Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory(Preview Version)

Xinjing Zhou
mortyzhou@tencent.com
Tencent Inc.

Joy Arulraj
arulraj@gatech.edu
Georgia Institute of Technology

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University

David Cohen
david.e.cohen@intel.com
Intel

Abstract

The design of the buffer manager in database management systems (DBMSs) is influenced by the performance characteristics of volatile memory (*i.e.*, DRAM) and non-volatile storage (*e.g.*, SSD). The key design assumptions have been that the data must be migrated to DRAM for the DBMS to operate on it and that storage is orders of magnitude slower than DRAM. But the arrival of new non-volatile memory (NVM) technologies that are nearly as fast as DRAM invalidates these previous assumptions.

Researchers have recently designed Hymem, a novel buffer manager for a three-tier storage hierarchy comprising of DRAM, NVM, and SSD. Hymem supports cache-line-grained loading and an NVM-aware data migration policy. While these optimizations improve its throughput, Hymem suffers from two limitations. First, it is a single-threaded buffer manager. Second, it is evaluated on an NVM emulation platform. These limitations constrain the utility of the insights obtained using Hymem.

In this paper, we present Spitfire, a multi-threaded, three-tier buffer manager that is evaluated on Optane Persistent Memory Modules, an NVM technology that is now being shipped by Intel. We introduce a general framework for reasoning about data migration in a multi-tier storage hierarchy. We illustrate the limitations of the optimizations used in Hymem on Optane and then discuss how Spitfire circumvents them. We demonstrate that the data migration policy has to be tailored based on the characteristics of the devices and the workload. Given this, we present a machine learning technique for automatically adapting the policy for an arbitrary workload and storage hierarchy. Our experiments show that Spitfire works well across different workloads and storage hierarchies. We present a set of guidelines for picking the storage hierarchy and migration policy based on the workload.

1 Introduction

The techniques for buffer management in a canonical DRAM-SSD hierarchy are predicated on the assumptions that: (1) the data must be copied from SSD to DRAM for the DBMS to operate on it, and that (2) storage is orders of magnitude slower than DRAM [4, 16]. But emerging non-volatile memory (NVM) technologies upend these design assumptions. They support low latency reads and writes similar to DRAM, but with persistent writes and large storage capacity like an SSD. In a DRAM-SSD hierarchy, the buffer manager decides *what* pages to move between disk and memory and *when* to move them. However, with a DRAM-NVM-SSD hierarchy, in addition to these decisions, it must also decide *where* to move them (*i.e.*, which storage tier).

Prior Work. Recently, researchers have designed Hymem, a novel three-tier buffer manager for a DRAM-NVM-SSD hierarchy [37]. Hymem employs a set of optimizations tailored for NVM. It adopts a *data migration policy* consisting of four decisions: DRAM admission, DRAM eviction, NVM admission, and NVM eviction. ❶ Initially, a newly-allocated 16 KB page resides on SSD. When a transaction requests that page, Hymem *eagerly* admits the entire page to DRAM. ❷ DRAM eviction is the next decision that it takes to reclaim space. It uses the CLOCK algorithm for picking the victim page [34]. ❸ Next, it must decide whether that page must be admitted to the NVM buffer (if it is not already present in that buffer). Hymem seeks to identify *warm* pages. It maintains a queue of recently considered pages to make the NVM admission decision. It admits pages that were recently denied admission. Each time a page is considered for admission, Hymem checks whether the page is in the admission queue. If so, it is removed from the queue and admitted into the NVM buffer. Otherwise, it is added to the queue and directly moved to SSD, thereby bypassing the NVM buffer. ❹ Lastly, it uses the CLOCK algorithm for evicting a page from the NVM buffer.

Hymem supports cache-line-grained loading to improve the utilization of NVM bandwidth. Unlike SSD, NVM supports low-latency access to 256 B blocks. Hymem uses cache line-grained loading to extract only the hot data objects from an otherwise cold 16 KB page. By only loading those cache lines that are needed, Hymem lowers its bandwidth consumption.

Hymem supports a *mini page* layout for reducing the DRAM footprint of cache-line-grained pages. This layout allows it to efficiently keep track of which cachelines are loaded. When the mini page overflows (*i.e.*, all sixteen cache lines are loaded), Hymem transparently promotes it to a *full page*.

Limitations. These optimizations enable Hymem to work well across different workloads on a DRAM-NVM-SSD storage hierarchy. However, it suffers from two limitations. First, it is a single-threaded buffer manager. Second, it is evaluated on an NVM emulation platform. These limitations constrain the utility of the insights obtained using Hymem (§6.5). In particular, the data migration policy employed in Hymem is not the optimal one for certain workloads. We show that the cache line-grained loading and mini-page optimizations must be tailored for a real NVM device. We also illustrate that the choice of the data migration policy is significantly more important than these auxiliary optimizations.

	DRAM	NVM	SSD
Latency			
Idle Sequential Read Latency	75 ns	170 ns	10 μs
Idle Random Read Latency	80 ns	320 ns	12 μs
Bandwidth (6 DRAM and 6 NVM modules)			
Sequential Read	180 GB/s	91.2 GB/s	2.6 GB/s
Random Read	180 GB/s	28.8 GB/s	2.4 GB/s
Sequential Write	180 GB/s	27.6 GB/s	2.4 GB/s
Random Write	180 GB/s	6 GB/s	2.3 GB/s
Other Key Attributes			
Price ($/GB)	10	4.5	2.8
Addressability	Byte	Byte	Block
Media Access Granularity	64 B	256 B	16 KB
Persistent	No	Yes	Yes
Endurance (cycles)	10^{16}	10^{10}	10^{12}

Table 1: Device Characteristics – Comparison of key characteristics of DRAM, NVM (Optane DC PMMs), and SSD (Optane DC P4800X).

Our Approach. In this paper, we present Spitfire, a multi-threaded, three-tier buffer manager that is evaluated on Optane Data Center (DC) Persistent Memory Modules (PMMs), an NVM technology that is being shipped by Intel [7]. As summarized in Table 1, Optane DC PMMs bridges the performance and cost differentials between DRAM and an enterprise-grade SSD [26, 32]. Unlike SSD, it delivers higher bandwidth and lower latency to CPUs. Unlike DRAM, it supports persistent writes and large storage capacity.

We begin by introducing a taxonomy for data migration policies that subsumes the specific policies employed in previous three-tier buffer managers [22, 37]. Since the CPU is capable of directly operating on NVM-resident data, Spitfire does not eagerly move data from NVM to DRAM. We show that *lazy* data migration from NVM to DRAM ensures that only *hot* data is promoted to DRAM. An optimal migration policy maximizes the utility of movement of data between different devices in the storage hierarchy. We empirically demonstrate that the policy must be tailored based on the characteristics of the devices and the workload. Given this, we present a machine learning technique for automatically adapting the policy for an arbitrary workload and storage hierarchy. This *adaptive data migration* scheme eliminates the need for manual tuning of the policy.

Contributions. We make the following contributions:

- We introduce a taxonomy for reasoning about data migration in a multi-tier storage hierarchy (§3).
- We introduce an adaptation mechanism that converges to a near-optimal policy for an arbitrary workload and storage hierarchy without requiring any manual tuning (§4).
- We implement our techniques in Spitfire, a multi-threaded, three-tier buffer manager (§5).
- We evaluate Spitfire on Optane DC PMMs and demonstrate that it works well across diverse workloads and storage hierarchies (§6).
- We present a set of guidelines for choosing the storage hierarchy and migration policy based on workload (§6.6, §6.7).
- We demonstrate that an NVM-SSD hierarchy works well on write-intensive workloads (§6.6).

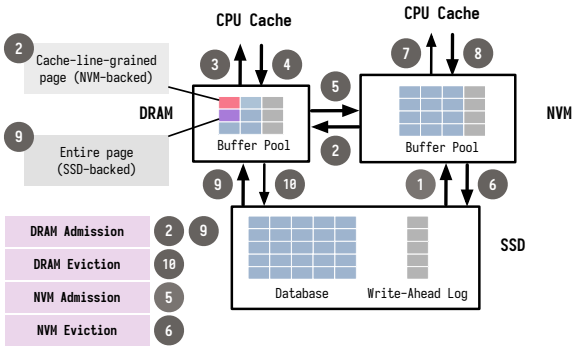


Figure 1: Data Migration Policy in Hymem – The four critical decisions in the data migration policy adopted by Hymem.

2 Background

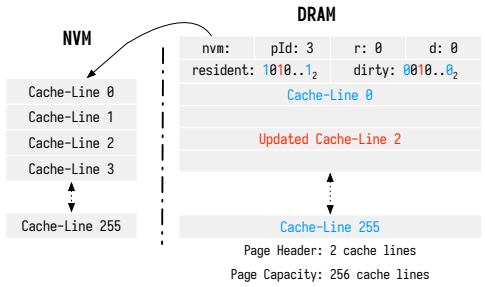
We first provide an overview of the NVM-aware optimizations adopted by Hymem in §2.1. We then discuss how Spitfire makes use of Optane DC PMMs in §2.2.

