AN EFFICIENT FRAMEWORK FOR IMPLEMENTING PERSIST DATA STRUCTURES ON REMOTE NVM

A PREPRINT

Teng Ma
Department of Computer Science
Tsinghua University
mt16@mails.tsinghua.edu.cn

Mingxing Zhang
Department of Computer Science
Tsinghua University
Sangfor Inc

Kang Chen
Department of Computer Science
Tsinghua University
chenkang@tsinghua.edu.cn

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Yongwei Wu
Department of Computer Science
Tsinghua University

September 26, 2018

ABSTRACT

The byte-addressable Non-Volatile Memory (NVM) is a promising technology since it simultaneously provides DRAM-like performance, disk-like capacity, and persistency. In this paper, we rethink the current symmetric NVM deployment which directly attaches NVM devices with servers. Following the trend of disaggregation, we propose *rNVM*, a new asymmetric deployment of NVM devices, that decouples servers from persistent data storage. In this architecture, the NVM devices can be shared by multiple servers and provide recoverable persistent data structures. We develop a prototype of *rNVM* leveraging the advantages of the powerful RDMA-enabled network that naturally fits the asymmetric deployment model. With several optimizations, we demonstrate that *rNVM* can achieve comparable performance as symmetric deployment while enjoying the benefits of the large data size not limited by local memory, high availability and shared data structures. Specifically, thanks to operation batching, local memory caching and efficient concurrency control, the throughput of operations on eight widely used data structures is improved by 6∼22 × without lowering the consistency promising.

***K*eywords** NVM · Distributed Data Structure · RDMA

1 Introduction

The emerging Non-Volatile Memory (NVM) is blurring the line between memory and storage. These kinds of memories, such as 3D X-Point [1], phase change memory (PCM) [2, 3, 4], spin-transfer torque magnetic memories (STTMs) [5], and memristors are byte-addressable, providing DRAM-like performance, high density, and persistency at the same time. To fully unleash the potential of these memories, most of the existing works suggest attaching NVM directly to processors [6, 7, 8, 9, 10]. Thanks to the performance and flexibility of NVM, many high-performance implementations of persistent data structures are proposed using *load* and *store* instructions on local memory. While accessing NVM via *local* memory load/store provides good performance, it may not be the most suitable setting in the data center for three reasons.

The first reason is the *desire of sharing NVM*. On one side, since NVM has higher density than DRAM, it can provide much larger capacity [11], which would likely exceed the need of a single machine. Indeed, the applications only need a less expensive NVM with smaller capacity [12]. However, in the data center servers, since each type of resource is isolated from each other, they cannot fully utilize all data center resources [13, 14] (i.e., NVM). Google [15] shows the resource utilization is lower than 40% on average.

As a result, a reasonable scenario is to share a single NVM device among multiple machines for better utilization. Therefore, deploying one NVM device for each low-end machine in data centers incurs inevitable high cost, — not a very economic choice in the long term.

The second reason is the *requirement of availability*. NVM itself only provides persistent. Once an NVM device is attached to a specific machine, its data become unavailable if the host machine goes down. This violates the availability requirement of many data center applications. Thus, even an NVM device is attached to the local processors, one still needs to replicate the data to a remote NVM to achieve high availability. However, it requires that the remote NVM contains the same data as in the local NVM, limiting the size of data that the same NVM devices can support.

The third reason is the *trend of disaggregation architecture*. As described by Gao *et al* [16], the recent industry trend suggests a paradigm shift from servers each tightly integrated with a small amount of various resources (i.e., CPU, memory, storage) to the disaggregated data center built as a pool of standalone resource blades and interconnected using a network fabric. In the industry, Intel's RSD [17] and HP's "The Machine" [18] are state-of-the-art disaggregation architecture products. The disaggregated architecture is not limited by the capability of a single machine, and can provide better resource utilization and the ease of hardware deployment. Due to these advantages, it is considered to be the future of data centers [19, 20, 21, 18, 22]. With this setting, the NVM devices should be treated as the disaggregated resource not associated with any server. It also improves the availability because the crash of a server will not affect the data availability.

In essence, these challenges are due to the *symmetric* nature of the current deployment of NVM [23, 24, 25], where each machine is attached with a local NVM device so that most NVM accesses are performed locally. To fundamentally overcome the drawbacks, we envision an *asymmetric* NVM deployment approach, in which the NVM devices are not associated with individual machine and should be accessed only or mostly via the network. In this asymmetric setting, the number of NVM devices can be much smaller than the number of machines, and they can be provided as specialized "blades".

Asymmetric setting (and resource disaggregation) is enabled due to the high performance network e.g., RDMA. RDMA provides the direct remote access capability with RDMA verbs (`RDMA_Write`, `RDMA_Read`, etc.) as well as reliable connection. In order to realize the persistent data structures to an asymmetric setting, a straightforward implementation is to replace the local store/load instructions with `RDMA_Write` and `RDMA_Read` operations. Unfortunately, although the throughput of RDMA over InfiniBand is comparable to the throughput of NVM, the NIC typically cannot provide enough IOPS for fine-grained data structure accesses. To solve this problem, Mojim [26] modifies the Linux kernel to provide two additional APIs (`msync`/`gmsync`), which allow users to synchronize one/multiple continuous memory regions to remote NVM via a user-transparent log channel. With the two APIs, Mojim successfully reduces remote access rounds to one per synchronization point, gaining a good performance. However, Mojim is still designed for symmetric setting with the inherent limitations discussed earlier. In particular, it requires that the remote memory contains same data structure as the local NVM.

In this paper, we propose *rNVM* (Remote NVM), a general NVM deployment framework for implementing data structures with high performance and availability on remote asymmetric NVM devices. Our key insight is that a naïve application of batching and caching technique would lead to data loss during a crash. In *rNVM*, we solve this problem by recording modifications as log entries and sending them to remote NVM as transactions. The availability can be ensured by replicating logs in remote NVM devices to mirrors. Importantly, different from Mojim that requires the entire data structure in the local memory, *rNVM* only requires a small local volatile space for caching the hot data. According to our evaluations, a cache that as small as 10% of the data in the NVM is enough for achieving a good performance.

To demonstrate the capability of *rNVM*, we implement eight widely-used data structures and compare them with: *1)* a naive implementation that simply replace local store/load with `RDMA_Write/Read`, whose usage is similar to NVMe over Fabric [27, 28] (a storage architecture which accesses remote storage media via RDMA directly); and *2)* an implementation that both use local load/store instruction to modify the local NVM and records its operations to a remote NVM, which is used to mimic the best-possible performance in a symmetric setting. Using various optimizations including operation batching, local memory caching and efficient concurrency control, the *rNVM* throughput for operations on the data structures is improved by 6∼22× compared to the naive asymmetric implementation. Overall, we demonstrate that *rNVM* can provide comparable performance as the highly available data structures implemented in the symmetric settings.

The remainder of this paper is organized as follows. Section 2 discusses the deployment of NVM device with the detail analysis and propose our novel asymmetric NVM deployment architecture. Section 3 presents *rNVM* and gives solutions based on three challenges over asymmetric deployment. Section 4 describes our basic APIs and the implementation details. Section 7 and Section 9 introduce how to apply general optimizations and special improvement according to the different types of data structures and how to support concurrent accesses. Section 10 depicts the evaluations of *rNVM*. Section 11 shows the related works followed by conclusions in Section 12.

2 NVM Devices Deployment

We consider three different settings of deploying NVM devices: *1)* single-node setting that only contains one machine and one NVM device; *2)* symmetric distributed setting, where each machine in the cluster is attached with an NVM device; and *3)* asymmetric distributed setting, in which the number of NVM devices can be much smaller than the number of machines and they can be even attached to only a few specialized “blades”. Thus, NVM device/blade is shared by multiple client machines, and the memory space of these client machines may be much smaller than the capacity of the NVM devices.

2.1 Single-Node Local NVM

Due to the DRAM-like performance of the byte addressable NVM, previous studies suggest that NVM should be directly accessed via the processor-memory bus using load/store instructions. This design avoids the overhead of legacy block-oriented file systems or databases. Persistent memory also allows programmers to update persistent data structures directly at byte level without the need for serialization to storage.

Based on this setting, many kinds of persistent data structures are proposed. For example, CDDS-Tree [10] uses multi-version to support atomic updates without logging. NV-Tree [29] is a consistent and cache-optimized B+Tree, which reduces CPU cache line flush operations. HiKV [30] constructs a hybrid index strategy to build a persistent key-value store. Since all the data accesses are performed by local store/load instructions, these implementations can offer the best performance. However, although these persistent data structures can survive a failure of machine, they are not accessible during the recovery/restarting.

2.2 Symmetric Distributed NVM

Symmetric architecture is widely used in distributed systems (e.g., shared memory and distributed file systems). In symmetric architecture, each machine in the system has its own NVM device for discussion in this paper. In order to achieve availability on top of persistency, one needs to replicate its data structures to multiple NVM devices. Mojim [26] implements this mechanism by adding two more synchronization APIs (msync/gmsync) in the Linux kernel. Specifically, it allows users to set up a pair of primary node and mirror node. Once these synchronization APIs are invoked, Mojim efficiently replicates fine-grained data from the primary node to the mirror node using an optimized RDMA-based protocol. This synchronization is implemented by appending logs of primary node with end marks to the mirror node’s log buffer, thereby tolerating a failure of the primary node. Mojim also allows users to set up several backup nodes that are the only weakly-consistent replication of data in primary node.

With a similar interface, Hotpot [31] extends Mojim to a distributed shared persistent memory system. It provides a global, shared, and persistent memory space for a pool of machines with NVMs attached at the main memory bus. As a result, applications can perform native memory load and store instructions to access both local and remote data in this global memory space and can at the same time make their data persistent and reliable. To achieve this, when a committed page is written, Hotpot creates a local copy-on-write page and marks it as dirty. These pages are not write-protected until they become committed after an explicit invoking of the synchronization APIs. At this point, the modifications to this page from all nodes are finished.

The two systems are designed for symmetric usage of NVM. As a result, Mojim requires a full replication of the data structure in local memory. Similarly, Hotpot assumes that the dirty page can always be held in memory and only uses a simple LRU-like mechanism to evict redundantly and committed pages. These implementations make them not suitable to be used in an asymmetric setting, where the local memory of client machines can be much smaller than the data.

2.3 Asymmetric Distributed NVM

To the best of our knowledge, there are currently very few works on asymmetric distributed NVM. Octopus [25] can support asymmetric service with file interface but its current implementation is using the symmetric setting. Mitsume [32] is the only work that can naturally support asymmetric deployment of NVM devices for providing object store interface. We make a step further to support **byte addressable data structures**.

