processing in vNVML. Then, we explain the implementation of shared regions of vNVML. Finally, we discuss some general issues in vNVML implementation.

4.1 vNVML API

```
Function nv_init (void)
    if caller is the first caller then
        initialize vNVML;
        construct linked lists for NVM cache and log buffer;
    end
    mmap NVM files such as cache, log buffer, and metadata
      into caller's virtual memory space;
Function nv_release (void)
    wait for redo background worker to apply committed
      logs to NVM write cache;
    flush all dirty pages to the storage;
    munmap all NVM files;
    if caller is the last caller then
        release all resources allocated by vNVML;
        erase all NVM files;
    end
Function nv_allocate (path filepath, size n, mapping_mode
  mode)
    acquire fd by open(filepath);
    get fileptr from mmap(n, mode, fd);
    return fileptr;
Function nv_free (pointer fileptr, size n)
    munmap(fileptr, n);
Function nv_txbegin (void)
    generate a unique transaction tid;
    return tid;
Function nv_write (id tid, address dst, address src, length n)
    if log buffer is needed and no log buffer is available then
        return the number of written data;
    end
    Allocates a page from log buffer if necessary;
    Add written data from address src to src+n as log entries
      of tid to one of the open log lists;
    return the number of written data;
Function nv_commit (id tid)
    update the read cache by parsing logs of tid;
    move logs of tid from one of open log lists to the tail of a
      committed log list;
Function nv_abort (id tid)
    remove logs of tid from open log lists;
```
Algorithm 1: vNVML API.

Algorithm 1 shows all APIs that vNVML offers and their brief implementation.

Every application (process) first needs to call `nv_init` once before it starts to utilize vNVML. The first caller creates (a log buffer, a cache, along with associated meta-data) files in NVM and a shared memory object by calling `shm_open`. The first caller is also responsible for constructing one linked list for pages of NVM cache as a free list and one linked list for pages of the log buffer. Section 4.2 describes the linked list data structure in more detail. The shared memory object contains and provides global information accessible by all vNVML users such as number of total current vNVML users, unique application ids assigned to each application, and unique transaction ids for each transaction. Because NVM pages in the free list

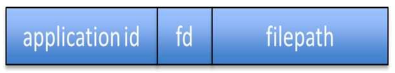


Fig. 3 Structure of a file record for recovery process

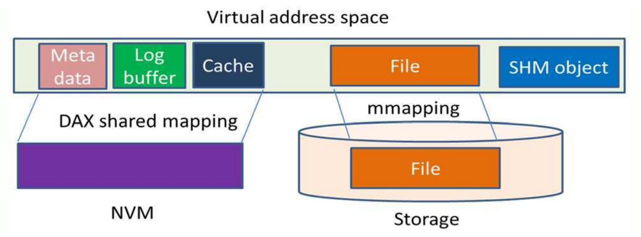


Fig. 4 The mapping of virtual address space of a process after calling `nv_init` and `nv_allocate`

and in the log buffer do not contain useful information and therefore do not relate to recovery process as well as they also need to be accessible by all applications, their linked list heads are stored in this shared memory object, too.

All callers must *shared* `mmap` all files created in NVM using the `nv_init` command in their virtual address space. These files are mapped by vNVML and are invisible to applications. Therefore, applications have no information of mapped address regions of these files, and all accesses to NVM files from applications can only be through vNVML.

To allocate virtual NVM regions, applications call `nv_allocate` by passing a path *filepath* in the storage, a file size *n*, and the mapping mode (*private* or *shared*). If a file exists in the *filepath*, then that file is opened; if it does not, a new file is created at *filepath* and is `posix_fallocated` with size *n*. The file descriptor *fd* returned from `open` command, along with application id and *filepath* are stored as a file record entry (application id, fd, filepath) of the metadata file for recovery process if needed. Figure 3 illustrates an entry of file record. Finally, A file pointer *fileptr* obtained by (*private* or *shared*) `mmapping` this file is returned to the caller.

Figure 4 illustrates the virtual address space of a process after calling `nv_init` and `nv_allocate` for a file. Only the mapping regions of files in the storage devices are known by applications.

After virtual NVM regions are allocated, applications can access virtual NVM like accessing real NVM through *fileptr* (virtual address returned from `nv_allocate`) for reading or the `nv_txbegin`, `nv_write`, `nv_write`, ..., `nv_commit` command series for writing. The `nv_txbegin` generates and returns a unique transaction *tid* for the following `nv_write(s)` and `nv_commit` commands to construct a single transaction.

The `nv_write` commands are used to write data into virtual NVM. Through `nv_write` commands, all data are written as *redo* logs in the log buffer. The first `nv_write` command must allocate a log page from log buffer. If a log page is needed but no log page is available, then `nv_write` returns the size of written data so far.

To write logs, a log object to store the log pages of this transaction is allocated from NVM and is put into one of 32 open lists of this process according to its transaction *tid*%32 (modulus operator). Log pages, allocated from the linked list of log buffer by the same transaction, are appended to the tail of the corresponding linked list of the log object in the open lists.

A single `nv_write` command may create several log entries. It first depends on the destination position and then depends on the left space of the current log page. This is because we want the data from a single log entry to be placed entirely within a single NVM cache page to simplify the design and implementation of redo background worker described at Sect. 4.3.

Inspired by some prior works (Condit et al. 2009; Wang et al. 2015; Zhang and Swanson 2015), we know when dealing with byte-addressable, memory-bus attached NVM, write ordering is required. To do so, modern CPUs provide cacheline flushing (`clflush`, `clflushopt`, and `clwb`) instructions and memory fence (`sfence` and `mfence` instructions to help to enforce write ordering. For example, `store-clflush` pair combined with `mfence` (or `sfence`) is adopted by several prior works (Dulloor et al. 2014; Qiu and Reddy 2013; Wu and Reddy 2011; Xu and Swanson 2016). Some other prior works (Giles et al. 2015; Kwon et al. 2017; Xu and Swanson 2016; Zhang and Swanson 2015) further replace the `store-clflush` pair with non-temporal store (`ntstore`) instructions. The `ntstore` instructions can bypass the CPU caches and directly write data to DRAM or NVM. By doing so, `ntstore` eliminates the expensive cacheline flushing operations (Zhang and Swanson 2015) and significantly improves the NVM persisting performance. Therefore, we also employ `ntstore` to write data to log buffer at `nv_write` commands.

When the `nv_commit` command is called, vNVML sequentially traces all log entries of log pages from the linked list of log objects for the committed transaction *tid*, and copies all committed data from log entries to the DRAM read-only cache. Because writing to DRAM does not require ordering, the standard `memcpy` function is sufficient for copying and therefore may be used to attain better performance. After copying to the DRAM cache, vNVML moves this log object (along with all log pages linked to this log object) from the corresponding open list to the tail of the only committed list of this process and persists all log entries and a log object in NVM. This committed list head is stored at the metadata of the applications in NVM. All log entries in the committed list are guaranteed to be preserved across power failures.

Finally, applications call `nv_free` (`nv_release`, respectively) if they do not want to access a certain file (do not want to access entire virtual NVM, respectively). `nv_free` is used to `munmap` a file mapped by the `nv_allocate`. After `nv_release` is called, the applications must wait for all committed logs, if they exist, to be applied to NVM cache by the redo background worker, actively flush all dirty cache pages back to the storage devices, and `munmap` all NVM files mapped at `nv_init`.

Algorithm 2 shows a typical example of using vNVML.

```
nv_init();
ptr = nv_allocate(filepath, filesize, mode);
tid = nv_txbegin();
x = 100;
y = 200;
nv_write(tid, ptr, &x, sizeof(x));
nv_write(tid, ptr+sizeof(x), &y, sizeof(y));
nv_commit(tid);
nv_free(ptr, filesize);
nv_release();
```
Algorithm 2: Example of vNVML.

4.2 vNVML data structures

The NVM log buffer and NVM cache are partitioned into units of 4KB pages and organized as linked lists. The implementation of linked lists for pages is via metadata; that is, for each page, a corresponding page object is created from NVM metadata file and connected with each other as a linked list. Therefore, allocating a page object from the linked list also equals to allocating the corresponding page.