2.1 NVM-aware Optimizations in Hymem

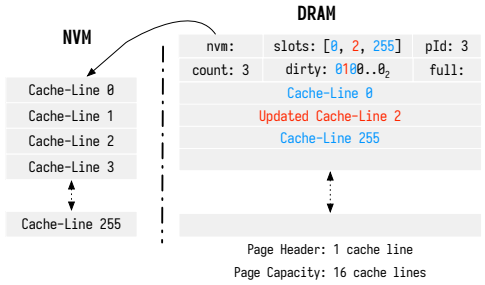
The traditional approaches for buffer management in DBMSs are incompatible with the advent of byte-addressable NVM. The reasons for this are twofold. First, to process SSD-resident data, the buffer manager must copy it to DRAM before the DBMS can perform any operations. In contrast, the CPU can directly operate on NVM-resident data (256 B blocks) [7]. Second, NVM bridges the performance gap between DRAM and SSD. Given these observations, researchers have designed Hymem, a novel buffer manager for a DRAM-NVM-SSD hierarchy [37]. The key NVM-centric optimizations in Hymem include:

Data Migration Policy. Figure 1 presents the data flow paths in the multi-tier storage hierarchy. NVM introduces new data flow paths in the storage hierarchy (❶, ❷, ❺, ❻, ❼,❽). By leveraging these additional paths, Hymem reduces data movement between different tiers and minimizes the number of writes to NVM. The former results in improving the DBMS's performance, while the latter extends the lifetime of NVM devices with limited write-endurance. The default read path comprises of three steps: moving data from SSD to NVM (❶), then to DRAM (❷), and lastly to the CPU cache (❸). Similarly, the default write path consists of three steps: moving data from processor cache to DRAM (❹), then to NVM (❺), and finally to SSD (❻).

As we discussed in §1, the four critical decisions in the data migration policy adopted by Hymem include: DRAM admission, DRAM eviction, NVM admission, and NVM eviction. When a transaction requests a page, Hymem checks if the page is in the DRAM buffer. If it is not present in the DRAM buffer, it checks if the page is present in the NVM buffer or not. In the former case, it directly accesses the page residing on NVM (❷). In the latter case, it eagerly admits the entire page residing on SSD into the DRAM buffer (❾). It does not leverage the path from SSD to NVM (❶). DRAM eviction is the second decision that Hymem takes to reclaim space. It must decide whether that page must be admitted to the NVM buffer. Each time a page is considered for admission, Hymem checks whether the page is in the admission queue. If so, it is removed from the



(a) Cache-line-grained Page



(b) Mini Page

Figure 2: Page Layouts in Hymem– The page layouts that Hymem uses to lower the NVM bandwidth consumption and the DRAM footprint, respectively [37].

queue and admitted into the NVM buffer (❺). Otherwise, the page's identifier is added to the queue, and it is directly moved from DRAM to SSD (❿). Lastly, it evicts pages from the NVM buffer when it needs to reclaim space (❻).

Cache-line-grained Loading. Hymem supports cache line-grained loading to extract only the hot data objects from an otherwise cold 16 KB page residing on NVM (❷). This lowers the NVM bandwidth consumption by only loading those 64 B cache lines that are needed. As shown in Figure 2a, to support cache-line-grained pages, Hymem maintains a couple of bitmasks (labeled `resident` and `dirty`) to keep track of the cache lines that are already loaded and those that are dirtied. The `r` and `d` bits indicate whether the entire page is resident and dirty, respectively. In this example, the first, third, and last cache-lines are loaded, as indicated by the corresponding set bits in `resident`. To support on-demand loading of cache-lines, each cache line-grained page in DRAM contains a pointer to the underlying NVM page. The header fits within two cache lines.

Mini Page. Hymem supports a mini page layout for reducing the DRAM footprint of cache-line-grained pages. This layout allows it to efficiently keep track of the loaded cachelines. A mini page stores up to sixteen cache-lines. As shown in Figure 2b, it contains a `slots` array for directing the accesses. Each slot stores the logical identifier of the cache-line residing in the mini page. For instance, the 255^{th} cache line in the underlying NVM page is loaded into the second slot of the DRAM-resident mini page. Hymem uses the `count` field to keep track of the number of occupied slots (*i.e.*, loaded cache lines). When the mini page overflows, Hymem transparently promotes it to a *full page*. The `dirty` bit mask determines the cache

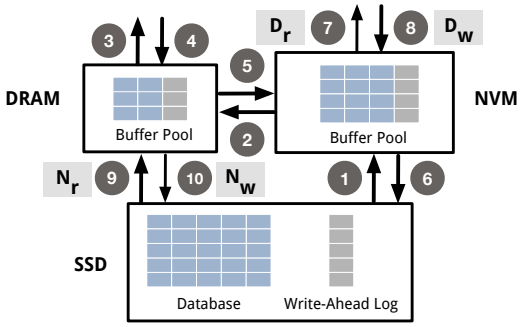


Figure 3: Data Flow Paths – The different data flow paths in a multi-tier storage hierarchy consisting of DRAM, NVM, and SSD.

lines that must be written back to NVM when the page is evicted from the DRAM buffer. In this example, the second cache line must be written back. The header of a mini page fits within a cache line.

2.2 Leveraging Optane DC PMMs

Optane DC PMMs can be configured in one of these two modes: (1) `memory` mode and (2) `app direct` mode. In the former mode, the DRAM DIMMs serve as a hardware-managed cache (*i.e.*, direct mapped write-back L4 cache) for frequently-accessed data residing on slower PMMs. The `memory` mode enables legacy software to leverage PMMs as a high-capacity volatile main memory device without extensive modifications. However, it does not allow the DBMS to utilize the non-volatility property of PMMs. In the latter mode, the PMMs are directly exposed to the processor and the DBMS directly manages both DRAM and NVM. In this paper, we configure the PMMs in `app direct` mode to ensure the durability of NVM-resident data.

Evaluation Platform. We evaluate Spitfire on a platform containing Optane DC PMMs. We create a namespace on top of the NVM regions in `fsdax` mode. This allows Spitfire to directly access NVM using the load/store interface. It maps a file residing on the NVM device using the following commands:

```
fd = open(("/mnt/pmem0/file", O_RDWR, 0);
res = ftruncate(fd, SIZE);
ptr = mmap(nullptr, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);
```

Spitfire uses the resulting pointer to directly manage the NVM-resident buffer. To ensure the persistence of NVM-resident data, it first writes back the cache lines using the `clwb` instruction and then issues an `sfence` instruction so that the data reaches NVM. The `clwb` instruction writes back the cache lines without evicting them. Since this instruction is non-blocking, Spitfire must issue the `sfence` instruction to ensure that the cache write back is completed and the data is persistent.

We next describe how Spitfire leverages the additional data flow paths introduced by NVM.

3 NVM-Aware Data Migration

We propose a probabilistic technique for deciding where to move data. This multi-tier data migration policy works in tandem with the page replacement policy used in the DRAM and NVM buffers. Similar to Hymem, Spitfire uses the CLOCK page replacement policy for deciding what data should be evicted from a buffer [34]. Once a page is selected for eviction or promotion, Spitfire uses

the probabilistic data migration policy for determining the storage tier to which the page must be migrated.

As shown in Figure 3, the default read path comprises three steps: moving data from SSD to NVM (❶), then to DRAM (❷), and lastly to the processor cache (❸). Similarly, the default write path consists of three steps: moving data from processor cache to DRAM (❹), then to NVM (❺), and finally to SSD (❻). We now describe how Spitfire leverages the additional data flow paths introduced by NVM (❼,❽,❾,❿) to minimize the performance impact of NVM and to extend the lifetime of the NVM and SSD devices.

3.1 Bypass DRAM during Reads (D_r)

Unlike SSDs, the CPU is capable of directly accessing data on NVM during read operations (❼) [18]. To access a page on SSD, in a disk-centric DBMS, the DBMS copies it over to DRAM (❾), before it operates on the copied data. With NVM, Spitfire leverages this new data flow path (❼) to lazily migrate data from NVM to DRAM while serving read operations.

Spitfire uses a probabilistic technique for deciding where to move data. Let D_r represent the probability that Spitfire moves data from NVM to DRAM during read operations. When D_r is 1, we refer to this policy as *eager migration* of data to DRAM. Spitfire employs a wider range of *lazy migration* policies with smaller values for D_r. For example, when $D_r = 0.01$, Spitfire moves a page from NVM to DRAM only once every hundred times. Otherwise, it directly serves the read operation using the page residing on NVM. If the page is not present in the DRAM and NVM buffers, then it must fetch the data from SSD (§3.3). These lazy policies reduce upward data movement between NVM and DRAM during read operations. They are beneficial when the capacity of DRAM is smaller than that of NVM. A lazy migration strategy ensures that warm pages on NVM do not evict hot pages in DRAM.

The optimal value of D_r depends on the application's workload. An eager policy ($D_r \geq 0.7$) works well if the working set fits within the DRAM buffer. However, a policy with lower D_r works well if the working set does not fit in DRAM. This strategy ensures that only hot data is stored in DRAM.

With the eager policy, Spitfire frequently brings the page to DRAM while serving the read operation. Consequently, if the application then updates the same page, the writes are performed on DRAM. In contrast, a lazy policy increases the number of writes on NVM. This is because it is more likely that the page being updated is residing on NVM This is not a problem for applications with skewed access patterns [3, 33]. Such applications tend to modify hot data that is cached in the DRAM buffer even when Spitfire adopts a lazy policy.

