Mikhail Zarubin, Thomas Kissinger, Dirk Habich, Wolfgang Lehner

Efficient compute node-local replication mechanisms for NVRAM-centric data structures

Erstveröffentlichung in / First published in:

SIGMOD/PODS '18: International Conference on Management of Data, Houston, 11.06.2018. ACM Digital Library, Art. Nr. 7. ISBN: 978-1-4503-5853-8

DOI: https://doi.org/10.1145/3211922.3211931

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

Efficient Compute Node-Local Replication Mechanisms for NVRAM-Centric Data Structures

Mikhail Zarubin Thomas Kissinger Dirk Habich Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
first.last@tu-dresden.de

ABSTRACT

Non-volatile random-access memory (NVRAM) is about to hit the market and will require significant changes to the architecture of in-memory database systems. Since such hybrid DRAM-NVRAM database systems will keep the primary data solely persistent in the NVRAM, efficient replication mechanisms need to be considered to prevent data losses and to guarantee high availability in case of NVDIMM failures. In this paper, we argue for a software-based replication approach and present compute node-local mechanisms to provide the building blocks for an efficient NVRAM replication with a low latency and throughput penalty. Within our evaluation, we measured up to 10x less overhead for our optimized replication mechanisms compared to the basic replication mechanism of the Intel persistent memory development kit (PMDK).

CCS CONCEPTS

• **Information systems** → **Storage class memory**; **Storage replication**; *Main memory engines*;

KEYWORDS

nvram, replication, in-memory, database, data strcutures

ACM Reference Format:
Mikhail Zarubin, Thomas Kissinger, Dirk Habich, Wolfgang Lehner. 2018. Efficient Compute Node-Local Replication Mechanisms for NVRAM-Centric Data Structures. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3211922.3211931

1 INTRODUCTION

The approaching availability of *non-volatile random-access memory (NVRAM)* – i.e., Intel 3D XPoint – will lead to significant changes in the architecture of in-memory database systems. Since this novel kind of memory allows persistent writes while featuring DRAM-like characteristics such as byte respective cache line addressability and low access latencies, it is likely to replace block-based secondary storage (e.g., HDDs or SSDs) for storing the primary data of the DBMS [2, 10, 12, 16]. Moreover, the increased packing density of data cells on non-volatile DIMMs (NVDIMM) results in higher capacities at lower cost compared to traditional volatile DRAM DIMMs leading to a higher NVRAM-to-DRAM ratio in the system. Nevertheless, NVRAM faces certain drawbacks such as a lower write endurance, lower read and especially write latency, and an increased error-proneness compared to DRAM. Since NVRAM-centric database systems will keep the primary data solely persistent in the NVRAM, efficient replication mechanisms need to be considered to prevent data losses and to guarantee high availability in case of NVDIMM failures. Such NVDIMM failures can range from single data cell failures to region, chip, or entire NVDIMM failures. The impact of such a failure becomes even more severe if the NVDIMMs are operated in an interleaved mode, which is usually the case to fully utilize all memory channels and reach the peak bandwidth.

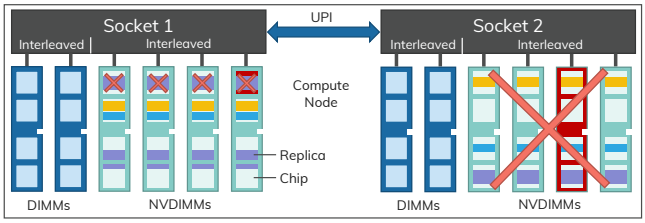


Figure 1: Data structure replication on a single compute node to protect primary data against NVDIMM failures.

Figure 1 schematically shows a hybrid DRAM-NVRAM platform consisting of two sockets, each running the DRAM DIMMs and NVDIMMs in interleaved mode. Hence, logical non-volatile memory regions are physically stored in an interleaved way across the local NVDIMMs of a single socket. In case of a single chip failure, the data of all horizontally adjacent chips becomes unavailable too and in case of a full NVDIMM failure, the entire NVRAM of the affected socket becomes unavailable. To harden the DBMS against all of those failures, non-volatile memory regions require a certain degree or redundancy. The common approach to achieve this is to replicate the data.

Discussion. Figure 2 presents a general overview of commonly used replication flows. First of all, data copying can be implemented on a physical or logical (log shipping) level. The former case is typically most efficient as it performs pure duplication of memory regions and can be tuned for a particular hardware platform. The latter case is usually used for hot standby remote replication and induces significant delays due to the log synchronization and replay [23]. Secondly, various backup storages can be deployed ranging from expensive, though fast DRAM, which sacrifices the

DOI: https://doi.org/10.1145/3211922.3211931

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

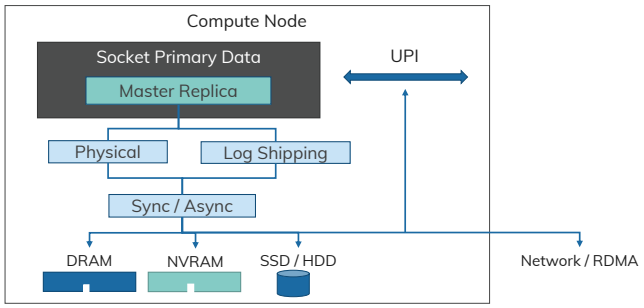


Figure 2: Replication flows to improve DB reliability and high availability.