As mentioned in (Swanson 2017), however, constructing the linked list in NVM is not the same as constructing a typical linked list in memory. The virtual address cannot be directly used as a pointer to be stored in NVM because there is no guarantee that the NVM files can be mounted into the same virtual space regions (1) by multiple concurrent processes or (2) by a single process before and after system crashes. Thus, we replace the address with the index of the page starting from 0 to construct the linked lists in NVM. Similarly, we substitute the offset from the starting address to the current position, when an access needs to be made, for the address to be stored into NVM.

Page objects for the log buffer and cache are created by different metadata files at `nv_init`. After a page object of log buffer is allocated, the application id is stored into page object, and the log entries (from `nv_write`) can be written directly into the corresponding log pages. The first field of the log page is the total written bytes to this page, and log entries are appended sequentially. Figure 5 illustrates the

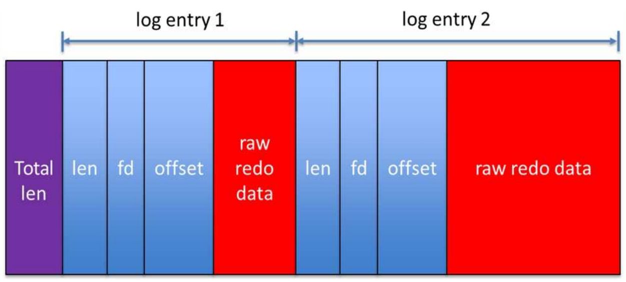


Fig. 5 Structure of a log page and its log entries

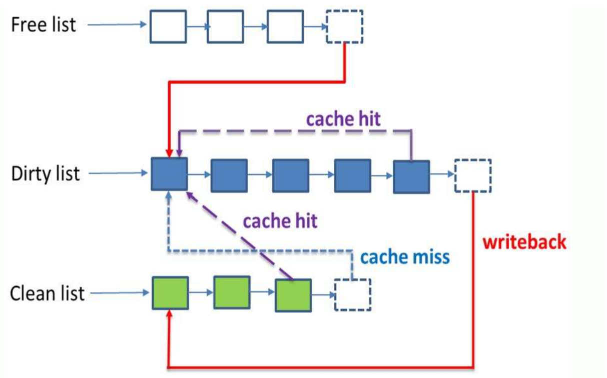


Fig. 6 Page movements of LRU policy between free list, dirty list, and clean list

fields of a log page and log entries in the log page. The log entry contains log header, including length of this entry, file descriptor, and offset (from the first byte of this file to the destination location), followed by the *redo* raw data. The page object (application id), the log entry header (len, fd, and offset), and file record (application id, fd, and filepath) already contain all necessary information for the recovery worker to write the committed log entries directly back to corresponding files of storage devices.

To handle cache pages, one free list is created through the shared memory object, and the others, dirty and clean lists, are created within each application. The dirty and clean lists implement the LRU replacement policy.

Figure 6 illustrates the page movements between the free list, dirty list, and clean list. Cache pages are always allocated from the free list, until the page share of an application is reached, and become dirty pages inserted to the head of dirty list after the redo background worker copies the corresponding pages from files in the storage devices and applies corresponding log entries on them. Dirty pages (of the tail of dirty list) become clean ones and are inserted to the head of the clean list after the writeback background worker writes the dirty pages back to files in the storage device. When a cache is hit, regardless of whether the page is in the dirty or clean list, the page is applied the redo log and inserted into

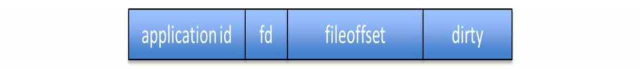


Fig. 7 Structure of a page object for writeback and recovery process

the head of the dirty list. When a cache miss happens, the tail page of clean list is always picked and is filled with the corresponding page from file in the storage devices. This page is inserted into the head of the dirty list after writing the redo log on it.

For the individual cache page, besides the application id of the page owner, some extra information is also stored into the corresponding page object, such as the fd (file descriptor), file offset, and dirty flag. The file offset is the offset which is used to `seek` the file and to access the page of files in the storage devices. The fd and file offset can be known from the header of log entries by redo worker when it redoes. The dirty flag is set only if this page is dirty (in the dirty list). This flag is cleaned after writeback worker writes this dirty page back to files and puts it into the clean list. Thus, the recovery worker only needs to handle the pages whose dirty flag are set. Also, the information contained in the page object (application id, fd, file offset, and dirty flag) and the file record (application id, fd, and filepath) are enough for recover worker to write the dirty pages back if system crashes. Figure 7 illustrates an entry of page object.

Partitioning the log buffer and cache page at a 4 KB page-size granularity and organizing them as linked lists has some advantages. First, the allocation and deallocation of pages from log buffer and free list are both *O(1)*. Second, the management of log buffer and cache space becomes easier since the space is managed in terms of pages, rather than bytes or variable size segments. Third, it makes it easier to share pages of the log buffer and free list across applications through linked lists maintained at the shared memory object.

To prevent a single application from allocating all log pages and all cache pages, vNVML adopts the equal share policy through the shared memory object, containing the total number of current applications. Applications can allocate log pages from the log buffer or cache pages from the free list if and only if the number of allocated pages does not reach their shares. A new joining user of vNVML may result in all current users exceeding their shares. Two background workers, described in the following section, help to return extra pages back to log buffer and free list.

Figure 8 illustrates the relation between open list, committed list, log object, and page object.

4.3 Background workers

Two background workers (threads) are created for each process at `nv_init`. The redo background worker keeps

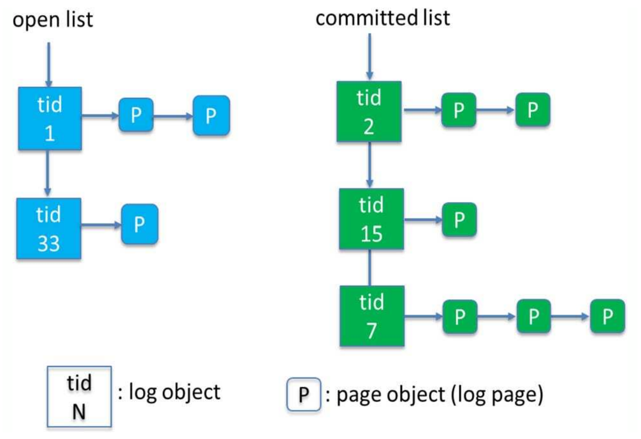


Fig. 8 Open list contains log objects of transactions. Each log object may link several page objects (of log pages). After transaction commits, the log object (along with its log pages) of the transaction is appended to the tail of the committed list. Redo worker always redoes from the head of the committed list; therefore, the transactions which is committed early would also be replayed early

checking the committed list. If the committed list is empty, the worker goes to sleep for a while (10us in the current configuration) and then checks the committed list again after it wakes up. If the committed list is not empty, the redo worker obtains the first log object (of some transaction) from the head of the committed list, and replays all the log entries sequentially from log pages of this log object to NVM cache pages. Since in this redo operation, data are moved from NVM to NVM, write ordering is also required; therefore, we also employ `ntstore` instructions here for writing to NVM cache pages. If a cache miss happens, redo worker is also responsible for reading this page from files in the storage device to NVM cache page. All log pages can be discarded and returned to the log buffer pool only after the entire logs of a transaction are completely replayed by the redo worker.

Here we want to highlight the limitation that, since the committed logs are always appended to the tail of the only one committed list (per process) at `nv_commit` (mentioned at Sect. 4.1), and redo background worker always redoes logs from the head of this committed list; therefore, programmers should keep in mind that the transaction which writes to an object first should also be committed first for the consistency of DRAM cache and the file on the storage device when multiple threads write to the same objects, but there is no constraint when threads write to different objects. This is a much weaker constraint compared to accessing a file by file system, which requires locking the whole file, even if different objects of the same file are being accessed.

The other writeback background worker is responsible for writing the dirty NVM cache pages back to the storage devices. To avoid accessing storage devices too frequently, we employ a threshold on dirty NVM pages accumulated in the dirty list (we use 30% of cache page share). Dirty pages are written back to the storage device by the writeback worker after the number of dirty pages is more than threshold. The dirty pages are inserted to the head of the clean list after writing back to storage devices. Furthermore, if the number of allocated cache pages exceeds the share due to new joining applications, the writeback worker may further release some clean pages back to the free list. After the number of the dirty pages drops below some threshold (we set 10% of cache page share), the writeback background worker is stopped and dirty pages may accumulate again.

