vNVML: An Efficient User Space Library for Virtualizing and Sharing Non-volatile Memories

Chih Chieh Chou*, Jaemin Jung*†, A. L. Narasimha Reddy*, Paul V. Gratz*, and Doug Voigt‡
*Department of Electrical and Computer Engineering
Texas A&M University
Email: {ccchou2003, reddy, pgratz}@tamu.edu
†Samsung Semiconductor
Email: j.jaemin@samsung.com
‡Hewlett Packard Enterprise
Email: doug.voigt@hpe.com

***Abstract*—The 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, like SSD, to create the impression of a larger and faster byte-addressable NVM which can be shared across many applications.**

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 and persistency guarantees across system failures. vNVML exploits DRAM for read caching, to enable improvements in performance and potentially to reduce the number of writes to NVM, extending the NVM lifetime. vNVML is implemented and evaluated with realistic workloads to show that our library allows applications to share NVM, both in a single O/S 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, allowing applications to safely share the virtual NVM.

I. Introduction

Emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM) [1], NVDIMM [2], and 3D XPoint [3], will dramatically shake up future system designs [4]–[8]. In particular, these NVM technologies promise not only much faster access times than existing SSDs, to within an order of magnitude of DRAM, but they also are "byte" addressable and will be placed directly on the memory buses. As a result, these NVM technologies could be used to replace existing permanent storage devices or even volatile memory (single level system).

To date there have been significant works in this domain. Some [6], [7], [9]–[12] engineer novel file systems suitable for exploiting NVM. Some [8], [13] carefully design their data store manipulation mechanism to directly access data structures from NVM, to maximize performance. These prior works, 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 expensive system calls incurred by accessing through the file system, however, squander the low-latency provided by NVM. Considering boosting maximum gain from NVM, in this paper, we focus on memory mapped file access form. Currently, when `mmapping` files on storage devices to volatile memory (DRAM), the POSIX interface can support sharing the same memory region of files between processes (i.e. *shared* `mmap`). Also, thanks to the swapping mechanism of virtual memory, which writes dirty pages back to the backing storage devices and produces clean pages for the future use, the physical available DRAM is extended. These two attractive properties of existing virtual memory, however, are not supported by the prior works of NVM.

This paper considers this problem of virtualizing and sharing byte-addressable NVM across many applications. Here we introduce "vNVML", an efficient user space library for virtualizing and sharing NVM. What we mean by "sharing NVM" here is that not only can the same NVM pages be reallocated and reused across users, but the data content on NVM can also be seen/accessed by applications concurrently.

One of the aims of vNVML is to provide the impression of large 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 read cache to improve the performance of persistent memory access. There are certain advantages in employing the DRAM even when the applications access virtual NVM. First, DRAM may have better performance than most types of NVM, except for NVDIMM. 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

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

storage devices to DRAM, bypassing the NVM entirely. This might be a better design choice compared to employing NVM as both read and write cache in terms of reducing number of write access.

We design and implement vNVML with the hope that programmers could access NVM with a similar interface as for existing memory mapped files. That is, after `mmapping` a file on the storage device as virtual NVM, a pointer is returned. This pointer can be directly accessed in the program 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 [8], [11], [13]–[15]. Many solutions have been proposed to address the write ordering problem within persistent memories, including hardware capacitors to ensure eviction of all data from volatile memories to NVM [11], epoch-based writes [11], [14], transaction like semantics [14]–[17], versioning [13], and special data stores and algorithms designed for NVM [8], [13]. Here vNVML proposes to use transaction like semantics to guarantee the atomicity and durability of NVM accessing.

vNVML employs DRAM as read cache, NVM as log buffer and write cache, and the backing storage device as the final destination of writes. Through our evaluations, vNVML incurs less than 10% throughput overhead compared to directly accessing NVM without an atomicity guarantee.

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 read caching in DRAM, coupled with write logging and caching in NVM with lazy write-back to the backing store to provide a high performance, virtualized, and shareable NVM to applications.
- We evaluate this proposed vNVML under synthetic as well as realistic (YCSB+MongoDB) workloads and show it is competitive with prior techniques which do not support virtualization.

The remainder of this paper is organized as follows. Section II describes background and prior work in this area. Section III presents a design overview and discusses the design decisions of vNVML. Section IV explains the implementation of vNVML in detail. Section V presents our results of evaluation of vNVML and section VI concludes.

II. Background

Much of the early work to-date incorporating NVM in systems assumes basic hardware changes. New memory controllers are proposed by [18]–[20]. Kiln [21] proposes a victim cache for buffering and Atom [22] deploys a hardware logging approach to avoid software logging overhead. BPFS [11] 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 first expect that NVM DIMMs will become available which will directly connect to the system memory bus with little or no changes to the basic processor caching and memory management hardware. For example, the recent Intel and Micron's collaborations to produce Optane NVM DIMMs [3].

