Managing Non-Volatile Memory in Database Systems

Alexander van Renen
Technische Universität München
renen@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Takushi Hashida
Fujitsu Laboratories
hashida.takushi@jp.fujitsu.com

Kazuichi Oe
Fujitsu Laboratories
ooe.kazuichi@jp.fujitsu.com

Yoshiyasu Doi
Fujitsu Laboratories
yosh-d@jp.fujitsu.com

Lilian Harada
Fujitsu Laboratories
harada.lilian@jp.fujitsu.com

Mitsuru Sato
Fujitsu Laboratories
msato@jp.fujitsu.com

ABSTRACT

Non-volatile memory (NVM) is a new storage technology that combines the performance and byte addressability of DRAM with the persistence of traditional storage devices like flash (SSD). While these properties make NVM highly promising, it is not yet clear how to best integrate NVM into the storage layer of modern database systems. Two system designs have been proposed. The first is to use NVM exclusively, i.e., to store all data and index structures on it. However, because NVM has a higher latency than DRAM, this design can be less efficient than main-memory database systems. For this reason, the second approach uses a page-based DRAM cache in front of NVM. This approach, however, does not utilize the byte addressability of NVM and, as a result, accessing an uncached tuple on NVM requires retrieving an entire page.

In this work, we evaluate these two approaches and compare them with in-memory databases as well as more traditional buffer managers that use main memory as a cache in front of SSDs. This allows us to determine how much performance gain can be expected from NVM. We also propose a lightweight storage manager that simultaneously supports DRAM, NVM, and flash. Our design utilizes the byte addressability of NVM and uses it as an additional caching layer that improves performance without losing the benefits from the even faster DRAM and the large capacities of SSDs.

ACM Reference Format:
Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196897

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

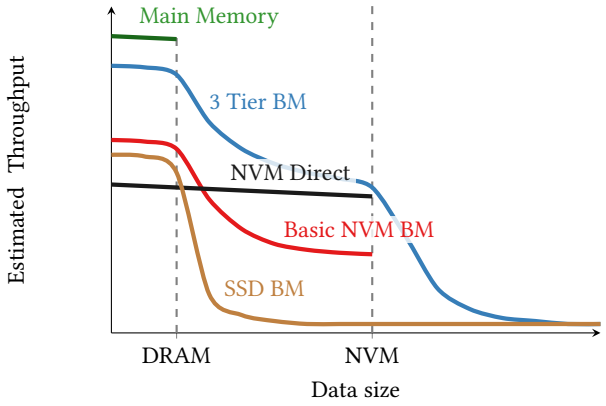


Figure 1: System designs under varying data sizes.

1 INTRODUCTION

Non-volatile memory (NVM), also known as Storage Class Memory (SCM) and NVRAM, is a radically new and highly promising storage device. Technologies like PCM, STT-RAM, and ReRAM have slightly different features [35], but generally combine the byte addressability of DRAM with the persistence of storage technologies like SSD (flash). Because commercial products are not yet available, the precise characteristics, price, and capacity features of NVM have not been publicly disclosed (and like all prior NVM research, we have to resort to simulation for experiments). What is known, however, is that for the foreseeable future, NVM will be slower (and larger) than DRAM and, at the same time, much faster (but smaller) than SSD [13]. Furthermore, NVM has an asymmetric read/write latency—making writes much more expensive than reads. Given these characteristics, we consider it unlikely that NVM can replace DRAM or SSD outright.

While the novel properties of NVM make it particularly relevant for database systems, they also present new architectural challenges. Neither the traditional disk-based architecture nor modern main-memory systems can fully utilize NVM without major changes to their designs. The two components most affected by NVM are logging/recovery and storage. Much of the recent research on NVM has optimized logging and recovery [5, 16, 22, 36, 45]. In this work, we instead focus on the storage/caching aspect, i.e., on dynamically deciding where data should reside (DRAM, NVM, or SSD).

Two main approaches for integrating NVM into the storage layer of a database system have been proposed. The first, suggested by Arulraj et al. [4], is to use NVM as the primary storage for relations as well as index structures and perform updates directly on NVM. This way, the byte addressability of NVM can be fully leveraged. A disadvantage is that this design can be slower than main-memory database systems, which store relations and indexes in main memory and thereby benefit from the lower latency of DRAM. To hide the higher NVM latency, Kimura [25] proposed using a database-managed DRAM cache in front of NVM. Similar to a disk-based buffer pool, accesses are always performed on in-memory copies of fixed-size pages. However, accessing an uncached page becomes more expensive than directly accessing NVM, as an entire page must be loaded even if only a single tuple is accessed. Furthermore, neither of the two approaches supports very large data sets, as the capacity of NVM is limited compared to SSDs.

In this work, we take a less disruptive approach and implement NVM as an additional caching layer. We thus follow Michael Stonebraker, who argued that NVM-DIMMs are …

> *"…not fast enough to replace main memory and they are not cheap enough to replace disks, and they are not cheap enough to replace flash."* [41]

Figure 1 sketches the performance characteristics and capacity restrictions of different system designs (*Buffer Manager* is abbreviated as *BM*). Besides the two NVM approaches ("Basic NVM BM", "NVM Direct"), we also show main-memory systems ("Main Memory"), and traditional SSD buffer managers ("SSD BM"). Each of these designs offers a different tradeoff in terms of performance and/or storage capacity. As indicated in the figure, all existing approaches exhibit steep performance cliffs ("SSD BM" at DRAM size and "Basic NVM BM" at NVM size) or even hard limitations ("Main Memory" at DRAM size, "NVM Direct" at NVM size).

In this work, we propose a novel storage engine that simultaneously supports DRAM, NVM, and flash while utilizing the byte addressability of NVM. As the "3 Tier BM" line indicates, our approach avoids performance cliffs and performs better than or close to that of specialized systems. NVM is used as an additional layer in the storage hierarchy supplementing DRAM and SSD [7, 13]. Furthermore, by supporting SSDs, it can manage very large data sets and is more economical [18] than the other approaches. These robust results are achieved using a combination of techniques:

- To leverage the byte-addressability of NVM, we cache NVM accesses in DRAM at cache-line granularity, which allows for the selective loading of individual hot cache lines instead of entire pages (which might contain mostly cold data).
- To more efficiently use the limited DRAM cache, our buffer pool transparently and adaptively uses small page sizes.
- At the same time, our design also uses large page sizes for staging data to SSD—thus enabling very large data sets.
- We use lightweight buffer management techniques to reduce the overhead of in-memory accesses.
- Updates are performed in main memory rather than directly on NVM, which increases endurance and hides write latency.

The rest of this paper is organized as follows. We first discuss existing approaches for integrating NVM into database systems in Section 2. We then introduce some key techniques of our storage engine in Section 3 before describing how our approach supports DRAM, NVM, and SSDs in Section 4. Section 5 evaluates our storage engine by comparing it with the other designs. Related work is discussed in Section 6, and we conclude the paper in Section 7.

2 BACKGROUND: NVM STORAGE

Several architectures for database systems optimized for NVM have been proposed in the literature. In this section, we revisit the two most promising of these designs and abstract their general concepts into two representative system designs. They are illustrated in Figure 2 alongside the approach that we propose in this paper.

2.1 NVM Direct

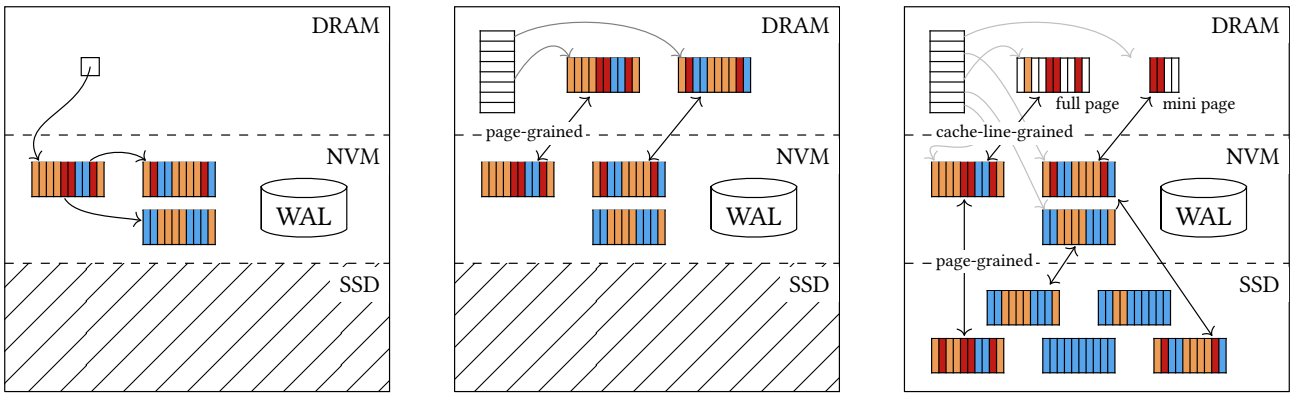
NVM, which offers latencies close to DRAM and byte addressability, can be used as the primary storage layer. A thorough investigation of different architectures for NVM Direct systems has been conducted by Arulraj et al. [4]. Their work categorizes database systems into in-place update, log-structured, and copy-on-write engines, before adapting and optimizing each one for NVM. Experimental results suggest that in most cases an in-place update engine achieves the highest performance as well as the lowest wear on the NVM hardware. Therefore, we chose this in-place update engine as a reference system for working directly on NVM (Figure 2a).

One challenge of using NVM is that writes are not immediately persistent because NVM is behind the same CPU cache hierarchy as DRAM and changes are initially written to the volatile CPU cache. It is only when the corresponding cache line is evicted from the CPU cache that the update becomes persistent (i.e., written to NVM). Therefore, it is not possible to prevent a cache line from being evicted and written to NVM, and each update might be persisted at any time. It is, however, possible to force a write to NVM by flushing the corresponding cache lines. These flushes are a building block for a durable and recoverable system.

