AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture

Teng Ma
mt16@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Mingxing Zhang
zhang.mingxing@outlook.com
Tsinghua University & Sangfor
Shenzhen, China

Kang Chen*
chenkang@tsinghua.edu.cn
Tsinghua University
Beijing, China

Zhuo Song
songzhuo.sz@alibaba-inc.com
Alibaba
Beijing, China

Yongwei Wu
wuyw@tsinghua.edu.cn
Tsinghua University
Beijing, China

Xuehai Qian
xuehai.qian@usc.edu
University of Southern California
Los Angles, CA

Abstract

The byte-addressable non-volatile memory (NVM) is a promising technology since it simultaneously provides DRAM-like performance, disk-like capacity, and persistency. The current NVM deployment with byte-addressability is *symmetric*, where NVM devices are directly attached to servers. Due to the higher density, NVM provides much larger capacity and should be shared among servers. Unfortunately, in the symmetric setting, the availability of NVM devices is affected by the specific machine it is attached to. High availability can be achieved by replicating data to NVM on a remote machine. However, it requires full replication of data structure in local memory — limiting the size of the working set.

This paper rethinks NVM deployment and makes a case for the *asymmetric* byte-addressable non-volatile memory architecture, which decouples servers from persistent data storage. In the proposed *AsymNVM architecture*, NVM devices (i.e., back-end nodes) can be shared by multiple servers (i.e., front-end nodes) and provide recoverable persistent data structures. The asymmetric architecture, which follows the industry trend of *resource disaggregation*, is made possible due to the high-performance network (e.g., RDMA). At the same time, AsymNVM leads to a number of key problems such as, still relatively long network latency, persistency bottleneck, and simple interface of the back-end NVM nodes.

We build *AsymNVM framework* based on AsymNVM architecture that implements: 1) high performance persistent data structure update; 2) NVM data management; 3) concurrency control; and 4) crash-consistency and replication. The key idea to remove persistency bottleneck is the use of *operation log* that reduces stall time due to RDMA writes and enables efficient batching and caching in front-end nodes. To evaluate performance, we construct eight widely used data structures and two transaction applications based on AsymNVM framework. In a 10-node cluster equipped with real NVM devices, results show that AsymNVM achieves similar or better performance compared to the best possible symmetric architecture while enjoying the benefits of disaggregation. We found the speedup brought by the proposed optimizations is drastic, — 5∼12× among all benchmarks.

CCS Concepts • **Computer systems organization** → **Processors and memory architectures**; • **Information systems** → *Data centers*; • **Hardware** → *Non-volatile memory*.

Keywords memory architectures; persistent memory; RDMA

ACM Reference Format:
Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378511

1 Introduction

Emerging non-volatile memory (NVM) is blurring the line between memory and storage. These kinds of memories, such as Intel Optane DC persistent memory [43], phase change memory (PCM) [9, 54, 103], spin-transfer torque magnetic memory (STTM) [4], and memristor are byte-addressable,

*Corresponding Author.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378511

and provide DRAM-like performance (300ns/100ns latency for read/write), high density (TB-scale), and persistency at the same time [45, 99]. To unleash the potential of NVM, most existing solutions attach NVM directly to processors [14, 28, 62, 92, 94], enabling high-performance implementations of persistent data structures using load and store instructions on local memory.

While accessing NVM via local memory provides promising performance, it is not the most suitable setting in the context of data center due to the *desire of sharing NVM*. Due to the higher density, NVM can provide much larger capacity [45, 64], which may exceed the need of a single machine [5]. In data center servers, the resources are often under utilized [19, 23], — Google's study [29] shows the resource utilization lower than 40% on average. We expect that persistent resource utilization will follow the same trend.

To enable NVM sharing, recent work [81] builds a distributed shared persistent memory system, which provides a global, shared, and persistent memory space for a pool of machines with NVMs attached to each at the main memory bus. This setting inherently affect availability: once an NVM device is attached to a specific machine, its data become unavailable when the host machine goes down. One solution to this problem is to replicate the data to a remote NVM [44]. However, it requires full replication of data structures in local memory, limiting the size of the working set. To access the replicated data, the lower-bound of network overhead is at least one network round-trip for each operation.

In essence, these challenges are due to the *symmetric* nature of most of the current NVM deployment [44, 81, 83]. To fundamentally overcome the drawbacks, we rethink NVM deployment and propose the *byte-addressable asymmetric* NVM architecture, in which NVM devices are not associated with the individual machine and accessed at byte-level only *passively via fast network*. In *AsymNVM* architecture, the number of NVM devices, which can be provided as specialized "blades", can be much smaller than the number of machines.

AsymNVM architecture follows the recent trend of disaggregation architecture, which was first proposed for capacity expansion and memory resource sharing by Lim *et al.* [59, 60]. As described by Gao *et al.* [26], disaggregated architecture is a paradigm shift from servers each tightly integrated with a small amount of various resources (e.g., CPU, memory [30, 67, 68], storage [65]) to the disaggregated data center built as a pool of standalone resource blades and interconnected using a network fabric. In industry, Intel's RSD [39] and HP's "The Machine" [74] are state-of-the-art disaggregation architecture products. Such systems are 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, disaggregation is considered to be the future of data centers [34, 37, 38, 40, 74]. In AsymNVM architecture, NVM devices are instances of disaggregated resources that are not associated with any server. It also improves the availability since the crash of a server will not affect NVM devices.

The main principle of AsymNVM architecture is to use high-performance network (RDMA) to access remote NVM and local DRAM as cache to ensure high performance. The concrete design poses several key challenges. First, directly replacing local store and load instructions with `RDMA_Write` and `RDMA_Read` still suffers from long network latency. Although the throughput of RDMA over InfiniBand is comparable to the throughput of NVM, NIC still cannot provide enough IOPS for fine-grained data structure accesses. Second, using local DRAM as cache should be carefully considered with the persistency semantics. Third, we need to ensure that the interface of back-end nodes is both simple and efficient.

Based on AsymNVM architecture, this paper builds *AsymNVM framework* that implements *byte-level data structures updates* with high performance and availability. To efficiently solve the three challenges, the framework efficiently implements four components: 1) to remove persistency bottleneck, high performance byte-level persistent data structure update is supported by *operation log* that reduces the stall due to RDMA writes and enables efficient communication batching and DRAM caching (Section 4); 2) *NVM data management* that handles non-volatile memory allocation and free, and metadata storage (Section 5); 3) *concurrency control* that supports both lock-free and lock-based data structures (Section 6); and 4) *crash-consistency and replication* that ensures correct recovery and availability (Section 7). Moreover, we apply several data structure specific optimizations to further improve performance (Section 8).

To evaluate the performance of AsymNVM framework, we choose eight widely used data structures and two applications (TATP/SmallBank), and use traces of industry workloads. The data structures/applications are executed in a 10-node cluster, in which at most three machines are used as the back-end node and mirror nodes. The results show that AsymNVM achieves similar or better performance compared to the best possible symmetric architecture while enjoying benefits of disaggregation. Speedup brought by the proposed optimizations is drastic, — 5∼12× among all benchmarks.

2 Background

Single-Node Local NVM. In this setting, NVM device is directly accessed via the processor-memory bus using load-/store instructions. It avoids the overhead of legacy block-oriented file-systems/databases. Instead, it allows persistent data structure updates at byte level without the need for serialization. Based on the byte-level accesses, many kinds of persistent data structures are proposed [11, 56, 56, 73]. CDDS-Tree [92] uses multi-version to support atomic updates without logging. NV-Tree [100] is a consistent and cache-optimized B+Tree, which reduces cacheline flush operations. HiKV [96] constructs a hybrid index strategy to build a persistent key-value store.

Symmetric Distributed NVM. Symmetric architecture is widely used in distributed systems, in which each machine has its own NVM device. To achieve good availability on top of persistency, one needs to replicate its data structures to multiple NVM devices. Mojim [44] 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 the 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 primary node's logs 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 only perform the weakly-consistent replication of data in the primary node.

With a similar interface, Hotpot [81] 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. Thus, applications can access both local and remote data in the global memory space by performing native memory load and store instructions. At the same time, the system ensures data persistency and reliablity. To achieve this, when a committed page is written, Hotpot creates a local Copy-On-Write (COW) page and marks it as dirty.