Near-term systems will incorporate this type of NVM be 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 form of NVM primarily focuses on constructing new file systems to handle the underlying NVM [6], [7], [9]–[12], [23], [24]. Some of those works have considered building file systems across multiple types of NVM and storage technologies. These include NVMFS [10] (NVM and SSD), and Strata [6] (DRAM, NVM, SSD, and HDD). The design concept of Strata is close to our vNVML, both of which 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. Also, Strata does not support memory mapped files, which are used by our target applications. Generally speaking, accessing NVM through the file system interface is not suitable for random, small access of NVM since the overhead of expensive system calls will squander the low-latency that NVM offers. Another drawback is that such software approaches require users and applications to deploy their dedicated file systems.

Some work considers replacing DRAM and disk with NVM to build a single level system, and manipulating data structure operations directly on NVM, like NV-Tree [8] and CDDS-Tree [13]. However, such approaches restrict themselves to only some specific data structures and cannot be easily applied to general memory access. SPAN [25] proposes some new swapping enhancements in the O/S kernel to exploit NVM as extended system memory. NV-Heaps [14] 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 does.

Some other approaches create user space libraries [15]–[17], [26]. Intel PMDK [26] and Memaripour et al. [17] consider employing only NVM and utilizing undo logging to provide in-place updates. Giles et al. [16] combine NVM and DRAM and use 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 related work to our vNVML is Mnemosyne [15], 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 difference between these two approaches. Mnemosyne achieves NVM

virtualization by swapping, which is controlled entirely by the kernel. Further, it 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 processes.

In this work we propose a system software-based 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 and endurance guarantees while offering a larger than available physical NVM space to the applications and allowing them to safely share the virtual NVM regions, while maintaining performance goals by leveraging caching in DRAM.

III. 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. When a file system writes data to a location in persistent memory, that location cannot be reused or reallocated by another application. If NVM is similarly allocated and used, NVM cannot be easily shared across applications if NVM is the only persistent storage devices 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 SSDs or HDDs, with NVM is not good enough*. Although by doing so, all applications can benefit from performance improvements brought by NVM immediately without any modifications required; however, this ignores the byte-addressability of NVM and can require 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 data through system calls, may not be a significant part of the access latencies. However, as devices get faster, these system call overheads become much more significant and hence must be avoided.

Finally, while it is possible (and even desirable) to continue running existing or older software on new hardware, *software may have to be rewritten to get the most of the hardware*. This can take the form of new file systems, new applications [8], [14] or new libraries [15]–[17], [26], [27]. 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 library, integrating DRAM, NVM, and storage devices to construct the abstraction of virtualized NVM. Like virtual memory, adopted almost universally in the modern computer systems, the main idea of virtual NVM is to provide the illusion 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).

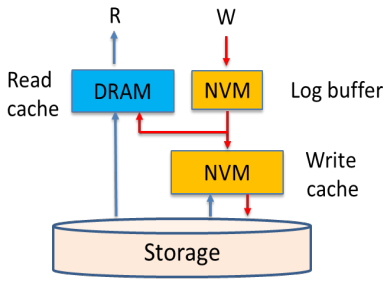


Fig. 1: The read/write data flow of vNVML private mmapping between DRAM, NVM, and storage device.

Usually applications have two means to access the data on a storage device, one is through file system commands such as `open`, `read`, and `write`, the other is through memory mapped files. In order to expose the byte addressability of NVM, we focus on memory mapped files, such that the applications access NVM much like memory, through byte-level load/store interfaces.

vNVML places no limits on which file system may be used and only requires that the file system must support `mmap`. Here we hope that programmers adopt vNVML as much the same way as they employ existing POSIX `mmap` for volatile memory, except that they must follow the transaction like semantics. Meanwhile, the benefits of performance improvement, atomicity, and durability are provided.

From here, we focus on vNVML's *private* `mmap` access mode, meaning that virtual NVM regions can only be accessed by a single process. The mechanism for NVM sharing mode is slightly different and, to avoid confusion, is explained in section IV-D. 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 written to the NVM log buffer first and then copied to NVM (write cache) and DRAM (read cache) only when the logged data are committed. If pages containing accessed data are not already in NVM or DRAM, they are copied from the storage devices to NVM or DRAM. Data are evicted from the NVM write cache back to storage devices only when NVM usage demands exceed some threshold of the available NVM. Before programs are terminated safely, all data are completely flushed from NVM 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 libraries, we also use transaction semantics as interface for vNVML to provide atomicity and durability. All written data are 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:** Typically, there are two approaches to logging: *undo* and *redo* logging. Both have pros and cons toward different workloads [28]. *Undo* logging requires that we persist old data as logs before we update new data in place. These two actions (logging and in place update) are in the write critical path. For *redo* logging, we persist all new data as logs to non-volatile media first, then we in place update new data on the storage device.