Logging is implemented as follows. A tuple is updated by first writing a write-ahead log (WAL) entry that logs the tuple id and the changes (before and after image). Then, the log entry needs to be persisted to NVM by evicting the corresponding cache lines. Intel CPUs with support for NVM like the Crystal Ridge Software Emulation Platform [14], which is also used in our evaluation, provide a special instruction for that: `clwb` allows one to write a cache line back to NVM without invalidating it (like a normal `clflush` instruction would). In addition, to ensure that neither the compiler, nor the out-of-order execution of the CPU reorders subsequent stores, a memory fence (`sfence`) has to be used. Thereafter, the log entry is persistent and the recovery component can use it to redo or undo the changes to the actual tuple. At this point, the transaction can update and then persist the tuple itself. After the transaction is completed, the entire log written by the transaction can be truncated because all changes are already persisted to NVM.

As illustrated in Figure 2a, the design keeps all data in NVM. DRAM is only used for temporary data and to keep a reference to the NVM data. The log is written to NVM as well.

The NVM direct design has several advantages. By keeping the log minimal (it contains only in-flight transactions), recovery is very efficient. In addition, read operations are very simple because the system can simple read the requested tuple directly from NVM.



(a) NVM Direct **(b)** Basic NVM BM **(c)** Our NVM-Opt Three-Tier BM

Figure 2: NVM-Based Storage Engine Designs – NVM Direct (a) stores all data on NVM, which allows for cache-line-grained accesses. Basic buffer managers (b) fixed-size pages in DRAM, but require page-grained accesses to NVM. Our design (c) uses fixed-size pages to enable support for SSD and, in addition, supports cache-line-grained loading for NVM-resident data to DRAM. (hot, warm, cold cache lines)

However, there are also downsides. First, due to the higher latency of NVM compared to DRAM, it becomes more difficult to achieve a very high transaction throughput. Second, working directly on NVM without a buffer wears out the limited NVM endurance, thus potentially causing hardware failures. Third, an engine that works directly on NVM is difficult to program, because there is no way to prevent eviction and any modification is potentially persisted. Therefore, any in-place write to NVM must leave the data structure in a correct state (similar to lock-free data structures, which are notoriously difficult).

2.2 Basic NVM Buffer Manager

Given the downsides of the NVM direct approach, using DRAM as a cache in front of NVM may seem like a promising alternative. A well-known technique for adaptive memory management between a volatile and persistent layer is a buffer manager. It is used by most traditional disk-based database systems and can easily be extended to use NVM instead of SSD. We illustrate this idea in Figure 2b.

In buffer-managed systems, all pages are stored on the larger, persistent layer (NVM). The smaller, volatile layer (DRAM) acts as a software-managed cache called a buffer pool. Transactions operate only in DRAM and use the `fix` and `unfix` functions to lock pages into the buffer pool while they are accessed. In the traditional buffer manager (DRAM + SSD/HDD), this was necessary because it is not possible to make modifications directly on a block-oriented device. In the case of NVM, we argue that it is still beneficial because of the higher latency and the limited endurance of NVM.

An NVM-optimized variant of this approach has been introduced in the context of the research prototype FOEDUS [25]. The memory is divided into fixed-size pages, and transactions only operate in DRAM. Instead of storing page identifiers, like a traditional buffer manager, FOEDUS stores two pointers. One identifies the NVM-resident copy of the page, the other one (if not `null`) the DRAM one. When a page is not found in DRAM, it is loaded into a buffer pool. FOEDUS uses an asynchronous process to combine WAL log entries and merge them into the NVM-resident pages to achieve durability. Thus, a page is never directly written back but only indirectly via the log. The system is optimized for workloads fitting into DRAM. NVM is mostly used for durability and cold data.

Our goal, in contrast, is to support workloads that also access NVM-resident data frequently. Therefore, we extend the idea of a buffer manager and optimize it to make NVM access cheap. A non-optimized version is used as a baseline to represent layered systems.

2.3 Recovery

Besides the storage layout, the logging and recovery components of database systems are also greatly impacted by the upcoming NVM hardware. Log entries can be written to NVM much faster than to SSD. Therefore, from a performance perspective, it is always beneficial to replace SSD storage with NVM as the logging device.

In this work, we focus on the storage layout and therefore implement the same logging approach in each evaluated system. This makes it possible to compare only the advantages and disadvantages of the storage engine itself with less interference of other database components.

We use write ahead logging with redo and undo information. The undo entries enable one to perform rollback and to undo the effect of loser transactions during recovery. The redo entries are used to repeat the effects of committed transactions during recovery if it was not yet persistent. The buffer-manager-based systems keep a log sequence number per page to identify the state of the page during recovery. In the NVM direct approach, this is not necessary, as changes are always immediately written to NVM. Therefore, only in-flight transactions need to be undone and every committed transaction is durable already.

Logging for NVM-based systems can be and has been optimized in prior work [5, 36]. While each of the described storage architectures can benefit from more advanced logging techniques, we believe that the impact on the storage engine is largely orthogonal and the two problems can be treated independently.

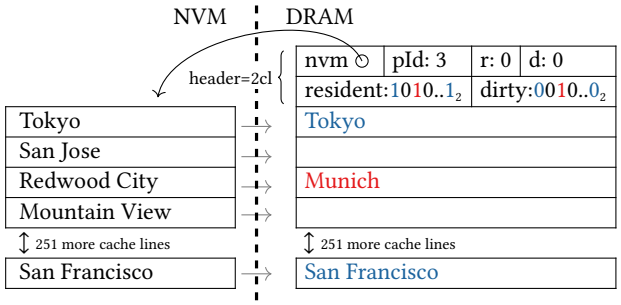


Figure 3: Cache-Line-Grained Pages – The bit masks indicate which cache lines are resident and which are dirty.

3 NVM BUFFER MANAGEMENT

The goal of our architectural blueprint is a system that performs almost as well as a main-memory database system on smaller data sets but scales across the NVM and SSD storage hierarchy while gracefully degrading in performance (cf. Figure 1). For this purpose, we design a novel DRAM-resident buffer manager that swaps cache-line-grained data objects between DRAM and NVM—thereby optimizing the bandwidth utilization by exploiting the byte addressability of NVM. As illustrated in Figure 2c, scaling beyond DRAM to SSD sizes led us to rely on traditional page-grained swapping between NVM and SSD. Between DRAM and NVM, we adaptively differentiate between full-page memory allocation and mini-page allocation to further optimize the DRAM utilization. This way, individual "hot" data objects resident on mostly "cold" pages are extracted via the cache-line-grained swapping into smaller memory frames. Only if the mini page overflows, is it transparently promoted to a full page—but it is still populated one cache-line at a time. We also devise a pointer swizzling scheme that optimizes the necessary page table indirection in order to achieve nearly the same performance as pure main-memory systems, which obviate any indirection but incur the memory wall problem once the database size exceeds DRAM capacity.

3.1 Cache-Line-Grained Pages

Compared to flash, the low NVM latency (hundreds of nanoseconds) makes it feasible to transfer individual cache lines instead of entire pages. Using this, our so-called *cache-line-grained pages* are able to extract only the hot data objects from an otherwise cold page. Thus, we preserve bandwidth and thereby increase performance. In the following, we discuss the details of this idea.

A single SSD access takes hundreds of micro seconds. It is therefore important to transfer large chunks (e.g., 16 kB) at a time in order to amortize the high latency. Therefore, a traditional buffer manager has to work in a page-grained fashion: When a transaction fixes a page, the entire page is loaded into DRAM and the data stored on it can be processed. Our buffer manager, in contrast, initially only allocates a page in DRAM without fully loading it from NVM. Upon a transaction's request for a certain memory region, the buffer manager retrieves the corresponding cache lines of the page (if not already loaded).

The page layout is illustrated in Figure 3. The cache-line-grained pages maintain a bit mask (labeled `resident`) to track which cache lines are already loaded. In the example, the first, third, and last cache line is loaded, as indicated by a bit set to 1 at the corresponding position in the bit mask. Similar to the `resident` bit mask, the `dirty` bit mask is used to track and write back dirty cache lines. The `r` and `d` bits indicate whether the entire page is resident and dirty, respectively. To allow for the loading of cache lines on demand, pages additionally store a pointer (`nvm`) to the underlying NVM page. With 16 kB pages, there are 256 cache lines and therefore the two bit masks require 32 byte each. Together with the remaining fields ($|\texttt{nvm}| + |\texttt{pId}| + |\texttt{r}| + |\texttt{d}| = (8 + 8 + 1 + 1)$ byte $= 18$ byte), the entire header ($(2 * 32 + 18)$byte $= 82$ byte) fits into 2 cache lines (128 byte) and thus incurs only a negligible space overhead of less than 0.8 %.

While this cache-line-grained design includes an extra branch on every access (to check the bit mask), it often reduces the amount of memory loaded from NVM into DRAM drastically: As an example, consider the leaf of a B-tree [6] where pairs of keys and values are stored in sorted order. Assuming a page size of 16 kB and a key and value size of 8 byte each, there are at most $16\,\text{kB} \div (8\,\text{byte} + 8\,\text{byte}) = 1024$ entries on a single page. A lookup operation only requires a binary search, which uses $log_2(1024) = 10$ cache lines at most. Therefore, our design only needs to access $64\,\text{byte} * 10 = 640\,\text{byte}$, instead of 16 kB. While this is already a huge difference, it can be even greater. In the case of the YCSB and TPC-C benchmarks, which we use in our evaluation (Section 5), we measured an average of 6.5 accessed cache lines per lookup.

A system allowing for cache-line-grained accesses is more difficult to program than a conventional page-based approach. The reason for this is that all data needs to be made resident explicitly before accessing it and marked dirty after modifying it. But working with a cache-line-granularity is optional and it is also possible to load and write back entire pages. Therefore, we only implement the operations, that provide the most benefit, in a cache-line-grained fashion: like lookup, insert, and delete. Other infrequent or complicated operations (like restructuring the B-tree) are implemented by loading and processing the full page (avoiding the residency checks). The overhead of checking the residency of every single cache line only pays off if we access a small number of cache lines. During scan operations or the traversal of inner nodes in a B-tree, many cache lines are accessed and cache-line-grained loading should therefore not be used.