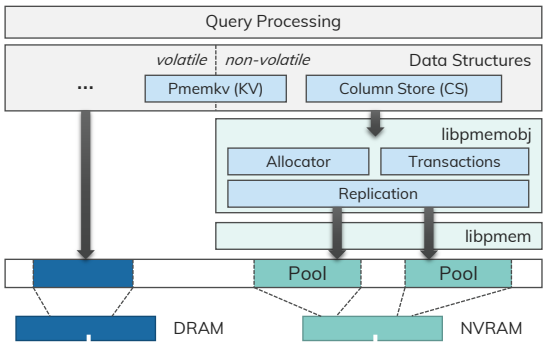


Figure 3: PMDK-Integrated DBMS Architecture.

non-volatility property of NVRAM, to SSDs or network-based storages which either impose prohibitively high delays in synchronous mode due to the network latency or reduce level of consistency in asynchronous mode. We target compute node-local synchronous physical NVRAM-backed approach since other alternatives can not replace it for failure model discussed in Section 1 without the drawbacks above. Under "compute node-local" we understand replication on the local socket and/or across multiple sockets, which also improves data locality on non-uniform memory access (NUMA) systems. Thus, NVRAM replication on a single compute node is vital to avoid data loss and is possibly augmented by synchronous or asynchronous disk or network replication [6, 7, 26] to further improve high availability in case of complete machine crash. To cope with DIMM failures, a commonly employed hardware-based approach for DRAM is *memory mirroring*, which is comparable to a RAID 1 as it partitions the local DRAM of a socket and issues all write operations to both memory partitions simultaneously. However, the drawbacks of applying the same technology to NVRAM are (1) that replication is limited to the local socket, (2) non-primary data that is also kept in NVRAM is unnecessarily replicated and occupies memory, and (3) the missing flexibility for the DBMS to decide on the appropriate replication strategy for a specific memory region. Hence, in this paper, we argue for a software-based approach and present compute node-local mechanisms to provide the foundation for an efficient NVRAM replication with a low latency and throughput penalty. Our considerations are based on the Intel *persistent memory development kit (PMDK)*[1] [18] that already provides NVRAM-centric data structures and a basic replication mechanism. Within our evaluation, we measured up to *10x* less overhead for our optimized replication mechanisms compared to the basic PMDK replication.

Contributions. Our contributions are summarized as follows:

(1) We show that the built-in basic PMDK NVRAM physical replication mechanism imposes prohibitively high costs – in terms of overhead per replica – by providing measurements for the common DBMS-employed data structures such as column stores and indexes (key-value stores) with and without transactional support (Section 3).
(2) We propose a rich set of building blocks for optimizations of the basic PMDK NVRAM replication mechanism ranging from instruction-level optimizations to thread-level optimizations and propose a template-based approach to automatically generate advanced replication mechanisms (Section 4).
(3) We provide a thorough evaluation of our optimizations and show that we were able to reduce the replication overhead up to an order of magnitude. Moreover, we show that the choice of the most efficient replication mechanism depends on multiple parameters (Section 5).

Paper Organization. The remainder of the paper is structured as follows. Section 2 provides the necessary NVRAM and Intel PMDK background. Subsequently, Section 3 describes the basic replication mechanism of the PMDK followed by Section 4, which discusses our advanced replication mechanisms. Section 5 provides an in-depth evaluation of proposed optimizations. Finally, Section 6 briefly summarizes the related work and Section 7 concludes the paper.

2 NVRAM & PMDK BACKGROUND

In this section, we cover the necessary NVRAM and PMDK background. We start with an overview of the PMDK software stack and its relation to a hybrid DRAM-NVRAM database system. Afterwards, we discuss important details like access methods for NVRAM and the programming model. Finally, we introduce the NVRAM-centric data structures employed for our experiments.

Hardware Setup. For all experiments in this paper, we use a dual socket system equipped with Intel Xeon Gold 6154 processors (Skylake-SP) and 384 GiB DDR4 memory. Each processor features 18 physical cores (36 w/ HyperThreading) and was pinned to 3 GHz core frequency. We run a Ubuntu 16.04.4 with kernel version 4.15 (w/o page table isolation) and binaries are compiled with gcc 5.4. Since NVRAM is not publicly available at this point in time and prototype hardware was not provided to us, we use DRAM emulation[2]. Nevertheless, our target metric is the relative replica overhead, which does not depend on absolute latencies.

2.1 PMDK-Integrated DBMS Architecture

The PMDK [18] is a set of open source libraries that support the development of NVRAM applications. In particular, it provides

[1] https://github.com/pmem/pmdk

[2] NVRAM emulation either uses `/dev/shm` or specific physical memory regions defined at boot time via the `memmap` kernel parameter.

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

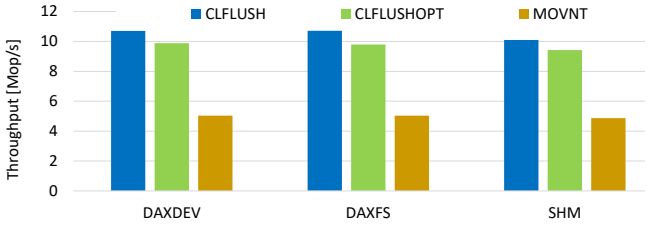


Figure 4: Column store update throughput w/o replication for different write back options and access methods.

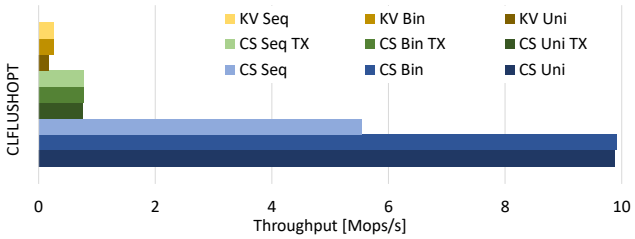


Figure 5: Update throughput w/o replication for different data structures and key distributions using `CLFLUSHOPT`.