Both background workers are killed upon the `nv_release` command.

4.4 Sharing NVM between processes

The implementation of shared NVM regions between processes is slightly different from what we have implemented for private regions. What we mean by "shared region" here is that a memory region can be accessed concurrently by multiple processes and all processes could see the same view of this region. This is exactly the same as existing POSIX *shared* `mmap` for volatile memory.

Shared regions are constructed by `nv_allocate` command. When `nv_allocate` is called, a file is *shared* `mmapped` to construct the DRAM read/write cache and a committed list head is created and maintained as metadata in NVM for each *shared* `mmapped` region. By DRAM cache constructed from *shared* `mmap` of the file, processes which require to access this region can share the same view.

Because a committed list is required for each shared region, the same limitation, the transaction writing to an object first should also be committed first, must also be obeyed. In addition, one more limitation is required when writing to shared regions; that is, all true destinations of writes from a single transaction must lie within the same single shared region. Without this limitation, if a transaction can write to multiple shared regions, then multiple log objects, each log object is for a shared region, should be created to be appended to multiple committed lists when transaction commits. This would cause a serious problem if system crashes while a transaction commits, resulting in some log objects have been appended to some committed lists but some not.

When we write to a shared region, data are still written to log buffer first. At transaction commits, the data from log buffer are replayed to the DRAM of a single *shared* `mmap` region, and all logs of this transaction are moved from one of 32 open lists of a process to the committed list of this shared region. We do not utilize NVM cache here to simplify our design and implementation. Figure 9 illustrates the read/write data flow of accessing a shared virtual NVM region.

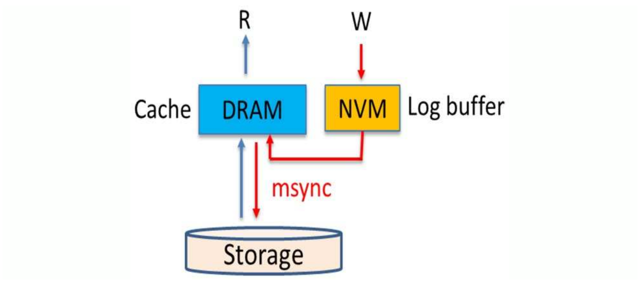


Fig. 9 The read/write data flow of vNVML shared mmap mode between DRAM, NVM, and storage device

Since *shared* `mmap` is employed to create our shared regions, the `msync` system call is required to write the modified data back to storage devices. Different from `fsync`, however, calling `msync` requires to know all the modified virtual address regions (i.e. start addresses and their lengths). To avoid parsing all logs when flushing (`msync`) data back to storage devices later, a global bitmap of dirty pages dedicated to this shared region is maintained in DRAM and is updated at the end of each `nv_commit` command by the local bitmap of each transaction; the local bitmap is constructed when logs are parsed and replayed to DRAM at `nv_commit`.

After logs have been accumulated to a certain threshold, upon a call to `nv_commit`, a background thread is triggered and atomically copies the global bitmap of this shared region to a "copied global bitmap" variable, zeroes this global bitmap, and marks the tail transaction of the committed list. Then several `msyncs` might be issued to flush dirty pages back to storage devices according to the copied global bitmap. Only after flushing is completed, can logs of transactions (from the head to the marked tail) be removed from the committed list.

During the flushing procedure, new transactions can still proceed, be committed, and update the "zeroed" global bitmap. Because their logs are always appended after the marked tail transaction and "read committed" policy is employed in vNVML, unexpectedly flushing some data of transactions committed after "marked tail" transaction during the flushing procedure will not harm since all data in DRAM must have been committed and therefore must be written back to storage devices eventually.

The logs from head transaction to marked tail transaction are discarded only after flushing is completed. If system crashes, recovery procedure always replays all logs from the head of committed list of this shared region.

4.5 Transaction aborts and long running transactions

For some extreme cases, the log buffer may run out of space if too many long running transactions, which keep writing data before commitment, execute concurrently. This situation can be detected when pages cannot be allocated from the log buffer pool for a while. vNVML could actively abort long running transactions by recording the timestamp into the log objects when log objects are allocated by transactions. The redo worker can periodically check the log objects from the head of each open lists and can abort the transactions whose elapsed time exceed some predefined threshold. Applications can also abort transactions for various reasons.

When a transaction is aborted, since all its logs are still in an uncommitted state (in the open list), these logs can be discarded directly and log pages are returned back to the log buffer pool. Moreover, because transactions of vNVML support the "read committed" isolation property, when one of the nested transactions needs to be aborted, aborting all transactions involved in the nested transactions may not be necessary and it should depend on the discretion of users.

4.6 Data recovery

Systems or applications may crash due to an unexpected failure at any moment such as power shortage, bugs of applications, or inadequate kernel resources. The mandatory function any NVM solution should provide is to ensure the data persistency after systems or applications crash. In vNVML, we handle this by a recovery program run by *root*. After systems reboot, a recovery worker (process) first `mmaps` the all NVM files (log, cache, and metadata) into its virtual memory space. From Sect. 4.2 we know the recovery worker already has all necessary information to recover the dirty pages and log entries back to files in storage devices by tracing the page objects, file records, and committed lists.

We always recover/write the dirty pages (by checking if dirty bit is set) back to files before we recover the committed logs back to files because the committed logs contain the newest data. Reversing this order might result in that the new data are covered by older data from dirty pages.

The order of objects in the committed list is important and we should replay the objects sequentially. With the help of 8-byte atomic update feature natively supported by processors, the order of objects can be maintained correctly by carefully handling the order of pointer updates between objects of linked lists.

Figure 10 illustrates the process of insertion and deletion of an object to and from a linked list at NVM. For object insertion [the correct sequence is from (a) to (b) to (c)], we could assume object C has been inserted into linked list only when system crashes after (c); otherwise, we assume object C is not inserted yet. On the other hand, for object deletion [the correct sequence is from (c) to (b) to (a)], when system crashes after (c) we would assume object C has been deleted from the linked list.

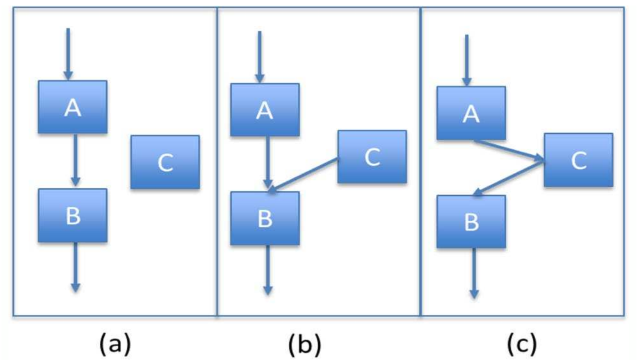


Fig. 10 The correct order of pointer updates for the objects of linked lists in NVM. From **(a)** to **(c)** is for the object insertion; from **(c)** to **(a)** is for object deletion

After the recovery process finishes, all NVM files are erased, and vNVML can be restarted again. This recovery process may be re-executed as many times as needed if the system ever crashes again during the recovery process since all the required data and metadata are conserved in NVM and are erased only after a successful recovery. Thus, all data have been written back to their true destination of files.

4.7 Security

Security is a major concern in the modern computer systems, especially in the data center, where infrastructure has to protect against any attacks from third party applications. In vNVML, the security is guaranteed in two aspects. First, the private regions are produced by *private* `mmap`. Due to the Copy-on-Write mechanism brought from *private* `mmap`, all the direct writes within this private address region will remain within the memory (virtual address space of the user process) and cannot impact the contents at the storage device.

Second, all the writes to private regions must be executed through `nv_txbegin`, `nv_write`, `nv_write`,..., `nv_commit` command series. Those APIs are entirely controlled by vNVML and accessing NVM (log buffer, cache, and metadata) files, which are invisible to applications, is not allowed outside vNVML. When applications try to write beyond the mapped regions (or outside allocated virtual address regions), the protections within the existing memory system will detect these violations. In addition, the vNVML bound checks will not allow these writes to proceed.