The two systems are both designed for the 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 simple LRU-like mechanism to evict redundant or committed pages.

Network Attached Storage. Network Attached Storage (NAS) [27] is designed based on the principle of decoupling storage and computation. Compared to AsymNVM architecture, the key distinction is that NAS heavily relies on *file system service* in the storage node. It is based on the *block* access interface, which suffers from the fundamental problem of read/write amplification (e.g., LSMTree) when performing fine-grained access. Apart from the file system interface, (de)-serialization and other transformation overhead also significantly affect the performance of NAS.

Resource Disaggregation. The resource disaggregation architecture [2, 30, 59] can largely extend the capability of a single machine by considering the available resources in dedicated blades as a sharing pool, thereby providing fine-grained control over their resources. For instance, INFINISWAP [30] manages unused memory with *the power of many choices* to perform slab placements/evictions, and thus gain a high memory utilization. RackOut [67] mitigates load imbalance with rack-scale memory pooling. Disaggregation architecture allows decoupled implementation of system functionalities and naturally provides an efficient way to deploy hardware as well as bringing down monetary cost [80].

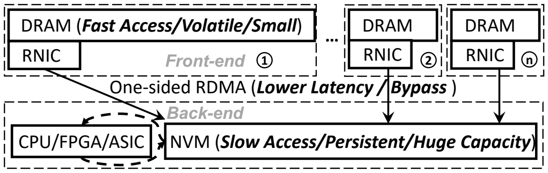

DRAM (Fast Access/Volatile/Small)
RNIC
Front-end ①
...
DRAM
RNIC ②
DRAM
RNIC ⓝ
One-sided RDMA (Lower Latency / Bypass)
Back-end
CPU/FPGA/ASIC
NVM (Slow Access/Persistent/Huge Capacity)

Figure 1. *AsymNVM* Architecture

3 AsymNVM Overview

3.1 AsymNVM Architecture

AsymNVM architecture is an asymmetric non-volatile memory architecture, 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 devices/blades are shared by multiple client machines, and the memory space of these client machines may be much smaller than the capacity of the NVM devices.

As shown in Figure 1, *back-end* nodes have NVM attached to their memory bus; and *front-end* nodes operate the data structures on NVM. Front-end can only access back-end via network. Specifically, the relationship between front-end and back-end is "*many-to-many*" — 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 the symmetric architecture, AsymNVM architecture offers four advantages: *a)* it enjoys the benefits of disaggregation; *b)* it naturally matches the desire of sharing NVM; *c)* it ensures availability with multiple back-end; and *d)* the back-end nodes can be implemented in simple manner that leads to better reliability [53].

Compared to NAS, AsymNVM architecture provides fine-grained and variable size byte-addressable access with higher IOPS/flexibility. Unlike previous NAS systems, AsymNVM is deployed in RDMA-based rack-scale data-centers, releasing the CPU resource in the back-end with one-sided communication. While recent works [63, 88] build file or object access interface on asymmetric architecture, the block access interface leads to the fundamental problem of read/write amplification when NAS is used to perform fine-grained data access.

3.2 Key Challenges

The first challenge is *network latency.* Although the bandwidth of InfiniBand is comparable to NVM, the latency is not. RDMA operation RTT is about 2 μs, much larger than the latency of NVM (about 100/300 *ns* for read/write) [64, 99]. Simply replacing local read/write operations with `RDMA_Read` and RDMA_Write operations will significantly degrade performance.

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 [102]) can always offer the best performance. However, it contradicts the original purpose of the asymmetric setting. The high density of NVM devices makes it capable to hold terabytes of data, but the memory space of a typical front-end node is only tens of gigabytes. This asymmetry implies that only the necessary data in the current work-set should be loaded into the front-end nodes during execution.

The third challenge is the design of back-end interface that is *both effective and simple with good reliability*. Specifically, back-end nodes should be only responsible of performing a small collection of simple APIs, such as remote memory read/write/allocation/release, lock acquire/release, etc. We also envision that the back-end nodes can be equipped with specialized hardware to support the operations on NVM. Indeed, a small set of *fixed* APIs makes it truly feasible.

3.3 AsymNVM Framework Overview

Based on AsymNVM architecture, we build the *AsymNVM framework*, a general framework for implementing high-performance data structures. The AsymNVM framework assumes that all persistent data are hosted in the remote back-end NVM devices, and can be much *larger* than the limited size of the local volatile memory in the front-end nodes. Moreover, the accesses of back-end nodes are always *passive*: they never initiates a communication with the front-end nodes, but only passively response to the API invocations from the front-end.

We assume the back-end nodes are equipped with advanced NIC that supports RDMA. The front-end nodes can directly access their data via one-sided operations (RDMA_Read/ RDMA_Write) as the basic APIs shown in Table 1 without notifying the processing unit on the remote side. This leads to better performance than two-sided operations [20] Although it is possible to implement any kind of data structures using only RDMA verbs, the performance will suffer due to long network latency. Moreover, back-end nodes need to expose methods to allow front-end nodes to manage NVM data. To this end, AsymNVM framework implements three sets of *simple and fixed* API functions in the back-end on top of RDMA verbs as shown in Table 1.

The first set of APIs provides a *transactional interface* that allows the front-end nodes to push a list of update logs to the back-end to achieve persistency in an all-or-nothing manner. The transactional interface is simple and has two variants. Specifically, a transaction can include: 1) a collection of *memory logs*, — {memory address, value} pairs; or 2) an *operation log*, which includes the operation and parameters applied to a certain data structure and is used to reduce the stall due to remote persistency. The back-end nodes ensure that all these addresses are updated atomically.

Table 1. AsymNVM APIs (rnvm: remote_nvm)

Type	API	Explanation
Basic (native)	*rnvm_read* *rnvm_write*	read data from local cache or remote NVM write data to remote NVM
Transaction	*rnvm_mem_log* *rnvm_op_log* *rnvm_tx_write*	write a memory log to front-end buffer write a operation log to remote NVM write a batch of memory logs to remote NVM
Management	*rnvm_malloc* *rnvm_free*	allocate NVM space in back-end free NVM space in back-end
Concurrency	*writer_(un)lock* *reader_(un)lock*	exclusive write lock concurrent read lock

The second set of APIs handles *memory management*, which includes remote memory allocation, releasing, and global naming. The AsymNVM framework implements a two-tier slab-based memory allocator [8]. The back-end runs in the remote NVM to ensure persistency and provide the fixed-size blocks. The front-end supports memory allocations at a finer granularity. To support recovery, several specific metadata are stored in the back-end *global naming space*.

The third set of APIs deals with *concurrency control* for Single Writer Multiple Reader (SWMR) access model. This means that if two front-end nodes perform writes on the same address, they should be synchronized by locks. In addition, the framework assumes that reads and writes to the same address are also properly synchronized by locks. They are implemented by leveraging existing RDMA atomic verbs and retry-based optimistic lock strategy.

In AsymNVM architecture, RDMA provides several atomic verbs to guarantee that any update to a *64-bit* data is atomic. Thus, we can apply RDMA atomic operations to the critical metadata, e.g., root pointer of data structure, in order to manage metadata consistently. Due to the non-volatile nature of the remote NVM, the data may be corrupted if the back-end crashes during a single RDMA_Write operation. AsymNVM guarantees the data integrity via checksum. To support recovery and replication, AsymNVM framework adopts a consensus-based voting system to detect machine failures.

As a complete example, Figure 2 shows the how to use AsymNVM underlying APIs to implement the insert operation of skiplist (line 2~12). The program first locates the insert position by using multiple rnvm_read operations. It then allocates remote NVM resource (line 14) and flushes one operation log to the back-end immediately for recovery (line 15). Next, it appends memory log to front-end buffer for each data modification (line 16~19). Finally, if a number of operations get executed successfully or the buffer is full, the buffered memory logs will be flushed to the back-end NVM (line 20, 21).