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 log and updating to DRAM are in the write critical path (from the perspective of the whole transaction). 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 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 [29], [30]. 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 are time-consuming. 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.

Update committed data to read cache: Before logs of uncommitted transactions can be placed into real positions, 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) [31] 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 actions, 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 IV-A will explain the detailed mapping of read cache, log buffer, and write cache into virtual address space of each process.

Two restrictions are related to the read cache: (1) reads can only be served by the read cache and (2) written data is copied to read cache only when the transaction commits to accomplish the isolation property; that is, only committed data is visible. This is sometimes referred as "read committed" transaction isolation level [32]. However, our transactions are defined differently from that of transactions in the traditional database systems. In database systems, the focus is on the consistency of transactions to ensure correct data is accessed between multiple concurrent transactions. In vNVML, we emphasize the persistency [33] of transactions. Here we define the *committed* (*uncommitted*, respectively) data are that the written data must be valid (invalid, respectively) after system crashes. We leave the consistency of transactions to the discretion of programmers/applications.

Use NVM as write cache: All the 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. Committed data are written to NVM by utilizing part of NVM as a write cache. This allows us to migrate the logs quickly to more permanent locations and to maintain 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 applications because logs could 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.

Write to storage devices through write cache or read cache: During the write command, data are written to the log buffer, and then written to NVM cache. These data are also written to DRAM during the commit command. So, the data have two paths, from DRAM or from NVM, to reach the storage devices. Depending on whether the regions of virtual NVM are to be shared across applications or not, we employ different strategies. For private virtual NVM regions, *private* `mmap`, which adopts copy-on-write mechanism and all written data remain only at DRAM, is used to construct the read cache. Therefore, the data can only be written back to storage devices from NVM cache.

On the other hand, when shared NVM regions are required, *shared* `mmap` is adopted to construct the read 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. Further details are provided in section IV-D.

Fig.2 illustrates the read/write flow of accessing private virtual NVM region and the propagation of data 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) The

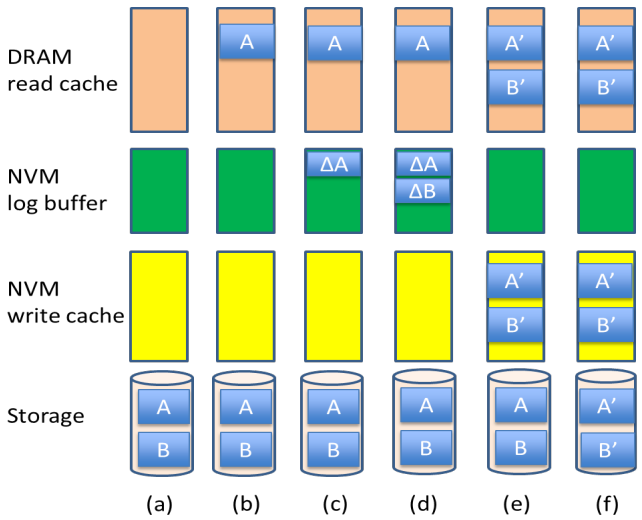


Fig. 2: The flow of read/write data of vNVML.

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.

IV. 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 sharing regions of vNVML. Finally, we discuss some general issues in vNVML implementation.

A. 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 metadata) 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 the other linked list for pages of the log buffer. Section IV-B describes the linked list data structure in more detail. The shared memory object contains and provides global information accessible by all users such as number of total current vNVML users, unique application ids assigned to each application, and unique transaction ids for each transaction. Because pages in free list and log buffer do not contain information and therefore do not relate to recovery process as well as they also need to

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.

be accessible by all applications, their linked list heads are stored in this shared memory object, too. All callers *shared* `mmap` all files created in NVM by `nv_init` command to their virtual address space. These files are mapped by vNVML and applications have no information of mapped address regions of these files, so 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 the

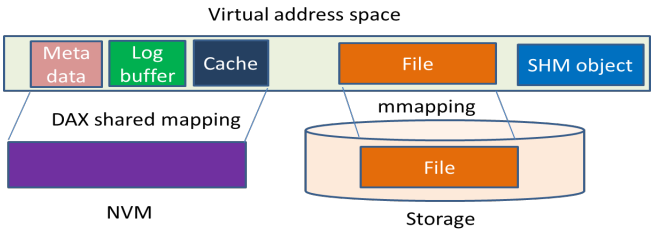


Fig. 3: The mapping of virtual address space of a process after calling `nv_init` and `nv_allocate`.

file exists, then it is opened; if it does not, a new file is created at *filepath* and `posix_fallocate` it 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, file_path) of the metadata file for recovery process if needed. Finally, A file pointer *fileptr* obtained by `mmapping` this file is returned to the caller.