5 Evaluation

In this section, we conduct experiments to answer fundamental questions about vNVML as follows:

- What are the characteristics of the vNVML?
- How does vNVML impact the performance when used by real applications?
- How to decide the size of the log buffer and the cache given a fixed and limited size of NVM in the platform?
- How does the vNVML perform when multiple processes concurrently access the NVM through vNVML?
- What is the impact of using vNVML within the container environment?
- What is the impact of different log buffer sizes, cache sizes, and single cache page size on life span of backend SSD?
- How does the vNVML perform compared to other user space libraries?

5.1 Experimental setup

Due to the absence of real NVM, we emulate NVM with DRAM for all our experiments. We mount the NVM with the Ext4 file system in order to utilize the DAX (direct access) feature provided by Ext4.

We evaluate vNVML on a platform with 16GB DRAM, 12GB emulated NVM, and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Samsung enterprise PM863 480GB SSD (SATA 6.0 Gbps) is adopted as our example of the storage devices. We implement vNVML on the Linux kernel 4.13 version.

5.2 MongoDB and YCSB

In this subsection, we explore and analyze the impact of accessing NVM through vNVML by real applications. We adopt a popular open-source database MongoDB version 3.6.0 (mongoDB 2008b) as our target application because its MMAPv1 storage engine uses memory mapped file form to access the data in the storage devices, which is perfect for our vNVML to employ. We modify part of the source code of MongoDB for our transactional interface to deploy vNVML.

We choose YCSB (Cooper et al. 2010) to generate the read/write traffic of MongoDB. The setup of experiment is delineated in Fig. 11. To simplify our analyses, we configure the size of all records' fieldcount as 128 and fieldlength as 512 and readallfields and writeallfields are both set as true in the configuration file of YCSB workloads, meaning that each read/write request will access exactly 64 KB data, which is also the data written per transaction. 100 K operations (read/write requests) are executed for all experiments. We deploy the different read/write ratios and two request distributions (zipfian or uniform) to observe the impact of performance.

YCSB has two phases: one is inserting records into the target data store, the other is accessing (read or write) records in the target data store. To avoid polluting the NVM cache of vNVML before the accessing phase, in the insertion phase MongoDB employs `nv_allocate` to *private* `mmap` files in the storage devices, and then, instead of using `nv_write` vNVML commands, MongoDB only adopts its original (unmodified) insertion functions to access the memory mapped regions, meaning that all records are only inserted into the memory (due to the Copy-on-Write mechanism provided by *private* `mmap` used by `nv_allocate`).

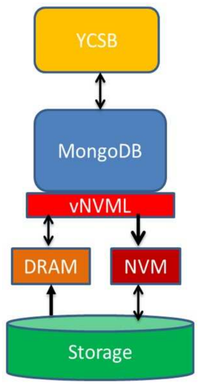


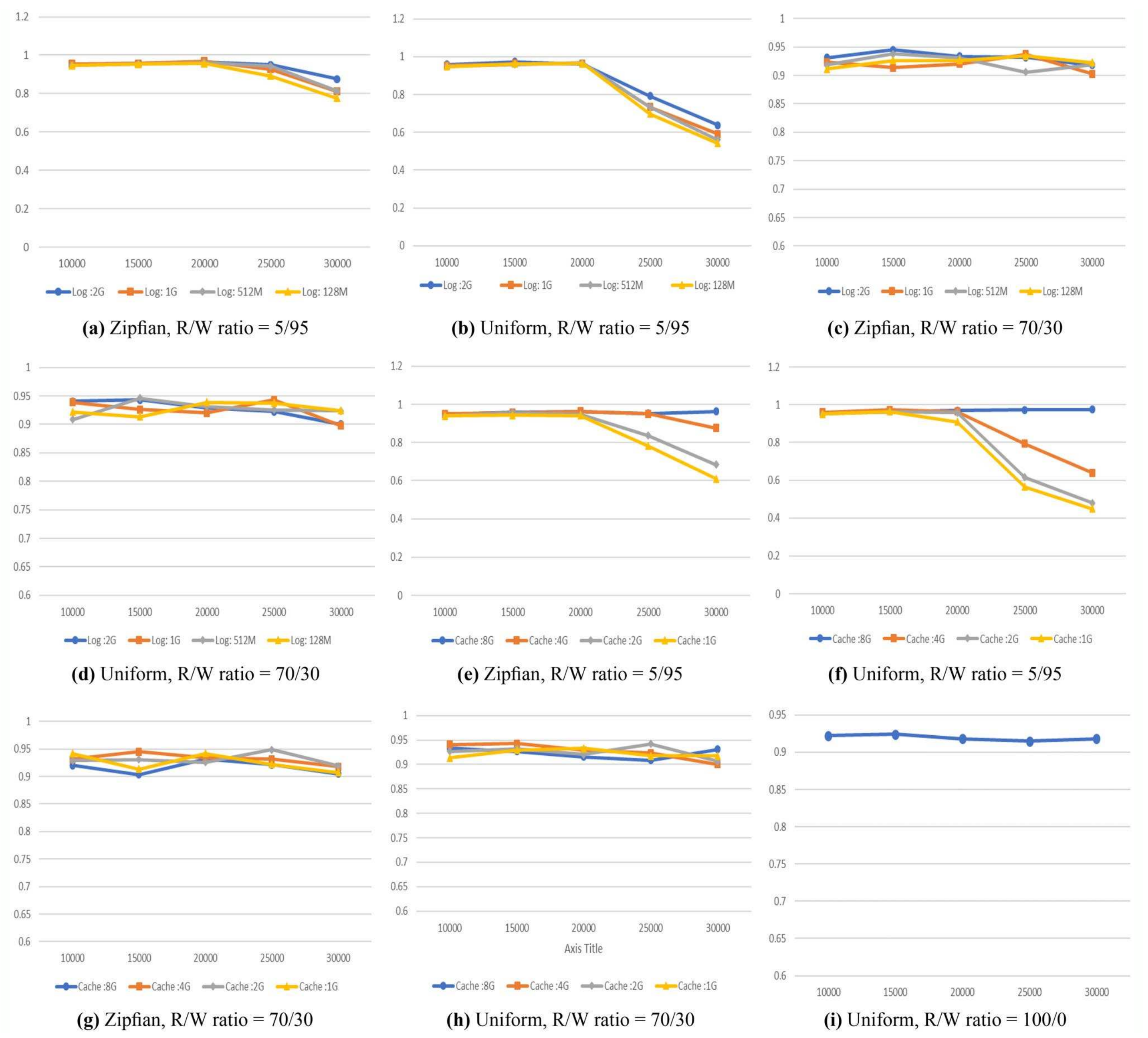
Fig. 11 The experimental setup of YCSB, MongoDB, and vNVML

Table 2 Throughputs of single MongoDB instance with different numbers and distributions of inserted records when NVM is the only storage device

# of records (K)	Uniform (op/s)	Zipfian (op/s)
30	1500	1505
10	1509	1498

All experiments are conducted by accessing four MongoDB instances concurrently in a single OS. However, since the MMAPv1 storage engine uses padding and the power of two sized allocation mechanisms (mongoDB 2008a), four instances would generate total 8.8 GB files in the storage devices when each instance is inserted 10 K records, and total 33 GB files after 30 K records are inserted into each of four instances. Therefore, 12 GB emulated NVM in our platform can only accommodate files created by four MongoDB instances inserted at most 13 K records, respectively. However, from Table 2, we find that the YCSB throughputs of one instance are very close to each other even with different numbers and different distributions of inserted records if all files generated are stored only in NVM. We assume this observation still holds in the 4-instance case. Therefore, we insert 10 K records to each of four instances, remove the periodic `msync` calls by MongoDB, disable journaling with *nojournal* option, and employ NVM as the only storage device of MongoDBs as our baseline[5].

[5] Note: our platform does not have enough NVM to accommodate all files generated by eight instances inserted 10 K as baseline, respectively. Also, when eight MongoDB instances adopting vNVML are running concurrently, some of instances would crash because of out-of-memory (OOM) error from *private* `mmap`. Therefore, executing at most four instances concurrently is the limitation of our platform.