3.2 Bypass DRAM during Writes (D_w)

Ensuring the persistence of pages containing log and checkpoint records is critical for the recoverability of the DBMS [30]. The DBMS's performance is constrained by the I/O overhead associated with persisting these pages on SSD [15]. As transactions tend to generate multiple log records that are each small in size, DBMSs use the *group commit* optimization to reduce this I/O overhead [9]. The DBMS first batches the log records for a group of transactions in the DRAM buffer (❹) and then flushes them together with a single write to SSD (❻). This optimization improves the operational throughput and amortizes the I/O overhead across multiple transactions.

Unlike SSDs, the CPU is capable of directly persisting data on NVM via write operations (❽) [18]. Spitfire leverages this path to provide *synchronous persistence* with lower overhead [22]. The write operation bypasses DRAM since the data must be eventually persisted. This optimization shrinks the overall latency of the operation. In addition to eliminating the redundant write to DRAM, it avoids potential eviction of other hot pages from the DRAM buffer.

Let D_w represent the probability with which the Spitfire uses DRAM during write operations. In a canonical DRAM-SSD system without NVM, $D_w = 1$. This is because the CPU cannot directly persist data on SSD. With faster NVM technologies, Spitfire employs a lazy policy with smaller D_w. This reduces the frequency of downward data migration to DRAM during write operations, thereby ensuring that pages containing log and checkpoint records do not evict hot pages in DRAM. Our evaluation demonstrates that this optimization improves Spitfire's performance on write-intensive workloads (§6.3).

3.3 Bypass NVM During Reads (N_r)

The data migration optimizations presented in §3.1 and §3.2 improve performance at the expense of increasing the number of writes to NVM. We next describe how Spitfire leverages two additional data flow paths for minimizing writes to NVM.

The default read path consists of moving the data from SSD to NVM (❶) and eventually migrating it to DRAM (❷). Spitfire instead makes use of the data flow path from SSD to DRAM (❾). When it observes that a requested page is not present in both the DRAM and NVM buffers, it copies the data on SSD directly to DRAM, thus bypassing NVM during read operations. If the data read into the DRAM buffer is not subsequently modified, and is selected for replacement, then it simply discards it. If the page is modified and later selected for eviction from DRAM, Spitfire considers admitting it to NVM (❺).

Let N_r represent the probability with which Spitfire copies data from SSD to NVM during read operations. With the default read path, $N_r = 1$. When a page is fetched from SSD and later evicted from DRAM, an eager policy necessitates two writes to NVM: once at fetch time and again when the page is evicted from DRAM. With a lazy policy (*e.g.*, $N_r = 0.01$), Spitfire installs a copy of a modified page on NVM only after it has been evicted from DRAM. This eliminates the first write to NVM when Spitfire fetches the page from SSD.

Hymem adopts a lazy migration policy for NVM [37]. Unlike Spitfire, it directly migrates pages from SSD to DRAM. This design ensures that the data is not duplicated in the NVM buffer. We quantify the degree of duplication using the *inclusivity ratio*:

$$\text{INCLUSIVITY} = \frac{\text{\# OF PAGES IN BOTH DRAM AND NVM BUFFERS}}{\text{\# OF PAGES IN EITHER DRAM OR NVM BUFFERS}}$$

Spitfire adopts a different approach for minimizing data duplication across the DRAM and NVM buffers. It employs: a lazy policy for migrating data from NVM to DRAM ($D_r = 0.01$), and a comparatively eager policy while moving data from SSD to NVM ($N_r = 0.2$). While this scheme increases the number of writes to

NVM compared to the lazy policy, it enables Spitfire to deliver higher performance than Hymem (§6.5).

3.4 Bypass NVM During Writes (N_w)

Lastly, Spitfire reduces the number of writes to NVM by bypassing it while serving write operations. The default write path consists of moving the data from DRAM to NVM (❺) and then eventually migrating it to SSD (❻). Instead of using the default path, Spitfire makes use of the direct data flow path from DRAM to SSD (❿).

By bypassing NVM during writes, Spitfire ensures that only pages frequently swapped out of DRAM are stored on NVM. This optimization is similar to the NVM admission queue mechanism employed in Hymem [37]. Unlike Hymem, Spitfire does not explicitly maintain such a queue and instead takes a probabilistic approach. Besides reducing the number of writes to NVM, this optimization also ensures that only warm pages are stored in the NVM buffer. If Spitfire employs an lazy policy while copying data from NVM into DRAM (D_r = 0.01), this optimization prevents colder DRAM-resident pages from polluting the NVM buffer.

Let N_w represent the probability with which the BM copies data from DRAM to NVM during write operations. With the default write path, N_w = 1. Lower values of N_w reduce downward data migration into NVM. Such a lazy policy is beneficial when the capacity of DRAM is comparable to that of NVM since it ensures that colder data on DRAM does not evict warmer data on NVM.

3.5 Data Migration Policy

We define the multi-tier data migration policy in terms of the probabilities with which the Spitfire bypasses DRAM and NVM while serving read and write operations (§3.1,§3.2, §3.3,§3.4). A policy P is given by the tuple:

$$\langle D_r, D_w, N_r, N_d \rangle$$

All of the above-mentioned data migration optimizations are moot unless Spitfire dynamically adapts the policy P based on the characteristics of the workload and the storage hierarchy (§6.4). We next present an adaptation mechanism that Spitfire uses to converge to a near-optimal policy for an arbitrary workload and storage hierarchy without requiring any manual tuning.

Theoretical Analysis.

We now present a theoretical analysis of how the data migration policy works. Consider a page P that is not present in the DRAM buffer. If there are $\mathcal{N}$ read requests for P, then the probability of P being brought into DRAM is approximated by $1 - ((1 - D_r)^{\mathcal{N}})$. If we assume that D_r is non-zero and accesses to pages are independent events, as $\mathcal{N}$ increases (*i.e.*, P is accessed more frequently), this probability converges to one. We empirically demonstrate the steady-state behavior of the policy in §6.3.

4 Adaptive Data Migration

Spitfire seeks to find a near-optimal policy P_{opt} that delivers the highest performance on the given workload and storage hierarchy. It is infeasible to determine the optimal policy for migrating data in a multi-tier buffer, even when full knowledge of the future requests is available [12]. This is because the benefit of a buffer hit depends on the level in which it occurred (*e.g.*, two hits in the DRAM buffer will be better than three hits in the NVM buffer). Thus, Spitfire relies on empirical analysis to identify P_{opt}.

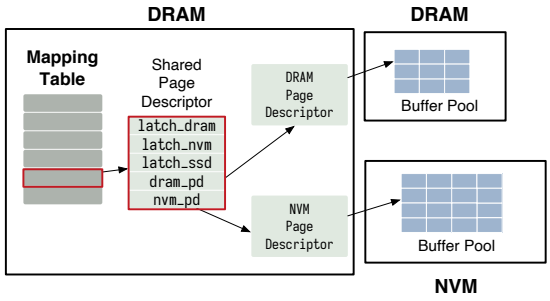


Figure 4: Architecture of Spitfire – Unified mapping table for the DRAM and NVM buffers.

Cost Function. The crux of our approach is to periodically track a set of target metrics, and then adapt P in the background. Spitfire evaluates a candidate policy across millions of buffer pool requests to ensure that its impact on the target metrics is prominently visible to the search algorithm. Over time, the search algorithm automatically optimizes P for the target workload and storage hierarchy.

Currently, Spitfire keeps track of the transactional throughput (T) metric to guide the adaptation mechanism. The cost function associated with a candidate policy P is given by:

$$\text{cost}(P) = \frac{1}{T}$$

Search Algorithm. Spitfire adapts the policy using simulated annealing (SA) [21]. This iterative search technique finds a policy P_{opt} that minimizes the cost function. SA is guaranteed to find the globally optimal policy P_{opt} with high probability. Internally, it uses a *temperature* parameter `t` for controlling the magnitude of cost fluctuations. During the initial tuning steps of SA, when `t` is high, the probability of picking worse policies is high. Despite temporarily increasing the cost, such non-beneficial steps allow for a more extensive search for P_{opt}. SA gradually decreases `t` over time. This corresponds to slowly decreasing the probability of accepting worse policies as it explores the state space.

5 System Architecture

We next present the system architecture of Spitfire. In particular, we discuss how Spitfire supports the NVM-centric data migration policies presented in §3, including an adaptive data migration mechanism. We begin with an overview of how Spitfire manages the DRAM and NVM buffers in §5.1. We then describe the concurrency control and recovery mechanisms of Spitfire in §5.2. We conclude with a discussion of the storage system design problem in §5.3.

5.1 Multi-Tier Buffer Management

We may configure Spitfire to manage one or two buffers residing in DRAM and/or NVM. Consider a three-tier configuration with DRAM and NVM buffers. With this configuration, Spitfire seeks to keep the hot, warm, and cold database pages on DRAM, NVM, and SSD, respectively. Since the CPU is capable of directly operating on both the DRAM and NVM buffers, Spitfire maintains a mapping table on DRAM to bound latency. This mapping table serves to keep track of pages buffered in DRAM and NVM.