Fig. 3 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 and 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` command is used to write data into virtual NVM. Through `nv_write` command, all data are written as *redo* logs in the log buffer. The first `nv_write` must allocate a log page from log buffer. If a log page is needed and 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 section IV-C.

During `nv_commit`, vNVML traces the log entries sequentially from the linked list of log object for this committed transaction *tid* and writes all committed data from log entries to the read cache. Next, 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 as well as log object in NVM. The committed list head is the metadata of the applications in NVM. All log entries in the committed list are guaranteed 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 at all, respectively). `nv_free` is used to `munmap` the file mapped by the `nv_allocate`. After `nv_release` is called, the application waits for all its committed logs, if exist, to be applied to NVM cache by a redo background worker, actively flushes all dirty cache pages back to the storage devices, and `munmaps` 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.

B. 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 list for pages is through metadata; that is, for each page a corresponding page object is created from NVM metadata file and connected to each other as a linked list. Therefore, a page object from the linked list is allocated is the same as the corresponding page is allocated.

However, as mentioned in [34], constructing the linked list in NVM is not the same as constructing a typical linked list in memory. In NVM, the virtual address cannot be used as pointer because there is no guarantee that the NVM files can be mounted into the same virtual space regions by multiple applications. 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 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. The log entry contains log header, including offset, file descriptor, and length of this entry, followed by the *redo* data. The page object (application id), the log entry header (offset, fd, and length of log), and file

record (application id, fd, file_path) already contain enough 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. Cache pages are allocated from the free list and become dirty pages attached to the 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 become clean ones and are attached to the clean list after the writeback background worker writes the dirty pages back to files in the storage devices. The dirty and clean lists implement LRU replacement.

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), offset, and dirty flag. The fd is known from the header of log entries by redo worker during it redoes. The offset is the file offset of this page. 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, file_path) are enough for recover worker to write the dirty pages back if system crashes.

Partitioning the log buffer and page cache at a 4KB 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 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.

Fig. 4 illustrates the relation between open list, committed list, log object, and page object.

C. Background workers

Two background workers (threads) are created by each process at `nv_init`. The redo background worker keeps checking the committed list. If the committed list is empty, the worker goes to sleep for a while (10us in the current configuration) and checks the committed list again after it wakes up. If the committed list is not empty, the redo worker

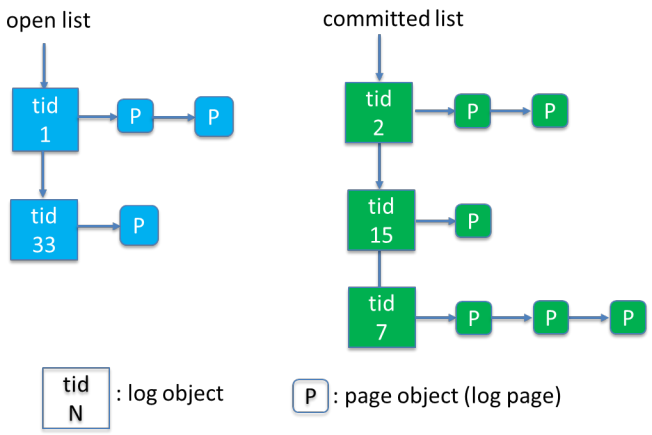


Fig. 4: Open list contains log objects of transactions. Each log object may link several page objects (of log pages). After transaction commits, the log object (along with its log pages) of the transaction is appended to the tail of the committed list. Redo worker always redoes from the head of the committed list; therefore, the transactions which is committed early would be replayed early, too.

gets the first log object (for some transaction) from the head of the committed list, and replays the log entries sequentially from log pages of this log object 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. The log pages can be returned to the log buffer pool only after the entire logs of a transaction are completely replayed by the redo worker.

The other writeback background worker is responsible for writing the dirty NVM 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), then dirty pages are written back to the storage device by the writeback worker. The dirty pages are attached to the clean list after written back. However, if the number of allocated cache pages exceeds the share because of new joining applications, the writeback worker will further release the clean pages back to the free list. Also, 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 are accumulated again.

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

D. Sharing NVM between processes

The implementation of sharing NVM regions between processes is slightly different from what we have implemented for private regions. At `nv_allocate`, a file is *shared* `mmapped` and a committed list head is maintained at metadata of NVM for each shared `mmapped` file. By sharing mmapping of read cache, processes which require to access this region can share the same view.