3.2 Mini Pages

Cache-line-grained pages reduce the consumed bandwidth to a minimum by only loading those cache lines that are actually needed. However, the page layout described previously still consumes much more DRAM than necessary. In this section, we introduce a second, smaller page type called *mini page*, which reduces the wasted memory. Consider the B-tree leaf example from above: Merely 640 byte out of 16 kB are accessed, but the system still allocates the 16 kB (excluding the header) in DRAM for the page. This problem is known from traditional disk-based systems: An entire page is loaded and stored in DRAM even if only a single tuple is required, wasting valuable NVM bandwidth and DRAM capacity. In the following, we will use the term *full page* to refer to a traditional page, as it was introduced before. Note that both pages (mini and full) are able to use

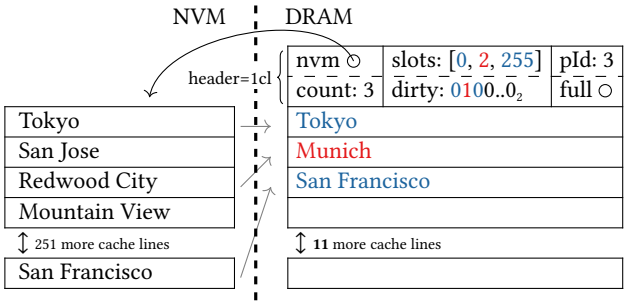


Figure 4: Mini Pages – The `slots` array indicates which cache lines are loaded (max 16). If promoted, `full` points to the full page.

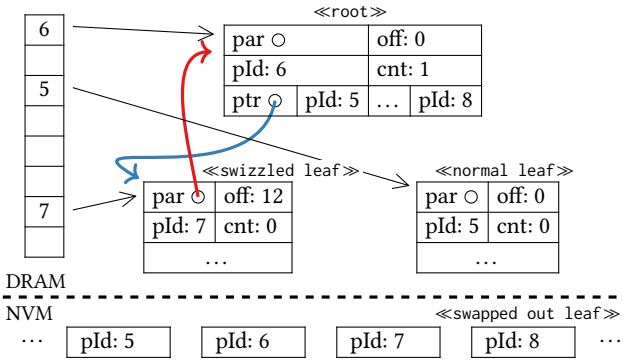


Figure 5: Pointer Swizzling – A B-tree with a root (pId: 6) and three child pages: A swizzled page (pId: 7), a normal DRAM page (pId: 5) and a page currently not in DRAM (pId: 8).

cache-line-grained loading to optimize bandwidth utilization—but not DRAM utilization. Hence, the term *cache-line-grained page* can refer either to a *mini page* or a *full page*.

The implementation of mini pages is illustrated in Figure 4. It consumes only 1088 byte of memory and is able to store up to 16 cache lines. The `slots` array implements the indirection. It stores the physical cache line id for each of the 16 cache line slots. For example, the cache line with the content "San Francisco" is located at the physical index 255 on the page but loaded into slot 2 on the mini page. Therefore, the `slots` array stores the index 255 at position 2. The array only requires 16 byte because each physical cache line id fits into one byte. In total, the mini page header fits into a single cache line: $|\texttt{nvm}| + |\texttt{slots}| + |\texttt{pId}| + |\texttt{count}| + |\texttt{dirty}| + |\texttt{full}| = (8 + 16 + 8 + 1 + 2 + 8)$ byte $= 43$ byte. This is a very low overhead (0.3 %) when compared to the size of a full page, which would be used in a system without mini pages. The `count` field indicates how many cache lines are loaded, e.g., a value of three means that the first three cache line slots of the mini page are occupied. The additional `dirty` bit mask indicates which cache lines must be written back to NVM when the page is evicted. In our example, the cache line "Redwood City" changed to "Munich" and needs to be written back.

Accessing memory on a mini page is more complicated than on a full page. Due to the mapping of cache lines, data members on the page can-not be directly accessed. Therefore, we use an abstract interface that enables transparent page access:

```
void* MakeResident(Page* p, int offset, int n)
```

The function takes a page `p` as an input and returns a pointer to the memory at the specified `offset` with a length of `n` bytes. In case `p` is a full page, the specified cache lines are loaded (if not yet resident) and a pointer to it is returned. Otherwise, in case of a mini page, the function searches the `slots` array for the requested cache lines. If these are not yet resident, they are loaded and added to the slots array. Afterwards, a pointer to the offset in the (now) resident cache line is returned. Thus, this basic interface transparently resolves the indirection within the mini pages. Compared to a traditional page, the only difference is that memory on mini pages can no longer be accessed directly but only via this function.

Mini pages need to guarantee that the memory returned by these functions is always contiguous, i.e., if more than one cache line is requested (e.g., cache lines with id 3 and 4), they need to be stored consecutively (i.e., 4 needs to be stored directly after 3) in the mini page's data array. Otherwise, the returned pointer would only be valid for the first cache line. To guarantee this, our implementation maintains the cache lines in sorted order (w.r.t. their memory locations). The overhead of maintaining order is small because, there are at most 16 elements and it is not on the critical path (only after loading from NVM). The benefit of this approach is that it simplifies implementation, as it avoids complicated reordering logic.

When a mini page does not have enough memory left to serve a request, it is promoted to a full page. To do this, an empty full page is allocated from the buffer manager and stored in the mini page's `full` member. Next, the current state of the mini page is copied into the newly allocated full page: all resident cache lines, the residency and dirty information. If the example mini page in Figure 4 was promoted, the newly initialized full page would look like the one in Figure 3. Finally, the page mapping table in the buffer manager is updated to point to the full page. From now on, the mini page is called *partially promoted* and all requests to the mini page are forwarded to the full page. It is only safely garbage collected, once the last transaction unfixes it. This is guaranteed to happen, as the page mapping table points to the full page and therefore no new references to the mini page are created. This feature is convenient for the data structures using mini page because this way its reference to the mini page is not invalidated when a promotion happens. Thus, a promotion is hidden from data structures and does not incur additional complexity.

3.3 Pointer Swizzling

While the buffer pool, allows the system to benefit from the low latency DRAM cache, it also introduces a non-neglectable overhead. In this section, we introduce pointer swizzling, a technique that reduces this overhead (mainly the page table lookup) by dynamically replacing page ids with physical pointers. In a traditional buffer manager (DRAM+SSD/HDD), this overhead is only noticeable if most of the working set fits into DRAM. Otherwise, the page has to be loaded from SSD/HDD anyway, which is orders of magnitude slower compared to the hash table lookup. In contrast to traditional buffer managers, in our proposed system, this overhead is also relevant for larger workloads. We cannot hide the overhead

behind even slower loads because (a) these loads are not that much slower (NVM latency is a lot lower than that of flash) and (b) the amount of data read is much less due to the cache-line-grained loading. Therefore, it is important for our system to minimize these management overheads as much as possible.

Pointer swizzling has recently been proposed in the context of traditional buffer managers (DRAM + SSD/HDD) [17]. The idea is to encode the address of the page directly in the page identifier for DRAM-resident pages. The most significant bit of the page identifier determines whether the remaining bits constitute a page identifier or a pointer. Thus, when accessing a page, the system first checks whether the most significant bit is set. If so, the remaining bits encode a pointer that can be dereferenced directly. Otherwise, the remaining bits are a page identifier and the system has to check the hash table in the buffer manager to find the page or load it from NVM if not present. This way, the hash table lookup is avoided for DRAM-resident pages and thereby the overhead is reduced to a single branch.

Figure 5 illustrates our implementation of pointer swizzling. On the left-hand side, the buffer manager's page mapping table is shown. It maps page identifiers (numbers in the table) to pages (represented by arrows). The example shows a B-tree with one `root` node (`pId` 6) and three leaf nodes: The first one (pId 7) is a `swizzled leaf`. The `root` can use a pointer (blue arrow) to access it instead of the page id. The second one (`pId` 5) is a `normal leaf` (not swizzled) and the third one (`pId` 8) is a `swapped out leaf`—currently not located in DRAM.

In the example, the root page has one swizzled child (as indicated by the `cnt`) field. A page with swizzled children can never be swapped out because the pointer to the swizzled child would be persisted. When a swizzled page (the left child with page identifier 7 in the example) is swapped out, it needs to update its parent (located via the `par` pointer): First, it decreases the child count, which is located at a fixed offset. Second, it converts the pointer pointing to itself back into a normal page identifier. The location of this pointer can be found using the offset field (`off`). The parent pointer (8 byte) and the offset (2 byte) require an additional 10 byte of space in the page header and therefore still fit into the mini page (1 cache line) and full page header (2 cache lines). Pointer swizzling is compatible with various data structures (trees, heaps, hashing); it only requires fix-sized pages and these additional header fields.

Consider a swizzled mini page. When a mini page is promoted to a full page, the swizzling information needs to be updated as well. This happens when the partially promoted mini page is unfixed. Until then it acts as a wrapper around the full page. When it is unfixed, the pointer in the parent page needs to be redirected to the full page. In addition, the pointer to the parent (`par`) and offset (`off`) in the mini page need to be copied to the full page.

4 THREE-TIER BUFFER MANAGEMENT

So far we have presented cache-line-grained loading, mini pages, and pointer swizzling as building blocks for an efficient DRAM buffer pool over an NVM storage layer. The next step towards our goal of building a storage engine that scales across the NVM and SSD hierarchy (cf. Figure 1) is to add flash (SSD) as a third layer. Such a three-tier design drastically increases the maximum workload

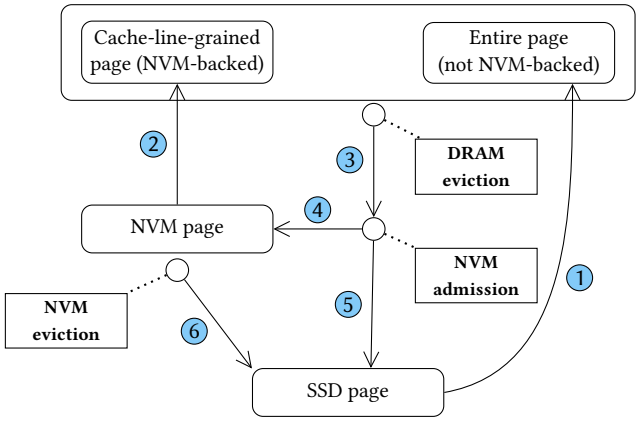


Figure 6: Page Life Cycle – There are five possible page transitions and the three critical decisions (DRAM eviction, NVM admission, and NVM eviction).

size in comparison to that of an in-memory or NVM-only system. This section describes the involved buffer replacement strategies and a low overhead way of adding this third layer.