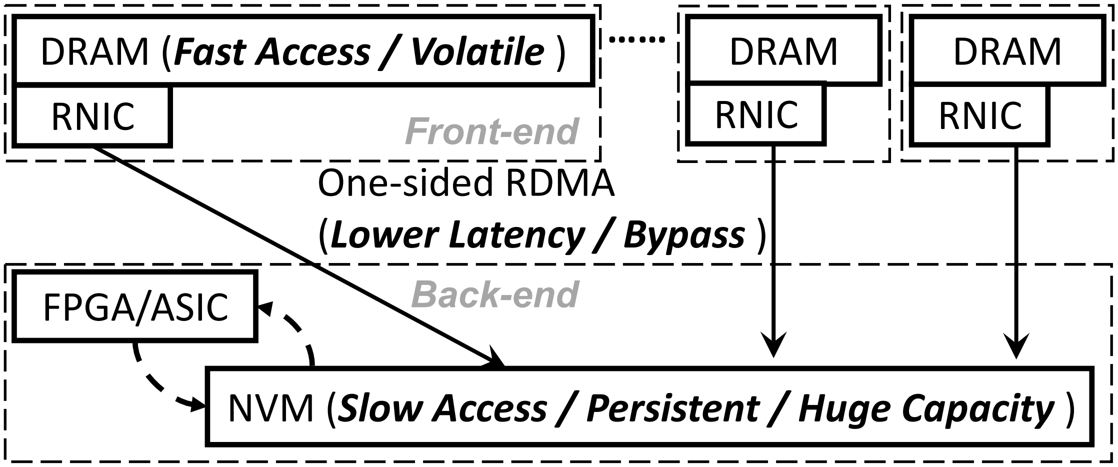

DRAM (*Fast Access / Volatile*)
RNIC
Front-end
DRAM
RNIC
DRAM
RNIC
One-sided RDMA
(*Lower Latency / Bypass*)
Back-end
FPGA/ASIC
NVM (*Slow Access / Persistent / Huge Capacity*)

Figure 1: *rNVM* Architecture

3 *rNVM* Design Overview

3.1 Asymmetric Deployment Challenges

Figure 1 shows the asymmetric deployment of NVM devices, which includes two components performing different functions: *1)* the *back-end nodes* that have NVM attached to their memory bus;a and *2)* the *front-end nodes* that actually operate the data structures on NVM. In the asymmetric architecture, front-end nodes can only access the back-end nodes via the network. Specifically, the relationship between front-end and back-end nodes is "**many-to-many**" (i.e., a front-end node can access multiple back-end nodes and a back-end node can also be shared by multiple front-end nodes). Compared to symmetric deployment, this setting offers three advantages: *a)* it follows the promising trend of disaggregated data centers; *b)* it matches the desire of sharing NVM devices; and *c)* it provides proper availability with multiple back-end nodes. Despite the benefits, this architecture also brings a few challenges.

The first challenge is the *network latency*. Although the bandwidth of InfiniBand is comparable to NVM, the RDMA operation RTT is about 2 μs, which is much larger than the latency of NVM write (about 200 ns). If we simply replace local read/write operations with `RDMA_Read` and `RDMA_Write` operations, the performance will significantly degrade.

The second challenge is how to efficiently use *the small volatile space* of the front-end nodes. Keeping a full copy of the data structure in the front-end (like Mojim) can always offer the best performance. However, it contradicts with the original purpose of the asymmetric setting. The high density of NVM devices makes it capable to hold terabytes of data. On the other side, the memory space of the front-end is typically to be only tens of gigabytes. This asymmetry means that only the necessary data in the current work-set should be loaded into the front-end for caching.

The third challenge is how to design the interface of the back-end nodes that is *both simple and effective*. The most straightforward interface is to assume that the back-end nodes can be programmed to perform arbitrary RPC calls. However, to ensure high reliability, the back-end nodes need to be simple. Therefore, they should be only asked to perform a small collection of simple APIs, such as remote memory read/write, remote memory allocation/release, lock acquire/release, etc. With a limited interface that constitutes a small set of *fixed* APIs, it is possible to implement the simple logic of the back-end nodes in specialized hardware (e.g., using ASIC or FPGA), instead of using a general-purpose CPU.

To address the challenges, we design *rNVM*, a general framework for implementing high-performance data structures in an asymmetric deployment of NVM. In our framework, we assume that all persistent data are hosted on the back-end remote NVM devices and can be much *larger* than the limited size of the local volatile memory in the front-end. Moreover, the back-end nodes are always passive: they never actively start a communication with the front-end nodes, but only passively response to the API invocations from the front-end.

We separately consider the problem of the exclusive use of the data structure (i.e., a certain data structure is only accessed by one front-end node) from the concurrent accesses. In the following discussion, we firstly assume that only one front-end node accesses the data, and then study the concurrent control related issues in Section 9.

3.2 Remote NVM Interface

We assume the back-end nodes are equipped with advanced NIC that supports RDMA. In other words, the front-end can directly access their data via `RDMA_Read/Write` operations. Although it is possible to implement any kind of data structures with only RDMA operations, we find that the implementation can be much more efficient with a few *simple and fixed* functions provided by the back-end. Specifically, the back-end provides two sets of APIs in addition to RDMA verbs.

The first set of APIs handles *memory management*, which includes remote memory allocation, releasing, and global naming. We implement a two-tier slab-based memory allocator [33]. The back-end runs in the remote NVM to ensure persistency and provide the fixed-size blocks. The front-end deals with the finer-grained memory allocations.

The second set of APIs provides a transactional interface that allows the front-end nodes to push a list of logs to the back-end, which are guaranteed to be executed in an all-or-nothing manner. To keep the transactional interface simple, only a collection of {memory address, value} pairs are included in a transaction, and the back-end nodes only need to ensure that all these addresses are updated atomically. Each pair is called ***memory log***. It might be natural to also design the interface to handle concurrency control, e.g., to implement certain lock mechanism. However, the similar functions also exist in RDMA standard, there is no need of re-implementing a special set of APIs.

3.3 Performance Optimization

Strawman Design. In general, a data structure implementation should support two types of operations: read-only operations that retrieve data from the data structure; and modification operations that update the internal state of the data structure. With the back-end (remote NVM) APIs, the two operations can be implemented by retrieving all the needed data via a series of RDMA reads, and pushing all the modifications to the back-end via a single write transaction.

However, this straightforward implementation poses two challenges. First, it may incur considerable rounds of network communications and the network latency is much higher than memory writes. Second, since the volatile local memory is typically much smaller than the whole data structure in remote NVM, the front-end nodes may not be able to hold all the needed data even for only a single data structure. As a result, a proper data eviction mechanism is needed, which at the same time should still ensure good performance.

Batching and Caching. To reduce the network latency, we can use *batching* and *caching*, two classical techniques that are applicable to any data structure. With batching, multiple updates from the front-end could be sent together to the back-end. With caching, certain portion of data could be stored in the local volatile memory of the front-end for fast access.

While effective and supported by many existing systems, the straightforward use of the two techniques in our scenario violates the consistency and persistency requirement. Once a modification operation is returned at the front-end node, the system must ensure that the modification will never be lost. Unfortunately, the transfer of this operation to the back-end nodes can be delayed due to caching or waiting to be batched with other operations.

Batching and Caching with Operation Log. To correctly ensure the consistency and persistency, we record an ***operation log*** in the remote NVM before returning to the user in the front-end. With these operation logs, we can make sure that all the data are recoverable even the front-end node is crashed. Specifically, in *rNVM*, we maintain two different data areas in the remote NVM for each data structure: the data area holding the real data structure, and the log area for append-only operation logs. When a modification operation is executed by the user, *rNVM* only appends the operation log to record this operation and its parameters and leaves the data area totally untouched. With the transactional interface, this log-appending operation can be achieved with a *single* one-sided RDMA write that records the memory modifications at the remote NVM. Once the operation logs are recorded, the modifications on the real data structure can be postponed and batched to improve the performance while ensuring crash consistency (e.g., asynchronous execution to remove network latency from the critical path, and combining redundant writes to reduce write operations). This is because, even after a crash, the proper state can be restored by replaying the operations that are not executed (i.e., have not yet modified the data area).

Since the un-executed operation logs may be accessed frequently, we always cache them in the local memory of the front-end nodes. This requirement does *not* lead to lower the front-end node's memory space, because we just need to flush out the logs by executing a batch-modification operation to transfer the state of all these logs from un-executed to executed. Thus, a smaller space for caching un-executed logs only means smaller space for batching. The detailed implementation of the batched operation is tightly bounded to the specific data structure. We will describe the details of the optimization for several common data structures in Section 8.

Under such design, the first problem is to correctly generate the read results. Since the modifications are all first recorded only in the operation logs, the workflow of reading data from the data structure is different from current system. Specifically, the up-to-date data is a combination of both the "base" data in the data area in remote NVM at the back-end nodes and the recent un-executed modifications in the record log. The exact method of this combination depends on the specific data structure. For example, for a pop operation to the stack data structure, we need first to count the number of un-executed push and pop operations in the operation log. If the number of pushes is larger than the number of pops, there is no need to access the data area. The second problem is the proper data replacement policy in local memory for better performance. One option is to simply use LRU to cache recent touched pages in local memory, however, according to our evaluation, a more advanced custom caching mechanism can largely improve the performance of certain kinds of data structures. The details are discussed in Section 7.2.

4 Back-End Implementation

The *rNVM* framework mainly consists of two parts, i.e., the back-end component that is implemented in remote NVM device and the front-end algorithm that is executed by the front-end node. In this section, we describe the implementation of the back-end APIs, which is an extension of the basic RDMA verbs to provide some of the advanced functions. This extension includes two sets of APIs, one for memory management, and the other one provides transaction semantics. The implementation goal is to ensure crash-consistent with the least number of RDMA operations. The front-end algorithm to implement high performance and persistent remote data structures in *rNVM* will be discussed in the next section.

4.1 Usages of RDMA

There are two common programming paradigms of RDMA. The straightforward way is the server-reply [34, 35] paradigm which directly replaces traditional send/recv with RDMA_Send/Recv. The other one is server-bypass paradigm using one-sided RDMA, which requires the system re-design to exploit such feature [36, 37, 38]. *rNVM* uses one-sided RDMA to improve the performance. This means that the front-end nodes can access the memory space on remote NVM devices directly via `RDMA_Write`, `RDMA_Read`, and even RDMA atomic operations without notifying the process unit (e.g., CPU or FPGA/ASIC chips) on the remote side.

With asymmetric architecture, back-end should manage metadata consistently. Fortunately, RDMA provides several atomic verbs to guarantee that any update to a 64-bits data is atomic. So we apply RDMA atomic operations to these critical metadata, e.g., root pointer of data structure.