One limitation needs to be complied when writing to sharing regions; that is, all writing destinations of a single transaction must lie within a single shared region. When writing 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 memory region, and 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 the design. To avoid parsing all logs when flushing data back to storage devices later, a global bitmap of dirty pages belonging to this shared region is maintained in DRAM and is updated at the end of `nv_commit` command by the local bitmap of each transaction, which is constructed by parsing and replaying logs to DRAM at `nv_commit`.

After logs are accumulated to a certain threshold, at a `nv_commit` command, a background thread is triggered, which atomically copies the global bitmap to the local variable, wipes the global bitmap out to zero, and marks the tail transaction of committed list. Then several `msyncs` might be issued to flush dirty pages back to storage device according to the local variable of global bitmap. Only after flushing is completed, can logs of transactions from head to the marked tail be removed from the committed list. During the flushing procedure, other transactions can still proceed and be committed since their logs are appended after the marked tail transaction.

E. 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 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 depends on the discretion of applications.

F. 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 NVM files (log, cache, and metadata) into its virtual memory space. From section IV-B we know the recovery worker already has enough 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.

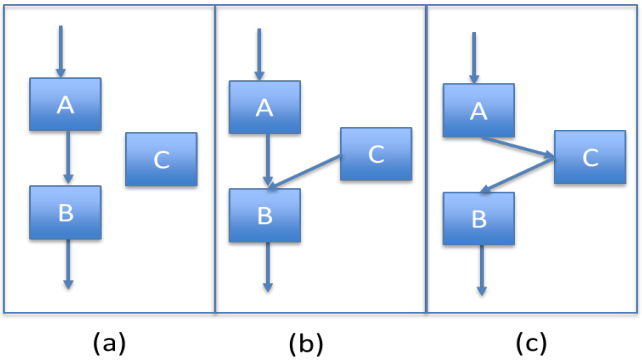


Fig. 5: The correct order of pointer updates for the objects of linked lists in NVM. From (a) to (c) is for the object insertion; from (c) to (a) is for object deletion.

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. Fig. 5 illustrates the process of insertion and deletion of a object.

After this recovery process finishes, all NVM files are erased, and vNVML can be restarted again. This recovery process can always 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.

G. 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, we guarantee the security 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 addresses), the protections within the memory system will detect these

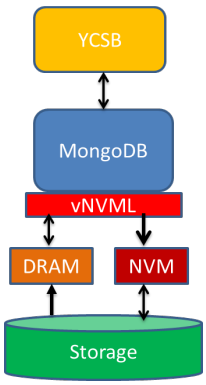


Fig. 6: The experiment setup of YCSB, MongoDB, and vNVML.

violations. In addition, the vNVML bound checks will not allow these writes to proceed.

V. 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?
- How does the vNVML perform compared to other libraries?

A. Experimental setup

Due to the absence of real NVM, we emulate NVM by DRAM [35] for all our experiments. We mount the NVM with the Ext4 file system [36] in order to utilize the DAX (direct access) feature provided by Ext4 [37].

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. All experiments are conducted three times and take the average.

B. Macrobenchmark

In this subsection, we analyze the impact of accessing NVM through vNVML by real applications. We adopt a popular open-source database MongoDB version 3.6.0 [38] as our target application because its MMAPv1 storage engine utilizes memory mapped files to access the files 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.

TABLE I: Throughputs of single MongoDB instance with different number and distribution of inserted records when NVM is the only storage device

# of records	uniform	zipfian
30K	1500 (op/s)	1505 (op/s)
10K	1509 (op/s)	1498 (op/s)

We choose YCSB [31] to generate the read/write traffic of MongoDB. The setup of experiment is delineated in Fig. 6. 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 64KB data, which is also the data written per transaction. 100K operations are executed for all experiments. We deploy the different read/write ratio and request distributions (zipfian or uniform) to observe the impact of performance.