(a) Zipfian, R/W ratio = 5/95 **(b)** Uniform, R/W ratio = 5/95 **(c)** Zipfian, R/W ratio = 70/30 **(d)** Uniform, R/W ratio = 70/30 **(e)** Zipfian, R/W ratio = 5/95 **(f)** Uniform, R/W ratio = 5/95 **(g)** Zipfian, R/W ratio = 70/30 **(h)** Uniform, R/W ratio = 70/30 **(i)** Uniform, R/W ratio = 100/0

Fig. 12 Normalized total throughput of four instances. Numbers of X-axis stand for inserted records to each database, and numbers of Y-axis stand for normalized throughput. **a**–**d** Fix 4 GB cache size and adjust log buffer size from 2 GB to 128 MB. **e**– **h** Fix 2 GB log buffer and change cache size from 8 to 1 GB. **i** 100% read uniform request

Figure 12 shows the normalized throughputs of different request distributions (zipfian and uniform) and different read/write ratios (5/95, 70/30, and 100/0). The results (normalized throughputs) are the summation of throughputs[6] (op/s), generated from YCSB when YCSB accesses one of four MongoDB instances employing vNVML, divided by the summation of throughputs of four MongoDB baseline instances with the same request distributions and the same read/write ratios.

From these results we can make some useful observations. First, the case that cache size is 1 GB, log size is 2 GB, and four MongoDB instances with 30 K inserted records (4 * 30K * 64 KB = 7.32 GB) already proves that vNVML can provide virtualization and shareability of NVM successfully.

Second, not only can vNVML achieve over 90% of the throughput of baseline (if the log buffer and cache can "absorb" the input write traffic, such as the line of 8GB cache in Fig. 12e, f), but vNVML can also provide the guarantees of atomicity, persistency, and write ordering, which

[6] Note: since all four throughputs from YCSB are almost the same (with usually less than 1% difference).

are our baseline, the MongoDB without journaling and write ordering, cannot. This less than 10% overhead results from writing data to NVM log buffer and from redoing logs to DRAM read cache.

Third, larger write working sets (more inserted records or uniform access requests), and more write requests (lower R/W ratio) degrade the throughput of vNVML. Larger write working sets require more NVM cache to store all data at run-time. Furthermore, if the write working sets are even larger than the capacity of NVM cache owned by applications, then cache pages are frequently written back to storage devices. Finally, when all cache pages become dirty, the overall performance would deteriorate to write throughputs of storage devices.

Fourth, through (a) to (d), when cache sizes are all fixed, the adjustment of log buffer only affects at most 10% normalized throughput[7] of baseline in all these cases.

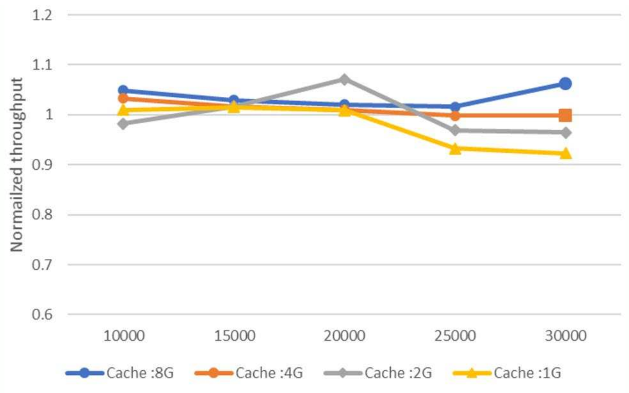
Fifth, from (e) to (h), when sizes of log buffer are fixed, their throughputs vary highly, especially in the case of (f): read/write ratio is 5/95, uniform request, and 30K inserted records. In (f), the throughputs differ by almost 50%, meaning that cache size impacts vNVML throughput more significantly than that of the log buffer. For the bottleneck of vNVML performance is the access throughput of storage devices, the cache size can impact vNVML performance significantly. As more NVM caches are able to store more write traffic, less writes (if NVM cache hits) to storage devices will be needed. The actual throughput of vNVML should be a function dominated by factors of NVM cache size and access performance of storage device.

On the other hand, unlike NVM cache, NVM log buffer only temporarily stores the write traffic as logs before logs are written to NVM cache pages. As a result, log buffer can only impact/improve performance slightly until log buffer is full; usually the log buffer will be full much quickly than NVM cache if the write working sets are huge and the storage devices are frequently accessed in order to fill the NVM pages before applying the corresponding log entries to cache pages.

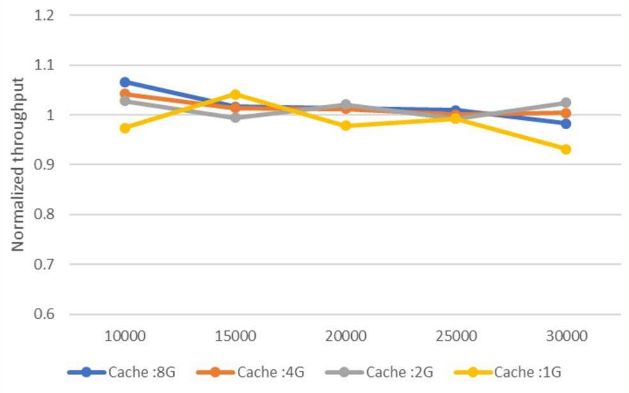
Finally, (i) shows at 100% read, uniform distribution request case, vNVML can achieve around 92% throughput regardless of the number of inserted records. It matches our expectation of vNVML since the read is entirely handled by the read cache (memory) and 16 GB memory is enough to handle the 30 K records working set since 30 K $\times$ 64 K $\times$ 4 $\sim$ 7.32 GB.

Therefore, from aforementioned observations, we suppose that under limited NVM resources, only some reasonable amount of NVM should be allocated as log buffer, and the rest should be cache to achieve higher performance of vNVML.

Next, we would like to examine the impact of using vNVML within docker containers (Docker 2013). Docker is a popular virtualization technique in data centers and recently has drawn significant attention from industry and academia due to its lightweight execution environment compared to traditional virtual machines. In this experiment, we launch four docker containers, use *bind mount* (Docker 2018) to mount 12 GB emulated NVM into each container so all containers can access and share content in NVM, and run single MongoDB instance within each container. Log buffer is configured as fixed 2 GB, the cache size as well as read/write ratio are adjusted to various settings. Each data point is normalized with individual counterpart, which is the same configuration without using containers. Figure 13 shows that all the data are close to 1; that is, using vNVML within docker containers does not affect the performance.



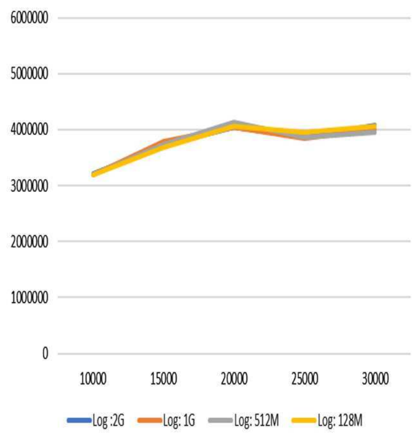
(a) Zipfian, 5/95 R/W ratio



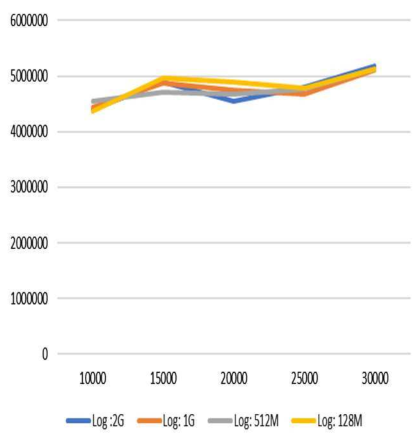
(b) Zipfian, 70/30 R/W ratio

Fig. 13 Normalized throughput of four instances inside Docker container

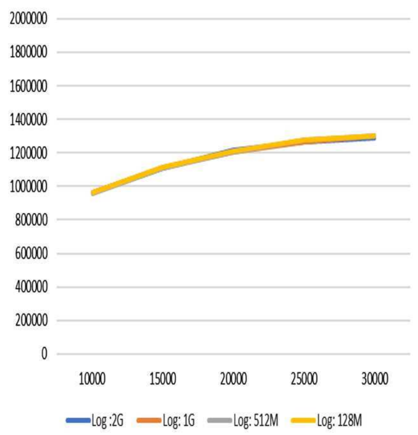
[7] At (a), the normalized throughput is 0.876 (0.776, respectively) when log buffer is 2 GB (128 MB, respectively) and inserted records are 30,000. At (b), the normalized throughput is 0.638 (0.542, respectively) when log buffer is 2 GB (128 MB, respectively) and inserted records are 30,000.