When a page is requested, Spitfire performs a table lookup that returns a *shared page descriptor* containing the locations (if any) of

the logical page in the DRAM and NVM buffers. If the page is found on DRAM, then it returns a reference to the DRAM-resident buffer frame. Otherwise, if the page is found on NVM and the migration policy allows Spitfire to bypass DRAM, then it returns a reference to the NVM-resident buffer frame. If the page is not found in both DRAM and NVM buffers, Spitfire retrieves the page from SSD and places it in one of those two buffers based on the multi-tier data migration policy.

As shown in Figure 4, Spitfire maintains a shared page descriptor for every logical page $\mathcal{P}$ in the mapping table. The descriptor contains latches for thread-safe data migration and pointers to the DRAM and NVM page descriptors. The device-specific page descriptors contain additional metadata: (1) number of users of the page, (2) a bit indicating whether the page is dirty, (3) and a physical pointer to the page frame residing on the device.

Similar to Hymem, Spitfire adopts the CLOCK cache replacement policy to reclaim space in the DRAM and NVM buffers [34]. It relies on the cache replacement policy and the data migration policy to work in tandem to place the pages in the appropriate tiers based on their access frequency.

5.2 Concurrency Control and Recovery

To support concurrent operations, we leverage the following data structures and protocols: (1) a concurrent hash table for managing the mapping from logical page identifiers to shared page descriptors [17], (2) a concurrent bitmap for the cache replacement policy [39], (3) multi-versioned timestamp-ordering (MVTO) concurrency control protocol [38], (4) concurrent B+Tree for indexing with optimistic lock-coupling [24], and (5) lightweight latches for thread-safe page migrations.

Concurrent Index. The introduction of NVM into the storage system increases the overall buffer capacity, thereby reducing the number of operations hitting SSD. The reduction in I/O overhead increases the importance of computational overhead associated with index lookups. We implement a concurrent B+Tree with optimistic lock-coupling on top of Spitfire [24]. The optimistic lock-coupling technique reduces the contention overhead associated with its pessimistic counterpart, especially when the working set fits within the buffers.

Thread-Safe Page Migration. While migrating a page $\mathcal{P}$, Spitfire must ensure that there are no concurrent operations on $\mathcal{P}$. It accomplishes this using a bespoke page-level latching protocol designed to maximize concurrency. As shown in Figure 4, the shared page descriptor associated with $\mathcal{P}$ contains three latches corresponding to the three storage tiers. While migrating $\mathcal{P}$ from a tier $\mathcal{X}$ to tier $\mathcal{Y}$ (*e.g.*, NVM to SSD), it grabs the latches associated with these tiers before performing the I/O operation. This fine-grained latching protocol maximizes concurrency (as opposed to using a single latch for all data flow paths). For example, when $\mathcal{P}$ is to be written back from NVM to SSD, Spitfire only acquires the latches for the NVM and SSD tiers, thereby allowing concurrent operations on the copy of $\mathcal{P}$ in the DRAM buffer.

Spitfire handles the upward data flow path from NVM to DRAM (❷) differently. While migrating $\mathcal{P}$ along this path, there may be concurrent operations on the copy of $\mathcal{P}$ in the NVM buffer if Spitfire adopts a lazy data migration policy. In this case, the newly installed copy of $\mathcal{P}$ in the DRAM buffer might not contain the modifications made on the one in the NVM buffer. Spitfire circumvents this problem by: (1) acquiring the latches associated with the DRAM and NVM tiers, (2) waiting for all references to the copy on the NVM buffer to be dropped before migrating the page to DRAM, and (3) finally releasing the acquired latches.

Recovery. Spitfire ensures recoverability by implementing an NVM-aware write-ahead logging protocol. It initially persists the log records in a shared NVM log buffer to exploit its persistence and latency characteristics. A log record consists of: (1) transaction identifier and page identifier, (2) type of record, (3) log sequence number of previous log record for this transaction, and (4) before and after images. Once the commit log record of a transaction is persisted in the NVM log buffer, the transaction is considered committed. When the NVM log buffer size reaches a threshold, its contents are asynchronously appended to an on-disk log file. In the background, Spitfire periodically flushes dirty pages in the DRAM buffer to allow log truncation and to bound recovery time. However, the modified pages in NVM buffer are not flushed down to SSD since NVM is persistent. So, pages in NVM buffer might be newer than their SSD counterparts.

During recovery, Spitfire first reconstructs the contents of the NVM buffer to identify the latest version of a page. This is accomplished by scanning the NVM buffer to collect the page ids and to construct the mapping table. Second, the NVM log buffer needs to be appended to the log file since the buffer is persistent. Once the mapping table is recovered and log file is complete, Spitfire proceeds to recover the database using a traditional recovery scheme (analysis, redo, and undo phases).

5.3 Storage System Design

We have so far focused on identifying the optimal data migration policy for a particular workload and storage hierarchy. The tuning algorithm presented in §4 assumes that we have already provisioned a multi-tier storage hierarchy that is a good fit for the workload. However, it is unclear how to select such a hierarchy for a particular workload given a system cost budget. Prior research on NVM-aware storage management has not tackled this problem of designing a multi-tier storage system [2, 10, 20, 22, 31, 37]. We empirically investigate this problem by running Spitfire on diverse storage hierarchies identified using a grid search algorithm in §6.6.

6 Experimental Evaluation

Our evaluation aims to answer the following questions:

- **Benefits of NVM and App-Direct Mode:** How do equi-cost NVM-SSD and DRAM-SSD (memory-mode) hierarchies compare against each other? (§6.2)
- **Data Migration Policies:** How does the optimal data migration policy vary? (§6.3)
- **Adaptive Data Migration:** Does the adaptation mechanism work? (§6.4)
- **Revisiting Hymem's Optimizations:** How do the optimizations introduced in Hymem work on Optane DC PMMs? (§6.5)
- **Storage System Design:** How should we provision a storage system given a cost budget? (§6.6, §6.7)

6.1 Experimental Setup

Implementation. We implemented Spitfire with around 14,000 lines of C++ code. We leverage the concurrent data structures in Intel's Thread Building Blocks (TBB) library [17].

Evaluation Platform. We evaluate Spitfire on a two-socket platform with Optane DC PMMs. The platform contains 24 physical cores on each node. It is equipped with 384 GB DRAM (2 socket × 6 channel × 32 GB/DIMM), 3 TB NVM (2 socket × 6 channel × 256 GB/PMM), and a 375 GiB Intel Optane DC SSD (DC P4800X Series, PCIe NVMe 3.0 x4). We run Fedora 27 with Linux kernel version 4.18.10 compiled from source. We disable hyper-threading and set the CPU to use the highest possible clock frequency. We bind all the threads of Spitfire to the first socket to avoid NUMA effects. We measure the number of buffer manager operations processed per second (*i.e.*, throughput).

Workloads. We next describe the workloads that we use in our evaluation. These workloads differ with respect to skew and ratio of persistent writes.

- **YCSB:** This is a widely-used key-value store workload that is representative of the transactions handled by web-based companies [6]. It contains a single table comprised of tuples with a 4 B primary key and 10 columns of random string data, each 100 B in size. Thus, each tuple's size is approximately 1 KB. The keys are accessed by following a Zipfian distribution(z = 0.3) [14]. If not mentioned otherwise, we use a database with 100 million tuples (~100 GB).

  The workload consists of two transaction types: (1) a *read* transaction that retrieves a single tuple based on its primary key, and (2) an *update* transaction that modifies a single tuple based on its primary key. We use three workload mixtures that allow us to vary the I/O operations that Spitfire executes. These mixtures represent different ratios of *read* and *update* transactions:

  - **Read-Only (YCSB-RO):** 100% *reads*
  - **Balanced (YCSB-BA):** 50% *reads*, 50% *updates*
  - **Write-Heavy (YCSB-WH):** 10% *reads*, 90% *updates*

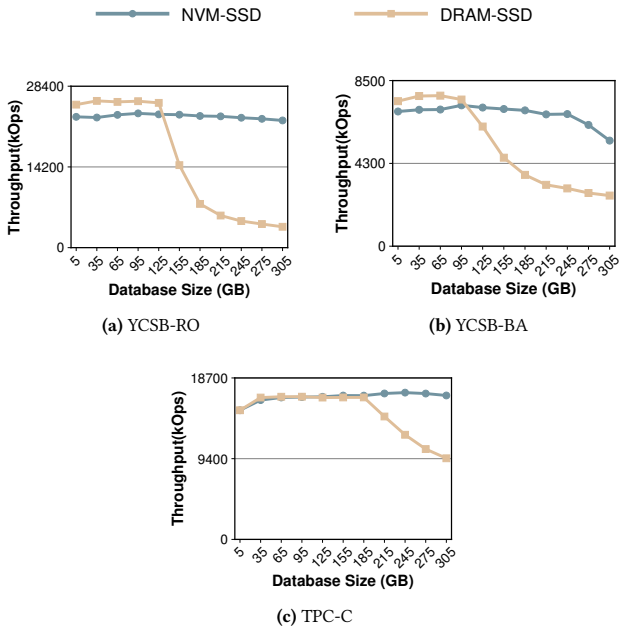
- **TPC-C:** This benchmark is the industry standard for evaluating the performance of transaction processing systems [35]. It simulates an order-entry environment of a wholesale supplier (*e.g.*, Amazon). The workload consists of five transaction types, which keep track of customer orders, payments, and other aspects of a warehouse. Transactions involving database modifications account for 88% of the workload. We configure the benchmark to manage 350 warehouses (~100 GB) by default.