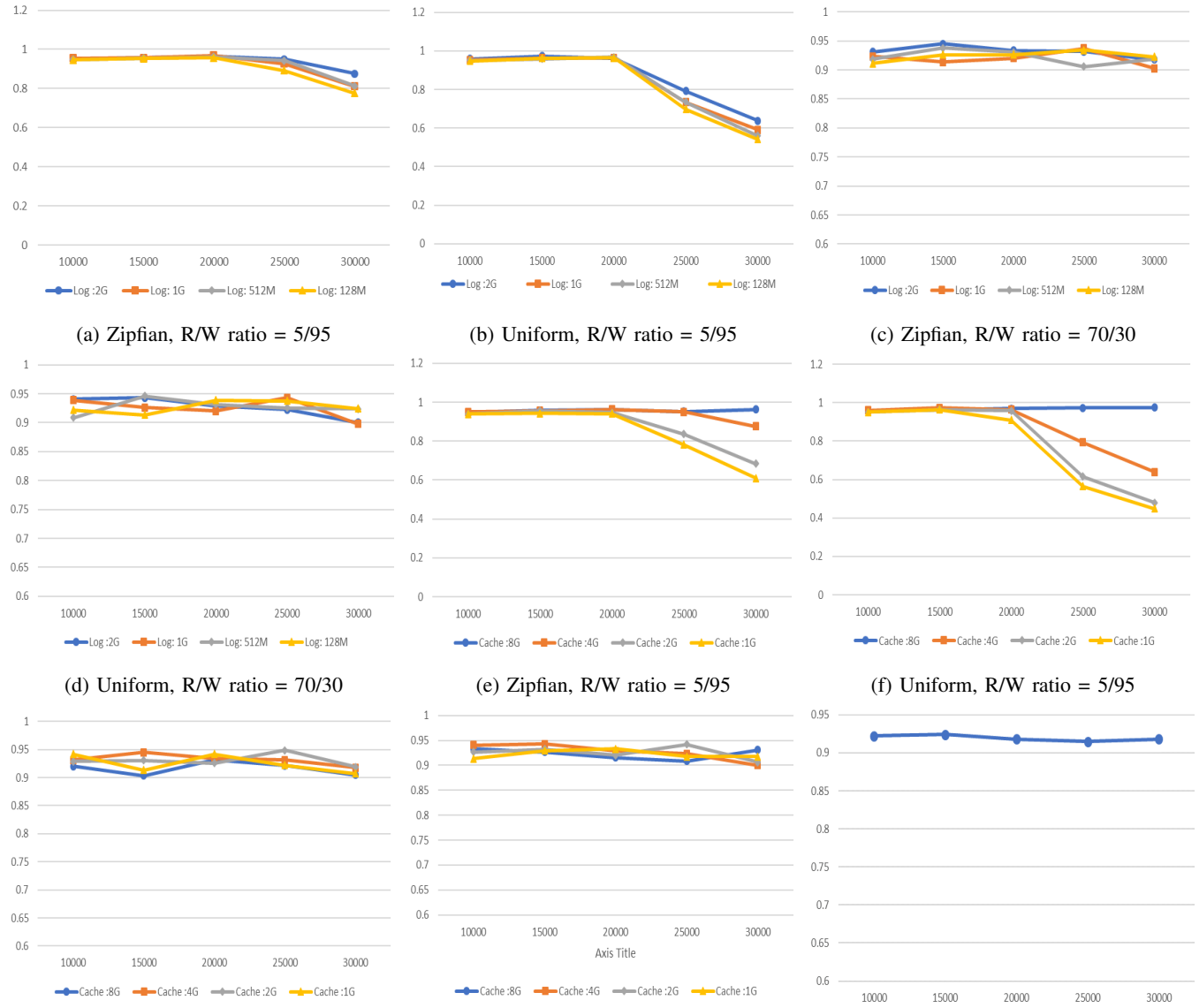
The YCSB has two phases: one is inserting the records into the databases, the other is accessing (read or write) records in the databases. To avoid polluting the NVM cache before the accessing phase, in the insertion phase MongoDB only uses its original insertion functions to access the memory mapped region, meaning that all records are inserted to the memory (due to the Copy-on-Write mechanism provided by *private* `mmap`).

All experiments are conducted by four MongoDB instances running concurrently. However, since the MMAPv1 storage engine uses padding and a power of two sized allocation mechanism [39], four instances generate 8.8GB files when each instance is inserted 10K records, and 33GB files after 30K records inserted. 12GB emulated NVM in our platform can only accommodate insertion of 13K records for each instance. However, from table I, we find that the throughputs of one instance are very close to each other even with different numbers of inserted records and distributions if all files are stored in NVM. We assume this observation still holds in the 4-instance case. Therefore, we insert 10K records to each of four instances, remove the periodic `msync` calls by MongoDB, disable journaling with *nojournal* option, and use NVM as the only storage device of MongoDBs as our baseline.

Fig. 7 shows the normalized throughputs of different request distributions (zipfian and uniform) and read/write ratios (5/95, 70/30, and 100/0). From these results we can make some useful observations.

First, the case of cache size is 1GB, log size is 2GB, and 30K inserted records (7.32GB) already proves that vNVML can provide virtualization of NVM successfully.

Second, vNVML can achieve not only over 90% of the throughput of baseline (if the log buffer and cache can handle the input traffic), but it also provides the property of persistency, which is our baseline, the MongoDB without journaling, does not. This less than 10% overhead results from writing data to log buffer and read cache.