Due to the non-volatility of the remote NVM, the data may be corrupted if the back-end crashes during a single `RDMA_Write` operation. *rNVM* relies on the software solution to guarantee the data integrity via checksum.

4.2 Transactional Operations

The transaction APIs are implemented in the back-end with two areas in the remote NVM: the *data* area holds the real data structures, and the *log* area records the transaction logs. The front-end can directly read the data area. All the write operations are achieved by invoking the transaction API, i.e., `remote_tx_write`. The input parameter of this function is a list of {address, value} pairs, each of which consists of a memory address and a value that should be written to this address. The library will construct a continuous set of memory logs and append to the corresponding log area in remote NVM via a single `RDMA_Write` operation. The format of these memory logs is shown in Figure 2. Every log entry includes address, length, data and one byte flag in the head to distinguish from commit flag. A transaction will produce several log entries, a commit flag and a *checksum* value. The checksum of a transaction is recorded as the end mark and can be used to validate the integrity of the appended log. After the restart of the back-end node, it needs to use the checksum of the last transaction to validate whether all the log entries of this transaction is flushed to the NVM.

The advantage of using the transactional API is that it can largely reduce the required rounds of RDMA operations. Otherwise, multiple rounds of RDMA operations are needed when the modification operation writes to *1)* multiple non-continuous areas of the NVM; *2)* a continuous area with size larger than a cache line. Other works [39, 40, 41] propose to add an additional flush operation to the RDMA standard. However, such resolution will at least add the additional latency of invoking this flush operation. Moreover, the additional operation itself does not make the other RDMA operation crash-consistent. Importantly, the implementation of the transactional API is fixed and simple, making a hardware solution possible. It improves the reliability of the back-end nodes.

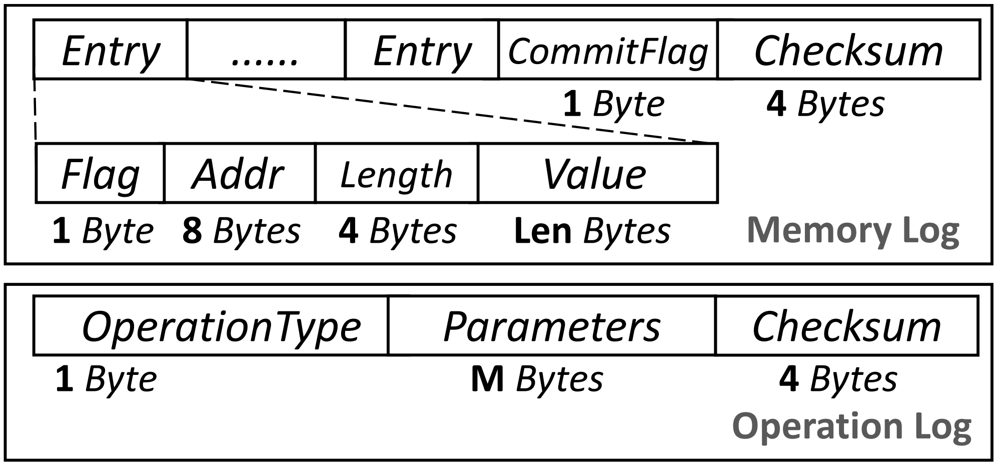


Figure 2: Memory Log vs. Operation Log

4.3 Replication

In Mojim (symmetric architecture) [26], it adopts a primary-mirror architecture to avoid complex protocols for achieving availability. Similar as Mojim, *rNVM* also needs **at least** one mirror-node with attached non-volatile device like SSD, Disk or even NVM. Moreover, it would be best to deploy mirror-nodes to different racks for improving tolerance. The back-end will replicate the memory/operation logs to mirror-node before committing the transaction. Mirror nodes are read-only. In the case of back-end crash, front-end can use the data of mirror nodes to recover the data structure to a new NVM device.

4.4 Memory Management

Memory management APIs are required because implementing such operations using only one-sided RDMA operations is difficult and inefficient. In addition, since it is a basic function needed by all applications, it is worthwhile to implement it directly in the back-end to reduce the network communication to only one round for RPC invocation. In *rNVM*, two memory management APIs are provided: `remote_nvm_malloc` and `remote_nvm_free`. The front-end node can use them to allocate and release NVM memory in the back-end. To ensure simplicity, we decided to only implement fixed-size memory management in the back-end. Moreover, the memory management needs to maintain persistent allocation status. We use a persistent bitmap to record the usage of NVM, with one bit indicates the allocation status of each block. These two design decisions ensure the fast recovery.

Given that the front-end nodes connect to the back-end via one-sided RDMA, we use the RFP [38] design to implement these two interfaces. Because the front-end puts the requests via `RDMA_Write` and gets the responses via `RDMA_Read`, the back-end does not need to deal with any network operation. This will simplify the implementation of the back-end.

The back-end nodes need to store metadata for recovery purpose since nothing will be left on the front-end after failure. In *rNVM*, the metadata are stored in the "well-known" locations to all front-end and back-end nodes. This is the *global naming* space for recovery. After restarting, both the front-ends or the back-end know the location to find the needed information/data before recovery. We have the following metadata stored in the global naming space. *1)* During recovery, the front-ends need to know their own portion of NVM area including the data and the logs. It is needed for translation from the physical to virtual address for the corresponding front-end. *2)* The front-ends need to know the location of corresponding data structures. It is achieved by storing the root reference of data structures, e.g., the address of root node for a tree. *3)* The allocation bitmaps indicate whether a block of NVM is allocated. This information is used to reconstruct the memory usage lists. *4)* Addresses of log areas and the LPNs (Log Processing Number indicating next entry in the log sequence for processing) are used to find the logs together with the location of the next logs. They can be used for the back-end to reproduce logs (memory log) and also be used for the front-end to recover the data structure operations (operation log).

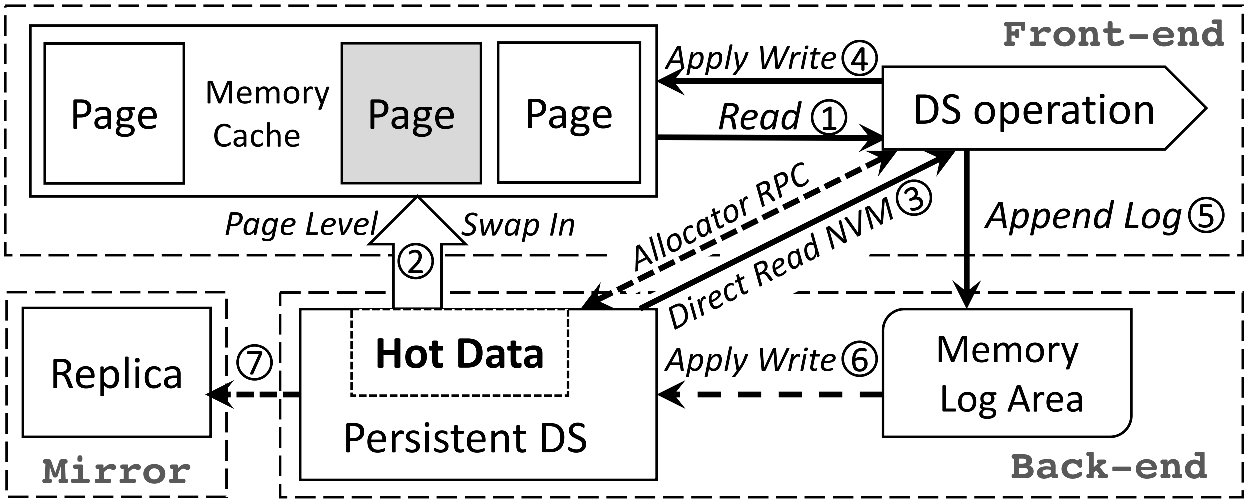


Figure 3: General Data Access Workflow

5 Data Structure Implementation

With the simple but efficiency interfaces provided by the back-end nodes, programmers can implement various kinds of data structures. To achieve high performance with persistency and consistency guarantee, programmers need to follow certain general guidelines and may need to apply specific optimizations for different data structures.

5.1 General Work Flow

Figure 4 shows the work flow of data structure operations. Each modification operation typically needs to first read the data and then write to perform the modifications. We divide each operation into two parts in a *Gather-Apply* model: gathering data and applying the modifications. Obviously, read-only operation only needs the gather part. We apply batching and caching to improve performance. Batching can execute multiple operations together and coalesce memory logs to both reduce `RDMA_Write` and `RDMA_Read` overhead. Besides, caching will reduce the `RDMA_Read` overhead during the gathering phase. They are applicable for all data structures. The general data access work flow is in Figure 4.

1) Gather Data: The data are fetched from the front-end cache whenever possible (cache-hit, ①). If not cache-hit, the data will either be read from the back-end directly by using `remote_nvm_read` (③) or a page will be swap-in (②) and put to the cache in the front-end memory (Accordingly, read data in front-end cache via ①). The choice between these two strategies depends on specific data structures and follows a principle that using ② for cold data and ③ for hot data.

2) Apply Modification: Each modification operation causes one operation log to be flushed to the back-end for recovery. After the operation log is stored in the back-end, meaning data structure is recoverable, the flush of corresponding memory logs can be delayed in batch. The cached data (if exist) are modified accordingly (④). If a bunch of operations get executed successfully or the buffer is full, the buffered memory logs, together with appended `TX_COMMIT`, will also be flushed to the back-end NVM via `remote_tx_write` (⑤). These logs are then handled by the back-end (⑥) and replicated to the mirror-node (⑦). If the back-end fails, the front-end takes care of exceptions, abort the transaction and clear the cache.

This *Gather-Apply* model can simplify the data work flow by merging data access operations.

5.2 Data Cache in Front-end

Several recent works build NVM systems using DRAM as cache [42, 9, 30]. Bw-tree [43] uses a cache layer to map logical pages (Bw-tree nodes) to physical pages. We use a similar data structure of hash map to translate address of data structure nodes in NVM to address in DRAM. Each item in the hash map represents the page cached. The page size is adjustable according to different data structures.

Our cache replacement policy combines the methods of LRU (Least Recently Used) and RR (Random Replacement). LRU works well in choosing hot data. However, its implementation is expensive. RR is easy to implement but does not provide any guarantee of preserving hot data. We use a hybrid approach, which firstly chooses a random set of pages for

Table 1: Comparison of Different Allocators. We set the slab size as 128 Bytes and 1024 Bytes separately.

Type/Tput(MOPS)	Alloc	Free
Glibc	21.0	57.0
Pmem	1.42	1.38
RPC allocator	0.33	0.88
Two-tier allocator (128 Bytes)	1.33	2.41
Two-tier allocator (1024 Bytes)	6.42	13.90

replacement (RR) and then a least used page from the set will be discarded (LRU). No page flush is needed because the write work flow already put the memory logs in the back-end. In the micro-benchmark, the hybrid approach (29.2%) can reduce the miss ratio by 33.5% compared to RR (62.7%) when the size of choosing set is 32, and gain a similar miss ratio as LRU with nearly 27.5% throughput improvement.