6.2 Benefits of NVM and App-Direct Mode

We begin by comparing Spitfire's throughput on a equi-cost memory-mode DRAM-SSD and NVM-SSD storage systems to highlight the utility of NVM and to make the case for app-direct mode (§2.2).

Evaluation Platform.

For this experiment, we use a different server that supports memory mode. This server contains 28 physical cores on each node. It is equipped with 192 GB DRAM (2 socket × 6 channel × 16 GB/DIMM), 1 TB NVM (2 socket × 4 channel × 128 GB/PMM), and a 3.2 TB Intel SSD (DC P4610 Series). We bind all the threads of Spitfire to the first socket to avoid NUMA effects. Within one socket, 96 GB DRAM and 512 GB NVM is available to applications. When the server is configured in memory-mode, DRAM serves as a L4 write-back cache for NVM resulting in 512 GB memory capacity. In this mode, we configure the buffer capacity to be 140 GB. Since this exceeds the capacity of real DRAM, at least 140 GB of NVM is used. This translates to an equi-cost 340 GB NVM buffer NVM-SSD hierarchy (app-direct mode). We run these workloads on Spitfire with 16 threads: YCSB-RO, YCSB-BA and TPC-C. We warm up the system until the buffer pool is full. We vary the database size from 5 GB to 305 GB.



(a) YCSB-RO **(b)** YCSB-BA **(c)** TPC-C

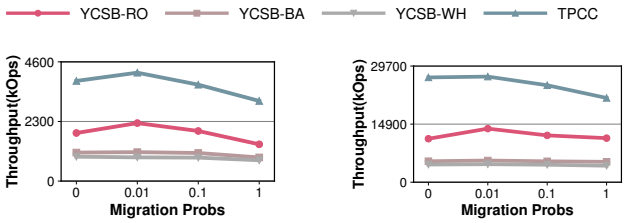
Figure 5: Benefits of NVM and App-Direct Mode: Comparison of throughput on NVM-SSD (app-direct mode) and DRAM-SSD (memory mode) hierarchies with varying database size settings.

From the results shown in Figure 5, we observe that when workload is cachable, DRAM-SSD hierarchy outperforms NVM-SSD hierarchy by up to 1.12x. However, once the database is not cacheable with DRAM-SSD, NVM-SSD hierarchy outperforms the DRAM-SSD by up to 6× on YCSB-RO. This is because the buffer capacity of NVM-SSD is 2.42× that of DRAM-SSD which is able to cache workloads across all the database sizes in the experiment. For YCSB-BA and TPC-C, we observe that NVM-SSD outperforms the DRAM-SSD hierarchy by up to 2.28× when database size exceeds the DRAM buffer capacity. This is due to the fact that NVM-SSD hierarchy reduces the number of SSD operations with a larger buffer and the elimination of flushing dirty pages in NVM buffer. For TPC-C, we observe that NVM-SSD performs as well as DRAM-SSD when workload is cacheable in DRAM. This is because Spitfire needs to flush dirty DRAM pages down to SSD, whose overhead offsets the latency advantage of DRAM over NVM.

From the-above observations, we make the case for using app-direct mode in buffer management. First, memory-mode requires an upfront NVM capacity at least equal to the size of DRAM. In

Migration Probabilities	0	0.01	0.1	1
Bypassing DRAM ($\mathcal{D}$)				
YCSB-RO	0	0.064	0.159	0.170
YCSB-BA	0	0.148	0.221	0.244
YCSB-WH	0	0.134	0.229	0.250
TPC-C	0	0.188	0.237	0.248
Bypassing NVM ($\mathcal{N}$)				
YCSB-RO	0	0.122	0.125	0.171
YCSB-BA	0	0.152	0.152	0.240
YCSB-WH	0	0.146	0.147	0.250
TPC-C	0	0.194	0.196	0.248

Table 2: Inclusivity Ratio of DRAM & NVM Buffers – We quantify the degree of duplication across the buffers (lower non-zero values are better).



(a) 1 Worker **(b)** 16 Workers

Figure 6: Performance Impact of Bypassing DRAM: Comparison of Spitfire's throughput when it adopts lazy and eager policies for DRAM.

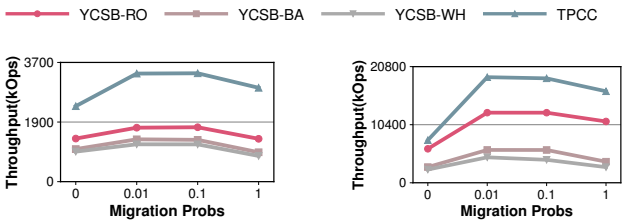
contrast, with app-direct mode, it could give a higher buffer capacity due to its cost advantage, though the NVM is bit slower. This is especially useful with a large working set. Second, with app-direct mode, Spitfire exploits the persistence property of NVM to reduce the overhead of recovery protocol by eliminating the need to flush modified pages in NVM buffer.

6.3 Data Migration Policies

In this section, we look at the impact of data migration policies on runtime performance and the number of writes performed on NVM. We begin by comparing the performance of the buffer manager when it employs the policies presented in §3. We consider a storage hierarchy with 12.5 GB DRAM and 50 GB NVM buffers on top of SSD. We quantify the performance impact of four data flow optimizations: (1) bypassing DRAM ($\mathcal{D}_r$, $\mathcal{D}_w$), and (2) bypassing NVM ($\mathcal{N}_r$, $\mathcal{N}_w$) while serving read and write operations. We run the experiments under single- and multi-threaded configurations. We report the inclusivity ratio of the DRAM and NVM buffers across different policies in Table 2.

Performance Impact of Bypassing DRAM. Figure 6 illustrates the impact of bypassing DRAM. We vary the DRAM migration probabilities ($\mathcal{D}_r$, $\mathcal{D}_w$) in lockstep from 0 through 1. We configure Spitfire to adopt an eager policy for NVM ($\mathcal{N}_r$, $\mathcal{N}_w$ = 1). Since the DRAM migration probabilities are updated in lockstep in this experiment, we denote them by $\mathcal{D}$. With the baseline policy ($\mathcal{D}$ = 1), the buffer manager eagerly moves data to DRAM. The results in Figure 6 demonstrate that the lazy policies work well for DRAM.

On the YCSB benchmark, the peak throughput of YCSB-RO workload observed is when $\mathcal{D}$ is 0.01, which is 58% higher than that with the eager migration policy. The reasons for this are threefold. First, the lazy policy reduces the data migration between NVM and DRAM by 16 times compared to the eager policy($\mathcal{D}$ = 1). Second, it



(a) 1 Worker **(b)** 16 Workers

Figure 7: Performance Impact of Bypassing NVM: Comparison of Spitfire's throughput when it adopts lazy and eager migration policies for NVM.

ensures that only frequently referenced data are moved to DRAM. Third, the lazy policy ensures a lower inclusivity ratio, as shown in Table 2, which results in allowing 10% more pages to be cached in the DRAM and NVM buffers in comparison to the eager policy ($\mathcal{D}$ = 1). This reduces the time spent on SSD operations by 1.13 ×. When $\mathcal{D}$ is 0, we observe a 20% drop in throughput compared to the peak throughput. This is because the DRAM buffer is effectively disabled, thereby reducing the total buffer space. With the YCSB-BA and YCSB-WH workloads, the performance gap between lazy ($\mathcal{D}$ = 0.01) and eager policy shrinks to 22% and 13%, since they involve more dirty page flushes. With the multi-threaded configuration, as shown in Figure 6b, the lazy policy outperforms eager policy across all workloads. The performance gap between these policies increases on write-intensive workloads, since the SSD constrains them in this configuration.

The lazy policy ($\mathcal{D}$ = 0.01) also works well on the TPC-C benchmark, as shown in Figure 6a. It outperforms the eager policy by 35%. We attribute this to the benefits of being able to directly update pages on NVM on a mixed workload [5]. With eager policies, more pages are updated in DRAM and they must be flushed down to lower tiers of the storage system (even when the update is localized to a small chunk of the page). In contrast, with a lazy scheme, Spitfire updates page in NVM, thereby reducing write amplification. We observe a similar trend with the multi-threaded configuration.

Performance Impact of Bypassing NVM. Figure 7 illustrates the performance impact of bypassing NVM. We vary the NVM migration probabilities ($\mathcal{N}_r$, $\mathcal{N}_w$) in lockstep from 0 through 1. Since these probabilities are updated in lockstep, we denote them by $\mathcal{N}$. The results in Figure 7 show that a lazy migration policy ($\mathcal{N}$ = 0.01) works well for NVM.