(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

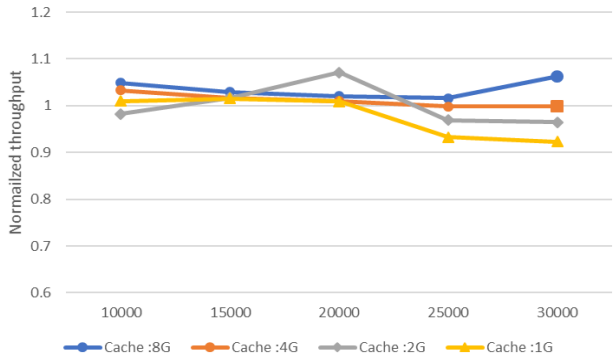
Fig. 7: Normalized total throughput of four instances. Numbers of X-axis stand for inserted records to each database, and numbers of Y-axis stand for normalized throughput. (a) to (d): Fix 4GB cache size and adjust log buffer size from 2GB to 128MB. (e) to (h): Fix the 2GB log buffer and change cache size from 8GB to 1GB. (i) 100% read uniform request.

Third, larger write working sets: (more inserted records, more random (uniform) requests), and more write traffic: (lower R/W ratio) degrade the throughput. Larger write working sets require more NVM cache to store the data at run-time; however, if the working set is even larger than the capacity of NVM cache owned by applications, which would result in cache pages written back to storage devices and therefore deteriorate the performance.

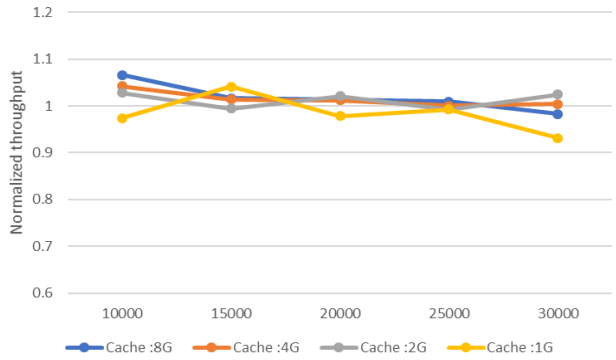
Fourth, through (a) to (d), when cache sizes are all fixed, the adjustment of log buffer only affects less than 12% throughput of baseline in all these cases.

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 throughput more significantly than that of the log buffer.

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 16GB memory is enough to handle the 30K records working set since 30K $\times$ 64K $\times$ 4 $\sim$ 7.32GB. Therefore, from above observations, we can conclude that under the limited NVM resources, NVM should be allocated more as cache than log buffer.



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



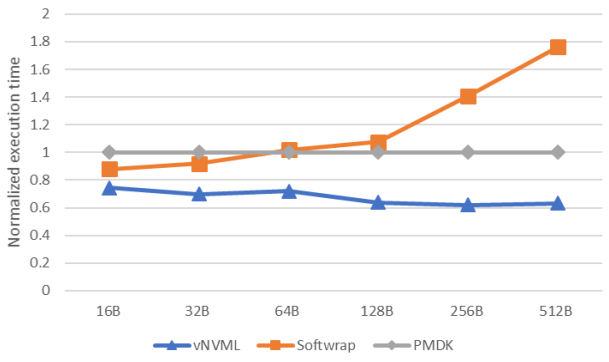
(b) Zipfian, 70/30 R/W ratio

Fig. 8: Normalized throughput of four instances inside Docker container.

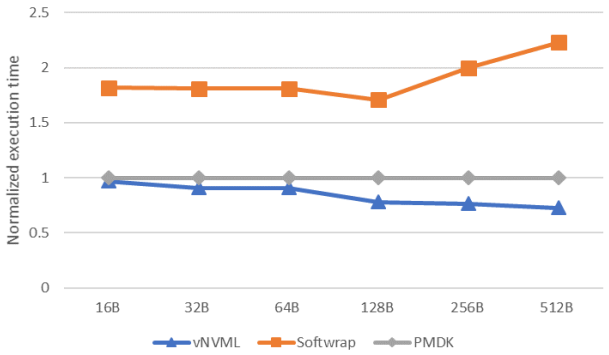
Next, we would like to examine the impact of using vNVML within docker containers [40]. Docker is a popular virtualization technique in data centers and recently has drawn significant attention from industry and academia due to its lightweight execution environment compared to traditional virtual machines. In this experiment, we launch four docker containers, use **bind mount** [41] to mount 12GB 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 2GB, 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. Fig. 8 shows that all the data are close to 1; that is, using vNVML within docker containers does not affect the performance.

C. Microbenchmark

We use a simple microbenchmark to compare the performance between Intel's PMDK [26], SoftWrAP [16], 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.



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



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

Fig. 9: Normalized vNVML and SoftWrap execution time. Numbers of x-axis stand for the amount of written data per page.