5.3 Batching Memory Logs

Since data structure operation logs will be flushed to remote NVM in back-end with persistency and recoverable guarantee, the flush of memory logs can be asynchronous and batched.

For example, when a new item is inserted into a binary search tree, the operation log with format {insert_op, key, value} as shown in Figure 2 will be put in the NVM log area. Then, the memory logs with the format {address, data} will be generated afterward. They do not need to be flushed immediately. The flush can be triggered by the conditions such as log buffer full or having finished a certain number of operations. At that time, a group of memory logs can be flushed in a batch. Batching can efficiently reduce the network rounds for invoking `remote_tx_write`. More data structure specific optimization using batching are discussed in Section 8.

5.4 Front-end Allocator

The design of the front-end allocator is inspired by the slab allocator [33]. Slabs are provided by the back-end allocator. The slabs in the front-end are organized in three lists of full/partial/empty list according to how much capacity is consumed in the corresponding page. We use a simple *best-fit* mechanism in the front-end. To improve the NVM utilization, a threshold is defined as the maximum free blocks number, and the front-end will reclaim free blocks periodically. When allocation size is larger than the size of a slab, front-end will switch to a mode to directly allocate memory from the back-end.

Benchmark. We measure our *rNVM* two-tier allocator with persistent allocator and standard Linux *Glibc* allocator. As table 2 shows, *Glibc* performs the highest throughput (21.0 MOPS and 57.0 MOPS) but without persistent guarantee. *Pmem* allocator is a persistent allocator from NVML project [44], and can reach 1.42 MOPS. Our original allocator's throughput is only 23% and 64% of Pmem allocator because of the network overhead. After applying the improvements, our persistent allocator can achieve similar or even better performance than Pmem allocator.

5.5 Data Structure Recovery

With the log mechanism, *rNVM* can keep crash consistency because all the data and logs are stored in the back-end. Due to the different component failure, recovery will take the different approach. In the case of front-end failure, another front-end will be selected as the sole writer and will recover the data structure according to the data and logs stored in the back-end. The back-end is critical for system to run correctly because all the data and logs are persistent. If the back-end faces transient failure, the back-end will reboot. After each reboot, the back-end will first reconstruct the mapping between the physical addresses and virtual addresses. The reconstruction is possible because such mapping is also stored in NVM. After that, the back-end can start its normal work immediately, i.e., reproducing memory logs to data structures if any log has not been applied. For the permanent failure of the back-end, the front-end will reconstruct the data structures to a new back-end by using the data and logs in the mirror nodes.

6 Data Structure Specific Optimization

The work flow and optimizations discussed so far are sufficient for most kinds of data structures to get the performance improvement. In this section, we present the data structure specific optimizations to further improve the performance.

6.1 Stack/Queue

Stack and Queue can be implemented by using the List data structure. Because the only data items that can be accessed in Stack or Queue are headers or tails, caching and batching can be specialized based on this property. Since the head and tail are more frequently accessed, the front-end typically only need to cache nodes pointed by them to reduce `remote_nvm_read`. For batching, the operations can be compacted because the operations are only allowed on stack header for Stack, and on queue tail for Queue. Thus, the effective pushes will be annulled by pops for Stacks, meanwhile the effective en-queues will be annulled by de-queues for Queue. The compaction will reduce the logs to only contain effective memory logs.

6.2 Tree-Like Data Structure

Tree-like data structures have the hierarchical organization. The nodes in higher (near to the root) level are more frequently accessed than lower level nodes. Based on this nice property, we can cache higher level data nodes. Specifically, the front-end sets a threshold N and the nodes with level larger than N will not be cached. They will be directly accessed through `RDMA_Read`. N is dynamically adjusted according to the cache miss ratio α, i.e., if $\alpha > 50\%$, $N = N - 1$ while if $\alpha < 25\%$, $N = N + 1$. Otherwise, N keeps unchanged.

Because trees are sorted data structures, the performance can be improved if the operations are sorted. We pack the sorted operations in *vector operation*. The operation goes from the root of the tree down to the leaf nodes. The vector can then be split accordingly. The operations in vector segments can be executed in parallel.

Algorithm 1 Vector Insert

```
procedure VECTOR INSERT($keys, values$)
    $node \leftarrow root$
    $queue.push(< 0, len, node >)$
    while queue is not empty do
        $begin, end, node \leftarrow queue.pop()$
        $mid \leftarrow$ binary_search($begin, end, node.key$)
        if $node.left = null$ then
            create_sub_tree($kvs[begin : mid]$)
                                                        ▷ construct a new sub tree
            $node.left \leftarrow sub\_tree$
        else
            $queue.push(< begin, mid, node >)$
        end if
        if $node.right = null$ then
            create_sub_tree($kvs[mid : end]$)
            $node.right \leftarrow sub\_tree$
        else
            $queue.push(< mid, end, node >)$
        end if
    end while
end procedure
```

The `vector_insert` in Algorithm 2 shows `vector_write`, one *vector operation*, in binary search tree following the Gather-Apply paradigm. It firstly reads the information to decide where to insert these nodes and then applies these insert in the correct position. Without batching, two read rounds are needed if insert operation A and B read the same node. When we execute A and B with one `vector_write` operation, it only needs one round read to access this node. Similarly, if several operations modify the same NVM memory, they will be compacted to one NVM write in `vector_insert`.

The caching and batching optimization described for tree-like data structure can also be applied to skip-list. Specifically, higher degree nodes in skip-list will be cached. Vector operation (containing sorted operations) for skip-list can similarly reduce the number of `RDMA_Read` calls.

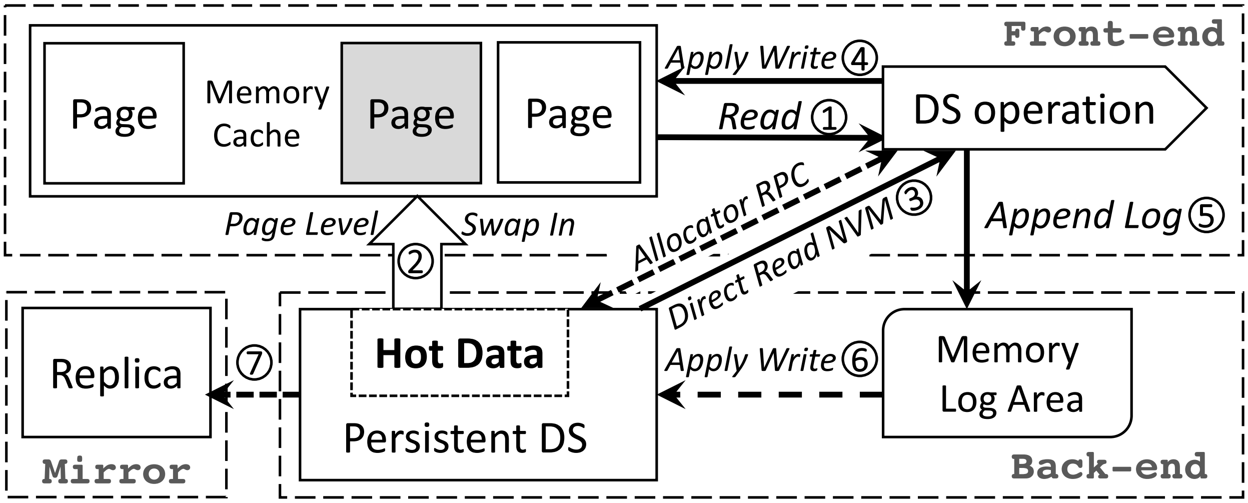


Figure 4: General Data Access Workflow

7 Data Structure Implementation

With the simple but efficiency interfaces provided by the back-end nodes, programmers can implement various kinds of data structures. To achieve high performance with persistency and consistency guarantee, programmers need to follow certain general guidelines and may need to apply specific optimizations for different data structures.

7.1 General Work Flow

Figure 4 shows the work flow of data structure operations. Each modification operation typically needs to first read the data and then write to perform the modifications. We divide each operation into two parts in a *Gather-Apply* model: gathering data and applying the modifications. Obviously, read-only operation only needs the gather part. We apply batching and caching to improve performance. Batching can execute multiple operations together and coalesce memory logs to both reduce `RDMA_Write` and `RDMA_Read` overhead. Besides, caching will reduce the `RDMA_Read` overhead during the gathering phase. They are applicable for all data structures. The general data access work flow is in Figure 4.

1) Gather Data: The data are fetched from the front-end cache whenever possible (cache-hit, ①). If not cache-hit, the data will either be read from the back-end directly by using `remote_nvm_read` (③) or a page will be swap-in (②) and put to the cache in the front-end memory (Accordingly, read data in front-end cache via ①). The choice between these two strategies depends on specific data structures and follows a principle that using ② for cold data and ③ for hot data.

2) Apply Modification: Each modification operation causes one operation log to be flushed to the back-end for recovery. After the operation log is stored in the back-end, meaning data structure is recoverable, the flush of corresponding memory logs can be delayed in batch. The cached data (if exist) are modified accordingly (④). If a bunch of operations get executed successfully or the buffer is full, the buffered memory logs, together with appended `TX_COMMIT`, will also be flushed to the back-end NVM via `remote_tx_write` (⑤). These logs are then handled by the back-end (⑥) and replicated to the mirror-node (⑦). If the back-end fails, the front-end takes care of exceptions, abort the transaction and clear the cache.

This *Gather-Apply* model can simplify the data work flow by merging data access operations.

7.2 Data Cache in Front-end

Several recent works build NVM systems using DRAM as cache [42, 9, 30]. Bw-tree [43] uses a cache layer to map logical pages (Bw-tree nodes) to physical pages. We use a similar data structure of hash map to translate address of data structure nodes in NVM to address in DRAM. Each item in the hash map represents the page cached. The page size is adjustable according to different data structures.

Our cache replacement policy combines the methods of LRU (Least Recently Used) and RR (Random Replacement). LRU works well in choosing hot data. However, its implementation is expensive. RR is easy to implement but does not provide any guarantee of preserving hot data. We use a hybrid approach, which firstly chooses a random set of pages for

Table 2: Comparison of Different Allocators. We set the slab size as 128 Bytes and 1024 Bytes separately.