Based on underlying APIs, AsymNVM framework implements eight lock-free and lock-based shared persistent data structures, with several data structure specific optimizations

```
// A insert request is coming
node_ptr pre = rnvm_read(head);
for(int level = MAX_LEVEL to 0){
  node_ptr cur = rnvm_read(pre.next[level]);
  key = rnvm_read(cur->key);
  suc = rnvm_read(cur->next[level]);
  while(cur_key < key){
    pre = cur; cur = suc;
    cur_key = rnvm_read(cur->key);
    suc = rnvm_read(cur->next[level]);
  } // Traverse the List
  if(lFound == -1 && key == cur_key) lFound=level;
}
node_ptr new = rnvm_malloc(size); //new node
rnvm_op_log({INSERT_OP, key, value}); //recovery
for(int level = 0 to MAX_LEVEL)
  rnvm_mem_log({new.next[level] = succs[level]});
for(int level = 0 to MAX_LEVEL)
  rnvm_mem_log({preds[level].next[level]=new});
if(counter++ == batch_size || is_fulled())
  rnvm_tx_write(backend_id);
```

Figure 2. Insert Operation for SkipList

applied to achieve even higher performance. With guidelines discussed in this paper, users can build new data structures using the defined underlying APIs.

4 Efficient Persistent Update

4.1 Basic Implementation

At low level, the persistent data structure implementation supports read/write operations. A *read* can return data that is not yet persisted, but if there is a persistent fence [51, 76] before the read, it should return the persisted data produced before the fence. When a *write* (update) returns, the data should always be persisted in the back-end NVM. The straightforward implementation of the two operations is to perform `RDMA_Read/Write` on the back-ends. To manage metadata consistently, we can apply RDMA atomic operations (it guarantees that any update to a *64-bit* data is atomic) to the critical metadata, e.g., root pointer of data structure. However, the simple implementation incurs considerable rounds of network communications, which is still much slower than memory accesses.

4.2 Decoupled Memory Log Persistency

To reduce persistency overhead, DudeTM [62] uses redo log and decouples the update of real data structure in NVM and the persistency of redo log. A write can return after the redo log is persisted and does not need to wait for the actual data structure modification. In AsymNVM, we also use memory log to improve performance. Unlike prior works, in AsymNVM architecture, the front-end and back-end nodes are distributed, so the only reasonable choice is to use redo log.

In AsymNVM framework, each write (update) will generate several *memory logs*, and the back-end node provides the transaction APIs to ensure that the memory logs are persisted atomically in an all-or-nothing manner. When memory logs in a transaction is persisted, the back-end node sends back an acknowledgement, so that the corresponding write in front-end can return and is guaranteed to be durable. To ensure correctness, the back-end node also guarantees that the modifications to the real data structure are performed, i.e., *replaying* the persisted logs, in the same order as the sequential log writing.

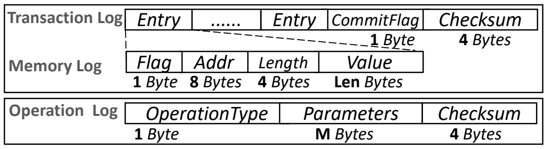


Figure 3. Memory Log vs. Operation Log

Specifically, the transaction API is `rnvm_tx_write`. The input parameter is a list of {address, value} pairs, each consists of a memory address and a value that should be written to this address. The back-end nodes keep two areas: the *data* area holds the real data structures; the *log* area records the transaction logs. The front-end can directly read the data area, but any updates have to go through the log area. To implement `rnvm_tx_write`, AsymNVM framework library constructs a continuous set of memory logs and appends to the corresponding log area in remote NVM via a single `RDMA_Write` operation. The format of these memory logs is shown in Figure 3. Every log entry includes *address, length, data*, and *one-byte flag* in the head. This flag indicates whether the value is in the memory log, it is used by an optimization related to batching, more details will be discussed in Section 4.3. A transaction will produce several log entries, a commit flag, and a *checksum* value. Since the data may be corrupted if the back-end crashes, 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 lastest transaction to validate the consistency.

Similar to DudeTM [62], the transactional API reduces the persistency latency due to modification of real data structure with decoupled log and data persistency. More importantly, different from the single machine system, it also largely reduces the required rounds of RDMA operations. Without the transaction API, multiple rounds of RDMA operations are needed when writing to multiple non-continuous areas of the NVM, or a continuous area with the size larger than a cache-line. Other works [13, 70, 75, 87] propose to add an additional flush operation to the RDMA standard. However, such solutions 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. Moreover, our implementation based on the transactional API is fixed and simple, providing better reliability.

4.3 Batching and Caching with Operation Log

To further reduce the latency for data persistency, we propose the notion of *operation log*, which is shown in Figure 3. Different from the memory logs, each write only incurs *one* operation log, which contains operation type, parameters, and checksum. A write can return after the operation log is persisted in the back-end node. Persisting operation log can be achieved by a *single* `RDMA_Write` to the back-end node.

The crucial benefit of operation log is that it enables *batching and caching*. 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 by 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 operation logs* that are not executed (i.e., have not yet modified the data area).

It is important to understand the key difference between the operation log and memory log. In short, memory log is *low-level* log *without data structure semantics* and used by back-end to *re-apply updates* on persistent data structures. Operation log is *high-level* log with data structure operation semantics. It is used by front-end to *replay operations* of data structures. Intuitively, memory log can only realize the "postpone" aspect — the real data structure modification can be delayed as long as the memory log is persisted. The key limitation of the decoupling is that the persistency of a write (high level) is realized with the persistency of its corresponding memory logs (low level). This is why it cannot achieve the "batched" aspect because the memory logs of a write are persisted anyway before executing the next write. It is sufficient for single machine NVM system [62] but not for remote NVM back-end nodes.

The essential benefit of operation log is making *persistency of a write realized at high level as well* by persisting operation log. The operation log achieves batching by combining the memory logs of multiple writes into one `rnvm_tx_write`. At lower level, operation log reduces the number of `RDMA_Write` operations. With only memory log, each write needs at least two `RDMA_Write` operations, — one for the commit and the others for at least one memory log. A write typically needs more than one memory log, thus the number of `RDMA_Write` operations is normally larger than two. With the operation log, each write needs exactly *one* `RDMA_Write` — no commit is required, because the operation log already serves the purpose of commit. The number of `RDMA_Write` operations for the memory logs is less since multiple writes can be coalesced into one `RDMA_Write` with batching, depending on the addresses. In addition, the commit for the batched memory logs (involving several writes) also needs an `RDMA_Write`. In summary, AsymNVM combines the use of operation log and memory log to enable batching and hence reduce the network round trips.

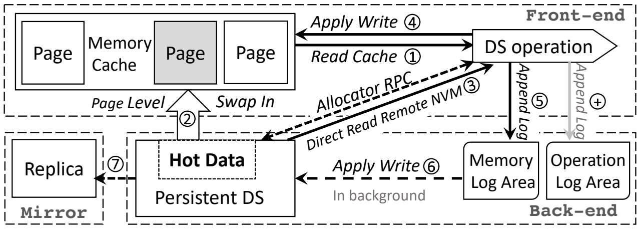


Figure 4. AsymNVM Framework Data Access Workflow

Figure 4 shows the workflow of AsymNVM framework data accesses. We divide each operation into two parts: gathering data and applying the modifications in a *Gather-Apply* model. We use the terms gather and apply to explain the holistic flow including batching and caching, which may involve multiple reads/writes. Batching can execute multiple operations together and *coalesce* memory logs to reduce the number of both `RDMA_Write` and `RDMA_Read` operations. Besides, caching reduces the number of `RDMA_Read` operations during the gathering phase. They are applicable to all data structures.

Gather Data. The data are fetched from the front-end cache whenever possible (cache-hit, ①). If not cache-hit, 1) the data will either be read from the back-end directly by using `RDMA_Read` (③) or, 2) its corresponding page will be swap-in (②) via `RDMA_Read` and put to the cache in the front-end memory. After finishing the page swapping, data is read from front-end cache (①). The choice between these two strategies depends on *specific* data structures and follows a principle that using swap-in (②) for hot data and direct remote read (③) for cold data. Hot data (e.g., the root of a B+Tree) are accessed frequently than cold data (e.g., the leaf of a B+Tree). On a persistent fence, the read after the fence needs to wait until memory logs before the fence persisted in the back-end node.