Although adding support for SSDs does not improve performance, it is still important, as it allows for the management of larger data sets and can also be more economical: Real-world data is often hot/cold clustered (e.g., older data is accessed less frequently). To process the hot data as fast as possible, it should reside in main memory. But it is neither a good practice to keep the cold data in a separate system nor is it cheap to buy huge amounts of DRAM to obtain enough storage to keep the cold data in DRAM as well. Our layered approach solves this problem by providing close to main memory speed for the hot data (provided it fits into DRAM) while also supporting cheap SSD storage for cold data in a single system. Beyond other systems, it even allows one to compactify the individual working sets of an application via mini pages.

4.1 Design Outline

In our three-tier architecture, buffer management is done by using both NVM and DRAM as selective caches over the SSD storage layer (Section 4.2). Pages are only accessed (read and written) in DRAM, and write ahead logging (WAL) is used to ensure durability. When a page is evicted from DRAM, it is either admitted to NVM or written back to SSD, depending on how hot the page is.

To locate pages on NVM, a page table (similar to the one in DRAM) is required. To avoid overheads, this can be implemented by using a combined page table for both DRAM and NVM—reducing the number of table lookups from two to one (Section 4.3).

For recovery (Section 4.4), we propose using textbook-style write ahead logging and an ARIES-based restart procedure. In a three-tier architecture, it becomes necessary to reuse the content of NVM-resident pages to allow for faster restarts. Therefore, the content of the combined mapping table is reconstructed after a restart.

4.2 Replacement Strategies

A three-tier architecture needs to manage two buffer pools (DRAM and NVM) instead of one. In this section, we detail the page transitions that can occur and describe the three necessary replacement decisions: DRAM eviction, NVM eviction and NVM admission. The process is illustrated in Figure 6 and can be used as an overview.

Initially, all newly-allocated pages start out on SSD. When a transaction requests a page, it is loaded directly and completely into DRAM (1). The pages are loaded completely because the block-based device only allows for a block-based access. We do not put pages into NVM when they are loaded from SSD because accesses are served directly from DRAM and putting them into NVM as well would only waste NVM memory. Pages are only admitted to NVM when they are swapped out of DRAM. Pages loaded directly from SSD are not NVM-backed and therefore cannot operate in a cache-line-grained fashion. This is only possible when the page is loaded from NVM (2) and therefore NVM-backed.

When there are no more free slots available in DRAM, the buffer manager needs to evict any of the DRAM-resident pages (3) in order to make room for a new one. **DRAM eviction** is the first out of three decisions our buffer manager has to perform. The goal is simply to keep the hottest pages in DRAM. We use the well-known clock (or second chance) algorithm, which performs reasonably well in both overhead and quality. It continuously loops through all pages in the buffer pool and swaps those pages out that have not been touched since the previous iteration.

Once a page is chosen to be evicted from DRAM (3) and is not already resident in NVM, it is considered for **NVM admission**, which is the second out of three decisions. This decision is a more difficult one because the goal is to identify warm pages instead of hot pages. It has been studied in the context of the ARC replacement strategy [34], where two queues are used to identify warm pages in order to optimize the replacement of hot pages in a two-layer system. Building on this idea, we use one set, which we call the *admission set*, to identify recently accessed pages. The idea is to admit pages to NVM that were recently denied admission. Each time a page is considered for admission, the system checks whether the page is in the admission queue. If so, it is removed from the set and admitted into NVM (4). Otherwise, it is added to the set and remains only on SSD (5). By limiting the size of the admission set, we make sure that it only contains pages that were recently considered for admission. This way, pages that are frequently swapped out of DRAM are admitted into NVM, but those that are only loaded once do not pollute NVM.

The third replacement decision is **NVM eviction**, i.e., choosing a page to be swapped out of NVM (6) when a new page is admitted. To keep our implementation simple, we also use the clock algorithm for this decision; however, as this is a rather expensive operation (writing a page (16 kB) to NVM and, if it was dirty, to SSD), one could opt for a slower algorithm that in turn yields better quality.

4.3 Combined Page Table

For workloads fitting into NVM, the third layer (SSD) is not used and should therefore not cause an overhead. We achieve this by the use of a *combined page table*, which stores both mappings (page identifier ↦ DRAM location and page identifier ↦ NVM location) in one hash table. The resulting structure is shown in Figure 7. When retrieving a page, the memory address of the page can be used to determine whether it is located in DRAM or NVM. If a page is not found in DRAM, its NVM-resident copy is still found and can be directly used without an additional hash table lookup. Therefore, there are no extra steps involved compared to a two-layer system. However, the size of the hash table differs because of the additionally stored mapping for NVM pages. According to our experimental results, the introduced overhead is less than 5 %.

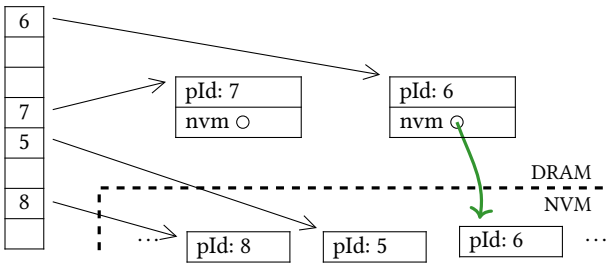


Figure 7: Single-Table Mapping – Using one hash table for DRAM and NVM-resident pages eliminates most overhead for managing the SSD layer. The hash table entries are identified by their location in memory (DRAM or NVM).

4.4 System Restart

The page mapping table is performance critical and is therefore stored in DRAM. After a restart, it needs to be rebuilt instead of recovering solely from SSD, which would have two major drawbacks: First, the time until the system can process the first transaction would be higher because more log entries have to be replayed, as the SSD version of the pages are older than the NVM version. Second, once the system is restarted, it takes a longer time to reach the pre-crash throughput again because not only the DRAM cache but also the NVM cache is empty. Therefore, our system reconstructs the page mapping table after a restart. This requires scanning over all NVM pages, reading their page identifiers and adding them to the DRAM-resident page table. This technique was not feasible for slow non random access mediums (like flash or HDD) but performs reasonably well for NVM. Reading the page identifiers for 100 GB of NVM takes slightly less than 1 second, but allows for a faster restart.

5 EVALUATION

In this section, we present an experimental analysis of our proposed architecture and compare it with other NVM management techniques. To provide a fair comparison, all evaluated architectures are implemented within the same storage engine. Consequently, all systems use the same logging scheme, B+-tree, and test driver. The only difference is the way DRAM, NVM, or/and SSD are accessed. This allows us to measure the difference between the architectures mostly independent of individual implementation choices.

5.1 Experimental Setup

We conduct our experiments on the Crystal Ridge Software Emulation Platform (SEP) provided by Intel [14]. It is a dual-socket

system equipped with Intel(R) Xeon(R) CPU E5-4620 v2 processors (2.6 GHz, 8 cores, and 20 MB L3 cache). This processor is extended to include the `clwb` instruction and is able to configure the NVM latency between 165 ns to 1800 ns (unless otherwise noted, 500 ns are used). The `clwb` instruction in combination with memory fences (`sfence`) is used to persist a certain cache line. Unlike the `clflush` instruction, it does not invalidate the cache line, thus triggering a reload on the next access, but only writes it back to the underlying memory and marks it as unmodified. We use the libpmem [1] library from the libpmem.io stack as a platform-independent wrapper around these instructions. The machine is equipped with 48 GB of DRAM, out of which 32 GB are connected to the first socket. To avoid NUMA effects, we conduct our experiments exclusively on this socket. The simulated NVM-DIMMs are exposed as a block device, which is formatted as an *ext4* file system and mounted with DAX (direct access) support enabled. This file system is then mapped into the address space of our process and can be directly accessed without file system overheads due to DAX. Note that buffer-managed systems do not require a special NVM allocator [38] because pointers into or the dynamic allocation of NVM are not used.

We implement each table as a B+-tree using C++ templates and 16 kB pages (page size is not restricted to the OS's virtual memory page size). The B-tree uses binary search and stores key and payloads in separate arrays (sorted by keys).

When ingesting data in preparation for a benchmark, the load factor of the B-tree is configured to 0.66. The term data size is used to describe the total memory consumption of the B-tree after loading the data. Therefore, if the flat data is 5 GB in size, the resulting data size would be around 7.5 GB due to the tree structure and the load factor. Transactions are executed on a single thread. By using no-steal and no-force in the buffer manager in combination with traditional write ahead logging (WAL), we ensure durability. In all experiments, the transactions and the code executing them are implemented in C++ and compiled together with our storage engine into one binary.

5.2 Workloads

We used YCSB and TPC-C in our experiments, which mostly use OLTP-style transactions. These are better suited to evaluate a storage engine, as they pose a greater challenge than OLAP-style full table scans.

YCSB is a popular key-value store benchmark framework [11]. It consists of a single table with a 4 byte primary key and 10 string fields of 100 byte each. YCSB defines simple "CRUD"-style operations on this table, which can be combined into workloads. In the YCSB experiments, we focus on point lookups, updates, and range scans (inserts and deletes are evaluated using TPC-C). We use Zipf-distributed ($z = 1$, non clustered popular keys) keys to model real-world data skew [20]. The corresponding row is located in the table and a (uniformly) randomly chosen field is read (lookup, range scan) or updated (updates). We define three workloads, which are generalizations of the five pre-defined example workloads (A-E) in YCSB.