Type/Tput(MOPS)	Alloc	Free
Glibc	21.0	57.0
Pmem	1.42	1.38
RPC allocator	0.33	0.88
Two-tier allocator (128 Bytes)	1.33	2.41
Two-tier allocator (1024 Bytes)	6.42	13.90

replacement (RR) and then a least used page from the set will be discarded (LRU). No page flush is needed because the write work flow already put the memory logs in the back-end. In the micro-benchmark, the hybrid approach (29.2%) can reduce the miss ratio by 33.5% compared to RR (62.7%) when the size of choosing set is 32, and gain a similar miss ratio as LRU with nearly 27.5% throughput improvement.

7.3 Batching Memory Logs

Since data structure operation logs will be flushed to remote NVM in back-end with persistency and recoverable guarantee, the flush of memory logs can be asynchronous and batched.

For example, when a new item is inserted into a binary search tree, the operation log with format {insert_op, key, value} as shown in Figure 2 will be put in the NVM log area. Then, the memory logs with the format {address, data} will be generated afterward. They do not need to be flushed immediately. The flush can be triggered by the conditions such as log buffer full or having finished a certain number of operations. At that time, a group of memory logs can be flushed in a batch. Batching can efficiently reduce the network rounds for invoking `remote_tx_write`. More data structure specific optimization using batching are discussed in Section 8.

7.4 Front-end Allocator

The design of the front-end allocator is inspired by the slab allocator [33]. Slabs are provided by the back-end allocator. The slabs in the front-end are organized in three lists of full/partial/empty list according to how much capacity is consumed in the corresponding page. We use a simple *best-fit* mechanism in the front-end. To improve the NVM utilization, a threshold is defined as the maximum free blocks number, and the front-end will reclaim free blocks periodically. When allocation size is larger than the size of a slab, front-end will switch to a mode to directly allocate memory from the back-end.

Benchmark. We measure our *rNVM* two-tier allocator with persistent allocator and standard Linux *Glibc* allocator. As table 2 shows, *Glibc* performs the highest throughput (21.0 MOPS and 57.0 MOPS) but without persistent guarantee. *Pmem* allocator is a persistent allocator from NVML project [44], and can reach 1.42 MOPS. Our original allocator's throughput is only 23% and 64% of Pmem allocator because of the network overhead. After applying the improvements, our persistent allocator can achieve similar or even better performance than Pmem allocator.

7.5 Data Structure Recovery

With the log mechanism, *rNVM* can keep crash consistency because all the data and logs are stored in the back-end. Due to the different component failure, recovery will take the different approach. In the case of front-end failure, another front-end will be selected as the sole writer and will recover the data structure according to the data and logs stored in the back-end. The back-end is critical for system to run correctly because all the data and logs are persistent. If the back-end faces transient failure, the back-end will reboot. After each reboot, the back-end will first reconstruct the mapping between the physical addresses and virtual addresses. The reconstruction is possible because such mapping is also stored in NVM. After that, the back-end can start its normal work immediately, i.e., reproducing memory logs to data structures if any log has not been applied. For the permanent failure of the back-end, the front-end will reconstruct the data structures to a new back-end by using the data and logs in the mirror nodes.

8 Data Structure Specific Optimization

The work flow and optimizations discussed so far are sufficient for most kinds of data structures to get the performance improvement. In this section, we present the data structure specific optimizations to further improve the performance.

8.1 Stack/Queue

Stack and Queue can be implemented by using the List data structure. Because the only data items that can be accessed in Stack or Queue are headers or tails, caching and batching can be specialized based on this property. Since the head and tail are more frequently accessed, the front-end typically only need to cache nodes pointed by them to reduce `remote_nvm_read`. For batching, the operations can be compacted because the operations are only allowed on stack header for Stack, and on queue tail for Queue. Thus, the effective pushes will be annulled by pops for Stacks, meanwhile the effective en-queues will be annulled by de-queues for Queue. The compaction will reduce the logs to only contain effective memory logs.

8.2 Tree-Like Data Structure

Tree-like data structures have the hierarchical organization. The nodes in higher (near to the root) level are more frequently accessed than lower level nodes. Based on this nice property, we can cache higher level data nodes. Specifically, the front-end sets a threshold N and the nodes with level larger than N will not be cached. They will be directly accessed through `RDMA_Read`. N is dynamically adjusted according to the cache miss ratio α, i.e., if $\alpha > 50\%$, $N = N - 1$ while if $\alpha < 25\%$, $N = N + 1$. Otherwise, N keeps unchanged.

Because trees are sorted data structures, the performance can be improved if the operations are sorted. We pack the sorted operations in *vector operation*. The operation goes from the root of the tree down to the leaf nodes. The vector can then be split accordingly. The operations in vector segments can be executed in parallel.

Algorithm 2 Vector Insert

```
procedure VECTOR INSERT($keys, values$)
    $node \leftarrow root$
    $queue.push(< 0, len, node >)$
    while queue is not empty do
        $begin, end, node \leftarrow queue.pop()$
        $mid \leftarrow$ binary_search($begin, end, node.key$)
        if $node.left = null$ then
            create_sub_tree($kvs[begin : mid]$)
                                                        ▷ construct a new sub tree
            $node.left \leftarrow sub\_tree$
        else
            $queue.push(< begin, mid, node >)$
        end if
        if $node.right = null$ then
            create_sub_tree($kvs[mid : end]$)
            $node.right \leftarrow sub\_tree$
        else
            $queue.push(< mid, end, node >)$
        end if
    end while
end procedure
```

The `vector_insert` in Algorithm 2 shows `vector_write`, one *vector operation*, in binary search tree following the Gather-Apply paradigm. It firstly reads the information to decide where to insert these nodes and then applies these insert in the correct position. Without batching, two read rounds are needed if insert operation A and B read the same node. When we execute A and B with one `vector_write` operation, it only needs one round read to access this node. Similarly, if several operations modify the same NVM memory, they will be compacted to one NVM write in `vector_insert`.

The caching and batching optimization described for tree-like data structure can also be applied to skip-list. Specifically, higher degree nodes in skip-list will be cached. Vector operation (containing sorted operations) for skip-list can similarly reduce the number of `RDMA_Read` calls.

9 Concurrency Control

So far, we only concern the one front-end case that can do read and write operations. However, *rNVM* can support SWMR (Single Writer Multiple Reader) access pattern by concurrent control mechanisms. Based on different applications, we

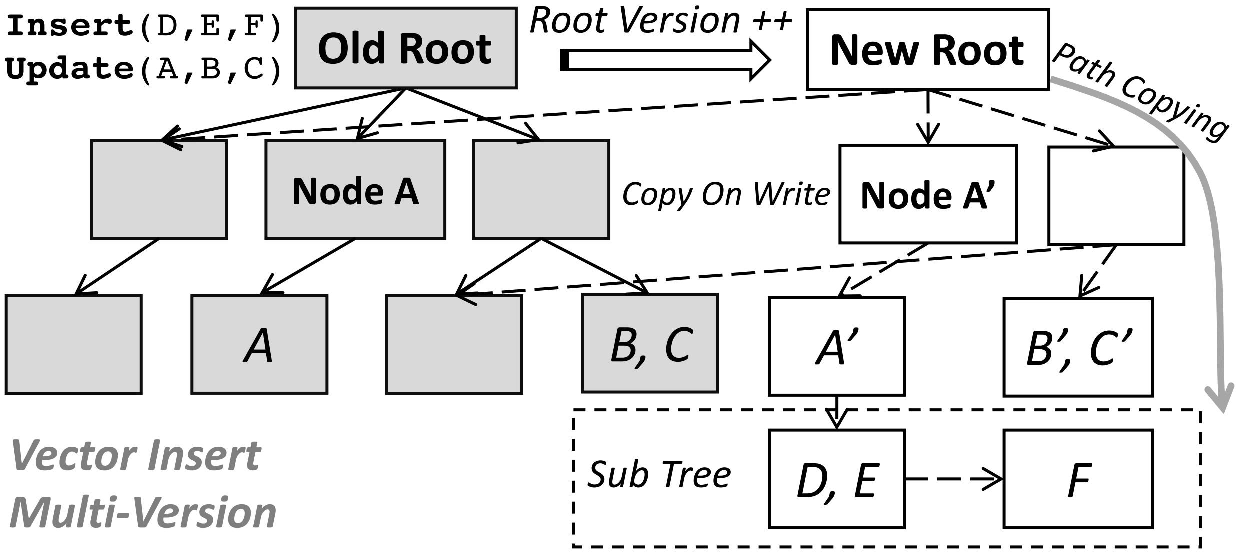


Figure 5: Overall Multi-version Data Structure

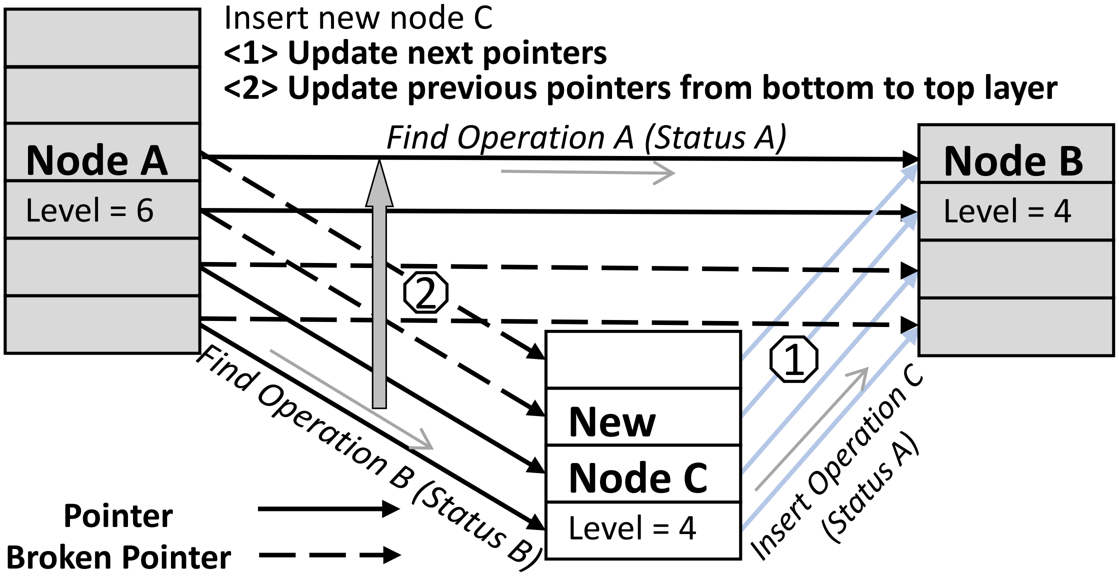

Insert new node C
<1> Update next pointers
<2> Update previous pointers from bottom to top layer
Find Operation A (Status A)
Node A
Level = 6
Node B
Level = 4
2
1
New
Node C
Level = 4
Find Operation B (Status B)
Insert Operation C (Status A)
Pointer
Broken Pointer

Figure 6: Naturally Lock-Free in Skip-list.

can support lock free and lock based data structures. In most cases, with serializing write operation to the single writer, SWMR mode is enough for providing concurrent access.