`libpmem` that is an abstraction layer for platform-specific details such as NVRAM access methods and required instructions for NVRAM programming. `Libpmem` maps physcial NVRAM regions into the virtual address space of the database system that are called pools. The next level of abstraction is provided by `libpmemobj`, which primarily adds an allocator to a pool that is able to persistently allocate dynamically sized objects with position-independent virtual addresses. Moreover, `libpmemobj` supports *pool sets* that handle the replication (cf., Section 3) and additionally supports persistent transactions to atomically update multiple memory locations within a pool or pool set. As depicted in Figure 3, a DBMS can leverage the PMDK to implement its internal NVRAM-centric data structures that are used for the actual query processing.

2.2 Access Methods

To efficiently access physical NVRAM, current Linux kernels implement the *Direct Access (DAX)* feature that maps physical NVRAM regions into the virtual address space of an application while bypassing kernel page and block-level caches. The following three specific ways of accessing NVRAM (real or emulated) are currently available:

DAXDEV. A DAX device is similar to a raw disk access. The pages of the NVRAM region are contiguously mapped into the address space of the application without any intervening file system.

DAXFS. The DAXFS access method requires the NVRAM region to be formatted with a DAX-enabled file system (e.g., ext4) and individual files are mapped into the virtual address space of the application without intervening kernel page caches.

SHM. While the previous access methods require a specific physical memory region to be defined as NVRAM at boot time, the *shared memory (SHM)* access method uses arbitrary DRAM memory pages (via tmpfs file system) and is thus solely meant for NVRAM emulation.

Evaluation. As shown by Figure 4, the performance numbers of the access methods do not vary significantly in case of a challenging random column store update workload. Nevertheless, we will use the *DAXDEV* access method for all subsequent experiments, because it provides the highest and most stable performance.

2.3 Programming Model

Because of the *DAX* feature, virtual memory pages are directly mapped to physical NVRAM pages and thus, no system calls are required to persist modified data to the NVRAM. However, the presence of CPU caches requires explicit cache line write backs to ensure the persistency of a modification. Those write backs are achieved via the following instructions:

CLFLUSH. This instruction writes back a specific cache line and evicts it from the cache hierarchy in the coherency domain.

CLFLUSHOPT. This instruction works similarly to `CLFLUSH`, but is executed asynchronously. The advantage is that multiple cache lines can be flushed in parallel. However, this instruction requires an additional `SFENCE` or `MFENCE` instruction to ensure the completion of all previously issued `CLFLUSHOPT` instructions, and therefore prevent write reorderings.

CLWB. This instruction asynchronously writes back a specific cache line without explicit eviction. `CLWB` also requires an `SFENCE` or `MFENCE` to ensure its completion. The `CLWB` instruction is present on our Skylake-SP system. However, it currently seems to be mapped to a `CLFLUSHOPT` as indicated by the CPU documentation [8] and the vast amount of experiments we conducted.

WBINVD/WBNOINVD. Both instructions are only executable in kernel mode and write back all modified cache lines either with or without eviction. The `WBNOINVD` instruction will be available on the upcoming icelake server CPU generation.

MOVNT. The non-temporal `MOV` instruction bypasses the caches and stores data directly to the memory. `MOVNT` is a SIMD extension (SSE2, AVX, AVX2 or AVX512) that is also executed out-of-order and thus requires an `SFENCE` or `MFENCE` to ensure its completion.

Evaluation. In Figure 4, we compared the performance of write back instructions that are available to user mode applications on our system using a random column store (1 GiB size) update workload without replication. As shown, `CLFLUSH` and `CLFLUSHOPT` performance numbers are close to each other with a slight advantage for `CLFLUSH`, because it does not require an additional `SFENCE`. The performance of the *MOVNT* is significantly lower.

2.4 NVRAM-Centric Data Structures

For our experiments, we consider two NVRAM-centric data structures that are fundamental for database systems:

Column Store (CS) The column store is an array of 4 Byte integers that is allocated via the `libpmemobj` API. We consider a non-transactional (CS) and a transactional workload (CS TX) for our experiments. The transactional workload also updates a single integer, but uses the transaction feature of the `libpmemobj`, which causes additional write overhead for the undo log.

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

Algorithm 1 Basic PMDK pool replication algorithm

```
1: Flush(MasterAddress, Size);
2: SFence();                                   ▷ not for CLFLUSH
3: for each Replica do                         ▷ actual replication
4:     Compute(ReplicaAddress);
5:     MemCpy(ReplicaAddress, MasterAddress, Size);
6:     Flush(ReplicaAddress, Size);            ▷ not for MOVNT
7:     SFence();                               ▷ not for CLFLUSH
```

Key-Value Store (KV) Key-value stores are employed as indexes within a DBMS. We use the pmemkv implementation[3], which is a hybrid DRAM-NVRAM data structure similar to the FPTree [14]. The leaf nodes of the tree are stored in NVRAM, while the recoverable inner nodes are stored in DRAM. Pmemkv uses the transactional facilities of `libpmemobj`, to ensure atomic updates to the persistent leaf nodes of the tree structure.