YCSB-RO uses 100 % point lookup operations (same as YCSB "Workload C").

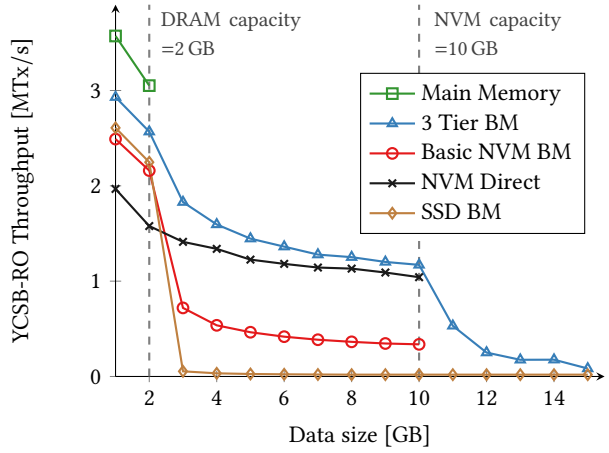


Figure 8: YCSB-RO – Performance for varying data sizes on read-only YCSB workload. The capacity of DRAM, NVM, and SSD is set to 2 GB, 10 GB, and 50 GB, respectively.

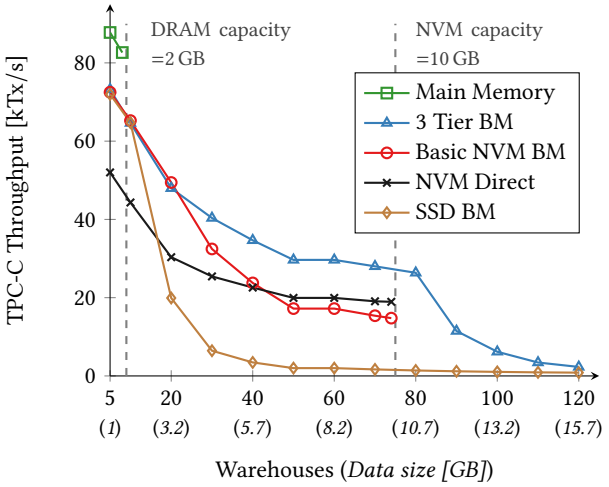


Figure 9: TPC-C – Performance in TPC-C for an increasing number of warehouses. The capacity of DRAM, NVM, and SSD is set to 2 GB, 10 GB, and 50 GB, respectively.

YCSB-R/W uses x% update and $(100 - x)$% point lookup operations, where x can be configured between 0 to 100 (configurable mix of "Workload A" and "Workload C").

YCSB-SCAN uses 100 % range scan operations. Each one has a random length between 1 and 100 (like "Workload E", but without the 5% inserts).

TPC-C is considered the industry standard for benchmarking transactional database systems. It is an insert-heavy workload that emulates a wholesale supplier. Like most research TPC-C implementations (e.g., [25, 42]), we do not implement think times.

5.3 Architecture Comparison

In this paper, we set out to design a system that performs well in all three layers (DRAM, NVM, and SSD) of next generation servers. This experiment compares the performance of different storage

engines with two workloads: YCSB-RO (in Figure 8) and TPC-C (in Figure 9). In both scenarios, we use 2 GB of DRAM, 10 GB of NVM and 50 GB of SSD. The horizontal axis increases the workload size by 1 GB at a time starting at 1 GB up to 15 GB.

The figures are divided into three areas by two dashed lines that show the capacity of DRAM and NVM. Therefore, the area on the left shows the performance for workloads fitting completely into DRAM; the one in the middle covers workloads fitting into NVM and the one on the right is for workloads exceeding NVM capacity. In the following, we describe the behavior of the different storage engines in these areas.

DRAM Area: For both workloads the main-memory variant (⊟) performs best (YCSB-RO with 3.6 MTx/s and TPC-C with 88 kTx/s). It is clear that it is impossible for any buffer-managed architecture (△, ○, ◇) to outperform a main memory system in this area. The overheads of fixing and unfixing pages and, in the case of our proposed system, checking the residency of individual cache lines can be minimized but never completely avoided. The basic buffer manager for DRAM and NVM (○) and the traditional buffer manager for DRAM and SSD (◇) have roughly the same throughput as our approach (△) in the DRAM area. The key differences related to performance are pointer swizzling and the cache-line-grained access. Mini pages do not play a role here, as they are rarely used (without the need to swap pages out, every page becomes a full page eventually). Our system needs to check whether a cache line is resident before accessing it. But, as the figure shows, the pointer swizzling speedup is higher than the cache-line-grained access slowdown. If the Basic NVM BM or the traditional one was extended to use pointer swizzling as well, they would come out slightly ahead. The NVM Direct system (⊸) performs worst in the first area because it does not use the fast DRAM but only the slower NVM.

NVM Area: In the NVM area, the line for the main memory system (⊟) vanishes, because such a system cannot support workloads exceeding the size of DRAM. The NVM Direct system (⊸) is not impacted by the fact that the workload no longer fits into DRAM, as it is not using DRAM. Its performance is decreasing because of the growing workload size and the fewer L3 cache hits. The two buffer management systems suffer a lot, as they have to start swapping pages in and out of SSD. The throughput of Basic NVM BM (○) and the SSD BM (◇) drop below that of NVM Direct because page misses are more likely (due to the lack of mini pages) and each page miss needs to retrieve an entire page (due to the lack of cache-line-grained pages). Our three-tier system (△) also drops in performance, but is still able to outperform the NVM Direct system due to the benefits of caching data in DRAM. The performance decrease is less severe in TPC-C than in YCSB-RO. This can be explained by the fact that the working set (the hot data) in TPC-C is only a portion of the entire data and can therefore be better cached in the buffer-managed systems.

SSD Area: In the last area, the NVM Direct system (⊸) and the Basic NVM BM (○) can no longer handle the large amounts of data and therefore no longer show up in the figure. Our system (△) has another performance drop, as it needs to load more and more data from SSD now. In the TPC-C experiment, this drop is delayed and does not occur right after the dashed line, as the hot data still fits into NVM at this point. Only when scaling the data size up to around 90 warehouses does it start accessing SSD and drops in performance. This drop is unavoidable because even a single SSD access (around 1 millisecond) every 26 transactions (number of transaction executed per 1 millisecond at 90 warehouses) will cut the transaction rate in half. On the very right of both figures (15 GB of data), the performance advantage of having NVM is still present. The SSD BM (◇) is a factor of 4.3 slower in YCSB-RO and 2.8 in TPC-C.

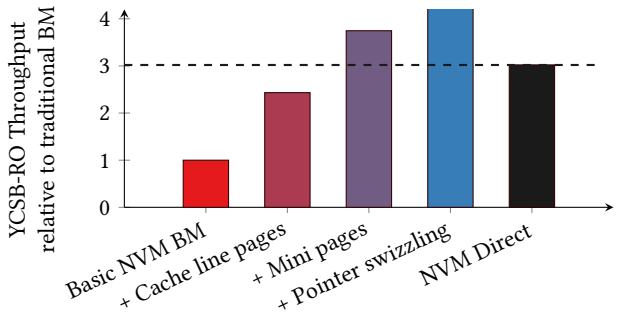


Figure 10: Performance Drill Down – Effect of proposed optimizations relative to a traditional buffer manager on NVM (YCSB-RO with 10 GB of data, read only, 2 GB DRAM, and 10 GB NVM).

These results show that a carefully engineered buffer manager can outperform an NVM direct system, be competitive with an in-memory system and greatly speed up workloads exceeding NVM capacity.

5.4 Performance Drill Down

Traditionally, buffer management is viewed as a technique with high overhead [21]. Therefore, working directly on NVM is, initially, a good idea. In this section, we show that by leveraging the novel properties of NVM (mainly byte addressability), it becomes possible to outperform an NVM in-place engine. We first break down the performance gains of the individual techniques proposed in our architecture and then analyze their overheads. In both experiments, we use 2 GB of DRAM, 10 GB of NVM, and around 6.5 million tuples, which amount to roughly 10 GB.

5.4.1 Performance Gains Breakdown. The benefits of the proposed techniques are shown using YCSB-RO in Figure 10. We first measure the performance of our system configured as a Basic NVM BM and then cumulatively enable the optimizations proposed in this paper (cache-line-grained pages, mini pages and pointer swizzling). The performance of the improvements is given relative to that of the Basic NVM BM. For comparison, we also added a line showing the performance of the NVM Direct engine.

As the first improvement, we add *cache-line-grained accesses*, as described in Section 3.1. It allows the buffer manager to load individual cache lines from NVM instead of having to load the entire page, which dramatically reduces the number of loaded cache lines by a factor of 55 from around 652.5 M to 11.8 M.

The next improvement are *mini pages*, which are detailed in Section 3.2. These pages can store less cache lines compared to a full page, but in turn also require less storage. This allows them to use the available DRAM more efficiently because, on many pages,

only a few cache lines are touched and thus more hot tuples are kept in DRAM. Using mini pages, the number of loaded cache lines is reduced by a factor of 2 and ends up at 5.6 M.

The last improvement, introduced in Section 3.3, is *pointer swizzling*. It essentially avoids the costly hash table lookup to map the page identifier to a page and replaces it with a single branch and a pointer chase for hot pages. Thus, lowering the overhead of the buffer manager indirection and making it more competitive with the architectures that do not require this indirection.

Overall, the experiment shows that it is possible to leverage the system's DRAM to increase the throughput by deploying a buffer manager. It also shows that it is necessary to specifically optimize buffer managers for NVM to achieve good performance on these new systems.