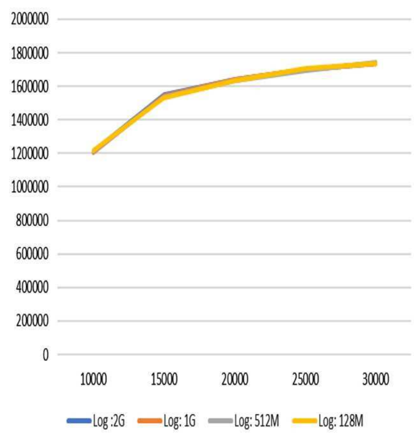
(a) Zipfian, R/W ratio = 5/95



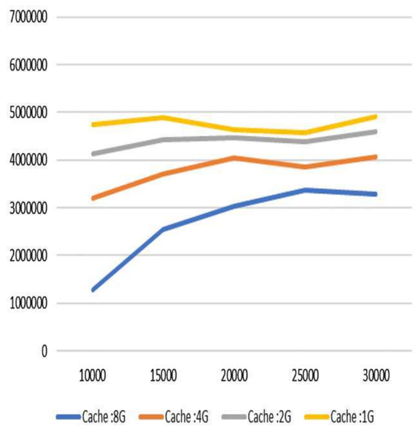
(b) Uniform, R/W ratio = 5/95



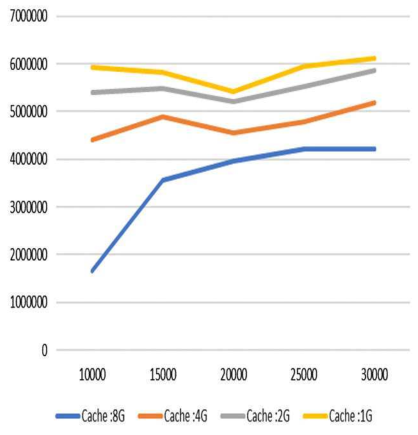
(c) Zipfian, R/W ratio = 70/30



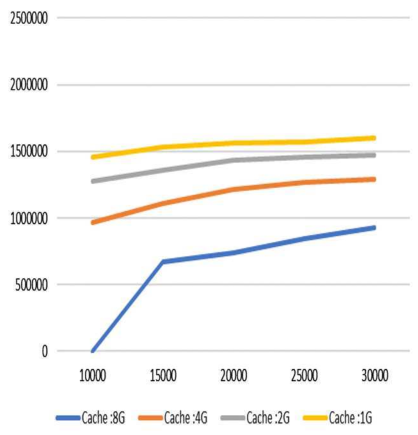
(d) Uniform, R/W ratio = 70/30



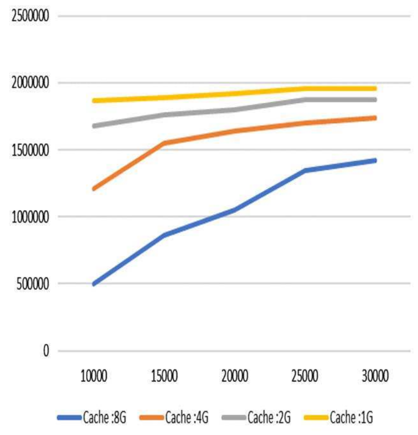
(e) Zipfian, R/W ratio = 5/95



(f) Uniform, R/W ratio = 5/95



(g) Zipfian, R/W ratio = 70/30



(h) Uniform, R/W ratio = 70/30

Fig. 14 Total number of NVM cache pages written to SSD of all four instances. Numbers of X-axis stand for inserted records to each database, and numbers of Y-axis stand for total number of pages written to SSD. **a**–**d** Fix 4 GB cache size and adjust log buffer size from 2 GB to 128 MB. **e**–**h** Fix 2 GB log buffer and change cache size from 8 to 1 GB

Moreover, we want to examine the impact of different sizes of NVM log buffer and NVM cache on life span (or write counts) of backend SSD. We measure the number of total NVM (cache) pages written to storage devices after all logs have been replayed from log buffer to NVM cache pages, and cache pages would then be written to backing SSD if the percentage of dirty pages are over 30%. Figure 14 shows the results, from which our conclusions are drawn.

First, larger write working sets usually have more numbers of written pages (more write counts), but some exceptions can also be found. For instance, cache size is 1 GB at (e) and (f). This phenomenon is because writeback background worker starts to write NVM cache pages back only when the percentage of dirty pages is over 30%. So, it is possible that even larger write working sets make all instances with higher percentage of dirty pages, but none of them is more than 30%. On the other hand, smaller write working sets cause instances with lower percentage of dirty pages, but once the percentage of one instance is over 30%, then smaller write working sets might result in more write counts than larger ones.

Second, from (a) to (d), all results (of various log buffer sizes) are almost the same. This also proves that log buffer only temporarily stores the write traffic and cannot influence the access frequencies and patterns of storage devices. It also matches the conclusion made from results of Fig. 12.

Third, one interesting point is the case of 8 GB cache at (g). We can find out that when number of inserted records is 10,000, the number of write count is zero; that is, all writes are stored entirely on NVM cache and none is written to SSD. This means that if we have enough NVM cache to store incoming write traffic, vNVML can achieve not only better performance but also longer life span of SSD.

Next, we consider the impact of different SSD page sizes on write counts and performance. We assume the size of NVM cache page should be the same as the page size of backend SSD; otherwise, the write amplification must be considerable. As a result, we only change the page size of NVM cache as 8 KB and compare the write counts and performance with NVM cache of 4 KB page size. Since we have learned from Fig. 14 that the size of log buffer cannot impact the write counts, we only employ different cache size here. R/W ratios of all experiments are fixed as 5/95. Figure 15 shows the result.

From our experiments, doubling the page size (as 8 KB) will cause the write counts slightly more than 50% of those of 4 KB page size at all experiments, and has almost no impact on the performance of vNVML. This is reasonable because larger cache pages can absorb more write logs than smaller cache pages before they are written to SSD and therefore require less writes to backend SSD.