On the YCSB benchmark, as shown in Figure 7a, the throughput peaks when $\mathcal{N}$ is 0.01. With the YCSB-RO workload, lazy policy outperforms the eager one 1.25×. This is because it allows Spitfire to buffer 7% more pages due to lower inclusivity, as shown in Table 2. We observe similar trends on the YCSB-BA, YCSB-WH, and TPC-C workloads. With the multi-threaded configuration, the performance gap between these policies shrinks since SSD is saturated with both policies. When $\mathcal{N}$ = 0, we observe a 25% compares to lazy policy on the YCSB-RO workload. This gap is further widened to 103% with sixteen worker threads (much larger than that seen in Figure 6 when $\mathcal{D}$ = 0). This is because we effectively reduced the total buffer space by 6× and scale the workload by 16×. Thus, $\mathcal{N}$ = 0 is likely not going to be a viable setting in practice due to the higher capacity and lower cost of the NVM buffer.

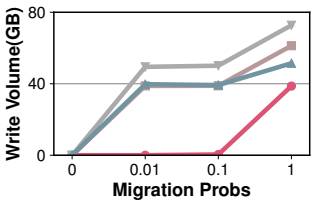


Figure 8: Impact of Bypassing NVM on Writes to NVM: Comparison of the number of writes performed on NVM when Spitfire adopts lazy and eager data migration policies for NVM.

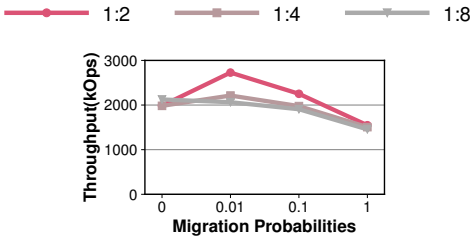


Figure 9: Impact of Storage Hierarchy: Comparison of the optimal data migration policy for bypassing DRAM across different storage hierarchies on the YCSB-RO workload.

The most notable observation from these experiments is that lazy policies work well for both DRAM and NVM buffers. This is because with Optane PMMs having comparable latency to DRAM, the best migration policy would be the one that maximizes the total buffer space available while keeping the hottest data in DRAM.

Impact of NVM Bypass on Writes to NVM. Besides improving runtime performance, lazy data migration policies also reduce the number of writes performed on NVM. Figure 8 presents the impact of these policies on the number of NVM writes. On the YCSB-RO workload, Spitfire performs 91.8× fewer writes to NVM with a lazy policy ($\mathcal{N}$ = 0.1) in comparison to its eager counterpart ($\mathcal{N}$ = 1). This is because, with the eager policy, Spitfire aggressively migrates pages from SSD to NVM when the page is not in DRAM and NVM buffers during reads. On the YCSB-BA, YCSB-WH, and TPC-C workloads, we observe that the writes to NVM is reduced by 1.58×, 1.45×, and 1.31× with the lazy policy. The is because these workloads are more write-intensive. These results illustrate that the optimal policy must be chosen depending on the performance requirements and write endurance characteristics of NVM.

Impact of Storage Hierarchy. We next consider how the optimal data migration policy varies across storage hierarchies. In this experiment, we configure Spitfire to use a 10 GB NVM buffer on top of SSD. We then vary the size of the DRAM buffer: 1.25 GB, 2.5 GB, and 5 GB. Thus, the ratio of the capacities of the DRAM and NVM buffers vary from 1:2, 1:4, and 1:8. The results for the YCSB-RO workload, as depicted in Figure 9, show that the utility of lazy data migration varies across these storage systems.

We observe that when the ratio is 1:8, the optimal policy is to set $\mathcal{D}$ = 0 (*i.e.*, completely disabling DRAM). This is because with the eager policy, the performance improvement brought by adding the comparatively smaller DRAM buffer (1.25 GB) is shadowed by cost of data migration between DRAM and NVM. However, as the ratio increases, a lazier policy ($\mathcal{D}$ = 0.01) works better by lowering inclusivity and thereby reducing SSD operations.

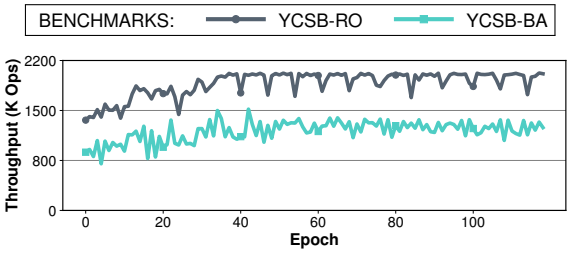


Figure 10: Adaptive Data Migration: The impact of data migration policy adaptation on runtime performance across different workloads.

These results show that the optimal policy depends not only on the workload and device characteristics, but also on the relative size of the DRAM and NVM buffers.

6.4 Adaptive Data Migration

In the previous experiments, we examined the utility of a *fixed* data migration policy. In the real world, identifying the optimal data migration policy is challenging due to diversity of workloads and storage hierarchies. Thus, we now examine the ability of Spitfire to automatically adapt the management policy at runtime.

In this experiment, Spitfire begins executing the workload with an eager policy for both DRAM ($\mathcal{D}$ = 1) and NVM ($\mathcal{N}$ = 1). During execution, it adapts the policy using the tuning algorithm presented in §4. This technique searches for the policy that maximizes the throughput given a target workload and storage hierarchy. We set α and γ to 0.9 and 10, respectively. We configure the initial and final temperatures of the annealing process to 800 and 0.00008. We configure the duration of a *tuning epoch* to be 5 s to ensure that the performance impact of policy changes are prominently visible to the tuning algorithm. We configure Spitfire to use 2.5 GB DRAM and 10 GB NVM buffers in this experiment.

The results in Figure 10 show that the Spitfire converges to a near-optimal policy for the YCSB-RO and YCSB-BA workloads without requiring any manual tuning. For the YCSB-RO workload, tuning the data migration policy increases throughput by 52%. Spitfire converges to a policy with lazy migration for both buffers on these workloads. The reasons are twofold. First, this policy reduces the I/O amplification between DRAM and NVM by utilizing the byte-addressability of NVM to directly read and update chunks of a page on NVM. Second, it ensures a smaller inclusivity ratio, allowing Spitfire to cache more pages in the DRAM or NVM buffers.

We observe that the throughput converges over time in Figure 10. We attribute this to the gradual cooling mechanism in SA that decreases the probability of accepting worse policies. The performance bumps are caused by dirty page flushes associated with the recovery protocol. Even on the YCSB-RO workload, Spitfire updates pages containing meta-data related to the MVTO protocol.

6.5 Revisiting Hymem's Optimizations

In this section, we examine how Hymem performs on real Optane DC PMMs. We configure both Hymem and Spitfire to use a 8 GB DRAM buffer and a 32 GB NVM buffer. We run the YCSB-RO and TPC-C workloads operating on ~20 GB SSD-resident databases.

We configure similar dataset sizes and buffer capacities to match the experiments described in Hymem [37].

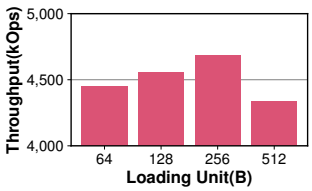


Figure 11: Optimal Granularity for Loading Data on NVM: Comparison of Hymem's throughput across different loading granularities. The results show that the cacheline-grained loading optimization does not work well on Optane DC PMMs.

Policy	$\mathcal{D}_r$	$\mathcal{D}_w$	$\mathcal{N}_r$	$\mathcal{N}_w$
Hymem [37]	1	1	0	AdmQueue
Spitfire-Eager	1	1	1	1
Spitfire-Lazy	0.01	0.01	0.2	1

Table 3: Migration Policies: List of policies used in the ablation study.

Admission Queue Size. As shown in Table 3, Hymem's migration policy consists of eager migration for DRAM ($\mathcal{D}_r$ = 1, $\mathcal{D}_w$ = 1). In case of NVM, each time a page is considered for admission, Hymem checks whether the page is in the admission queue (§1). If so, it removes the page from the queue and admits it into the NVM buffer. Otherwise, it adds the page identifier to the queue and directly moves the page to SSD, thereby bypassing the NVM buffer. Thus, Hymem eagerly moves the data to DRAM and then stores the pages evicted from DRAM on NVM using the queuing mechanism.

Unfortunately, the size of the admission queue is not mentioned in [37]. So, we conduct an experiment to determine a performant queue size. We observe that the queue size is proportional to the size of the NVM buffer. In particular, setting the queue size to be half the number of the pages in the NVM buffer works well on both workloads (~8 MB).

Optimal Granularity for Loading Data on NVM. Recent efforts have highlighted that Optane DC PMMs perform device-level operations at 256 B granularity (not 64 B granularity as assumed in Hymem) [25, 36]. So, we seek to determine the optimal granularity for loading data from NVM. We perform an experiment using YCSB-RO workload with an eager migration policy.

The most notable observation from the results shown in Figure 11 is that the cacheline-grained loading optimization proposed in Hymem does not work well on Optane DC PMMs. Hymem's throughput instead peaks at 256 B granularity. We attribute the 1.1× lower throughput at 64 B granularity to the I/O amplification stemming from the mismatch between the device-level block size and loading granularity.