5.4.2 Overhead Analysis. On the other side, the proposed optimizations also have overheads associated with them. To show these CPU overheads, we measure YCSB-SCAN using a fill factor of 100%. We start with our base line, the "Basic NVM Buffer Manager" in the first row, and cumulatively add our optimizations showing their throughput relative to the baseline:

	Small Scan (range = 100)	Full Scan (range = |table|)
Basic NVM BM (100%)	50 000 scans/s	0.34 scans/s
+ Cache-Line-Grained	104.2 %	91.3 %
+ Mini Pages	93.1 %	90.8 %
+ Pointer Swizzling	93.8 %	90.9 %

While the base line implementation loads each page completely during the scan, the *cache-line-grained* one loads each tuple individually. Due to the perfect fill rate of leaf pages, almost no loads are avoided when all tuples on a page are needed. But, for small scans, cache-line-grained loading still benefits, as only around 3 pages (50 tuples) are touched and the pages on the edges of the range are not read completely. In the case of the full table scan, each page is loaded completely and the tracking of individual cache lines has no benefit and thus reduces the throughput.

The use of *mini pages* is almost never beneficial for scans because due to the large YCSB tuples, the mini pages are promoted as soon as more than 1 tuple is accessed. This happens frequently in both cases, therefore, the system suffers in throughput.

Lastly, the use of *pointer swizzling* has little effect on the performance of scans, as it is only used for finding the starting point of the scan. The only difference is an additional branch when fixing and unfixing a page during the scan and the increased memory consumption (16 byte of additional data in the buffer frames header).

This experiment shows that the proposed techniques incur a maximum overhead of around 10 %. Note, however, that this overhead can trivially be avoided using a hinting mechanism that selectively disables cache-line-grained and mini pages for full table scans.

5.5 Hybrid Structures

Using only NVM can result in sub-optimal performance due to its fairly high latency. Therefore, some NVM-optimized data structures incorporate DRAM into their design. One recent proposal of a hybrid data structure is the FPTree [39]. It is a B+-Tree that places its leaf nodes in NVM and its inner nodes in DRAM, thus gaining fast lookups (as inner node traversal is fast due to low DRAM latencies) and still being durable (as leaf nodes are in NVM and can be used to reconstruct inner nodes upon a restart).

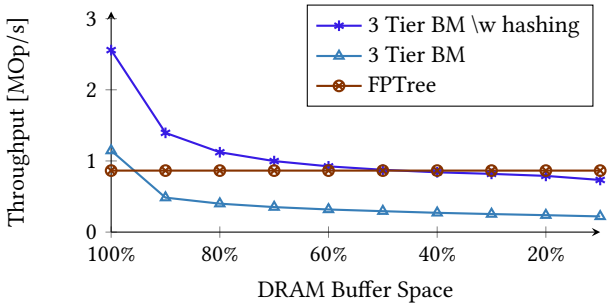


Figure 11: Hybrid DRAM-NVM Systems – Uniformly distributed lookup keys in tree with 100 M 8 byte keys values pairs.

In the experiment shown in Figure 11, we compare our approach with a reimplementation of the FPTree. For a fair comparison, we use the same experimental setup as the original FPTree paper: Uniformly-distributed point lookups in a single tree with 8 byte integer keys and values. As in the paper, the FPTree is configured to use 4096 entries in inner pages (64 kB) and 56 entries in leaf pages (960 B). Our tree uses a page size of 16 kB for all nodes (around 1000 entires). We use 100 M tuples, resulting in a tree size of roughly 2.5 GB in both systems. The horizontal axis depicts the percentage of pages that fit into DRAM for the buffer-managed systems.

The results show that our system (-△-) can only outperform the FPTree (-⊗-) when 100 % of the data fits into the DRAM cache. This is caused by a different leaf node layout in the trees: While our leaves use a sorted array of keys and binary search, the FPTree uses a hash-based leaf node layout ("fingerprints"). For point lookups, this layout allows the FPTree to reduce the number of NVM accesses in leaf nodes (from around 8 down to 2).

However, our proposed three-tier architecture is agnostic to the leaf node design. Therefore, we can optimize our leaves for point lookups by implementing a hashing structure (based on open addressing). The resulting system (-✳-) remains competitive with the FPTree, even with lower DRAM cache sizes. While a hashing layout performs well for point lookups, it introduces overheads for scans (the nodes need to be sorted just in time) and lower bound queries (nodes need to be scanned completely).

The most important advantage of a buffer-managed approach regarding performance is that it is able to adapt to skewed workloads. In this experiment (Figure 11), we measured a uniformly-distributed access pattern, which is the worst case for caching. When using a Zipf distribution ($z = 1$), the buffer-managed system is able to cache larger portions of the data with limited DRAM. This achieves a 30 % higher throughput at 50 % DRAM and only slightly drops beneath the FPTree line at 10 % DRAM.

5.6 Impact of NVM Characteristics

As of today, NVM is not commercially available as a byte addressable storage device. Therefore, there is still uncertainty about the

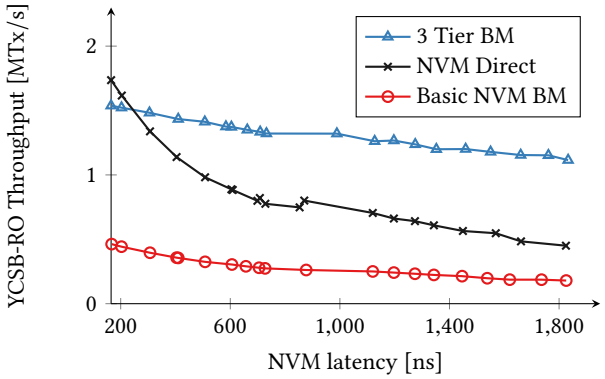


Figure 12: NVM Latency – The impact of varying NVM latencies on the YCSB-RO performance (YCSB with 10 GB of data, read only, 2 GB DRAM, and 10 GB NVM).

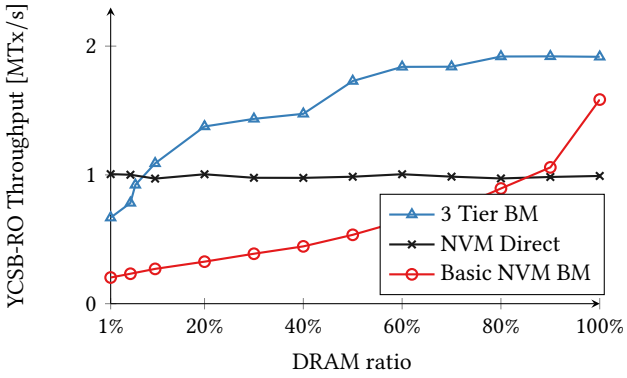


Figure 13: DRAM Buffer Size – YCSB-RO performance for varying amounts of DRAM and a fixed NVM capacity (YCSB with 10 GB of data, read only and 10 GB NVM).

exact characteristics of the hardware. In this section, we evaluate how the NVM latency and capacity impact the individual engines.

5.6.1 NMV Latency. We first investigate how the three systems perform under various NVM latencies. Our test platform allows us to vary the latency between 165 ns to 1800 ns. We use the YCSB-RO workload to determine in which areas the benefit of a buffer manager outweigh the incurred overheads. The data size is 10 GB, the NVM capacity is 10 GB, and the DRAM capacity is 2 GB. The results are shown in Figure 12. The vertical axis shows the throughput, while the horizontal one depicts the various NVM latencies.

At the lowest possible latency (around 165 ns), the NVM Direct system (–×–) is slightly faster than our NVM optimized buffer manager (–△–). At this point, the DRAM latency advantage is too marginal for a buffer manager to be beneficial. With increasing NVM latency, all systems become slower, but the buffer-managed ones are not as much impacted by the latency increase because they use the constantly fast DRAM as a cache. The NVM Direct system, on the other hand, loads all its data from the increasingly slow NVM. Starting at a latency of around 300 ns, the buffer manager outperforms the NVM Direct system.

The Basic NVM BM (–○–) also decreases in performance with an increasing NVM latency. The slope of the curve suggests that for even higher latencies, this non-optimized approach would also outperform the NVM Direct system. This is because the fast DRAM cache becomes more valuable as the latency difference between NVM and DRAM increases.

5.6.2 NVM Capacity. Besides latency, the exact capacity of NVM is another unknown parameter. In this experiment, we look at the ratio between the capacities of DRAM and NVM. The results are shown in Figure 13. The NVM capacity is fixed at 10 GB. The DRAM capacity is increased from 100 MB up to 10 GB (horizontal axis). Consequently, a value of 20 % implies a DRAM capacity of 2 GB.

The NVM Direct engine (–×–) is not impacted by the changing ratio between NVM and DRAM, as it is not using DRAM. The other two engines benefit from more DRAM. The Basic NVM BM (–○–), however, requires a lot more DRAM (around 80 % of NVM) to outperform the NVM Direct engine. The 3 Tier BM (–△–), on the other hand, outperforms the NVM Direct engine very early on (around 7 %). In addition, it is worth noting that starting at around 85 %, the 3 Tier BM does not require any NVM accesses anymore due to the mini pages. Thus, from this point on, the performance remains constant.

6 RELATED WORK

NVM will likely trigger a drastic redesign of existing database systems. In this section, we first analyze the state of the art for integrating NVM into database systems at the storage layer before discussing other aspects.

The SOFORT database engine [36, 40] proposes a copy-on-write architecture for NVM. All primary data is placed and modified directly in NVM, thus eliminating the need to load it into main memory after a restart. This way, SOFORT is able to achieve an almost instantaneous restart and can resume working at a pre-shutdown throughput instantly. The placement of secondary data, like indexes, can be considered a tuning parameter: Placing secondary data in main memory allows the user to increase system performance due to the lower latencies, but it also increases the restart time, as the data needs to be reconstructed.

For systems that modify primary data directly in NVM, one interesting question is the endurance of this storage technology. NVM has a limited endurance [35], i.e., a given NVM cell will fail after a certain number of write operations. Therefore, Arulraj et al. [4] compare three generic approaches for data management on an NVM-only architecture: in-place updates, copy-on-write and log-structured (LSM) system. Their experiments suggest that an in-place update architecture is usually best, as it delivers good throughput while minimizing the wear on the NVM hardware. This is not a coincidence, as these two goals (maximizing endurance and throughput) are in support of each other: By minimizing the number of accesses to NVM, one also improves performance, due to the, compared to DRAM, relatively high NVM latency.