Apply Modification. Each modification operation causes one operation log to be flushed to the back-end for recovery (⊕). The operation log with format {insert_op, key, value} as shown in Figure 3 will be put in the operation log area. Then, the memory logs of format {address, data} are be generated afterward. They do not need to be flushed immediately. We replace actual data in memory log with a pointer to the previous flushed operation log to reduce the size of data write, the data/pointer is indicated by the "Flag" of memory log in Figure 3. It is correct because, after the operation log is stored in the back-end, the data structure modification is persistent and recoverable. While flushing the logs, the cached data (if exist) are modified accordingly (④). If a number 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 `rnvm_tx_write`

(⑤). These logs are then handled by the back-end (⑥) and replicated to the mirror-node (⑦) (Section 7 discusses details on replication). If the back-end fails, the front-end node handles exceptions, aborts the transaction and clears the cache.

To support a data structure larger than the capacity of the NVM in a single back-end node, AsymNVM framework supports a distributed data structure partitioning across multiple back-ends as described in Section 8.3. When the front-end node executes a data structure operation, it first locates the appropriate back-end with key-hashing and the processing is similar to the single back-end scenario.

4.4 Data Cache in Front-end Nodes

Several recent works build NVM systems using DRAM as cache [55, 62, 77, 96]. Bw-Tree [57] uses a cache layer to map logical pages (Bw-Tree nodes) to physical pages. In AsymNVM, the front-end manages a similar data structure of hash map to translate the 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, but its implementation is expensive. RR is easy to implement but does not provide any guarantee of preserving hot data. We use a hybrid approach — first choosing a random set of pages for replacement and then selecting a least used page from the set to discard. No page flush is needed because the write workflow already puts the memory logs in back-end node. With Zipf distribution workload, 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 NVM Data Management

5.1 Back-end Interface and Metadata

At the back-end nodes, we implement NVM management APIs since using only one-sided RDMA operations is inefficient. In addition, since this module provides the basic functions needed by all applications, it is convenient to support the functionalities directly in the back-end nodes to reduce the network communication with one round for each RPC invocation). In the AsymNVM framework, two memory management APIs are provided: `rnvm_malloc` and `rnvm_free`. The front-end node can use them to allocate and release back-end NVM memory. For back-end, to manage NVM resource, we use the persistent memory pool in NVML library [41] and interact with it via `malloc/free`. In the back-end, we further use a persistent bitmap to record the usage of NVM, with one bit indicating the allocation status of each block. The two design decisions ensure fast recovery. Since front-end nodes connect to the back-end via one-sided RDMA, we use the RFP RPC [84, 95] to implement the interfaces. Back-end provides two circular buffers for front-end allocator to write the requests and fetch back the responses. Because the front-end puts the requests via `RDMA_Write` and gets the responses via `RDMA_Read`, the back-end is passive and does not need to handle any network operation.

The back-end nodes also need to store *metadata* for recovery since nothing will be left on the front-end after failure. In AsymNVM frameworks, the metadata are stored in the "well-known" locations to all front-end and back-end nodes. This is the *global naming* space in the back-end NVM for recovery [6]. After restarting, AsymNVM exploits NVM file system to find the specific file whose location is in global naming space. Both front-end and back-end nodes know the location to find the needed information/data before recovery. By using the meta-data, the back-end node can find the data file and mmap the virtual memory address to the previous NVM mapped regions [71]. With this mechanism, a pointer to the back-end NVM is still valid after restarting.

The following metadata are stored in the global naming space. 1) The NVM area address, including the data and log area. It is needed for physical to virtual address translation for the corresponding front-end node. 2) The location of data structure and its auxiliary data. It is achieved by storing the root reference of data structures, e.g., the address of the root node for a tree. Additionally, other necessary data such as exclusive lock (refer to Section 6.1) and mapping table between key range and partition (refer to Section 8.3) are stored next to the root reference. These metadata from different data structure instance are persisted as a mapping table. 3) The allocation bitmaps (indicating allocation status). This information is used to reconstruct the memory usage lists and soon recover the back-end allocator. 4) Addresses of log areas, LPNs (Log Processing Number, indicating the next entry in the memory log area) and the OPNs (Operation Processing Number, indicating the last operation log whose memory log is still not persistent) are used to find the logs together with the location of the next logs. They can be used to reproduce logs (memory log) and to recover the data structure operations (operation log). Actually, the footprint of metadata in the back-end NVM is negligible in most cases. We use a hash table to index those meta-data, and each item in the hash table contains type, name and physical address which links to the related meta-data file.

5.2 Front-end Allocator

In general, the back-end allocator provides slabs (fixed size) to the front-end allocator, and the front-end manages these slabs in finer granularity. When allocation size is *larger* than the size of a slab, the front-end node directly allocates memory in the back-end using RPC and back-end interface. The slabs in the front-end are organized in three lists of full-,

Table 2. Comparison of Different Allocators.

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

partial-, empty-list according to the capacity consuming in the corresponding page. To support finer granularity allocation, we use a simple *best-fit* mechanism for slabs in the front-end. To improve the NVM utilization, a threshold is defined as the maximum free blocks number, and the front-end nodes will reclaim free blocks periodically. While reclaiming, the front-end nodes send the request to the back-end nodes to free the reclamation slabs via RPC. Note that, with persistent bitmap in the global naming space, AsymNVM can reconstruct the allocation status only in the slab level while recovering.

Benchmark. We compare the two-tier allocator of AsymNVM framework with persistent allocator and Glibc allocator (allocated size varies 32 bytes to 128 bytes). As table 2 shows, Glibc achieves the highest throughput (21.0/57.0 MOPS) but without persistent guarantee. Pmem allocator is a *single-node* persistent allocator from NVML project [41], and can reach 1.42 MOPS. With only the back-end (RPC) allocator, the throughput is only 23% and 64% of Pmem allocator because of the network overhead. With two-tier allocator, the throughput is similar or even better performance than Pmem allocator.

6 Concurrency Control

6.1 Exclusive Write

Under SWMR mode, write operations are exclusive. Therefore, while executing write operation, the writer should acquire an exclusive lock first. The lock is created after initializing the data structure, and its location is alongside the root reference for recovery. If it succeeds, it fetches the LPN (Log Processing Number, refer to Section 5.1 for its management), and then executes the write operation. After finishing appending logs to the remote NVM based on LPN, it should release this exclusive lock. While the exclusive writes are being performed, other write operations (if any) will be blocked until the current writer has completed the current write operation.

As shown in Algorithm 1, we leverage the RDMA atomic verbs, RDMA_Compare_And_Swap [101] to implement it as a writer lock. When releasing the lock, the writer resets it via a RDMA_Write. In AsymNVM framework, to handle the failures while holding the lock, every write lock acquire/release operation should write a record (lock-ahead log) to the back-end node before appending the memory logs. Thus, if the front-end crashed before releasing the lock, we can identify the lock need to be released during recovery.

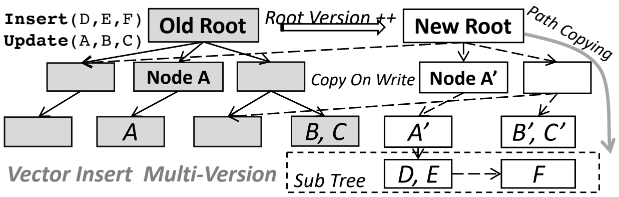


Figure 5. Overall Multi-version Data Structure

Algorithm 1 Writer Lock

```
procedure WRITER LOCK(L)
    while $rdma\_compare\_and\_swap(\mathbf{L}, Locked) = Locked$
procedure WRITER UNLOCK(L)
    $rdma\_atomic\_write(\mathbf{L}, UnLocked)$
```

6.2 Lock-Free Data Structure

Multi-version Data Structure. Our design of lock-free tree-like data structure is inspired by append-only B-Tree [3, 32] and persistent data structures [22, 69]. The 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 [78]. Then, the nodes in the path will update some of the pointers pointing to the old data. Finally, the data will be inserted into the new path. After finishing all these operations, the root will be atomically changed to the new root by updating the root pointer. *Vector operation* discussed in Section 8 can help here to reduce the number of network round trips significantly. Since the readers can always get consistent data, this kind of concurrent control does not affect the performance of readers.