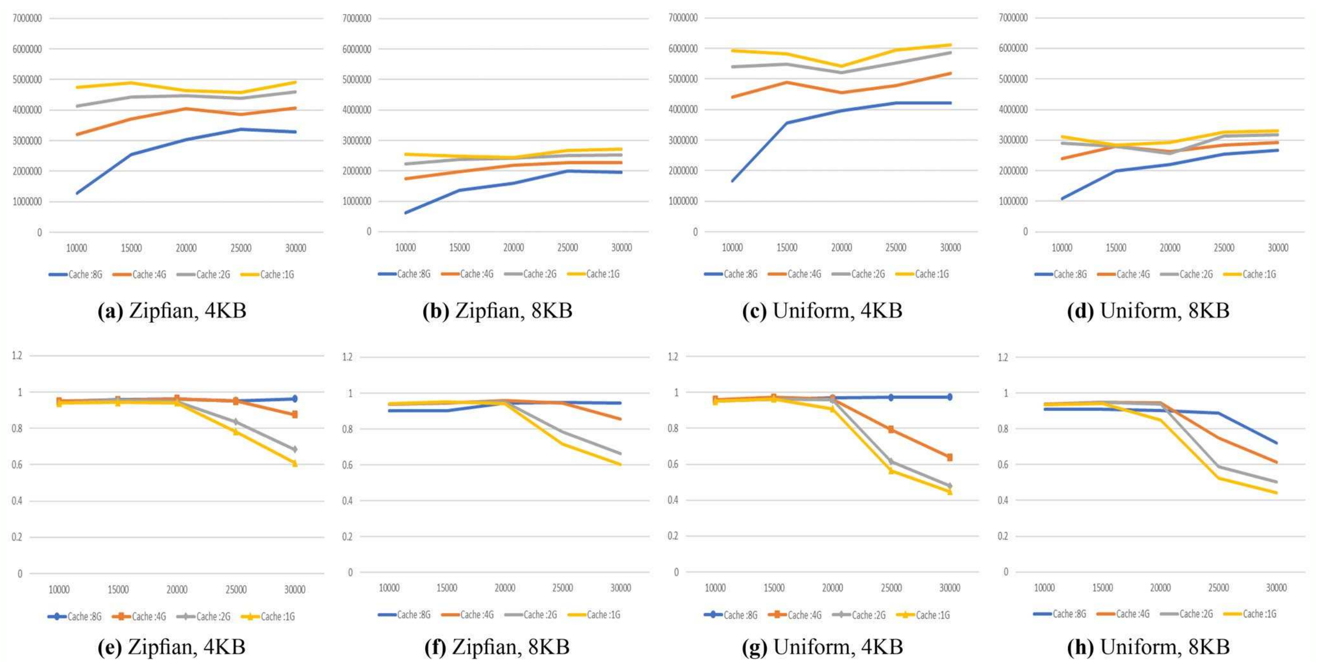
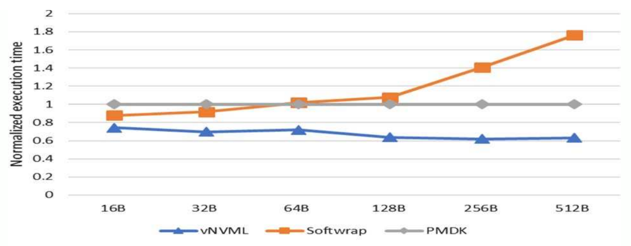
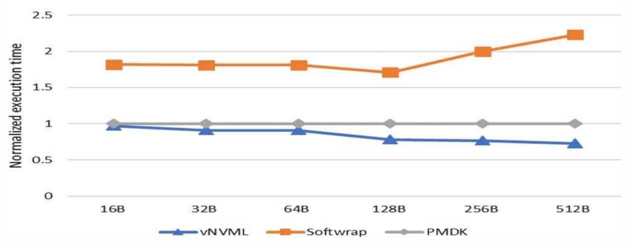


Fig. 15 Comparisons of 4 KB and 8 KB size of NVM cache page. All read/write ratios are 5/95. The log buffer size is 2GB and cache sizes are from 8GB to 1GB. Numbers of X-axis stand for inserted records to each database. **a**–**d** Total number of NVM cache pages written to SSD of all four instances (numbers of Y-axis) with different request distributions and page size. **e**–**h** Normalized throughputs (numbers of Y-axis) with different request distributions and page size



(a) Normalized vNVML and SoftWrap execution time with the same NVM size. (Lower is better.)



(b) Normalized vNVML and SoftWrap execution time assuming unlimited NVM size. (Lower is better.)

Fig. 16 Normalized vNVML and SoftWrap execution time. Numbers of x-axis stand for the amount of written data per page

5.3 Microbenchmark

We use a simple microbenchmark to compare the performance between Intel's PMDK (Intel 2015), SoftWrAP (Giles et al. 2015), and our vNVML. In this experiment, we create a 2GB array in NVM (virtual NVM, respectively) for PMDK and SoftWrAP (vNVML, respectively), and write different amounts of data (from 16B to 512B) per page sequentially. Each transaction contains 32 page writes.

To use PMDK, we use `pmemobj_create` to create a 4 GB NVM pool because 2 GB NVM pool is not enough to accommodate 2 GB array. We always set *PMEM_IS_PMEM_FORCE=1* when executing PMDK to avoid unnecessary `msync` or `fsync` when accessing NVM. For fairness, we use 2 GB log buffer and 2 GB cache when running vNVML. We only use default setting for SoftWrAP since it does not provide API for internal buffer size adjustment.

Figure 16a shows the result. We use the total execution time of PMDK as our baseline, and show the total time of writing the 2 GB array for once. The result indicates that among others our vNVML performs better as the total written data keeps increasing. Figure 16b shows another experiment, which we enlarge the NVM to 8 GB and want to compare the upper bound of each library. We write the 2 GB NVM array 16 times. Its result is similar as Fig. 16a.

6 Conclusion

In this paper we presented vNVML, a byte-level user space library to access NVM that provides transaction-like semantics for applications, ensures write ordering, and provides persistency guarantees across failures. Our system employs NVM as a write log and a write cache, while also employing DRAM as a cache.

We implemented vNVML and evaluated it with realistic workloads to show that our system allows applications to share NVM, both in a single OS and when docker-like containers are employed. The results from the evaluation show that vNVML incurs less than 10% overhead while providing a larger than available physical NVM space to the applications and allowing them to safely share the virtual NVM.

Acknowledgements The conference version of this paper was presented in MSST 2019. We thank the anonymous reviewers for their valuable comments and feedback to improve the content and quality of this work. We also thank the National Science Foundation, which supports this work through Grants I/UCRC-1439722 and FoMR-1823403, and generous support from Hewlett Packard Enterprise.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Chen, A.: A review of emerging non-volatile memory (NVM) technologies and applications. Solid-State Electron. **125**, 25–38 (2016)

Coburn, J., Caulfield, A.M,, Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: ASPLOS XVI Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 105–118. ACM, Newport Beach (2011)

Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better i/o through byte-addressable, persistent memory. In: SOSP '09 Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, Big Sky, Montana (2009)

Cooper, B.F., Silberstein, A., ErwinTam, Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC '10 Proceedings of the 1st ACM Symposium on Cloud Computing, pp 143–154. ACM, Indianapolis (2010)

Docker. Docker container. https://www.docker.com/ (2013)

Docker (2018) Docker container bind mounts. https://docs.docker.com/storage/bind-mounts/

Doshi, K., Giles, E.R., Varman, P.: Atomic persistence for scm with a non-intrusive backend controller. In: HPCA '16 Proceedings of the IEEE International Symposium on High Performance Computer Architecture. IEEE, Barcelona (2016)

Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory.