FOEDUS [25] tries to take advantage of this finding: Modifying data directly on NVM has the advantage of fast restart times, but it suffers from a higher access latency, leaves the available DRAM unused, and wears out the NVM. FOEDUS therefore uses a

two-layered approach: data can reside either on NVM or DRAM. The idea is similar to classic disk-based systems: The memory is divided into fixed-size pages, which are loaded from NVM into a DRAM-resident buffer pool for read and write operations. In order to update the persistent state on NVM, FOEDUS periodically runs an asynchronous process that updates the NVM-resident snapshot using the databases log.

The SAP HANA in-memory database system integrates NVM by utilizing its "delta" and "main" storage separation [2]. The immutable and compressed bulk of the data ("main") is stored on NVM, while the updatable part ("delta"), which contains recent changes, remains in main memory. This simple approach nicely fits HANA's architecture, but is not applicable to most database systems. Another specific way of exploiting NVM is to use it as a cache for LSM-based storage [30].

Microsoft Siberia [15] is an approach for extending the capacity of main-memory database systems. By logging tuple accesses [31], infrequently-accessed tuples are identified and eventually migrated to "cold" storage [15] (e.g., SSD). This high-level concept could also be used to add support for NVM-based cold storage for main-memory systems.

There are also proposals to integrate persistency into NVM-resident data structures. However, performing in-place updates on NVM requires customized data structures to avoid data corruption [9]. Multiple NVM-specialized data structures, including CDDS Tree [43], HiKv [46], NV-Tree [47], FPTree [39], WORT [26], wBTree [10], and BzTree [3], have been proposed. These data structures optimize the node layout for NVM and explicitly manage persistency using appropriate cache write back instructions. In our design, in contrast, the data structure design is largely transparent to the storage engine (except for the requirement of fixed-size pages). Furthermore, since the storage engine takes care of persistency, write back instructions are inserted automatically.

In contrast to the approaches discussed above, we propose transparently integrating NVM into the memory hierarchy. While some systems use NVM mostly as a means to achieve durability or to extend the main memory capacity, in our approach, NVM is an integral part: We leverage not only the persistency but also the byte addressability by loading individual cache lines from NVM into DRAM. This way, we can deploy variable-size pages, which allow us to keep hot tuples in DRAM instead of hot pages.

Our three-layer architecture unites DRAM, NVM, and SSD into one transparent memory, thus enabling workloads that far exceed the capabilities of main-memory databases. We are, to the best of our knowledge, the first to study memory management in a database context for DRAM, NVM, and SSDs. Three-layer architectures have already been investigated [8, 12, 23, 32, 33] for different storage layers (namely: DRAM, SSD, HDD). However, the vastly different properties of NVM (low latency and byte addressability) compared to traditional durable storage technologies (SSD and HDD) requires a drastically different architecture. For instance, when loading data from SSD, it has to be done in a page-grained fashion. This is neither required nor the most efficient way of dealing with NVM.

While this paper focuses on storage, NVM also poses challenges and opportunities for other aspects, for example, for testing [37], memory allocation [38], and query processing [44]. Another component affected by NVM is logging/recovery [5, 16, 22, 36, 45]. Write-behind logging [5], for example, is a recovery protocol specifically designed for multi-version databases that use NVM as primary storage. On commit, all newly-created changes of a transaction (versions) are persisted—instead of only persisting the traditional write-ahead log. While our current implementation uses write-ahead-logging and single-version storage, it would also be possible to combine our storage engine with write-behind logging. We defer evaluating different logging and recovery schemes to future work.

The pmem.io library [1] is the de facto standard for managing NVM. It offers various abstraction levels, from low-level synchronization utilities (libpmem) to full-fledged transactional support (libpmemobj). Like all NVM-optimized database systems, we use the low-level primitives in order to have full control over persistency.

7 CONCLUSIONS

NVM will have a major impact on current hardware and software systems. We evaluated three approaches for integrating NVM into the storage layer of a database system: One that works directly on NVM, a FOEDUS-style buffer manager based on fixed-size pages, and our novel cache-line optimized storage engine. We found that by taking the byte addressability into account, it becomes possible to outperform the other two approaches while supporting large data sets on SSD as well.

This result is achieved using a number of techniques. While a traditional buffer manager loads entire pages, we use NVM's byte-addressability to load individual cache lines instead, reducing the transferred memory between DRAM and NVM enormously. Enabled by the cache-line-grained pages, we introduce mini pages, which store only a few cache lines but also use less memory. These pages use the limited DRAM capacity more efficiently. We also optimized our buffer manager for in-memory and in-non-volatile-memory workloads by using pointer swizzling. This enables us to be competitive with in-memory DBMSs and systems working directly on NVM. In summary, we ended up with a system that achieved a performance close to that of main-memory database systems if the workload fits into DRAM. At the same time, our system performs better than NVM-only systems if the workload fits into NVM and similar to disk-based performance for even larger workloads.

In isolation, each of these techniques is either well-known or fairly simple. The novelty comes from combining these ideas into a coherent and effective system design. We argue that conceptual simplicity is a major advantage, or in the words of Jim Gray:

> *"Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex don't."* [19]

ACKNOWLEDGMENTS

We would like to thank Dieter Kasper and Andreas Blümle for helping with the SEP system.

A APPENDIX

A.1 Multi Threading

In this paper, we compared 5 radically different storage system designs. To keep the implementation effort reasonable and the comparison fair, all implementations and experiments are single-threaded. However, given current hardware trends, any modern storage engine should efficiently support multi-threading. The FOEDUS [25] and LeanStore [27] projects have shown that page-based storage engines can efficiently be synchronized for modern multi-core CPUs. The two key techniques for achieving this are version-based latches that allow readers to proceed optimistically without physically acquiring latches [29], and epoch-based memory reclamation [42]. Another alternative for implementing synchronization is hardware transactional memory, which has been shown to be effective at synchronizing B-trees [24, 28, 39]. In the following, we discuss some additional synchronization aspects of our design. We assume the use of per-page, version-based latches (and do not rely on hardware transactional memory).

Cache-line-grained accesses may cause a read to physically change a page if the requested page is not yet resident. Therefore, such "cache line faults" make it necessary to upgrade the page latch to exclusive mode. Note that this only affects a single (leaf) page and is therefore unlikely to cause contention. Furthermore, frequently-accessed (hot) pages will generally be fully resident and will therefore not cause latch upgrades for read accesses. Another aspect that requires some care is the promotion of mini pages to a full pages, which becomes necessary once more than 16 cache lines have been accessed. Promotion is done by exclusively latching the mini page (source) and the full page (destination) before copying the data.

In traditional (textbook-style) buffer managers, the mapping table itself often becomes a synchronization bottleneck because all page accesses have to touch this data structure. In our design, in contrast, all frequently accessed pages will be swizzled. Accessing a hot page therefore does not require accessing the mapping table. Finally, regarding the replacement strategies, to achieve good scalability, decentralized algorithms like Second Chance should be preferred over centralized ones like LRU.

A.2 Scaling the Data Size

In this section, we evaluate the performance of the proposed system under larger workloads. The results of this experiment are shown in Figure 14. The vertical axis shows the throughput in million transactions per second. On the horizontal axis, we scale up the number of tuples and the capacity of the database proportionally. The axis shows the data size, which is the required space for the tuples once loaded into a B-tree. The DRAM capacity is set to a fifth of that of NVM.

Starting at the bottom, the Basic NVM BM (⊸) is dominated by the cost of reading entire pages from NVM. Therefore, the throughput is barely impacted by the increasing number of tuples or the larger control structures in the database.

The NVM Direct system (⊸) and the 3 Tier BM (⊸) both drop in performance as the workload size is increased. But this decrease is a lot more severe for the NVM Direct system (almost a factor of two). This can be explained by the decreasing ratio of hot tuples fitting into the L3 cache. Therefore, it is important to measure larger workloads when working with NVM, as the L3 cache can speed up small ones and hide the difference between DRAM and NVM.

Finally, there is a bump in the performance of the 3 Tier BM. Before it starts to drop at around 30 GB, the performance increases. This is an artifact of the Zipf distribution generator we used. The ratio of accessed pages that fit into the buffer pool increases up to 30 GB.

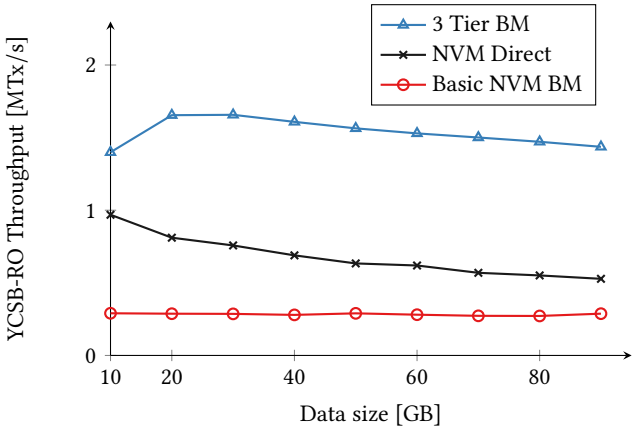


Figure 14: Large Workloads – YCSB-RO point lookup performance for large workload sizes. The NVM capacity is configured to match the data size, and the DRAM capacity is set to a fifth of that of NVM.

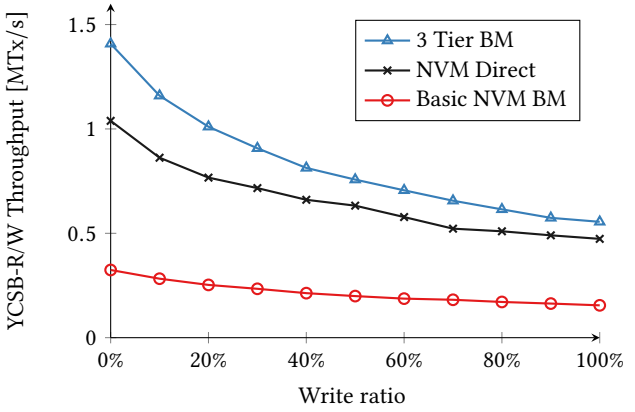


Figure 15: Update Performance – YCSB-R/W performance with an increasing amount of write transactions (YCSB with 10 GB of data, 2 GB DRAM, and 10 GB NVM).