Lightweight Recovery. In the multi-version data structure, the only in-place update is the root pointer. However, the pointer changing is atomic. Therefore, it does not need a recovery process as the discussion in [6]. While recovering, the front-end can use the root pointer (which is well-known via naming mechanism) to find out the whole data structure.

NVM Reclamation. The use of lock-free data structures needs to ensure that memory is safely reclaimed, which further complicates the garbage collection[25]. In AsymNVM framework, this requirement is achieved by a lazy garbage collection mechanism. After version changes, the front-end should release the old version's data. Back-end delays this operation for $n + l$ μs and then reclaims corresponding memory. It requires that the latency of each pending data structure operation should be less than n μs to avoid memory leak (access the reclaimed memory). A smaller n cause frequent retries of read operation, and a larger n causes lower NVM utilization. We fixed n/l as 4000/1000 after pre-run the whole system.

Algorithm 2 Writer Preferred Reader Lock

```
procedure Write Begin(SN)
    $gcc\_atomic\_increment(\mathbf{SN})$
procedure Write End(SN)
    $gcc\_atomic\_increment(\mathbf{SN})$
procedure Reader Lock(SN)
    do
        $ret \leftarrow rdma\_atomic\_read(\mathbf{SN})$
    while $ret$ is $odd$
    $start\_sn \leftarrow ret$
procedure Reader Unlock(SN)
    return $start\_sn \neq rdma\_atomic\_read(\mathbf{SN})$
```

6.3 Lock Based Data Structure

Write-Preferred Read Lock. RDMA atomic operations are appropriate APIs to implement distributed sequencer [48, 101] and lock [66, 101]. Algorithm 2 shows the implementation of retry-based optimistic read locks by using the sequence number (SN), an 8-byte integer variable. Distinct from Algorithm 1, which is invoked by the front-end, `Write_Begin` and `Write_End` is executed by the back-end. When a back-end applies the persisted memory log to the real data structure in NVM, it atomically increases the SN twice before and after the modification. `Reader Lock` and `Reader Unlock` are invoked by front-ends before and after a sequence of reads. To disallow reads when data are being updated, it needs to wait until the current SN is odd. To ensure reads in between get the consistent view, `Reader Unlock` needs to check that SN is unchanged since `Reader Lock`. If the data are inconsistent, the readers need to *retry and fetch* the data again.

Lock Benchmark. We make a ping-point test about the lock's performance as in Frangipani [86]. Six readers and one writer try to access the same data in the back-end and the workload is 10% write and 90% read. The results show that each reader's average throughput is 260 KOPS (1.56 MOPS in total) and writer's throughput is 539 KOPS. The reader's failed ratio (i.e., a try for reading data is failed) is only 3%. When setting the workload as 50% write, reader's throughput will drop to only 165 KOPS with a 26% fail ratio, and writer's throughput remains at 510 KOPS. The write-preferred lock makes writers to gain a higher throughput than readers.

Discussion Lock-free data structures benefit the reader but create multiple copies by writers. Lock-based data structures prioritize the writer without extra copies, but readers have to read multiple times until consistent data are obtained. The right choice depends on specific applications.

7 Recovery and Replication

7.1 Replication

The AsymNVM needs *at least* one mirror-node attached with the non-volatile device like SSD, Disk or even NVM. To improve fault tolerance, we deploy mirror-nodes to different racks. The back-end nodes replicate the memory/operation logs to mirror-nodes before committing the transaction and acknowledging the front-end. If the mirror-node is equipped with NVM, the mirror-node also implements a log replay function to apply logs to the replicated data structure. Replicated logs in mirror-nodes are read-only. When the back-end crashes, if the mirror-node is equipped with NVM, it will be voted as the new back-end. Otherwise, the front-end nodes use the logs and data structures from the mirror-node to recover the data structure to a new back-end.

In our implementation, the back-end is responsible for ensuring that the replica is persistent in its mirror-node. The front-end only needs to ensure that data is stored in the back-end NVM, but does not wait for an acknowledge after replication completes. Thus, the replication phase is performed asynchronously. In our AsymNVM deployment, two machines are used as mirror-nodes for each back-end. Note that mirror nodes do not lead to low utilization: since the mirror-node has no specific constrains (e.g., reproducing logs), multiple back-ends can share one mirror-node. In fact, one back-end can also become the mirror-node of another back-end.

7.2 Data Structure Recovery

With the log mechanism, AsymNVM ensures crash consistency with the non-volatile data and logs stored in the back-end node. Similar to most distributed systems, we use a consensus-based voting system, i.e., ZooKeeper [36] coordination service, to detect machine failures. The replicated Zookeeper instance can run on at least three other nodes. Leases are used to identify whether a node is still alive or not. If the lease expires and the node cannot renew its lease, the node is considered to be crashed. We implement this mechanism as *keepAlive service*. Next we discuss different crash scenarios (assume the crashed front-end will always reboot within a limited time).

Case 1: Front-end reader crash. If the front-end crashes when performing a read, it needs to gain the meta-data via naming mechanism and resume execution after rebooting.

Case 2: Front-end writer crash. If the front-end crashes when performing a write, the back-end will know this information through keepAlive service. After the front-end reboots, if there still exists memory logs not replayed from the front-end, the back-end will validate whether all log entries of the last transaction are flushed to the NVM or not via checksum. If this transaction log is consistent **(Case 2.a)**, the back-end will notify the front-end to resume as normal, same as **(Case 1)**. Otherwise **(Case 2.b)**, the back-end will notify the front-end that the last transaction log is inconsistent. Thus, the front-end will fetch the LPN, OPN and operation logs whose memory logs is not replayed, and then re-executes the uncommitted transaction. **(Case 2.c)** In most cases, there are several operation logs whose corresponding memory logs are not flushed to back-end yet. The front-end will process as **Case 2.b**.

Case 3: Back-end transient failure. When the back-end fails while executing RDMA_Read/Write, the front-end can detect it through the feedback from RNIC. Then it will wait for the notification for the back-end recovery or a new voted back-end. After rebooting, the back-end will first reconstruct the mapping between the physical addresses and virtual addresses. The mapping is stored in NVM and well-known via *global addressing* scheme as described in Section 5.1. After that, the back-end checks whether the last transaction log is consistent. If there is no transaction/operation logs left, or the transaction log is consistent **(Case 3.a)**, the back-end can start its normal execution immediately, i.e., reproducing memory logs if any log has not been applied, and then notify its liveness to the front-ends. If the transaction log is inconsistent **(Case 3.b)**, the back-end will notify the corresponding front-end nodes, and they will flush the memory logs again to redo this transaction. It is possible since the front-end must have not received the persistent acknowledgement. If existing operation logs are ahead of current memory logs **(Case 3.c)**, which means that the memory logs have not been flushed from front-end due to batching, the back-end will notify the front-end, and the front-end will continue to execute the next operation.

Case 4: Back-end permanent failure. In this case, one of the mirror-nodes will be voted as the new back-end and provides service to the front-end. The new back-end will broadcast to living front-ends to announce such event. After that, the front-end will reconstruct the data structures to a new back-end by using the data and logs in the mirror-nodes.

Case 5: Mirror node crash. The consensus-based service will detect the failure and remove it out of the group.

If both front-end and back-end crash, the keepAlive service coordinates front-end and back-end nodes, and lets the back-end nodes to recover first. They will first check the status as in **Case 2**. Then, the front-end will determine how to recover according to the back-end's failure cases in **Case 1**.

8 Data Structure Implementations

The underlying APIs of AsymNVM framework are general to implement various persistent data structures with high performance. In this section, we discuss the implementation of several commonly-used data structures and propose data structure specific optimizations.

8.1 List-Based Data Structure

We implement Stack and Queue by using the List data structure. Because the only data items that can be accessed in Stack or Queue are headers or tails which are more frequently accessed, the front-end only needs to cache each nodes pointed by them to reduce the number of rnvm_read. If there are not enough data items of headers and tails in the cache, i.e., less than a threshold, the front-end will fetch back corresponding data to the cache. Moreover, due to the access pattern, the operations may be combined 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, and the effective enqueues will be annulled by dequeues for Queue. Such an opportunity can be identified by checking the un-executed operation logs in the front-end memory. For instance, for a pop operation to the stack data structure, we first need 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. This optimization based on operation log reduces the RDMA operations significantly.