9.1 Lock-Free Data Structure

Multi-version Data Structure. We design lock-free tree-like data structure inspired by several previous works about append-only B-tree [45, 46] and persistent data structures [47, 48].

Multi-version is a widely used method in optimistic concurrent control [49, 50]. Multi-version data structures will first make copies of the corresponding data if needed. Then the data will be modified or new data items are inserted. For example in Figure 5, the writer copies all the affected nodes along the path to the root a.k.a., path copying. Then, the data will be inserted into the new path. After finishing all the jobs, the root will be changed to the new root atomically by updating the root pointer. Vector operation can help here to reduce the number of network round trips. Since the readers can always get the consistent data, this kind of concurrent control does not influence the performance of readers.

Algorithm 3 Writer Preferred Lock

```
procedure WRITER LOCK
    $gcc\_atomic\_increment$(SN)
end procedure
procedure WRITER UNLOCK
    $gcc\_atomic\_increment$(SN)
end procedure
procedure READER LOCK
    do
        $ret \leftarrow rdma\_atomic\_read$(SN)
    while $ret\ is\ odd$
    $start\_sn \leftarrow ret$
end procedure
procedure READER UNLOCK
    return $start\_sn \neq rdma\_atomic\_read$(SN)
end procedure
```

Table 3: Performance Improvement and Comparison (in KOPS) (R: using log reproducing, C: caching 10% NVM size in the front-end, B: batching with batch size 1024. The evaluation uses one-to-one setting with 100% write workloads harnessing all optimization. Reasons for empty cells: Data structure with time complexity $O(1)$ (i.e., HashTable/SmallBank) cannot apply batching optimization. In Queue/Stack implementation, batch and cache should be combined together.

	TX(SmallBank)	TX(TATP)	Queue	Stack	HashTable	SkipList	BST	BPT	MV-BST	MV-BPT
Symmetric	654	237	1199	1087	1097	125.2	84.5	305.2	42.2	18.6
Symmetric-B	-	260	2279	2255	-	209.0	151.0	343.0	146.1	76.0
rNVM-Naïve	254	10.2	301	285	315	5.0	19.0	11.5	7.0	7.4
rNVM-R	295	12.4	833	828	385	7.7	22.9	13.7	12.3	9.8
rNVM-RC	362	63.7	-	-	445	40.4	59.5	77.1	28.4	17.8
rNVM-RCB	-	127.5	1678	1449	-	66.0	134.2	184.3	88.9	60.2

Skip-List: Naturally Lock-Free. Some data structures like skip-list are naturally lock-free and the only work to be done is to carefully choose the order of operations [51, 52]. As shown in Figure 6, the writer first creates the newly allocated node and sets the pointers in the new node accordingly. After that, the previous pointers will be updated from the bottom to the top. Readers can still get the consistent views of Skip-list in such scenario through different views for different readers.

9.2 Lock Based Data Structure

Write Preferred Lock. RDMA library provides atomic verbs which is an appropriate way to implement distributed sequencer [53].

Algorithm 3 shows how to implement locks by using sequence number (SN), an 8 bytes integer variable. The back-end manipulates write lock and the front-ends manipulate read lock. The writer will not be locked out since the lock function only increases the sequence number atomically. However, the readers need to judge whether the lock is held by the writer (SN is odd) or the fetched data are consistent (whether SN is the same before and after the read). If the data are not consistent, the readers need to retry and fetch the data again. Notice that every lock acquire/release should also write operation logs to remote NVM to handle in the presence of failures.

Discussion. Lock-free data structures can benefit the reader but create multiple copies by writers. Lock-based data structures can benefit the writer without extra copies but readers have to read multiple times until consistent data are obtained. They They do not contradict each other and the right choice depends on specific applications. Moreover, MWMR (Multiple Writer Multiple Reader) can be implemented based on the currently available mechanism and data structure. A straightforward method is using sequencer to make write operations in order. Furthermore, we can use shared logs [54, 55] or flat-combining [56] technique to guarantee consistent concurrent writes in our future work. In addition, data partition can also be used to easily support multiple writers (SWMR for each partition) without any change of system.

10 Evaluation

Our evaluations will answer the following questions:

I. How *rNVM* performs? In addition, how it is compared to symmetric setting and naive asymmetric implementation?

II. How the batching and caching can benefit *rNVM*?

III. How *rNVM* performs under multiple front-ends?

IV. How *rNVM* performs under different workloads?

10.1 Evaluation Setup

Hardware Setup. The experiment cluster contains eight machines, each of which is equipped with 8-core CPU (Intel Xeon E5-2640 v2, 2.0 GHz), 96 GB memory, and one Mellanox ConnectX-3 InfiniBand with network bandwidth 40Gbps. One machine is used to simulate FPGA/ASIC based back-end.

NVM Emulator. We use 60 GB DRAM as remote NVM, and 6GB DRAM as the front-end DRAM for caching data. Similar to prior works [57, 26, 9], we set the write latency as $200ns$ and read latency as the latency of DRAM. This is due to the read/write asymmetry in NVM.

10.2 *rNVM* Performance and Comparison

We implement eight data structures that are widely used covering different access time complexity ($O(1)$, $O(log(n))$): stack, queue, hash-table, skip-list, binary search tree (BST), B+tree (BPT), multi-version binary search tree (MV-BST), and multi-version b+tree (MV-BPT). We also implement two end-to-end transaction applications: SmallBank and TATP [58].

Replication Performance. We measure the overhead of log replication with ten applications in SWSR and SWMR (7 readers and 1 writer). There is almost **no** performance degradation even with 100% write workload in such architecture. If log replication is performed by using the front-end instead of back-end, this will lead to 20∼40% performance degradation. It is reasonable because the back-end has very low CPU usage ratio as shown in Section 10.3.

Table 3 shows the overall performance as well as the comparison to symmetric and naive implementations.

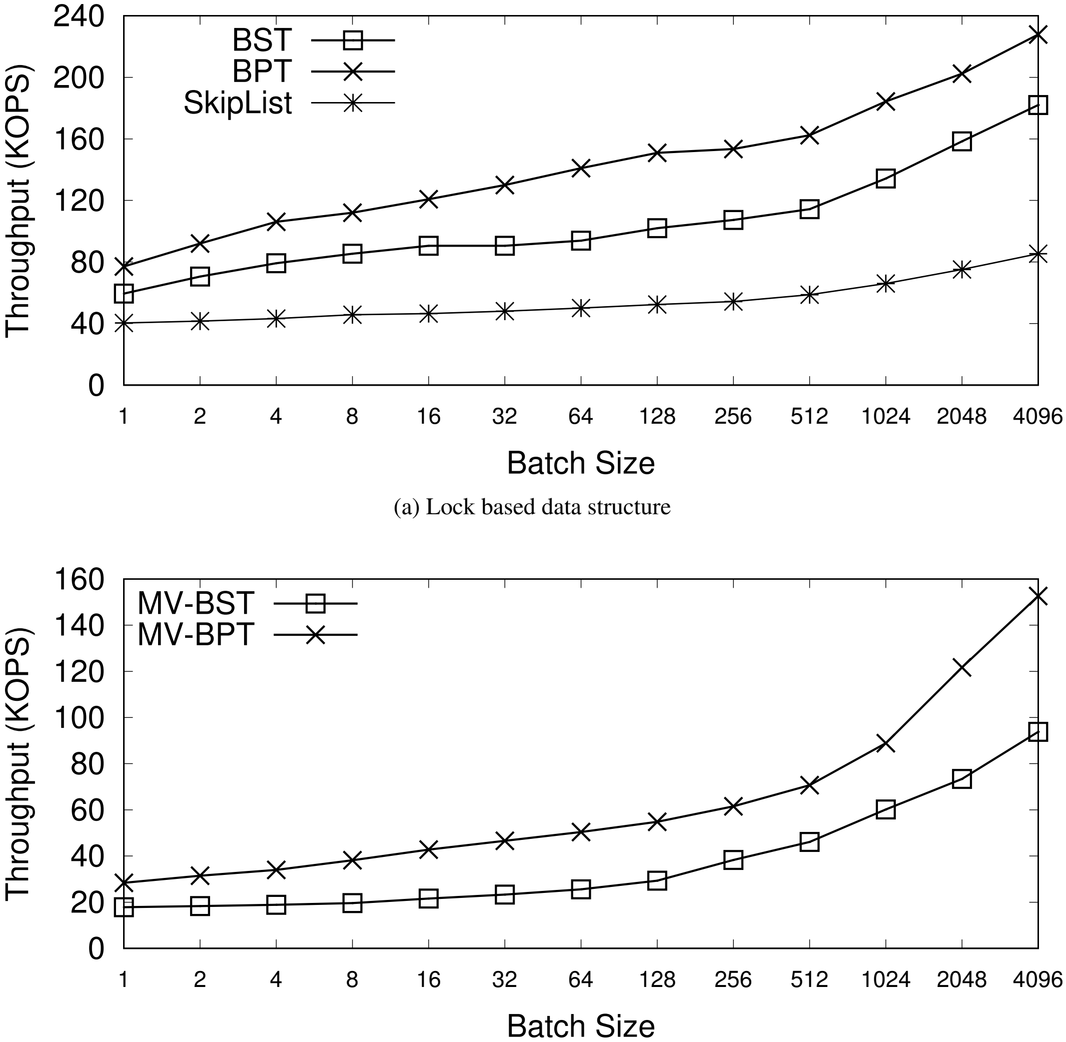
Compare to naïve implementation. Naïve implementation just puts the data structure in remote NVM and access the data directly without log reproducing, caching or batching. The complete implementation denoted as *rNVM*-RCB (means with log **R**eproducing, **C**atching, and **B**atching) can reach nearly 6∼22 × improvement than naive implementation. All the optimization methods can reinforce each other. Obviously, cache shows more effectiveness than other optimization (nearly 2∼7 × performance improvement). The reason is that the `RDMA_Read` operation always has the major portion in an data structure operation, and applying cache will eliminate this read overhead.

Compare to the symmetric setting. We implemented the symmetric persistent NVM data structure by storing data structures in local NVM and storing logs in remote NVM. The logs are flushed asynchronously (without waiting the responses from remote nodes) so as to reach the up-bound performance. This will bring the problem of inconsistency. However, thanks to the small DRAM cache in the front-end, *rNVM*-RCB still has the similar performance even with such implementation of symmetric NVM data structures, and its performance can even be better than the *no-batch* implementation for data structures in symmetric NVM deployment. As mentioned previously, *rNVM* has better storage efficiency.