Policy Comparison & Ablation Study. We next conduct an ablation study to delineate the performance impact of the key optimizations in Hymem and Spitfire with the data migration policies listed in Table 3. We configure Hymem's loading granularity to be 256 B and set the admission queue size to be 8 MB. For each of the policies, we incrementally add these two optimizations to the baseline configuration: (1) fine-grained loading to improve the utilization of NVM bandwidth, and (2) mini-page layout to reduce the DRAM footprint of fine-grained pages.

The results are shown in Figure 12. With the YCSB-RO workload, fine-grained loading improves performance by reducing the amount of I/O operations on Optane DC PMMs. The throughput

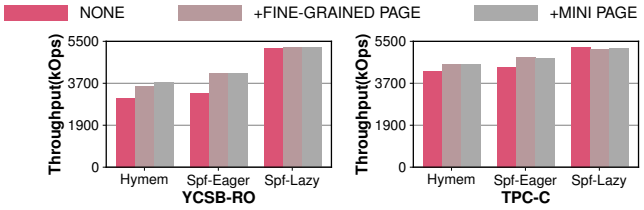


Figure 12: Ablation Study of Hymem and Spitfire: The performance impact of the optimizations in Hymem across different data migration policies and workloads. (Spf-Eager: Spitfire-Eager ; Spf-Lazy: Spitfire-Lazy)

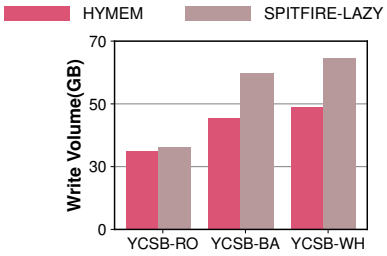


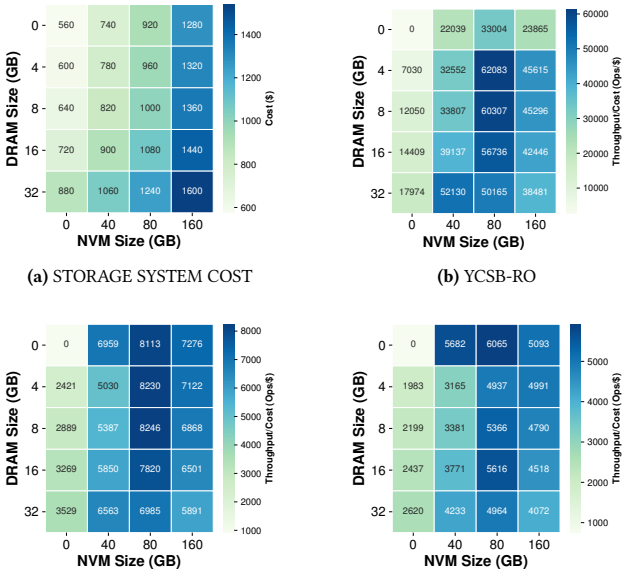
Figure 13: Impact of Data Migration Policies on NVM Device Lifetime: The number of NVM writes using different migration policies.

increases by 18% with Hymem's migration policy and 37% with the eager migration policy, respectively. In contrast, the mini-page optimization, only delivers 6% improvement for both policies. We attribute this to the increase in the time taken for sorting the slots, given the larger loading unit size. In case of the lazy policy in Spitfire, fine-grained loading has a minuscule impact. This is because with this policy, the amount of data migration between DRAM and NVM is already reduced by 6.5× since Spitfire directly operates on the larger NVM buffer with comparable latency. This renders the impact of I/O reduction between DRAM and NVM via fine-grained loading to be less prominent.

As for the TPC-C workload, fine-grained loading only improves performance by 7% and 9% with the Hymem and eager policies. This is because the time spent on data migration between DRAM and NVM is smaller than that spent on SSD operations. Thus, the mini-page optimization does not work well on write-intensive workloads across all migration policies.

On both workloads, we observe that even the baseline configuration of the lazy policy outperforms other policies with these optimizations enabled. This implies that the choice of the migration policy is more important than the other optimizations. Furthermore, the benefits of the optimizations proposed in Hymem vary based on the migration policy and work well with the eager policy.

Impact on NVM Device Lifetime. Lastly, we measure the number of NVM writes using different migration policies. We compare Spitfire's lazy migration policy(Spitfire-Lazy) against Hymem. To ensure an fair comparison, we enable the fine-grained page loading optimization in both policies as this helps reduce the amount of I/O to NVM. The results are shown in Figure 13. The Spitfire-Lazy policy performs 1.05–1.4× more writes than Hymem. This is because Spitfire eagerly writes data on NVM and aggressively bypasses DRAM to maximize runtime performance. This policy will shrink



(a) STORAGE SYSTEM COST **(b)** YCSB-RO

(c) YCSB-BA **(d)** YCSB-WH

Figure 14: Storage System Design: (a) The total cost of the DRAM, NVM, and SSD devices used in a multi-tier storage system. (b-d) The performance/price numbers of candidate storage hierarchies on different workloads. Given a cost budget and a target workload, we perform a grid search to identify the storage hierarchy with the highest performance/price number.

the lifetime of NVM devices with limited write-endurance. In contrast, Hymem's policy performs more writes on DRAM, thereby reducing the number of NVM writes.

6.6 Storage System Design

We next focus on the storage system recommendation problem presented in §5.3. In this experiment, we compare the *performance/price numbers* of multi-tier storage hierarchies. If the cost of a storage hierarchy is $\$ C$ and the throughput it delivers is $\mathcal{T}$ operations per second, then the performance/price number is given by $\frac{\mathcal{T}}{C}$. This represents the number of operations executed per second per dollar. Given a system cost budget and a target workload, we seek to identify the storage hierarchy with the highest performance/price number. Each storage system consists of at most three devices: DRAM, NVM, and SSD. We vary the capacity of the DRAM and NVM devices from 0 GB through 32 GB, and from 0 GB through 160 GB, respectively. We examine the runtime performance of Spitfire on both two- and three-tier storage hierarchies: DRAM-SSD, NVM-SSD, and DRAM-NVM-SSD. We configure the SSD to 200 GB for all the hierarchies. Since the SSD device is used in every hierarchy, the cost for SSD device remains unchanged for all hierarchies. For three-tier storage hierarchies, we use Spitfire-Lazy listed in Table 3 as the migration policy. For both YCSB and TPC-C workloads, we configure the database size to be 100 GB. For YCSB workloads, we configure the skew factor to be 0.5. We run all the experiments with eight worker threads.

Storage Hierarchy Recommendation. Figure 14 shows the performance/price numbers of candidate storage hierarchies across different workloads. These numbers are derived from the cost listed in Figure 14a. We perform a grid search to identify the storage system with the highest performance/price number on a target workload given a cost budget. With the YCSB-RO benchmark, as shown in Figure 14b, the storage system that delivers the highest performance/price number consists of a 4 GB DRAM buffer and a 80 GB NVM buffer and on top of SSD. Expanding the capacity of the DRAM buffer to 32 GB only improves the performance by 4%. However, it also raises the storage system cost by 29%. We observe that NVM-SSD hierarchy does not work well for YCSB-RO since NVM read latency is 3× higher than DRAM. In contrast, Spitfire-Lazy is able to migrate the hottest data in DRAM and directly read the warm data on NVM.

With the YCSB-BA benchmark, as shown in Figure 14c, the highest performance/price number is achieved with a hierarchy consists of a 8 GB DRAM buffer and a 80 GB NVM buffer on top of SSD. This hierarchy reduces the number of SSD operations by absorbing persistent writes in NVM and keeping the hot pages in DRAM. Increasing the DRAM buffer space does not improve the performance/price number. Another interesting observation is that the performance/price number of NVM-SSD hierarchy is close to the optimal DRAM-NVM-SSD hierarchy. On write-intensive workloads, as shown in Figure 14d, the optimal performance/price number is actually achieved with NVM-SSD hierarchy (though the maximal absolute performance metric is achieved with a DRAM-NVM-SSD hierarchy). This is because these workloads are bottlenecked by the performance of the recovery protocol that flushes dirty DRAM pages downward. With NVM-SSD hierarchy, this overhead is reduced since modifications to NVM pages are persistent.

The results in Figure 14 illustrate how the selection of a multi-tier storage system for a given workload depends on the working set size, the frequency of persistent writes, the performance and cost characteristics of NVM, and the system cost budget. We distill these results into a set of design principles:

- To achieve highest absolute performance, the hierarchy usually consists of DRAM (since DRAM has lowest latency).
- If the workload is read-intensive, DRAM-NVM-SSD hierarchy is the best choice from a performance/price standpoint, since it is able to ensure the hottest data resides in DRAM.
- If the workload is write-intensive, NVM-SSD hierarchy is the best choice from a performance/price standpoint, since NVM is able to reduce the recovery protocol overhead.

6.7 Impact of Database Size

In this section, we focus on the performance impact of database size on different migration policies and storage hierarchies. Specifically, we compare five configurations: DRAM-NVM-SSD (Spitfire-Eager, Spitfire-Lazy, Hymem), NVM-SSD, and DRAM-SSD. For fairness, we enable the two optimizations in Hymem for the first three configurations. We include the NVM-SSD and DRAM-SSD hierarchies to illustrate their strengths and weaknesses.