8.2 Hash Table

Key-value items in the hash table can be subdivided into hot ones and cold ones, and the front-end can buffer items with the hotkey. The caching granularity is each key-value item. Since with batching optimization, hash table gains no significant performance improvement, we do not enable batching.

8.3 Tree-Like Data Structure

Tree-like data structures (e.g., binary search tree, B+Tree) have the hierarchical organization. The nodes in higher (near the root) level are more frequently accessed than lower level nodes. Based on this observation, we choose to cache higher level non-leaf nodes with higher priority. 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 stays unchanged. The native LRU algorithm treats higher level nodes and lower level nodes in the same way, and hence incurs frequently cache misses. Compared to LRU, our mechanism gives a "hint" to cache the hot nodes. In addition, due to the sorted nature of tree-like data structures, the performance can be improved when the operations are also sorted. Based on this insight, we pack the sorted operations into a *vector*. The operations are performed 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 3 shows vector_write, one vector operation, in a binary search tree following the Gather-Apply paradigm. It first 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 the 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

Algorithm 3 Vector Write

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

modify the same NVM memory, they will be compacted to one NVM write in `vector_write`.

Data Structure Partition We use partitioning to eliminate the potential bottleneck due to the lock, and achieve both high throughput and better scalability [58, 89]. Similar to the support of large size data structure in Section 4.3, AsymNVM framework adopts key-hashing partitioning to improve the performance of various data structures. Each partition has its own write lock and index data structure. While the writer is executing write operation in one of the partitions, multiple readers can still concurrently access other partitions. These mapping tables between key range and partition are stored in the global naming space for recovery as describes in Section 5.1.

8.4 SkipList

The caching and batching optimization described for tree-like data structure can also be applied to skip-list to reduce the number of RDMA operations. In skiplist, we cache the nodes with higher degree. Skiplist are naturally lock-free and the only concern is to carefully choose the order of operations [17, 24]. The writer first creates the new allocated node and sets the successor pointers in this node accordingly. After that, the predecessor pointers will be updated *from the bottom to the top*. Readers can still get (potentially different) consistent views of skiplist in such scenario, thus the lock is not required [33].

9 Evaluation

9.1 Evaluation Setup

System Configuration. The experiment cluster contains ten machines configured as seven front-ends, two mirror nodes and one back-end. Each machine equips with one Mellanox CX-3 InfiniBand (40Gbps) and runs Ubuntu 14.04 (Linux 4.4.0 kernel) and MLNX_OFED_LINUX-4.2 driver. Each front-end or mirror node is equipped with an 8-cores CPU (Intel Xeon E5-2640 v2, 2.0 GHz), 96 GB memory. The back-end is a 4-sockets 72-cores (Intel Xeon Gold 5220, 2.2 GHz) server with four Intel Optane DC PM Module, totaly 3 TB (4 socket × 6 channel × 128 GB/DIMM) of NVMM. All these data structures and applications where implemented using C++11 and compiled using GCC 4.8.5.

Real NVM Devices. We set Intel Optane DC PM as NVM running in App Direct Mode, and it is initialized as a character file to enable DAX mapping capabilities [42] via `mmap`. We use one of the namespaces to be registered by RDMA for remote access, which has a capacity of 718 GB (6 channels).

9.2 AsymNVM Performance

We implement eight widely-used data structures covering different access time complexity ($O(1)$ and $O(log(n))$): 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). To simplify the evaluations, the key and value of our data structure are 8 Bytes and 64 Bytes, respectively. We set probability p of skiplist as 0.5 and fan-out d of B+tree as 32. In the following evaluations, we implement PUT with insert operations and GET with find operations, respectively, and each IO means a PUT/GET operation. In addition, we use two transaction applications: TATP [82] and SmallBank [90]. Table 3[1] shows the overall performance, breakdown and comparisons to symmetric/naive ones.

Compare to Naive Implementation. The naive implementation accesses remote NVM directly using RDMA reads and writes without any optimizations. The complete implementation denoted as AsymNVM-RCB can provide nearly 5∼12× improvements compared to naïve implementation.

Compare to the Symmetric Setting. We implement the symmetric NVM architecture by storing data structures in local NVM and storing logs in remote NVM for fault tolerance. The logs are flushed asynchronously (without waiting for the acknowledgement from remote nodes). It reaches the upper-bound performance of symmetric NVM architecture, but will obviously cause inconsistency. From the results, we see that, AsymNVM-RCB still achieves *comparable* performance to the optimistic performance of symmetric NVM data structures without consistency. Especially, in a few cases (i.e., Queue, Stack, BST, MV-BST, MV-BPT), the performance of AsymNVM-RCB is even better than symmetric NVM without batching. This is mainly due to the small front-end cache.

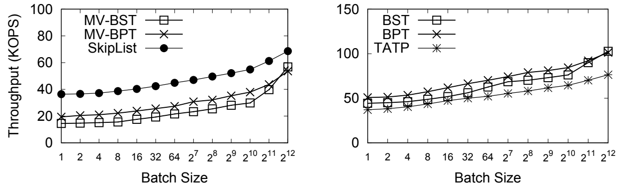
End-to-end Performance. We evaluate application performance by two transaction benchmarks: SmallBank [90] and TATP [82]. We use HashTable and BPT as the index data structure of SmallBank and TATP, separately. As shown in Table 3, the results show AsymNVM can improve the throughput to 1.39× in SmallBank and 7.20× in TATP.

Cost Comparison. We also make comparison about the usage of NVM. In the 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, the real usage of each NVM is S_i).

[1] 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.

Table 3. Performance Comparison (R: using log reproducing, C: caching 10% NVM size in the front-end, B: batching with size 1024. The evaluation uses one(front-end)-to-one(back-end) setting with 100% write workloads harnessing all optimization.

KOPS	TX(SmallBank)	TX(TATP)	Queue	Stack	HashTable	SkipList	BST	BPT	MV-BST	MV-BPT
Symmetric	643	112.7	1187	1076	1148	117.6	77.1	124.3	38.2	14.0
Symmetric-B	-	149.1	2459	2293	-	189.3	134.8	174.2	126.6	62.7
AsymNVM-Naïve	243	10.2	319	211	304	5.2	13.5	9.5	6.6	7.0
AsymNVM-R	278	12.4	615	589	344	6.8	15.7	11.7	11.5	9.7
AsymNVM-RC	338	37.1	-	-	439	36.4	44.4	50.9	19.4	14.5
AsymNVM-RCB	-	73.4	1392	1139	-	54.9	76.2	84.4	37.9	29.8



(a) Lock-Free Data Structure **(b)** Lock Based Data Structure

Figure 6. Throughput with different batch sizes

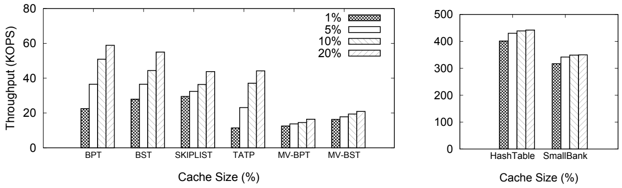
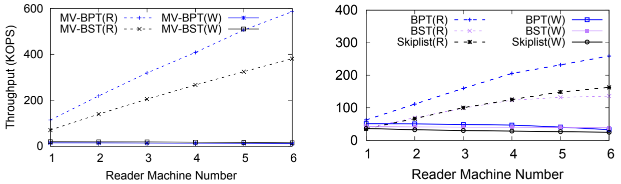


Figure 7. Throughput with different cache sizes



(a) Lock-Free Data Structure **(b)** Lock Based Data Structure

Figure 8. Scalability of multiple readers.

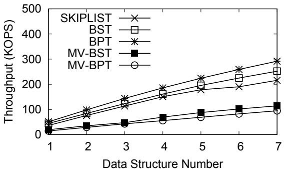


Figure 9. Multiple DS.