Evaluation. All experiments are executed single-threaded on the same socket as the NVRAM regions are physically allocated on, except it is explicitly stated differently. Figure 5 shows the throughput for updates on the data structures (w/o replication) for a uniform (Uni), binomial (Bin), and sequential (Seq) key distribution using the `CLFLUSHOPT` instruction. The non-transactional CS workload gives the best performance, because only a single cache line needs to be flushed. In contrast, the CS TX workload uses transactions, which causes additional cache line flushes as it is reflected by the throughput numbers. We observed the worst performance for the KV workload that faces the transactional overhead as well as additional costs for updating to the inner tree nodes in DRAM and may issues modifications to multiple NVRAM cache lines, when modifying the leaf nodes. We also observe that all three workloads are relatively insensitive to the key distribution, which is a result of the cache line eviction issued by the `CLFLUSHOPT` instruction. This especially hurts the performance of the sequential CS workload that constantly evicts the cache line it needs for the next update. Due to the adjacent cache line prefetch feature, the other key distributions have a chance that the cache line to be updated was already fetched during a previous operation. Nevertheless, this behavior is likely to change with the full implementation of the `CLWB` instruction.

3 BASIC POOL REPLICATION

In this section, we discuss the basic replication mechanism of the PMDK and present overhead measurements for our three workloads. As depicted in Figure 3, the `libpmemobj` already provides a replication feature that replicates writes to a pool across a set of pools (pool replication). The pool replication also works in combination with the transactional feature where transaction tables and the undo log are additionally replicated across the pool set.

Algorithm 1 lists the pseudocode for the compute node-local parts of the replication mechanism. The algorithm starts with flushing the modified cache lines of the master replica (line 1) followed by an `SFENCE` to ensure the completion of the flush operations (line 2). Subsequently, the actual replication is performed by looping over the individual replicas (lines 3-7). For each replica, the algorithm computes the memory address of the modification within the replicated pool (line 4); copies the data from the master replica to the calculated replica address either using a `memcpy` or a `MOVNT` (line 5); flushes the modified cache lines of the replica in case of a `memcpy` (line 6) and finally, issues an `SFENCE` to ensure the completion of the flushes respectively the `MOVNT` (line 7).

[3]https://github.com/pmem/pmemkv

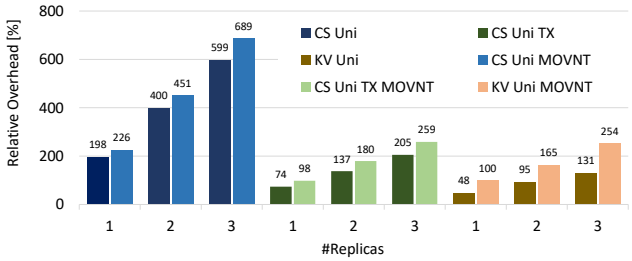


Figure 6: Relative update overhead w/ replication for different workloads and replica counts. Uniform key distribution.

Evaluation. Figure 6 shows the overhead measurements relative to the non-replicated throughput for our three workloads (cf., Section 2.4) using a uniform key distribution. The overheads are given for the temporal copy (`memcpy`) where replica modifications are flushed using the `CLFLUSHOPT` and the non-temporal copy (`MOVNT`). In both cases, master replica modifications are flushed using the `CLFLUSHOPT`. Each pool of the set has a size of 1 GiB. The measurements reveal that the non-transactional column store workload (CS) faces about 200 % additional overhead per replica, which is located on the same socket. Note that the updates are executed non-dependent on each other by the benchmark, which is a more realistic setting for writes. Hence, the large pipelines, out-of-order execution, and speculation efforts of modern processors hide the latency for fetching the cache line of the master replica, which speeds up the baseline and thus amplifies the replication overhead. In case of dependent updates, the overhead is about 100 % per replica as it would be expected. The other two workloads (i.e., CS TX and KV) face a replication penalty of about 250 % for three replicas. Such overhead numbers are prohibitively high, especially in case of the CS workload, and annihilate the advantage of using NVRAM as a storage for primary data. Hence, we will reason about the origin of this replication penalty and propose optimizations for the basic replication mechanism in the following section.

4 ADVANCED POOL REPLICATION

In this section, we discuss the issues of the basic pool replication mechanism that cause the high overheads during the NVRAM replica maintenance. Afterwards, we present our advanced pool replication mechanisms that are generated using a template-based approach for automatically composing the individual building blocks to a full replication mechanism. The basic pool replication mechanism faces the following issues:

(1) The algorithm is over-synchronized, because cache line flushes are synchronized via an `SFENCE` instruction after each replica.
(2) The master replica is updated first, which evicts the modified data from the cache when using the `CLFLUSH(OPT)` or `MOVNT`

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

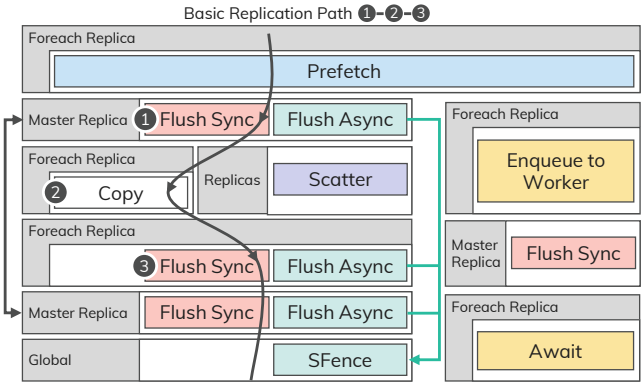


Figure 7: Building blocks for advanced pool replication and composition options including the basic replication path.

instruction. Hence, the subsequent copy operation that updates the first replica needs to refetch the data from the memory first.