To use PMDK, we use `pmemobj_create` to create a 4GB NVM pool because 2GB NVM pool is not enough to accommodate 2GB 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 2GB log buffer and 2GB cache when running vNVML. We only use default setting for SoftWrAP since it does not provide API for internal buffer size adjustment.

Fig.9(a) shows the result. We use the total execution time of PMDK as our baseline, and show the total time of writing the 2GB array for once. The result indicates that among others our vNVML performs better as the total written data keeps increasing. Fig.9(b) shows another experiment, which we enlarge the NVM to 8GB and want to compare the upper bound of each library. We write the 2GB NVM array 16 times. Its result is similar as Fig.9(a).

VI. Conclusion

In this paper we presented vNVML, a byte-level library interface to 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 employing DRAM as a read cache.

We implemented vNVML and evaluated it with realistic workloads to show that our system allows applications to share NVM, both in a single O/S 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and feedback to improve the content and quality of this paper. 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.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, pp. 131–141, Mar. 2010.

[2] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. London, England, UK: ACM, 2012.

[3] "Intel optane technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[4] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing*. Santa Clara, CA, USA: ACM, 2016.

[5] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "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*. Istanbul, Turkey: ACM, 2015.

[6] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*. Shanghai, China: ACM, 2017, pp. 460 – 477.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys '14 Proceedings of the Ninth European Conference on Computer Systems*. Amsterdam, The Netherlands: ACM, 2014.

[8] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "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*. Santa Clara, CA, USA: USENIX, 2015.

[9] X. Wu and A. L. N. Reddy, "Scmfs : A file system for storage class memory," in *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, Washington, USA: ACM, 2011.

[10] S. Qiu and A. L. N. Reddy, "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*. Long Beach, CA, USA: IEEE, 2013.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Big Sky, Montana, USA: ACM, 2009.

[12] J. Xu and S. Swanson, "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*. Santa Clara, CA, USA: USENIX, 2016, pp. 323 – 338.

[13] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "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.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "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*. Newport Beach, California, USA: ACM, 2011, pp. 105 – 118.

[15] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. Newport Beach, California, USA: ACM, 2011, pp. 91–104.

[16] E. R. Giles, K. Doshi, and P. Varman, "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*. Santa Clara, CA, USA: IEEE, 2015, pp. 1–14.

[17] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *EuroSys '17 Proceedings of the Twelfth European Conference on Computer Systems*. Belgrade, Serbia: ACM, 2017, pp. 499–512.

[18] K. Doshi, E. R. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *HPCA '16 Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. Barcelona, Spain: IEEE, 2016.

[19] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture*. Austin, TX, USA: ACM, 2009.

[20] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "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*. Austin, TX, USA: ACM, 2009.

[21] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "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*. Davis, CA, USA: IEEE, 2013.

[22] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *HPCA '17 Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. Austin, TX, USA: IEEE, 2017.

[23] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "Frash: Exploiting storage class memory in hybrid file system for hierarchical storage," *ACM Transactions on Storage (TOS)*, vol. 6, no. 1, 2010.

[24] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *EuroSys '14 Proceedings of the Ninth European Conference on Computer Systems*. Amsterdam, The Netherlands: ACM, 2014.

[25] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future NVM storage," in *MEMSYS '17 Proceedings of the International Symposium on Memory Systems*, Alexandria, Virginia, 2017.

[26] "Intel persistent memory development kit," https://pmem.io/pmdk/.

[27] L. A. Eisner, T. Mollov, and S. Swanson, "Quill: Exploiting fast non-volatile memory by transparently bypassing the file system," in *Technical report*, UCSD, 2013.

[28] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *NVMSA '16 Proceeding of the 5th Non-Volatile Memory Systems and Applications Symposium*. Daegu, South Korea: IEEE, 2016, pp. 1–6.

[29] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016.

[30] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[31] B. F. Cooper, A. Silberstein, ErwinTam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154.

[32] "Transaction isolation levels," https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-2017.

[33] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA '14 Proceeding of the 41st annual international symposium on Computer architecuture*, Minneapolis, Minnesota, USA, 2014, pp. 265–276.
[34] S. Swanson, "A vision of persistence," https://www.sigarch.org/a-vision-of-persistence/.
[35] "Emulated nvdimm in linux," https://nvdimm.wiki.kernel.org/.
[36] "How to emulate persistent memory," https://pmem.io/2016/02/22/pm-emulation.html.
[37] "Direct access for files," https://www.kernel.org/doc/Documentation/filesystems/dax.txt.
[38] "Mongodb," https://github.com/mongodb.
[39] "Mmapv1 storage engine," https://docs.mongodb.com/manual/core/mmapv1/.
[40] "Docker container," https://www.docker.com/.
[41] "Docker container bind mounts," https://docs.docker.com/storage/bind-mounts//.