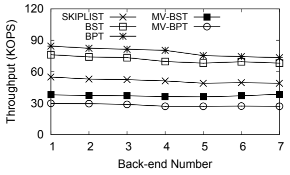
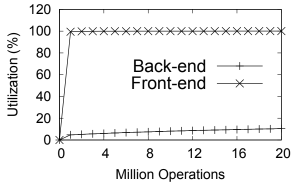


Figure 10. Partitions.

Besides, AsymNVM needs $n_2 = \lceil \sum_{i=1}^{i=m} S_i \rceil$ NVM devices and $n_2 \ll n_1$. As we mentioned in Section 1, each back-end only needs smaller capacity less than S_0, thereby the necessary NVM n_2 will be fewer than n_1 ($n_1 = m$).



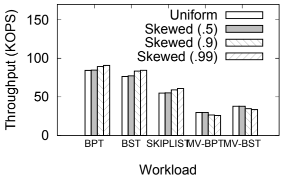


Figure 11. CPU utilization. **Figure 12.** Throughput(Zipf)

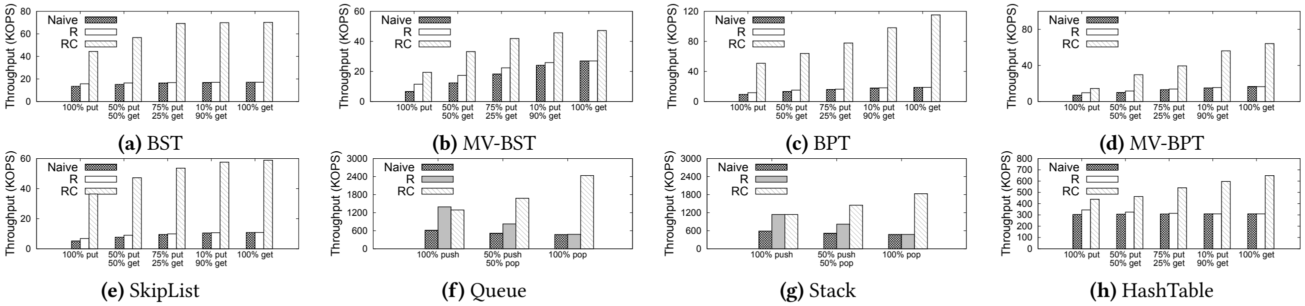
9.3 CPU Utilization

Figure 11 shows CPU utilization of front-end and back-end nodes. The workload is 10% put and 90% get in BST. The front-end node keeps running with nearly 100% CPU utilization but the request only incurs very small CPU usage (4%~10% CPU utilization). It matches the intuition that, the back-end has very little computing overhead. The major overhead comes from replaying the persisted logs and managing slabs.

9.4 Effects of Batching and Caching

Batching. We measure the performance of batching with vector operations under different batch sizes from 1 to 4096. The results are in Figure 6. MV-BST can be improved by 2.76× (from 19.4 KOPS without batch to 53.6 KOPS with batch size 4096). The improvement for MV-BPT is about 3.91× (from 14.5 KOPS to 56.7 KOPS with batch size 4096). The improvements are 131%, 102%, and 88% for BST, BPT, and SkipList, respectively. Multi-version data structures need to perform path copying which incurs many write operations. The batching can effectively reduce such overhead.

Caching. We measure the benefit of caching under different cache sizes. BST, B+Tree, hashtable, and skiplist are used here (We do not consider queue and stack since they need very small cache), and the results are shown in Figure 7. Overall, the throughput increases with the increase of cache sizes. Notice that MV-BPT and MV-BST do not get too much improvement with catching. This is due to the fact that the data modified are still kept in memory for multi-version data structures. We also measure the improvement due to our special optimizations of tree-like data structures. The results show that, when using native LRU strategy (access any data including the lower level nodes through the front-end cache), the BPT can only reach 31.4 KOPS which is 38% lower than AsymNVM.



(a) BST

(b) MV-BST

(c) BPT

(d) MV-BPT

(e) SkipList

(f) Queue

(g) Stack

(h) HashTable

Figure 13. Throughput with Different Workloads (100%put, 50%put+50%get, 25%put+75%get, 10%put+90%get, 100%get)

9.5 Multiple Front-end/Back-end Nodes

Since AsymNVM can support the operation in SWMR mode, we also measure the scalability of AsymNVM by using multiple readers and one writer. The results are shown in Figure 8 (The workload of the writer is 100% insert, R/W represents reader/writer). Note that AsymNVM does not support shared stack/queue since the write operation incurs a higher contention than other data structures, leading to low performance. We make comparisons between lock based and lock-free data structures as we mentioned in Section 6.

The readers' performance can scale well with the increasing number of front-end nodes. We see that, 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 is 39% lower than the value with only one reader. In the case of MV-BST, performance degradation is about 10%. The results confirm that the multi-version data structures do benefit the readers.

From another aspect, the lock-free data structures scale better than their lock-based counterparts. The readers in Figure 8b have about 2.0∼2.8× higher performance than the readers in Figure 8a. Retries incurred by the failed read is the main cause for the lower performance. The portion of retry is about 8%∼21% of total operations with 6 readers and 100% insert. Lower write workload will decrease the ratio of retries.

We also measure the throughput of multiple front-ends sharing one NVM, each accessing its own data structure. Each front-end uses the same type of data structure but with different instances. Figure 9 shows that the scalability is almost linear. The performance degradation for a single client is about 7%∼19% compared to the one-to-one deployment.

As shown in Figure 10, we measure the performance after partitioning data structure to multiple back-ends. The results show no significant performance degradation after partitioning. The reason is that the partition in each back-end is strictly isolated with other back-ends.

9.6 Different Workloads

We evaluate the performance of AsymNVM in different workloads in Figure 12. We use YCSB tool [15] to generate skewed workloads. The keys are generated according to Zipf distribution with parameter .5, .9 and .99. AsymNVM adapts well to skewed workload, showing a comparable performance even when the Zipf parameter is .99.

We also measure our data structure implementations using industry workloads from an online service of Alibaba. The workloads trace of real world application behaviors and satisfy the power-law distribution. In this workloads, the key is hashed to 64 bytes and value is 64 Bytes∼8 KB, and operations are PUSH/POP (queue/stack) and PUT/GET (other index data structures) respectively. Figure 13 shows the throughput of using different read/write ratios from a single writer front-end node. For simplicity, insert operation is used as write and find operation is used as read. With fewer read operations, the performance decreases due to more overhead brought by write operations. Comparing BPT/BST to their MV-counterparts, BPT/BST have relatively higher performance. For instance, with the full write workload, there are about 54%/71% performance gap. This is because, in the MV-version, the write operations need to write more data during path copying.

10 Related Work

Single-node NVM systems [10, 14, 28, 49, 50, 52, 61, 62, 77, 96, 100, 104] and management subsystems [7, 72, 79, 85, 91, 94, 97] provide direct access to NVM via memory bus but cause lower utilization of NVM and inaccessible facing node failures. To provide durability for transaction, ATOM [47] uses synchronous undo logging with hardware support and LB++ [46] supports lightweight epoch ordering by dynamically tracking inter-thread dependencies. Distributed NVM systems including Hotpot [81], Mojim [102], FaRM [20, 21], and FlatStore [12] combine the NVM devices together with RDMA, and they are all using symmetric deployment. With the same architecture, Octopus [63] and Orion [98] are distributed file system built for RDMA and NVM. In the hardware level, Hu *et al* [35] improve the persistence parallelism of flushing data from RDMA network to NVM. Currently, the asymmetric deployments such as [63, 88] provide storage interfaces including NVMe-oF [16] and Crail [83]. Especially, NVMe-oF is designed to work with any suitable storage fabric technology. However, they neither support byte addressable nor provides transaction interface, i.e., they cannot implement persistent data structures. Different from Aerie [93] which also uses transactional write logs, AsymNVM is a distributed system with suitable interfaces for NVM. Prior works [7, 62, 98, 102] model the persist ordering overhead by adding a fixed extra delay, and we choose the real NVM device to evaluate AsymNVM.