(3) The `memcpy` operation that copies the modified data from the primary replica to the other replicas effectively performs a read-modify-write operation. Thus, the cache lines of the replicas to be updated need to be present in cache, which is not considered by the algorithm. `MOVNT` copies do not require the data to be in the cache, but are executed faster if the virtual-to-physical address mapping of the destination is already cached in the translation lookaside buffer (TLB).

(4) Copy operations from the modified primary replica to the other replicas are not parallelized and execute sequentially.

To overcome those issues, we present our template-based approach for generating advanced replication methods in Figure 7. The template describes how the individual building blocks are arrangeable to guarantee persistent updates to all replicas of the pool. Generating starts from the top of the scheme and proceeds downwards by configuring the desired option for a particular building block (depicted in grey colour). In accordance to this approach, the basic replication algorithm can be generated following a path indicated as 1-2-3 in Figure 7, while, for example, multiple threads optimization is enabled by the right most path. In the following, we describe the building blocks in detail and highlight how they cope with the issues of the basic replication mechanism.

Flush Sync/Async (SF). To resolve issue (1), we add the *Flush Async* building block that omits the `SFENCE` as an alternative to *Flush Sync* (Flush followed by an `SFENCE`). Nevertheless, to ensure completion of the NVRAM writes a single `SFENCE` (SF) is required at the end (this flow is depicted by the green arrow). Doing so, *Flush Async* is able to execute `CLFLUSHOPT` and `MOVNT` instructions across replicas in parallel.

Master Replica Flush Reorder. To address issue (2), the template for our advanced pool replication mechanism allows to execute the master replica flush before (PRE) or after (POST) the other replicas are written and flushed (reflected by the left most arrow).

Prefetch (PF). To cope with issue (3), we add the *Prefetch* building block that preloads the first cache line of the memory location for a replica. Consecutive cache lines that may need to be updated are usually automatically loaded by the hardware prefetchers.

Scatter (SCA). To address issue (4) in case of small updates, we use the AVX512 *Scatter* instruction[4] that leverages data-level parallelism to update multiple replicas with a single instruction.

Threads. To address issue (4) in case of large updates, we employ thread-level parallelism to additionally leverage the load-store units (LSU) of other cores. Hence, for each slave replica a worker thread is instantiated during the opening procedure of a pool set that executes the *Copy* and *Flush Sync* building blocks for its replica, if instructed so by the master replica thread.

Note that the *Copy* and *Flush Sync/Async* building blocks can additionally be switched to a different implementation, i.e., `CLFLUSHOPT`, `MOVNT`, etc. With the help of our template-based approach, we are able to evaluate all combinations of allowed and meaningful optimizations for several workloads and workload parameters. Moreover, the template can be leveraged to automatically switch to the optimal replication mechanism for the current conditions, which is out of the scope for this paper and is left for future work.