For three-tier hierarchies, we provision the capacity of the DRAM and NVM buffers to be 20 GB and 60 GB, respectively. For DRAM-SSD and NVM-SSD hierarchies, we configure the capacity of DRAM buffer and NVM buffer to be 46 GB and 104 GB which are similarly priced with the three-tier hierarchies. We then vary the database size of YCSB and TPC-C workloads from 5 GB to 140 GB. This ensures that we cover a wide range of buffer hit ratios. For all

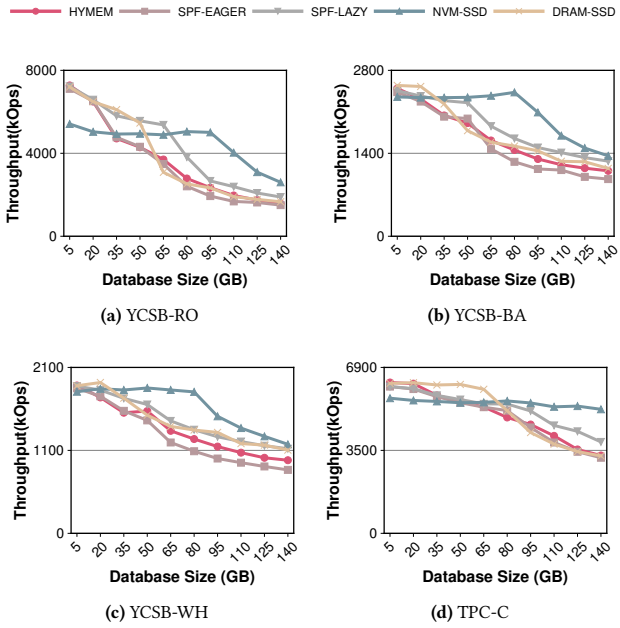


Figure 15: Impact of Database Size: Comparison of throughput onSpitfire-Eager, Spitfire-Lazy, Hymem, DRAM-SSD, and NVM-SSD with varying database sizes. The first three run on DRAM-NVM-SSD hierarchy.

experiments, we first warm-up the buffer pools for 300 s by running the workloads.

The results shown in Figure 15 indicate that varying database size for diffrent migration policies and hierarchies has significant performance impact. For YCSB-RO workload shown in Figure 15a, when the database size is cacheable in DRAM buffers, all policies and hierarchies with DRAM perform similarly well. For NVM-SSD hierarchy, its throughput is up to 1.33× lower than others because of the higher latency of NVM. However, the NVM-SSD hierarchy performs better by up to 2.5× after database size reaches 80 G. This is because NVM-SSD hierarchy reduces expensive SSD operations due to larger buffer capacity. For DRAM-SSD hierarchy, it shows the highest performance when the database is DRAM-cacheable and degrades significantly when that is not the case. For three-tier storage hierarchies, Spitfire-Lazy performs better than Spitfire-Eager and Hymem in almost all settings. The reasons are twofold. First, Spitfire-Lazy has a larger buffer space because of the lower inclusivity between DRAM and NVM buffers. This enables Spitfire-Lazy to reduce more SSD operations. Second, the lazy policy reduces the migration cost between DRAM and NVM. We also note that for YCSB-RO, Spitfire-Lazy is performing as well as DRAM-SSD. This is partly because the lazy migration policy ensures that hottest data resides in DRAM buffer.

On YCSB-WAL and YCSB-WH workloads, NVM-SSD hierarchy starts with similar or lower throughput than other configurations with DRAM when the database is cacheable in DRAM. It outperforms all of them when the database size is 50 GB. The reasons are two-fold. First, there are no dirty page flushes with the NVM-SSD hierarchy. Second, the buffer capacity of NVM-SSD hierarchy is 25% and 126% larger than that of three-tier hierarchies and DRAM-SSD hierarchy. This reduces the number of expensive SSD operations. The second-best performing policy is Spitfire-Lazy because of its larger buffer capacities and lower data movement. Hymem and Spitfire-Eager perform the worst in settings when database is not DRAM-cacheable.

With the TPC-C benchmark, as shown in Figure 15d, we observe that DRAM-SSD hierarchy maintains the highest throughput up to 65 GB of database size where the workload cannot be easily cached. Among three-tier policies, Spitfire-Lazy maintains similar or higher throughput across all databases sizes, especially when the database size exceeds the buffer capacities. NVM-SSD hierarchy delivers the highest throughgput after 80 GB of database size and maintains the throughput across all the database sizes. This is because TPC-C has a smaller working set than YCSB and the large capacity of NVM-SSD hierarchy is able to cache most of the page accesses.

From these observations, we could summarize the the strengthes and weaknesses of the policies and hierarchies as follows:

- For three tier hierarchies, Spitfire-Lazy is the ideal policy for delivering highest throughput.
- For read-intensive or small working-set workloads, Spitfire-Lazy is the ideal policy as it increases the effective buffer capacities and ensures that hottest data remains in DRAM.
- For write-intensive or large working-set workloads, NVM-SSD hierarchy stands out due to its buffer capacity advantage and its ability to reduce the flushing overhead of recovery protocol by performing persistent writes to the pages in NVM buffer.

7 Related Work

We now discuss the previous research on buffer management.

Buffer Management in Systems with NVM. Renen et al. presented Hymem, an NVM-aware multi-tier buffer manager that eagerly migrates data from SSD to DRAM [37]. We highlighted the differences between Hymem and Spitfire in §1 and §6.5. SOFORT [31] is a storage engine that targets a two-tier storage system with DRAM and NVM. FOEDUS is a scalable OLTP engine designed for a two-tier storage system with DRAM and NVM [20]. Unlike these systems, Spitfire focuses on managing and designing multi-tier storage hierarchy with DRAM, NVM, and SSD.

Buffer Management in Systems without NVM. Researchers have studied buffer management in storage systems without NVM [1, 11, 27, 29]. LeanStore is a concurrent DRAM-SSD buffer manager that is based on pointer swizzling [13] and a low-overhead replacement policy [23]. ERMIA is a memory-optimized system designed for handling heterogeneous workloads [19]. FlashStore is a key-value store that uses an SSD as a fast cache between DRAM and HDD and minimizes the number of SSD accesses [8]. Deuteronomy is a data caching system that exploits latch-free access to a log-structured store [28]. Unlike these systems, Spitfire focuses on NVM-aware buffer management.

8 Conclusion

This paper presented Spitfire, a multi-threaded buffer manager for managing a three-tier storage hierarchy comprising of DRAM, NVM, and SSD. We introduced a taxonomy for NVM-aware data migration policies and discussed why the data migration policy must be synthesized based on the workload and the storage hierarchy. We presented an adaptation mechanism in Spitfire that achieves a

near-optimal policy for an arbitrary workload without requiring any manual tuning. Our results demonstrate that Spitfire outperforms Hymem, the state-of-the-art multi-tier buffer manager, across diverse workloads. We demonstrate that the choice of the migration policy is more important than the fine-grained loading and mini-page optimizations. We discuss how these latter optimizations must be tailored for Optane DC PMMs.

References

[1] R. Appuswamy, R. Borovica, G. Graefe, and A. Ailamaki. The five minute rule thirty years later and its impact on the storage hierarchy. In *ADMS*, 2017.

[2] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, and F. Inc. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *SIGMOD Rec.*, 39(3):5–10, 2011.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[7] I. Cutress and B. Tallis. Intel launches optane dimms up to 512 gb: Apache pass is here. https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here, May 2018.

[8] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *VLDB*, pages 1414–1425, 2010.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[10] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing dram footprint with nvm in facebook. In *EuroSys*, page 42, 2018.

[11] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

[12] B. S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *FAST*, pages 1–17, 2008.

[13] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-memory performance for big data. *VLDB*, 8(1), 2014.

[14] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[16] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[17] Intel. Thread Building Blocks Library. https://github.com/oneapi-src/oneTBB.

[18] Intel Corporation. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf, August 2015.

[19] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, pages 1675–1687, 2016.

[20] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[22] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[23] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *ICDE*, pages 185–196. IEEE, 2018.

[24] V. Leis, M. Haubenschild, and T. Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[25] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *VLDB*, 13(4):574–587, 2019.

[26] L. Lersch, I. Oukid, I. Schreter, and W. Lehner. Rethinking dram caching for lsms in an nvram environment. In *ADBIS*, pages 326–340. Springer, 2017.

[27] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[28] D. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *DaMoN*, pages 1–10, 2018.

[29] L. Ma and *et al.* Larger-than-memory data management on modern storage hardware for in-memory oltp database systems. In *DaMoN*, 2016.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[31] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. DaMoN, pages 8:1–8:7, 2014.

[32] G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki. Bridging the latency gap between nvm and dram for latency-bound operations. In *DaMoN*, pages 1–8, 2019.

[33] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *SIGMOD*, SIGMOD '12, pages 731–742, 2012.

[34] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3), 1978.

[35] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[36] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory i/o primitives. In *DaMoN*, pages 1–7, 2019.

[37] A. van Renen et. al. Managing non-volatile memory in database systems. In *SIGMOD*, 2018.

[38] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *VLDB*, 10(7):781–792, 2017.

[39] M. Yui, J. Miyazaki, S. Uemura, and H. Yamana. Nb-gclock: A non-blocking buffer management based on the generalized clock. In *ICDE*, pages 745–756. IEEE, 2010.