Cost Comparison. We also make comparison about the usage of NVM. In symmetric setting with m machines, it needs $n_1 = max(\sum_{i=1}^{i=m} \lceil S_i/S_0 \rceil, m)$ NVM devices (assuming each NVM capacity is S_0). Besides, *rNVM* needs $n_2 = \left\lceil \sum_{i=1}^{i=m} S_i \right\rceil$ NVM devices and significantly $n_2 \leq n_1$. As we mentioned in Section 1, each server only needs smaller capacity less than S_0, so the necessary NVM n_2 will be fewer than n_1 ($n_1 = m$).

10.3 CPU Utilization

Figure 11 gives CPU utilization of front-end and back-end. The front-end keep running with nearly 100% CPU utilization but the request only incur very small CPU usage (4%∼10% CPU utilization). As we proposed in Section 1, the back-end has very little computing overhead and thus it is reasonable to use AISC/FPGA in the back-end.



(a) Lock based data structure

(b) Multi-version data structure

Figure 7: Throughput with Different Batch Sizes

10.4 Effects of Batch and Cache

We measure the effects of batch and cache using different parameters.

Batch. We measure batch by using vector operations under different batch sizes from 1 to 4048 and the results are in Figure 7. MV-BST can be improved by 3.38 × i.e. from 17.8 KOPS without batch to 60.2 KOPS with batch size 1024. The improvement for MV-BPT is about 3.13 × (from 28.4 KOPS to 88.9 KOPS). The improvements are 126%, 139% and 63% for BST, BPT and SkipList respectively. Multi-version data structures need to do path copying which incurs a lot of write operations. The batch can effectively reduce such overhead.

Cache. Cache effects can be measured under different front-end cache sizes. Binary search tree, B+Tree and skip-list are used here and the results are in Figure 8. Overall, the throughput increases with the increase of cache sizes. Notice that MV-BPT and MV-BST do not get too much improvement with cache. This is due to the fact that the data modified are still kept in the memory for multi-version data structures.

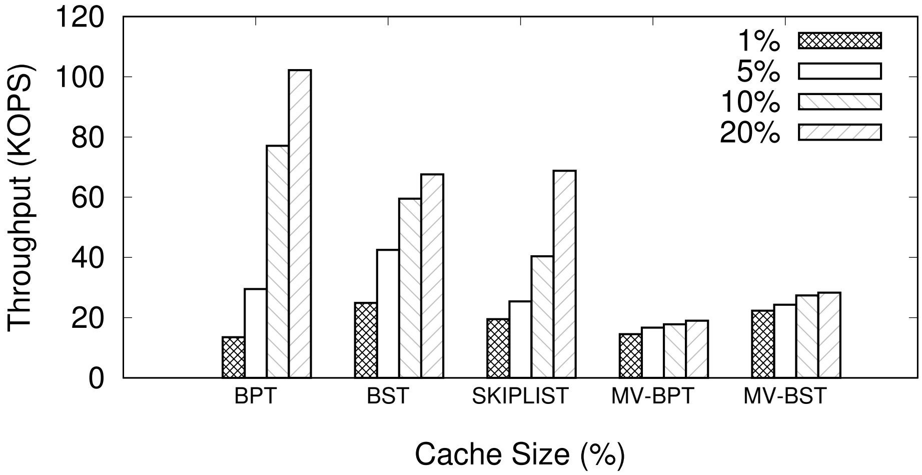


Figure 8: Throughput with Different Cache Sizes

10.5 Multiple Front-ends

Previous tests are performed using one front-end and one back-end. However, since *rNVM* can support the operation in SWMR mode, we also test the performance of scalability by using multiple readers. The results are shown in Figure 9.

The readers' performance can scale well with the increasing number of readers. The writer's performance is slight influenced. The writer performance of lock based data structure decreases more than that of multi-version data structures. This is because there are more RDMA rounds for lock based data structures that can influence the performance. With different mechanisms of concurrent control, the effects are different. With lock based BST, the average throughput with 6 readers performs 26% lower than the value with only one reader. In the case of MV-BST, the performance degradation is about 8% lower. The multi-version data structures do benefit the readers.

Also from the results, the lock-free data structures scale better than their lock-based counterparts. The readers in Figure 9b have about 3.0∼3.2 × higher performance than the readers in Figure 9a. Retries incurred by failed read take the main responsibility for the lower performance. The portion of retry is about 4%∼16% of total operations under the situation of 6 readers and 100% insert from the writer. Lower write workload will decrease the ratio of retries.

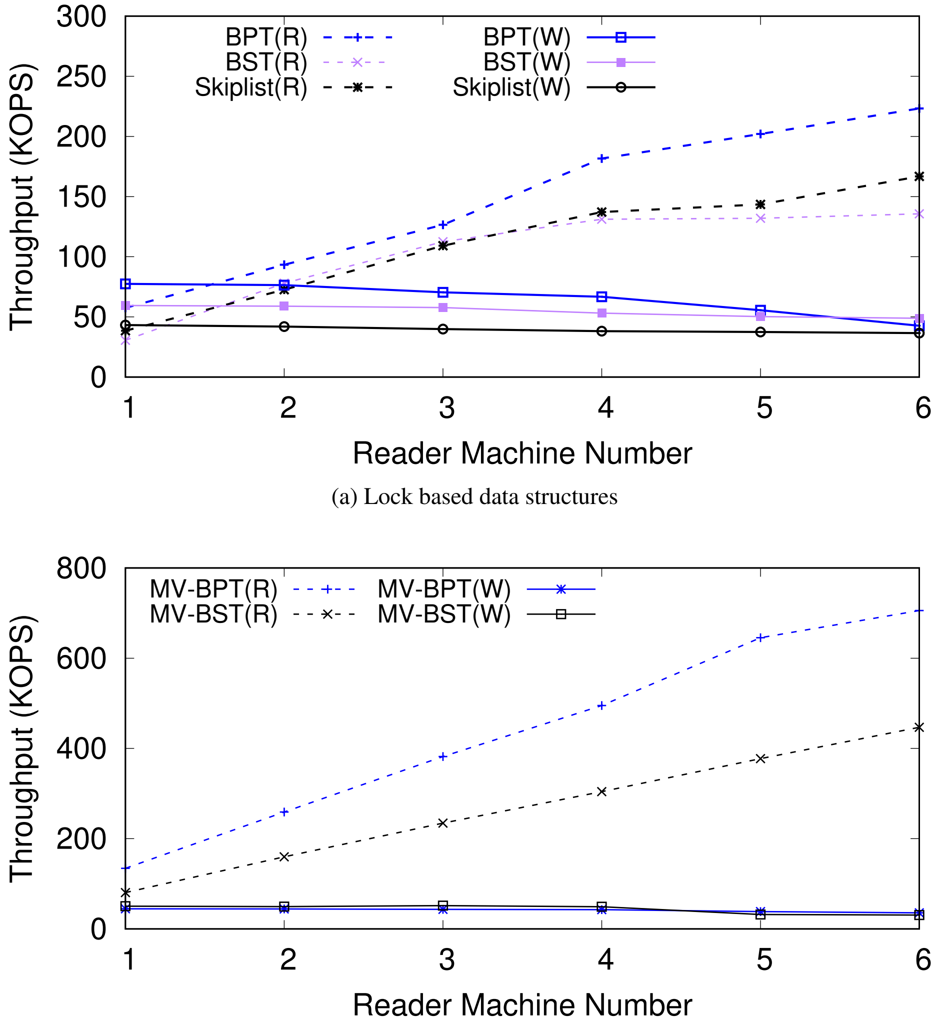
In the asymmetric deployment, one NVM device can be shared by multiple front-ends to increase the utilization. *rNVM* can also support such usage scenario of NVM device. We measured the throughput of multiple writable front-end sharing one NVM device. Each front-end has its own distinct data structure. Since there is a huge amount of possible combination, we only test the case that each front-end use the same type of data structure but with different instances. In Figure 10, the scalability is almost linear with 7 front-ends. The average performance degradation for a single client is about 7% ∼ 20% compared to the one-to-one deployment.

10.6 Different Workloads

Figure 12 shows the throughput of using different read/write ratios from the single writer front-end. For simplicity, insert operation is used as write and find operation is used as read. Obviously, with fewer read operations, the performance will be decreased due to more overhead brought by write operations. Compare BPT/BST to their MV-counterparts (MV-BPT/MV-BST), BPT/BST have relatively higher performance. For example, with the full write workload, there are about 23% and 48% performance gap. This is because, in the MV-version, the write operations need to write more data during path copying.

11 Related Work

There are two RDMA usage methods: straightforward server-reply paradigm [34, 35] and server-bypass [36, 37, 38] paradigm. We adopt server-bypass paradigm to relief the network stress in the back-end node and further keep the back-end from network operations.



(a) Lock based data structures

(b) Lock-free data structures

Figure 9: Scalability of Multiple Readers. The workload of writer is 100% insert. R/W represents reader/writer.

Single-node NVM systems [42, 29, 59, 30, 60, 6, 9, 7] provide direct access to NVM via memory bus but cause lower utilization of NVM and inaccessible facing node failures.

Distributed NVM systems including Octopus [25], Hotpot [31], Mojim [26], and FaRM [37, 61] combine the NVM devices together with RDMA, and they are all using symmetric deployment. Currently, the asymmetric deployments provide storage interfaces including NVMe over Fabric [28] and Crail [24]. However, they are not byte addressable i.e., they can not provide data structure level service.

A few file systems (e.g., Aerie [62]) adopt a hybrid paradigm similar as rNVM which allowing remote read access and transactional write access with logs.

There are some researches about how to implement a persistent allocation system over NVM including nvm_malloc [63], Makalu [57], PAllocator [64], Mneosyne [8]. They discuss different considerations of NVM allocators in a single machine. We make the first step towards distributed NVM allocator.