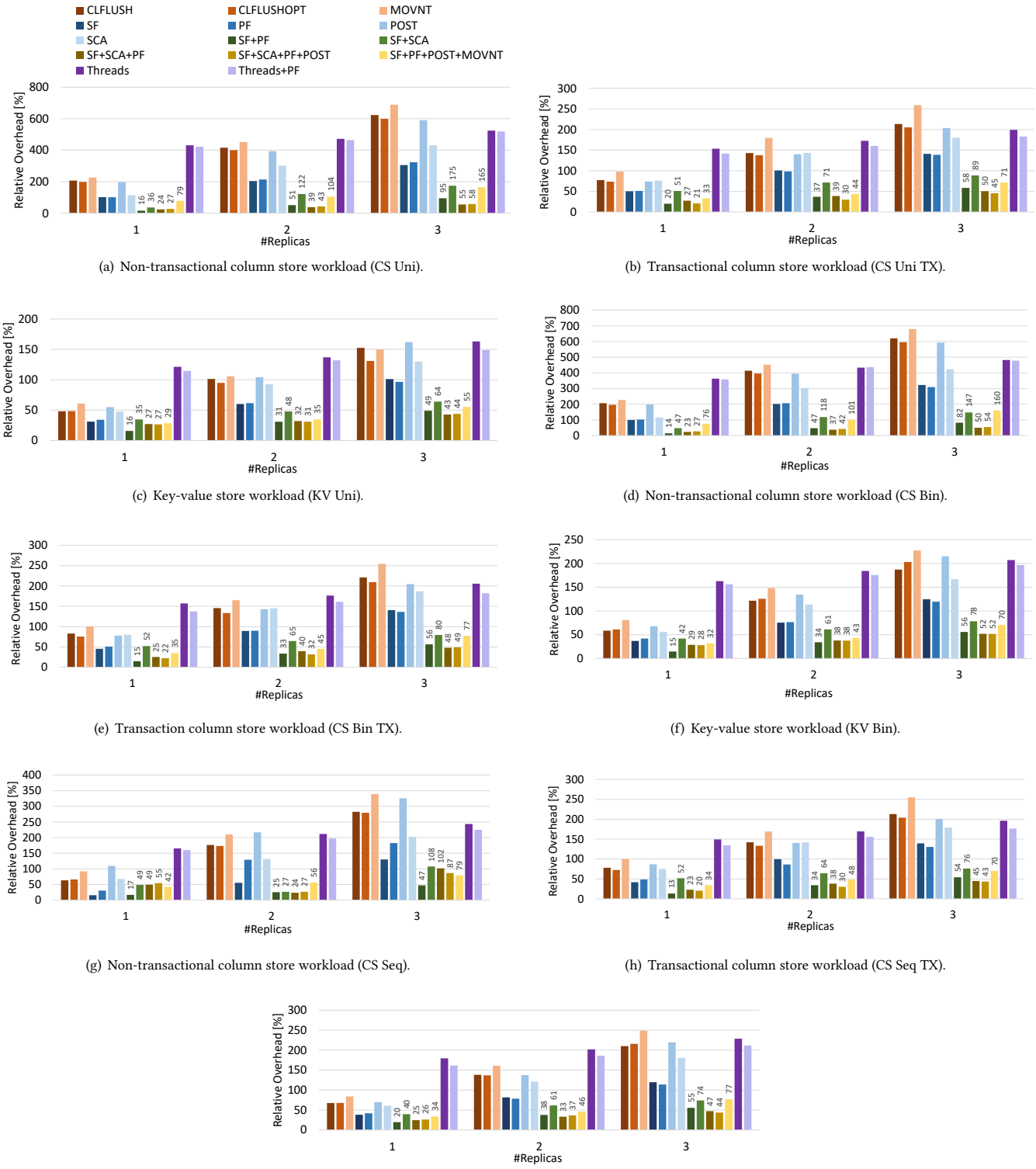
5 EVALUATION

In this section, we evaluate our advanced replication mechanisms and compare them to the basic replication mechanism of the PMDK using our three workloads and key distributions. Moreover, we investigate the influence of different pool sizes, the number of consecutively updated column store elements (chunks), high replica counts, and the socket placement of individual replicas.

Impact of Optimizations. Figure 8 shows the overhead numbers of a selected set of activated optimizations including the baseline for the nine workloads using the uniform, binomial and sequential key distributions and up to three replicas (each pool 1 GiB). The measurements show that enabling a single optimization already halves the overhead for uniform and binomial non-transactional workloads except for the `POST` optimization that postpones the master replica flush. Activating combinations of optimizations further decreases the replication overhead. For one replica, the `SF+PF` combination gives the best results. For more replicas, the `SF+SCA+PF` combination performs best for the CS workload and additionally enabling `POST` further improves the overhead of the transactional workloads. We observe that replication behavior differs for sequential key distribution due to cache line eviction issued by the `CLFLUSHOPT` instruction. The same cache line needs to be refetched several times to update its sequential keys. Nevertheless, combinations `SF+PF` and `SF+PF+SCA+POST` give still the best improvement, while enabling just one optimization does not allow to halve the respective overhead. Due to the adjacent cache line prefetch feature, uniform and binomial key distributions have a chance that the cache line to be updated was already fetched during a previous operation. As we argued in Section 2, this behavior is likely to change with the full implementation of the `CLWB` instruction. The `Thread` optimization has a high initial overhead, because of the thread synchronization costs that start to amortize with an increasing number of replicas. We measured overhead savings of *91 %* for the CS workload, *78 %* for the CS TX workload, and *67 %* for the KV workload. Moreover, our optimizations reduced the extra overhead costs for adding an additional replica to the pool set.

[4] `_mm512_mask_i64scatter_epi32` or `_mm512_mask_i64scatter_epi64` intrinsic

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931



(a) Non-transactional column store workload (CS Uni).

(b) Transactional column store workload (CS Uni TX).

(c) Key-value store workload (KV Uni).

(d) Non-transactional column store workload (CS Bin).

(e) Transaction column store workload (CS Bin TX).

(f) Key-value store workload (KV Bin).

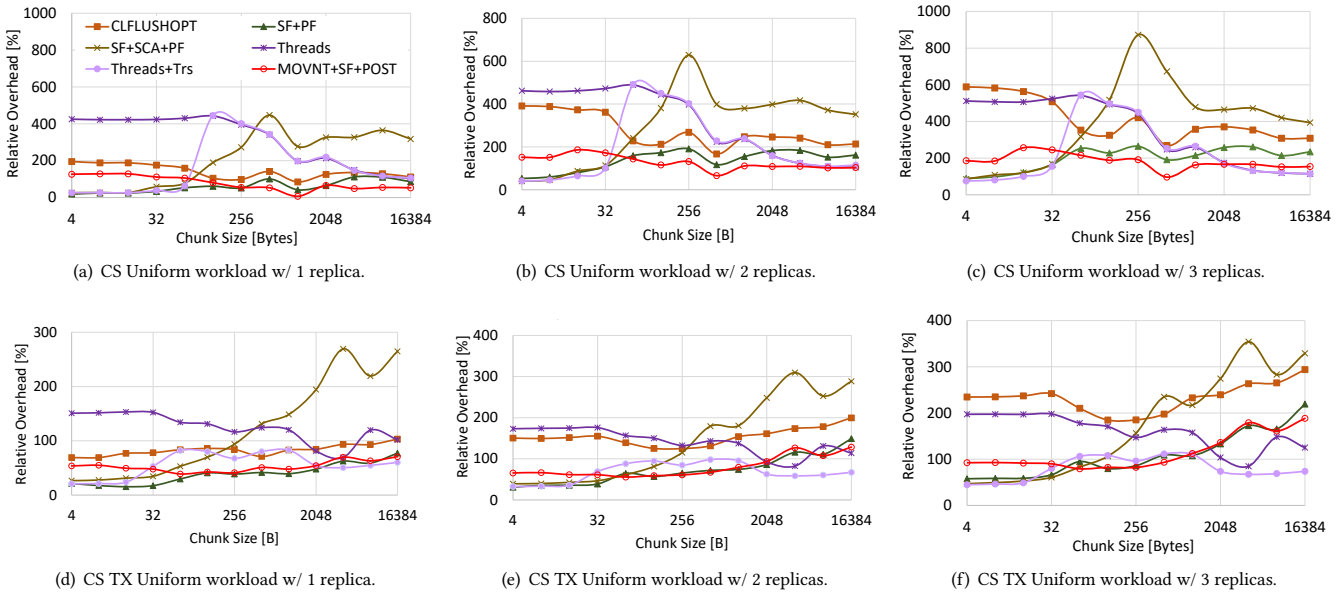
(g) Non-transactional column store workload (CS Seq).