A.3 Updates

We now analyze the impact of writes. To do this, we set up an experiment where we run YCSB with an increasing amount of update transactions. The results are shown in Figure 15. The horizontal axis shows the percentage of update transactions, while the vertical one depicts the throughput. We, again, measured the three competing systems: Our NVM-optimized buffer manager (⊸), the system working directly on NVM (⊸) and a basic buffer manager

for NVM (). The systems are configured to use 2 GB of DRAM and 10 GB of NVM. The size of the workload is 10 GB, just fitting into NVM.

With an increasing amount of write transactions, all systems degrade in performance, because more log entries and more tuple data needs to be written back to NVM. Compared to the read-only setting, the throughput of the Basic NVM BM is only half as high in the write-only case. The other two systems, NVM Direct and our NVM-optimized buffer manager, still outperform the Basic NVM BM in every setting, but also drop by a similar factor in performance. The experiment shows that our system is as stable as the other systems under a write-heavy workload. In addition, consider that the figure shows a rather unfavorable configuration. Our system would benefit from a change of the workload size in either direction: On the one hand, with a smaller workload, the ratio of tuples fitting into DRAM becomes higher and therefore, the buffer manager faster. On the other hand, with a larger workload, the NVM direct system could not run at all because it would not fit into NVM any more.

A.4 Endurance and Wear

In the previous experiments, our system has demonstrated a large performance benefit for workloads fitting in DRAM and has allowed for workloads larger than the capacity of NVM. We also showed that with write-heavy workloads, the performance remains competitive. In this section, we show another important advantage of using the buffer manager instead of working directly on NVM.

NVM has a limited endurance and therefore wears out and eventually fails. By using a buffer manager, the life-time of NVM can be greatly increased. To back up this hypothesis, we added counters that measure the number of writes to each individual NVM cache line. The results are shown in Figure 16.

The vertical axis shows the number of writes for each cache line. The cache lines are ordered by the number of writes and displayed on the horizontal axis. Both axes are logarithmic to better visualize the data. In the experiment, we compare a write only run of YCSB with 10 million transactions. As in the previous experiments, the data size is 10 GB, the NVM capacity 10 GB and the capacity of DRAM 2 GB.

The figure shows two advantages of our proposed system () compared to the NVM Direct system (). First, the total number of writes to NVM is reduced down to 4.7 M from 25 M. Second, and even more importantly, these write operations are spread out a lot more evenly. While the NVM Direct system has several cache lines that are written to 60 K times, the cache lines written to most with the buffer-managed approach are written to 3 times. The reason for this can easily be explained: The buffer manager caches pages that are frequently accessed in DRAM to increase performance. As a nice side effect, this also prevents many writes to these cache lines in NVM.

A.5 Restart Time

One major advantage of NVM is fast recovery and restart time. In our implementation, a write ahead log (WAL) is written to NVM. The creation of a WAL has two advantages: First, high availability is extremely important in a production environment and usually

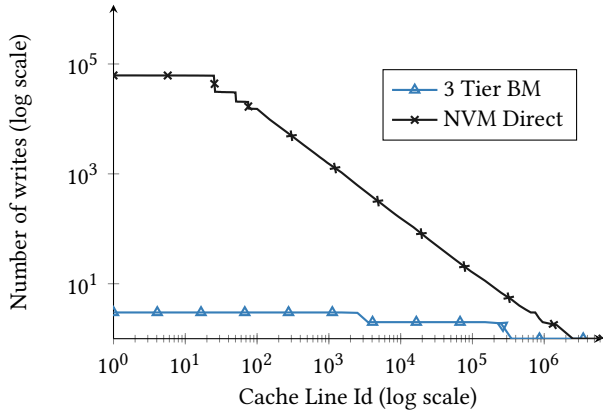


Figure 16: NVM Wear – The sorted number of writes for each cache line (YCSB-R/W (x = 100) with 10 GB of data, 2 GB DRAM, and 10 GB NVM).

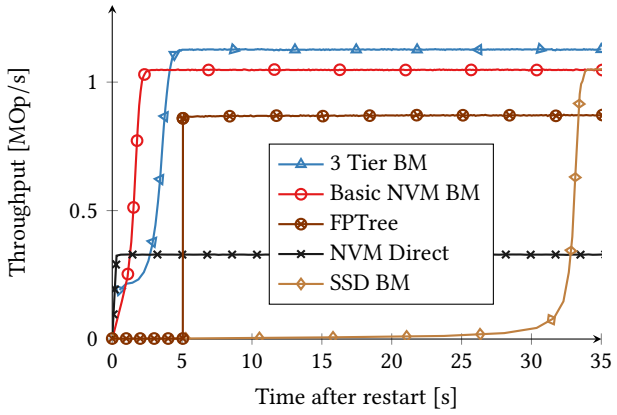


Figure 17: Restart Time – Ramp-up phase for uniformly distributed lookups with 100 M 8 byte key/8 byte value pairs. The entire workload fits into the buffer pool.

achieved with hot standbys in the event the primary system goes down. To keep the standby system up to date, it is necessary to supply it with a stream of changes from the primary. Second, (in comparison with hybrid and NVM-only systems) a write ahead log bundles many small writes into one large sequential write at the end of the transaction. This is beneficial considering the limited endurance and asymmetric write latency of NVM.

In the experiment shown in Figure 17, we compare the throughput immediately after a clean restart (until peak throughput is reached). In the "NVM direct" system (), the durable storage and the working memory are the same (both NVM). Therefore, these systems achieve a very fast, almost instantaneous, restart, because nothing has to be loaded explicitly (except for warming the CPU caches). FPTree (), in contrast, must reconstruct its inner pages by scanning all leaf nodes, which takes about 5 seconds in our experiment. Once the reconstruction is completed, FPTree reaches pre-restart throughput almost instantaneously. Traditional buffer managed systems () can begin processing transactions immediately after a restart, but they suffer in performance due to

a cold buffer cache and high SSD latencies. The basic NVM buffer manager (-o-) is similar, but recovers much faster, as it can fill its buffer cache from NVM instead of SSD. Our three-tier architecture (-▲-) needs to perform the reconstruction of the combined page table (cf. Section 4.3), which is quite fast, however (around 200 ms). It reaches peak performance slightly more slowly than the basic NVM BM due to mini pages, which only get lazily promoted.

A.6 Debugging Cache-Line-Grained Access

To easily detect usage failures in mini pages, we developed a debugging mode that checks both reads and writes. For reads, all cache lines within the page are marked as uninitialized memory when it is fixed. A memory checking tool (e.g., valgrind) can then be used to detect invalid reads. To detect invalid writes, the page initializes all cache lines to a specific ("magic") sequence. When it is unfixed, it validates that only those cache lines that are marked as dirty have been changed (false positives are possible but very unlikely).

REFERENCES

[1] Persistent memory programming. http://www.pmem.io. Accessed: 2018-02-13.
[2] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA adoption of non-volatile memory. *PVLDB*, 10(12):1754–1765, 2017.
[3] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5):553–565, 2018.
[4] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722, 2015.
[5] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.
[6] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, pages 107–141, 1970.
[7] P. Bonnet. What's up with the storage hierarchy? In *CIDR*, 2017.
[8] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.
[9] A. Chatzistergiou, M. Cintra, and S. Viglas. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.
[10] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.
[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.
[12] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, pages 1113–1124, 2011.
[13] S. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.
[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.
[15] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.
[16] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE*, pages 1221–1231, 2011.
[17] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-memory performance for Big Data. *PVLDB*, 8(1):37–48, 2014.
[18] J. Gray and G. R. Putzolu. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD*, pages 395–398, 1987.
[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[20] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.
[21] 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.
[22] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4):389–400, 2014.
[23] W. Kang, S. Lee, and B. Moon. Flash as cache extension for online transactional workloads. *VLDB Journal*, 25(5):673–694, 2016.
[24] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel transactional synchronization extensions. In *HPCA*, 2014.
[25] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, pages 691–706, 2015.
[26] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: write optimal radix tree for persistent memory storage systems. In *FAST*, pages 257–270, 2017.
[27] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-memory data management beyond main memory. In *ICDE*, 2018.
[28] V. Leis, A. Kemper, and T. Neumann. Scaling HTM-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2):297–310, 2016.
[29] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *DaMoN*, 2016.
[30] L. Lersch, I. Oukid, W. Lehner, and I. Schreter. An analysis of LSM caching in NVRAM. In *DaMoN*, 2017.
[31] J. J. Levandoski, P. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.
[32] X. Liu and K. Salem. Hybrid storage management for database systems. *PVLDB*, 6(8):541–552, 2013.
[33] T. Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *PVLDB*, 5(10):1076–1087, 2012.
[34] N. Megiddo and D. S. Modha. ARC: a self-tuning, low overhead replacement cache. In *FAST*, 2003.
[35] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1537–1550, 2016.
[36] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN*, 2014.
[37] I. Oukid, D. Booss, A. Lespinasse, and W. Lehner. On testing persistent-memory-based software. In *DaMoN*, 2016.
[38] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. In *VLDB*, 2017.
[39] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *SIGMOD*, pages 371–386, 2016.
[40] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.
[41] M. Stonebraker. How hardware drives the shape of databases to come. https://www.nextplatform.com/2017/08/15/hardware-drives-shape-databases-come/. Accessed: 2017-11-02.
[42] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
[43] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.
[44] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5):413–424, 2014.
[45] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.
[46] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for DRAM-NVM memory systems. In *USENIX ATC*, pages 349–362, 2017.
[47] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, pages 167–181, 2015.