12 Conclusion

This paper introduces an asymmetric NVM architecture *rNVM*, which is different from the current usage of NVM as attachable devices. The framework separates the work into front-end and back-end. The front-end can get the memory

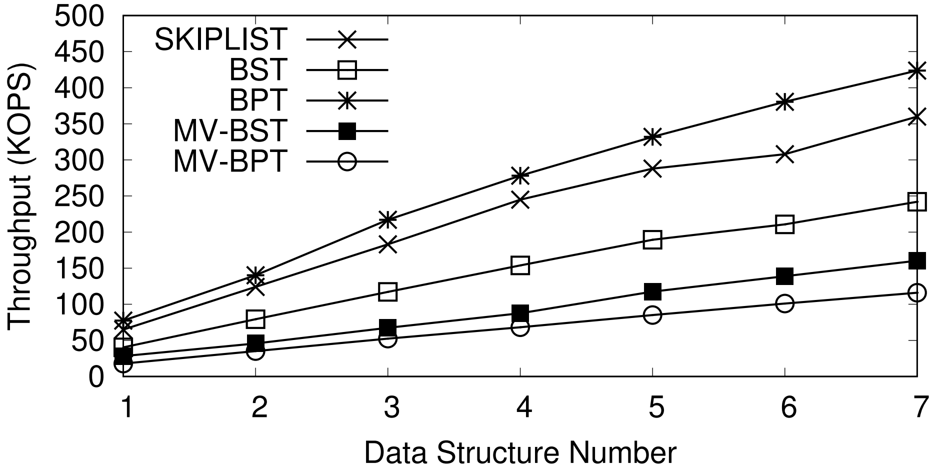


Figure 10: Throughput of Multiple Data Structures in the Same Back-end Machine

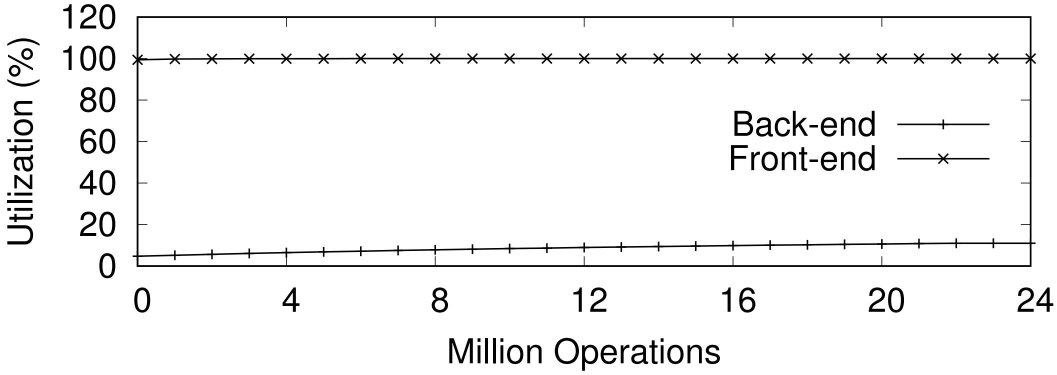


Figure 11: CPU Utilization with the operation increasing in BST. The workload is 10% put and 90% get.

blocks from the back-end and they together provide the persistent transaction data structures by using basic RDMA verbs and three simple APIs. The preliminary system *rNVM* was built to demonstrate the feasibility and benefits of such architecture for supporting byte addressable persistent data structures. *rNVM* applies several general optimizations including batching and caching and special optimizations according to data structure types to improve the performance. With the trend of resource disaggregation in data centers, *rNVM* is the right way of providing highly available persistent data structures.

References

[1] Hewlett Packard. Understanding the intel/micron 3d xpoint memory. `http://www.hpl.hp.com/research/systems-research/themachine/`, 2015.

[2] Geoffrey W Burr, Matthew J Breitwisch, Michele M Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan L Jackson, B N Kurdi, Chung H Lam, Luis A Lastras, Alvaro Padilla, et al. Phase change memory technology. *Journal of Vacuum Science and Technology B*, 28(2):223–262, 2010.

[3] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *international symposium on computer architecture*, 37(3):2–13, 2009.

[4] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. *international symposium on computer architecture*, 37(3):14–23, 2009.

[5] Dmytro Apalkov, Alexey Vasilyevitch Khvalkovskiy, Steven M Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian E Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems*, 9(2):13, 2013.

[6] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[8] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[9] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–343. ACM, 2017.

[10] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[11] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[12] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758. ACM, 2017.

[13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

[14] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[15] Google. Clusterdata2011 2 traces. `https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md`, 2018.

[16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, volume 16, pages 249–264, 2016.

[17] Intel. Intel rack scale design architecture white paper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf`, 2017.

[18] Hewlett Packard. The machine. `https://www.labs.hpe.com/the-machine`, 2018.

[19] Intel. Intel, facebook collaborate on future data center rack technologies. `https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/`, 2013.

[20] Intel. Intel rack scale design. `https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html`, 2018.

[21] Intel. Disaggregated servers drive data center efficiency and innovation. `https://www.intel.com/content/dam/www/public/us/en/documents/best-practices/disaggregated-server-architecture-drives-data-center-efficiency-paper.pdf`, 2018.

[22] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 535–548. ACM, 2018.

[23] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing*, page 8. ACM, 2016.

[24] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[26] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. *ACM SIGPLAN Notices*, 50(4):3–18, 2015.

[27] NVM Express. Nvme over fabrics overview. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf, 2018.

[28] Patrice Couvert. High speed io processor for nvme over fabric (nvmeof). *Flash Memory Summit*, 2016.

[29] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *FAST*, volume 15, pages 167–181, 2015.

[30] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362. USENIX, 2017.

[31] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.

[32] Shin-Yeh Tsai and Yiying Zhang. Mitsume: an object-based remote memory system. In *Workshop on Warehouse-scale Memory Systems (WAMS)*. ACM, 2018.

[33] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[34] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[35] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 743–752. IEEE, 2011.

[36] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[37] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[38] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. Rfp: When rpc is faster than server-bypass with rdma. In *EuroSys*, pages 1–15, 2017.

[39] Talpey Tom. Remote access to ultra-low-latency storage. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf, August 2015.

[40] Douglas Chet. Rdma with pmem: Software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf, 2015.

[41] OpenFabric. Rdma and nvm programming model. https://www.openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Monday/monday_12.pdf, 2015.

[42] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[43] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 302–313. IEEE, 2013.

[44] Intel. Nvm library. https://github.com/pmem/nvml, 2018.

[45] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. " O'Reilly Media, Inc.", 2010.

[46] Martin Hedenfalk. how the append-only btree works. http://www.bzero.se/ldapd/btree.html, 2009.

[47] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[48] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[49] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):264–275, 1996.

[50] Benjamin Sowell, Wojciech Golab, and Mehul A Shah. Minuet: A scalable distributed multiversion b-tree. *Proceedings of the VLDB Endowment*, 5(9):884–895, 2012.

[51] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

[52] Tyler Crain, Vincent Gramoli, and Michel Raynal. No hot spot non-blocking skip list. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 196–205. IEEE, 2013.

[53] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[54] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *NSDI*, pages 35–49, 2017.

[55] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[56] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[57] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 677–694. ACM, 2016.

[58] Neuvonen Simo, Wolski Antoni, manner Markku, and Raatikka Vilho. Tatp benchmark. `http://tatpbenchmark.sourceforge.net/`, 2011.

[59] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

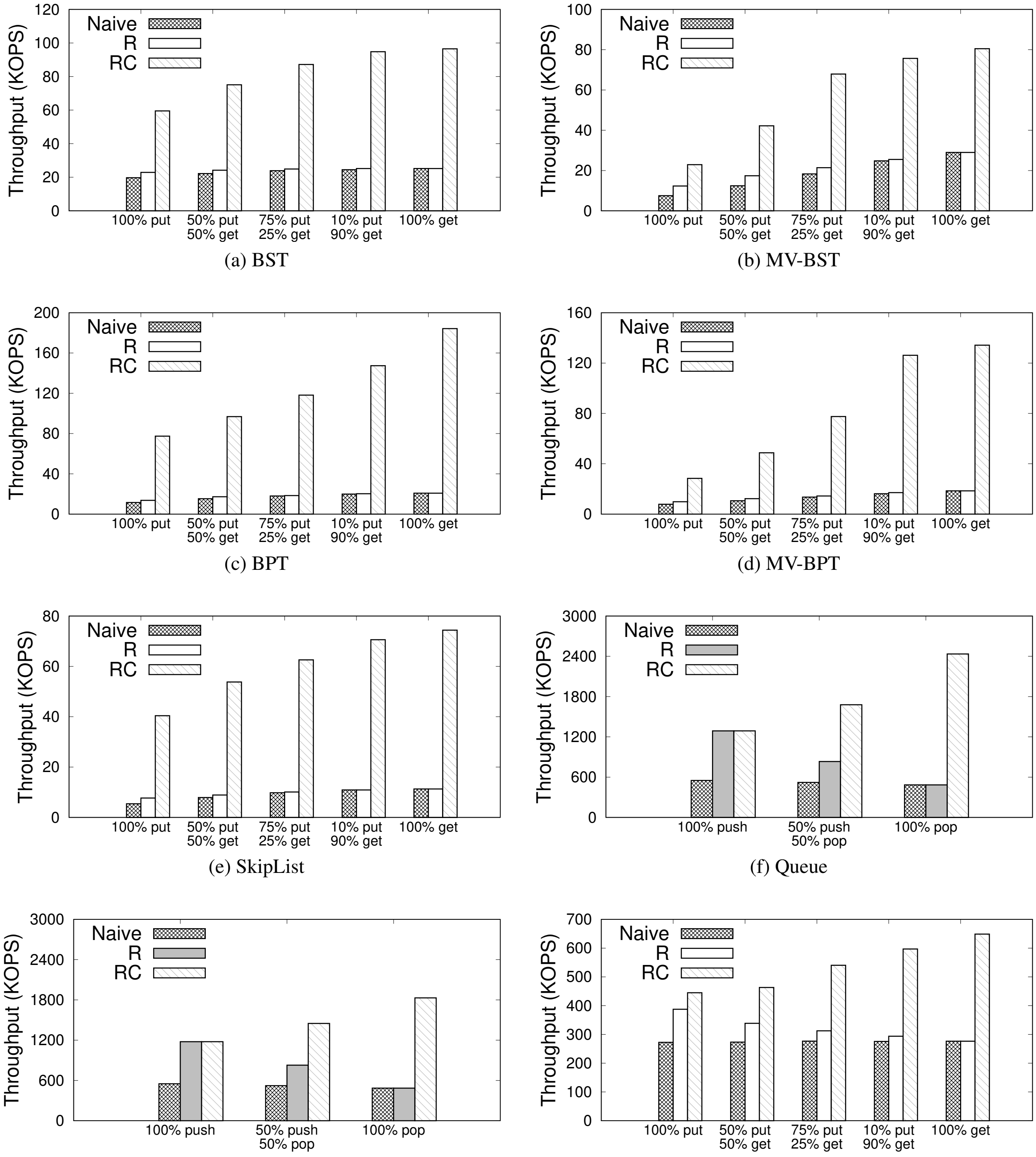
[60] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[61] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70. ACM, 2015.

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

[63] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@ VLDB*, pages 61–72, 2015.

[64] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.



(a) BST

(b) MV-BST

(c) BPT

(d) MV-BPT

(e) SkipList

(f) Queue

(g) Stack

(h) HashTable

Figure 12: Throughput with Different Workloads (100%put, 50%put+50%get, 25%put+75%get, 10%put+90%get, 100%get)