(h) Transactional column store workload (CS Seq TX).

(i) Key-value store workload (KV Seq).

Figure 8: Relative overheads for different workloads, optimizations, replica counts and key distribution.

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931



(a) CS Uniform workload w/ 1 replica. (b) CS Uniform workload w/ 2 replicas. (c) CS Uniform workload w/ 3 replicas.

(d) CS TX Uniform workload w/ 1 replica. (e) CS TX Uniform workload w/ 2 replicas. (f) CS TX Uniform workload w/ 3 replicas.

Figure 9: Relative overheads for different workloads, optimizations, updated chunk sizes, and replica counts.

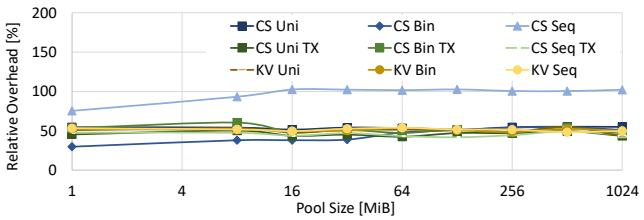


Figure 10: Relative overheads for different workloads, key distributions, and pool sizes. 3 replicas (SF+SCA+PF).

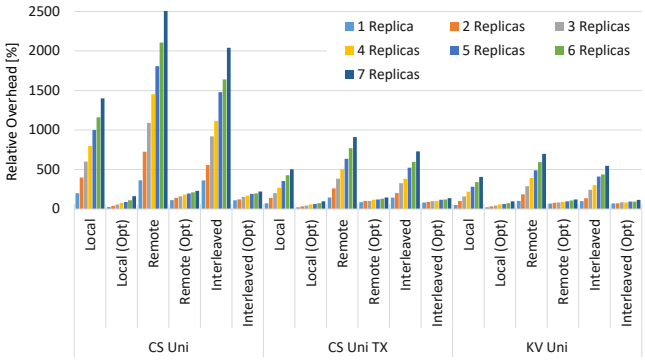


Figure 11: Relative overheads for different workloads, key distributions, and NVRAM socket allocations. `CLFLUSHOPT` and SF+SCA+PF (Opt) replication mechanisms.

Updated Chunk Size. In these series of experiments, we vary the size of the memory chunk that is updated by writing a varying number of consecutively located column store elements per update operation. Figure 9 visualizes the measured relative overheads for the CS and CS TX workload using different optimization combinations depending on the updated chunk size. The results show that the best replication mechanism depends on the updated chunk size as well as on the workload and replica count. For instance Figures 9(c) and 9(f) shows (1) that the `SCA` building block generates more overhead for larger chunk sizes, (2) that the `MOVNT` implementation of the `Copy` building block starts to become feasible at a chunk size of 64 Bytes, and (3) that the `Threads` building block gives the lowest overhead starting at a chunk size of 2 KiB and becomes feasible with replica count starting from 3. Since the `Threads` building block is designed for large chunks, we added a `Threshold` building block (`Threads+Trs`) that falls back on the alternative `SF+SCA+PF` path for small chunk sizes ($\leq$ 32 Bytes as visible in Figure 9(a)) as they occur in the transactional part of the CS TX workload. As seen from the Figures 9(d), 9(e) and 9(f) such combined approach gives the most stable chunk size independent improvement for transactional workloads.

Pool Size and Key Distribution. Figure 10 illustrates the relative overhead measurements for the `SF+SCA+PF` building block combination depending on the size of a single pool of the pool set for all workload and key distribution combinations. The results show that the overhead measurements are insensitive to the size of the pool and that all considered workloads face a similar overhead, except for the sequential CS workload (cf., Section 2.4). This insensitivity is a result of the `Prefetch (PF)` optimization that preloads all required cache lines before data is copied from the master replica to the slave replicas.

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

Replica Socket Placement. In Figure 11, we placed the emulated NVRAM either on the local socket, remote socket, or interleaved the allocation – first replica remote, second replica local, etc. – and run all workloads with the uniform key distribution with up to 7 replicas. The measurements reveal that placing replicas on the remote socket increases the replication overhead by about 75 %. Employing the `SF+SCA+PF` optimization combination (Opt) also faces an initial 75 % overhead penalty as soon as a single replica is located on the remote socket. However, it still dramatically decreases the absolute overhead and the additional overhead costs per replica.

Relation to Real NVRAM Hardware. It is important to mention that our hardware currently does not fully support the *CLWB* instruction and is not able to emulate the asymmetric read/write latency that is expected for real NVRAM. Since any attempt to transfer our measurements to upcoming NVRAM hardware is only speculation, we are looking forward to run our evaluation on true NVRAM hardware. Nevertheless, we are convinced that our template-based advanced replication mechanisms will still significantly outperform the basic built-in replication mechanism of the PMDK.