Several projects aim to design the future disaggregation data center, such as [1, 31, 38, 40, 65, 74, 80]. LegoOS [80] proposes splitkernel, an OS model disseminates functionalities into loosely-coupled monitors. Some of these works focus on how to design remote memory. Aguilera *et al.* [2] introduce benefits and challenges about applying remote memory. RackOut [67] mitigates load imbalance of memory disaggregation. Some RDMA extensions such as atomic object read [18] can further improve the access to remote memory in the disaggregation architecture. AsymNVM is an asymmetric architecture that can be used to organize the disaggregated NVM resource.

11 Conclusion

This paper rethinks NVM deployment and makes a case for the *asymmetric* NVM architecture, which decouples servers from persistent data storage. We build *AsymNVM framework* based on AsymNVM architecture that implements: 1) high performance persistent data structure update; 2) NVM data management; 3) concurrency control; and 4) crash-consistency and replication. The central idea is to use *operation logs* to reduce the stall due to RDMA writes and enable efficient batching and caching in front-end nodes. The results show that AsymNVM achieves similar or better performance compared to the best possible symmetric architecture while avoiding all the drawbacks with *disaggregation*.

Acknowledgments

We thank our shepherd Prof. Steven Swanson and the anonymous reviewers (OSDI' 18, ASPLOS' 19, ISCA' 19, MICRO' 19, ASPLOS' 20) for their valuable feedbacks. Teng Ma, Kang Chen and Yongwei Wu are with the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This Work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61877035, 61433008, 61373145, 61572280), China Postdoctoral Science Foundation (2018M630162), Young Scientists Fund of the National Natural Science Foundation of China (Grant No. 61802219) National Science Foundation (CCF-1657333,1750656).

References

[1] 2012. SeaMicro Technology Overview. http://seamicro.com/sites/default/files/SM_TO01_64_v2.5.pdf.

[2] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 121–127.

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

[4] 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. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems* 9, 2 (2013), 13.

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

[6] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.

[7] Kumud Bhandari, Dhruva R Chakrabarti, and Hansjuergen Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. *conference on object oriented programming systems languages and applications* 51, 10 (2016), 677–694.

[8] Jeff Bonwick et al. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator.. In *USENIX summer*, Vol. 16. Boston, MA, USA.

[9] 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. 2010. Phase change memory technology. *Journal of Vacuum Science and Technology B* 28, 2 (2010), 223–262.

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

[11] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: an Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[13] Douglas Chet. 2015. 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.

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

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[16] Patrice Couvert. 2016. High speed IO processor for NVMe over fabric (NVMeoF). *Flash Memory Summit* (2016).

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

[18] Alexandres Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. SABRes: Atomic object reads for in-memory rack-scale computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 6.

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

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

[21] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 54–70.

[22] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. 1989. Making data structures persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124.

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

[24] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third annual ACM Symposium on Principles of Distributed Computing*. ACM, 50–59.

[25] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[26] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 249–264.

[27] Garth A Gibson and Rodney Van Meter. 2000. Network attached storage architecture. *Commun. ACM* 43, 11 (2000), 37–37.

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

[29] Google. 2018. ClusterData2011 2 traces. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667.

[31] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 10.

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

[33] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer.

[34] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. 2018. 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*. ACM, 535–548.

[35] Xing Hu, Matheus Ogleari, Jishen Zhao, Shuangchen Li, Abanti Basak, and Yuan Xie. 2018. Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[36] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX Annual Technical Conference*, Vol. 8. Boston, MA, USA.

[37] Intel. [n. d.]. 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.

[38] Intel. 2013. Intel, Facebook Collaborate on Future Data Center Rack Technologies. https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-\data-center-rack-technologies/.

[39] Intel. 2017. 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.

[40] Intel. 2018. Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[41] Intel. 2018. NVM Library. https://github.com/pmem/nvml.

[42] Intel. 2019. Configure, Manage, and Profile Intel® Optane™ DC Persistent Memory Modules. https://software.intel.com/en-us/articles/configure-manage-and-profile-intel-optane-dc-persistent-memory-modules.

[43] Intel. 2019. What Is Intel Optane DC Persistent Memory? https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

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

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[46] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 660–671.

[47] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 361–372.

[48] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance {RDMA} Systems. In *2016 {USENIX} Annual Technical Conference (USENIX ATC 16)*. 437–450.

[49] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 613–626.

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

[51] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. 2016. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 58.

[52] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. 2018. Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 315–327.

[53] Butler W Lampson. 1983. Hints for computer system design. In *ACM SIGOPS Operating Systems Review*, Vol. 17. ACM, 33–48.

[54] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. *International Symposium on Computer Architecture* 37, 3 (2009), 2–13.

[55] Hyung Gyu Lee, Seungcheol Baek, Chrysostomos Nicopoulos, and Jongman Kim. 2011. An energy-and performance-aware DRAM cache architecture for hybrid DRAM/PCM main memory systems. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 381–387.

[56] Lucas Lersch, Ismail Oukid, Ivan Schreter, and Wolfgang Lehner. 2017. Rethinking DRAM caching for LSMs in an NVRAM environment. In *European Conference on Advances in Databases and Information Systems*. Springer, 326–340.

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

[58] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.

[59] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[60] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.

[61] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+ tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[62] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. 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*. ACM, 329–343.

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

[64] Sparsh Mittal and Jeffrey S Vetter. 2016. 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 (2016), 1537–1550.

[65] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage.. In *NSDI*. 17–33.

[66] Sundeep Narravula, A Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K Panda. 2007. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*. IEEE, 583–590.

[67] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The case for RackOut: Scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 182–195.

[68] Stanko Novakovic, Alexandros Daglis, Dmitrii Ustiugov, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2019. Mitigating load imbalance in distributed data serving with rack-scale memory pooling. *ACM Transactions on Computer Systems (TOCS)* 36, 2 (2019), 6.

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

[70] OpenFabric. 2015. RDMA and NVM Programming Model. https://www.openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Monday/monday_12.pdf.

[71] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 8.

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

[73] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 371–386.

[74] Hewlett Packard. 2018. The Machine. https://www.labs.hpe.com/the-machine.

[75] Grun Paul, Bates Stephen, and Rob Davis. 2018. Persistent Memory over Fabric. https://www.snia.org/PM-Summit/2018/presentations/05_PM_Summit_Grun_PM_Final_Post_CORRECTED.pdf.

[76] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 265–276.

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

[78] Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)* 3, 4 (2008), 2.

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

[80] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

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

[82] Neuvonen Simo, Wolski Antoni, manner Markku, and Raatikka Vilho. 2011. TATP Benchmark. http://tatpbenchmark.sourceforge.net/.

[83] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[84] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA.. In *EuroSys*. 1–15.

[85] Michael M Swift. 2017. Draft: Towards o (1) memory. In *The 16th Workshop on Hot Topics in Operating Systems (HotOS)*.

[86] Chandramohan A Thekkath, Timothy Mann, and Edward K Lee. 1997. Frangipani: a scalable distributed file system. *symposium on operating systems principles* 31, 5 (1997), 224–237.

[87] Talpey Tom. 2015. Remote Access to Ultra-Low-Latency Storage. https://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/Talpey-Remote_Access_Storage.pdf.

[88] Shin-Yeh Tsai and Yiying Zhang. 2018. Mitsume: an Object-Based Remote Memory System. In *Workshop on Warehouse-scale Memory Systems (WAMS)*. ACM.

[89] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[90] Brown University. 2018. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[91] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1541–1555.

[92] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *USENIX FAST 11*, Vol. 11. 61–75.

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

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

[95] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, CHEN Kang, and Zhenyu Guo. 2019. RF-RPC: Remote Fetching RPC Paradigm for RDMA-Enabled Network. *IEEE Transactions on Parallel & Distributed Systems* 30, 7 (July 2019), 1657–1671. https://doi.org/10.1109/TPDS.2018.2889718

[96] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX, 349–362.

[97] Jian Xu and Steven Swanson. 2016. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*. USENIX Association, 323–338.

[98] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 221–234.

[99] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2019. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. *arXiv preprint arXiv:1908.03583* (2019).

[100] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *USENIX FAST 15*, Vol. 15. 167–181.

[101] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1571–1586.

[102] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 3–18.

[103] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. *International Symposium on Computer Architecture* 37, 3 (2009), 14–23.

[104] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.