In: EuroSys '14 Proceedings of the Ninth European Conference on Computer Systems, ACM, Amsterdam (2014)
Eisner, L.A., Mollov, T., Swanson, S.: Quill: Exploiting fast non-volatile memory by transparently bypassing the file system. In: Technical report, UCSD (2013)
Fedorov, V., Kim, J., Qin, M., Gratz, P.V., Reddy, A.L.N.: Speculative paging for future NVM storage. In: MEMSYS '17 Proceedings of the International Symposium on Memory Systems, Alexandria, Virginia (2017)
Giles, E.R., Doshi, K., Varman, P.: Softwrap: A lightweight framework for transactional support of storage class memory. In: MSST '15 Proceedings of the 31st Symposium on Mass Storage Systems and Technologies, pp. 1–14. IEEE, Santa Clara (2015)
Intel.: Persistent memory development kit (2015). https://pmem.io/pmdk/
Intel. Storage performance development kit (2017). https://spdk.io/
Intel: Intel optane dc persistent memory (2019). https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
Joshi, A., Nagarajan, V., Viglas, S., Cintra, M.: Atom: Atomic durability in non-volatile memory through hardware logging. In: HPCA '17 Proceedings of the IEEE International Symposium on High Performance Computer Architecture. IEEE, Austin (2017)
Jung, J., Won, Y., Kim, E., Shin, H., Jeon, B.: Frash: exploiting storage class memory in hybrid file system for hierarchical storage. ACM Trans. Storage (TOS) **6**(1), 1–25 (2010)
Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., Anderson. T.: Strata: A cross media file system. In: SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles, pp. 460–477. ACM, Shangha (2017)
Lee, B.C., Zhou, P., Yang, J., Zhang, Y., Zhao, B., Ipek, E., Mutlu, O., Burger, D.: Phase-change technology and the future of main memory. IEEE Micro **30**, 131–141 (2010)
Liang, L., Chen, R., Chen, H., Xia, Y., Park, K., Zang, B., Guan, H.: A case for virtualizing persistent memory. In: SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, Santa Clara (2016)
Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., Swanson, S.: Atomic in-place updates for non-volatile main memories with kamino-tx. In: EuroSys '17 Proceedings of the Twelfth European Conference on Computer Systems, pp. 499–512.ACM, Belgrade (2017)
Microsoft: Transaction isolation levels (2017). https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-2017
mongoDB (2008a) Mmapv1 storage engine. https://docs.mongodb.com/manual/core/mmapv1/
mongoDB (2008b) Mongodb data base. https://github.com/mongodb
Narayanan, D., Hodson, O.: Whole-system persistence. In: ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems. ACM, London (2012)
Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. ISCA '14 Proceeding of the 41st annual international symposium on Computer architecuture, pp. 265–276. Minneapolis, Minnesota (2014)
Qiu, S., Reddy, A.L.N.: Nvmfs: a hybrid file system for improving random write in NAND-flash SSD. In: MSST '13 IEEE 29th Symposium on Mass Storage Systems and Technologies. IEEE, Long Beach (2013)
Qureshi, M.K., Srinivasan, V., Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology. In: ISCA '09 Proceedings of the 36th Annual International Symposium on Computer Architecture. ACM, Austin (2009)
Samsung (2017) Ultra-low latency with Samsung z-nand ssd. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf
SNIA: NVM Programming Model. In: Storage Networking Industry Association, rev., vol. 1, no. 2 (2017)
Swanson, S.: A vision of persistence (2017). https://www.sigarch.org/a-vision-of-persistence/
Swift, M.: Persistent memory ordering (2015). http://materials.dagstuhl.de/files/15/15021/15021.MichaelSwift1.Slides.pdf
Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H.: Consistent and durable data structures for non-volatile byte-addressable memory. In: FAST '11 Proceedings of the 9th USENIX Conference on File and Storage Technologies, USENIX (2011)
Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: Lightweight persistent memory. In: ASPLOS XVI Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 91–104. ACM, Newport Beach, California (2011)
Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., Swift, M.M.: Aerie: Flexible file-system interfaces to storage-class memory. In: EuroSys '14 Proceedings of the Ninth European Conference on Computer Systems. ACM, Amsterdam (2014)
Wan, H., Lu, Y., Xu, Y., Shu, J.: Empirical study of redo and undo logging in persistent memory. In: NVMSA '16 Proceeding of the 5th Non-Volatile Memory Systems and Applications Symposium, pp. 1–6. IEEE, Daegu (2016)
Wang, C., Wei, Q., Yang, J., Chen, C., Xue, M.: How to be consistent with persistent memory? an evaluation approach. NAS '15, pp. 186–194. IEEE, Boston (2015)
Watts, D.: Intel optane dc persistent memory product guide (2019). https://lenovopress.com/lp1066-intel-optane-dc-persistent-memory
Wu, X., Reddy, A.L.N., (2011) SCMFS: a file system for storage class memory. In: SC '11 Proceedings of International Conference for High Performance Computing. Networking, Storage and Analysis. ACM, Seattle (2011)
Xu, J., Swanson, S.: Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In: FAST '16 Proceedings of the 14th USENIX Conference on File and Storage Technologies, pp. 323–338. USENIX, Santa Clara (2016)
Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-Tree: Reducing consistency cost for NVM-based single level systems. In: FAST '15 Proceedings of the 13th USENIX Conference on File and Storage Technologies. USENIX, Santa Clara (2015)
Yu, S., Chen, P.Y.: Emerging memory technologies: recent trends and prospects. IEEE Solid-State Circuits Mag. **8**(2), 43–56 (2016)
Zhang, Y., Swanson, S.: A study of application performance with non-volatile main memory. In: MSST '15 Proceedings of the 31st Symposium on Mass Storage Systems and Technologies. IEEE, Santa Clara (2015)
Zhang, Y., Yang, J., Memaripour, A., Swanson, S.: Mojim: a reliable and highly-available non-volatile memory system. In: ASPLOS '15 Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, Istanbul (2015)
Zhao, J., Li, S., Yoon, D.H., Xie, Y., Jouppi, N.P.: Kiln: closing the performance gap between systems with and without persistence support. In: MICRO '13 Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, Davis (2013)
Zhou, P., Zhao, B., Yang, J., Zhang, Y.: A durable and energy efficient main memory using phase change memory technology. In: ISCA '09 Proceedings of the 36th Annual International Symposium on Computer Architecture. ACM, Austin (2009)

Chih Chieh Chou received the B.S. degree in Electrical Engineering in 2003 and M.S. degree in Communication Engineering in 2005, both from National Tsing Hua University, Hsinchu, Taiwan. From 2005 to 2011, he was a Software Development Engineer with MediaTek Inc., Hsinchu, Taiwan. He is currently a Ph.D. student in Electrical and Computer Engineering at Texas A&M University, College Station, Texas, USA, from 2014. He is also currently a Software Development Engineer with Amazon Lab126, Sunnyvale, California, USA, from 2020. His research interests include Non-Volatile Memories, Memory Systems, Operating Systems, and Computer Architecture. He is a student member of IEEE.

Jaemin Jung received the B.S., M.S., and Ph.D. degrees in Computer Science from Hanyang University, Seoul, Korea, in 2007, 2009, and 2016, respectively. He continued his research as a Postdoc in Texas A&M University, College Station, Texas, USA, from 2016. He is currently a Senior Engineer in Samsung Semiconductor, San Jose, California, USA, from 2018. His research interests include Storage Systems, Persistent Memories, and Near-Memory Computing.

A. L. Narasimha Reddy received a B.Tech. degree in Electronics and Electrical Communications Engineering from the Indian Institute of Technology, Kharagpur, India in August 1985, and M.S. and Ph.D. degrees in Computer Engineering from the University of Illinois at Urbana-Champaign in May 1987 and August 1990, respectively. Reddy is currently a J.W. Runyon Professor in the department of Electrical and Computer Engineering at Texas A&M University as well as the Associate Dean for Research with the Texas A&M Engineering Program and the Assistant Director of Strategic Initiatives & Centers with the Texas A&M Engineering Experiment Station. Reddy's research interests are in Computer Networks, Storage Systems, Multimedia Systems, and Computer Architecture. During 1990–1995, he was a Research Staff Member at IBM Almaden Research Center in San Jose where he worked on projects related to disk arrays, multiprocessor communication, hierarchical storage systems and video servers. Reddy holds five patents and was awarded a technical accomplishment award while at IBM. He received an NSF Career Award in 1996. He was a Faculty Fellow of the College of Engineering at Texas A&M during 1999–2000. His honors include an Outstanding Professor award by the IEEE student branch at Texas A&M during 1997–1998, an Outstanding Faculty award by the Department of Electrical and Computer Engineering during 2003–2004, a Distinguished Achievement award for teaching from the Former Students Association of Texas A&M University, and a citation "for one of the most influential papers from the 1st ACM Multimedia Conference". Reddy is a Fellow of IEEE Computer Society and is a member of ACM.

Paul V. Gratz (SM'16) received the B.S. and M.S. degrees from the University of Florida, Gainesville, FL, USA, in 1994 and 1997, respectively, both in Electrical Engineering, and the Ph.D. degree in Electrical and Computer Engineering from the University of Texas at Austin, Austin, TX, USA, in 2008. He is an Associate Professor with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA. From 1997 to 2002, he was a Design Engineer with Intel Corporation, Santa Clara, CA, USA. His current research interests include energy efficient and reliable design in the context of high performance computer architecture, processor memory systems, and on-chip interconnection networks. Dr. Gratz is a member of ACM.

Doug Voigt is a retired Distinguished Technologist who worked for HP and HPE storage for his entire 40 year career. He has developed firmware and software for disk controllers, disk arrays and storage management. He has led HP and HPE virtual array advanced development projects and strategy between 1990 and his retirement in 2018. Since 2012 his technical focus has been on non-volatile memory systems. Throughout his career Doug was a strong proponent of industry standards. He has been a member the Storage Network Industry Association (SNIA) since 2009. He served on the SNIA board of directors, technical council and as co-chair of the NVM Programming Technical Working Group. Doug has over 50 patents mostly in the areas of virtual arrays and persistent memory. Doug's hobbies include music, photography and reading science fiction/fantasy.