6 RELATED WORK

The PMDK [18] is the de facto standard for NVRAM programming and is a melting pot for NVRAM-related technologies that are agilely researched. In particular, persistent allocators [1, 3, 11, 13, 20, 25] and NVRAM-centric data structures [4, 5, 14, 15, 21, 22, 24] have been thoroughly researched. For instance, the FPTree [14] is a hybrid DRAM-NVRAM data structure that is implemented in a modified version[5] by the `pmemkv` library we leveraged for our experiments. The PMDK additionally implements a basic mechanism for compute node-local and RDMA-based replication across compute nodes [9]. Moreover, RDMA-based replication mechanisms for NVRAM have been addressed in the context of operating systems [26] and virtual machines [19]. The logical replication has been NVRAM-optimized in ERMIA DBMS [23]. To the best of our knowledge, efficient replication mechanisms within a single compute node have not been further researched so far. Another related research topic are hybrid DRAM-NVRAM database systems like SOFORT [12, 16, 17], FOEDUS [10], or Peloton [2] that store the primary data in NVRAM and provide instant recovery. Such DBMSs are able to leverage our advanced replication mechanisms to improve high availability.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we addressed the topic of efficient compute node-local replication of NVRAM regions that is essential to prevent data losses and ensure high availability on hybrid DRAM-NVRAM database systems in case of NVDIMM failures. We proposed a template-based approach for automatically assembling optimized building blocks to physical pool replication mechanisms. Within our evaluation, we experienced replication overhead savings of up to *10x* for replicas placed on the local as well as remote socket of the compute node. Promising directions for future work are the autonomous replication mechanism selection and the actual replica allocation strategy for hybrid DRAM-NVRAM DBMSs.

[5]https://github.com/pmem/pmemkv/blob/master/ENGINES.md

8 ACKNOWLEDGMENT

This work is partly funded (i) by the German Research Foundation (DFG) in the context of the project "Self-Recoverable and Highly Available Data Structures for NVRAM-centric Database Systems" (LE-1416/27-1), (ii) by DFG-CRC 912 (HAEC) and (iii) by the Cluster of Excellence "Center for Advancing Electronics Dresden" (Orchestration Path).

REFERENCES

[1] Alfons Kemper Thomas Neumann Takushi Hashida Kazuichi Oe Yoshiyasu Doi Lilian Harada Sato Mitsuru Alexander van Renen, Viktor Leis. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 SIGMOD International Conference on Management of Data, June, 2018, Houston*. 691–706. https://doi.org/10.1145/2723372.2746480

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016), 337–348. http://www.vldb.org/pvldb/vol10/p337-arulraj.pdf

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 677–694. https://doi.org/10.1145/2983990.2984019

[4] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 21–31. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper3.pdf

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797. http://www.vldb.org/pvldb/vol8/p786-chen.pdf

[6] Rohit Dhamane, Marta Patiño-Martínez, Valerio Vianello, and Ricardo Jiménez-Peris. 2014. Performance Evaluation of Database Replication Systems. In *18th International Database Engineering & Applications Symposium, IDEAS 2014, Porto, Portugal, July 7-9, 2014*. 288–293. https://doi.org/10.1145/2628194.2628214

[7] Jinwei Guo, Chendong Zhang, Peng Cai, Minqi Zhou, and Aoying Zhou. 2016. Low Overhead Log Replication for Main Memory Database System. In *Web-Age Information Management - 17th International Conference, WAIM 2016, Nanchang, China, June 3-5, 2016, Proceedings, Part II*. 159–170. https://doi.org/10.1007/978-3-319-39958-4_13

[8] Intel. 2018. Intel Instruction Reference Manual (Vol 2A, 3-147). (2018).

[9] Tomasz Kapela. 2015. An introduction to replication. (2015). http://pmem.io/2015/11/23/replication-intro.html.

[10] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 691–706. https://doi.org/10.1145/2723372.2746480

[11] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*. ACM, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/2524211.2524216

[12] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*. 8:1–8:7. https://doi.org/10.1145/2619228.2619236

[13] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *PVLDB* 10, 11 (2017), 1166–1177. http://www.vldb.org/pvldb/vol10/p1166-oukid.pdf

[14] 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 (SIGMOD '16)*. ACM, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[15] Ismail Oukid and Wolfgang Lehner. 2017. Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1759–1764. https://doi.org/10.1145/3035918.3054777

[16] Ismail Oukid and Wolfgang Lehner. 2017. Towards a Single-Level Database Architecture on Non-Volatile Memory. In *8th Annual Non-Volatile Memories Workshop 2017 (NVMW)*.

[17] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA,*

Final edited form was published in "SIGMOD/PODS '18: International Conference on Management of Data. Houston 2018", Art. Nr. 7, ISBN 978-1-4503-5853-8
https://doi.org/10.1145/3211922.3211931

USA, January 4-7, 2015, Online Proceedings. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper13.pdf

[18] Andy Rudoff. 2015. Persistent Memory Programming. *Login: The Usenix Magazine* 42 (2015), 34–40.

[19] Vasily A. Sartakov and Rüdiger Kapitza. 2017. Multi-site Synchronous VM Replication for Persistent Systems with Asymmetric Read/Write Latencies. In *22nd IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2017, Christchurch, New Zealand, January 22-25, 2017*. 195–204. https://doi.org/10.1109/PRDC.2017.33

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

[21] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[22] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *PVLDB* 7, 5 (2014), 413–424. http://www.vldb.org/pvldb/vol7/p413-viglas.pdf

[23] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 406–419. https://doi.org/10.1145/3164135.3164137

[24] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. 2016. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Computers* 65, 7 (2016), 2169–2183. https://doi.org/10.1109/TC.2015.2479621

[25] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. 2015. WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *34th IEEE International Performance Computing and Communications Conference, IPCCC 2015, Nanjing, China, December 14-16, 2015*. 1–8. https://doi.org/10.1109/PCCC.2015.7410326

[26] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 3–18. https://doi.org/10.1145/2786763.2694370