Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems

Ismail Oukid
TU Dresden & SAP SE
ismail.oukid@sap.com

Daniel Booss
SAP SE
daniel.booss@sap.com

Adrien Lespinasse
Independent
adrien.lespinasse@gmail.com

Wolfgang Lehner
TU Dresden
wolfgang.lehner@tu-dresden.de

Thomas Willhalm
Intel Deutschland GmbH
thomas.willhalm@intel.com

Grégoire Gomes
Grenoble INP
gregoire.gomes@gmail.com

ABSTRACT

Storage Class Memory (SCM) is a novel class of memory technologies that promise to revolutionize database architectures. SCM is byte-addressable and exhibits latencies similar to those of DRAM, while being non-volatile. Hence, SCM could replace both main memory and storage, enabling a novel single-level database architecture without the traditional I/O bottleneck. Fail-safe persistent SCM allocation can be considered *conditio sine qua non* for enabling this novel architecture paradigm for database management systems. In this paper we present PAllocator, a fail-safe persistent SCM allocator whose design emphasizes high concurrency and capacity scalability. Contrary to previous works, PAllocator thoroughly addresses the important challenge of persistent memory fragmentation by implementing an efficient defragmentation algorithm. We show that PAllocator outperforms state-of-the-art persistent allocators by up to one order of magnitude, both in operation throughput and recovery time, and enables up to 2.39x higher operation throughput on a persistent B-Tree.

1. INTRODUCTION

SCM is a group of emerging memory technologies that promise to combine the low latency and byte-addressability of DRAM, with the density, non-volatility, and economic characteristics of traditional storage media. Most SCM technologies exhibit asymmetric latencies, with writes being noticeably slower than reads. Table 1 summarizes current characteristics of two SCM candidates, Phase Change Memory (PCM) [16] and Spin Transfer Torque RAM (STT-RAM) [14], and compares them with current memory technologies. Like flash memory, SCM supports a limited number of writes, yet, some SCM candidates, from a material perspective, promise to be as enduring as DRAM. These candidates also promise to feature even lower latencies than DRAM. However, while its manufacturing technology matures, we expect the first few generations of SCM to exhibit higher latencies than DRAM, especially for writes. Other promising SCM candidates that were subject to industry announcements include Intel and Micron's 3D XPoint, Resistive RAM (RRAM) [13], and HP's Memristors [26]. Given its non-volatility, idle SCM memory cells do not consume energy, in contrast to DRAM cells that constantly consume energy to maintain their state. Consequently, SCM has the potential to drastically reduce energy consumption. Given its byte-addressability and low latency, data in SCM can be accessed via load and store semantics through the CPU caches, without buffering it in DRAM. Current file systems, such as *ext4* with Direct Access (DAX), already support this access method by offering zero-copy memory mapping that bypasses DRAM, offering direct SCM access to the application layer.

Table 1: Comparison of current memory technologies with SCM candidates [18].

Parameter	NAND	DRAM	PCM	STT-RAM
Read Latency	25 µs	50 ns	50 ns	10 ns
Write Latency	500 µs	50 ns	500 ns	50 ns
Byte-addressable	No	Yes	Yes	Yes
Endurance	10^4–10^5	$>10^{15}$	10^8–10^9	$>10^{15}$

Therefore, SCM can be architected as universal memory, i.e., as main memory and storage at the same time. This architecture enables a novel single-level database architecture that is able to scale to much larger main memory capacities while removing the traditional I/O bottleneck. Databases that implement this paradigm start to emerge, such as Peloton [3], FOEDUS [15], and our database SOFORT [22]. Persistent memory allocation is a fundamental building block for enabling this novel database architecture. In this paper we present SOFORT's memory management component, called PAllocator, a highly scalable, fail-safe, and persistent allocator for SCM, specifically designed for databases that require very large main memory capacities. PAllocator uses internally two different allocators: SmallPAllocator, a small block persistent allocator that implements a segregated-fit strategy; and BigPAllocator, a big block persistent allocator that implements a best-fit strategy and uses hybrid SCM-DRAM trees to persist and index its metadata. The use of hybrid trees enables PAllocator to also offer a fast recovery mechanism. Besides, PAllocator addresses fragmentation in persistent memory, which we argue is an important challenge, and implements an efficient defragmentation algorithm that is able to reclaim the memory of fragmented blocks by

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/07.

leveraging the hole punching feature of sparse files. To the best of our knowledge, PAllocator is the first SCM allocator that proposes a transparent defragmentation algorithm as a core component for SCM-based database systems. Our evaluation shows that PAllocator improves on state-of-the-art persistent allocators by up to two orders of magnitude in operation throughput, and by up to three orders of magnitude in recovery time. Besides, we integrate PAllocator and a state-of-the-art persistent allocator in a persistent B-Tree, and show that PAllocator enables up to 2.39x better operation throughput than its counterpart.

The rest of this paper is organized as follows: We contribute in Section 2 an analysis of the challenges posed by SCM that we drew from our experience in designing and implementing SOFORT. Then, Section 3 discusses related work, with a focus on state-of-the-art persistent allocators. Thereafter, Section 4 presents our PAllocator, its different components, its defragmentation algorithm, and its recovery mechanism. Section 5 presents a detailed performance evaluation of PAllocator against state-of-the-art SCM allocators. Finally, Section 6 concludes this paper.

2. PERSISTENT MEMORY CHALLENGES

With the unique opportunities brought by SCM comes a set of novel programming challenges, from which we identify: (1) data consistency; (2) data recovery; (3) persistent memory leaks; (4) partial writes; (5) persistent memory fragmentation; and (6) virtual address space fragmentation. We detail these challenges in the following.

Data consistency

SCM is managed using an SCM-aware file system that grants the application layer direct access to SCM through memory mapping. This enables CPUs to access SCM directly with load and store semantics. The path from SCM to CPU registers is long and mostly volatile, as illustrated in Figure 1. It includes store buffers, CPU caches, and the memory controller buffers, over all of which software has little to no control. Additionally, modern CPUs implement complex out-of-order execution and either partial store ordering (Intel x86) or relaxed-memory ordering (IBM PowerPC). Consequently, memory stores need to be explicitly ordered and persisted to ensure consistency. Current x86 CPUs provide the CLFLUSH, MFENCE, SFENCE, and non-temporal store instructions to enforce memory ordering and data durability. Additionally, CLFLUSHOPT and CLWB have been announced for future platforms [1]. CLFLUSH evicts synchronously a cache line and writes it back to memory. Thus, data needs to be aligned to cache-line boundaries to avoid accidentally evicting falsely shared data. SFENCE is a memory barrier that serializes all pending stores, while MFENCE serializes both pending loads and stores. Non-temporal stores bypass the cache by writing to a special buffer, which is evicted either when it is full, or when an SFENCE is issued. CLFLUSHOPT is the asynchronous version of CLFLUSH and requires an MFENCE to be serialized. Finally, CLWB writes back a cache line to memory without evicting it, which is beneficial when data is accessed shortly after it is persisted.

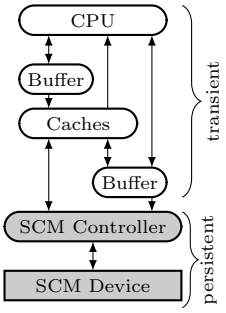

Figure 1: Volatility chain in x86-like CPUs.

Until recently the memory controller buffers were considered to be part of the volatility chain. Since then Intel has announced support for Asynchronous DRAM Self-Refresh (ADR) in all platforms that will support persistent memory[1]. ADR protects data still pending in memory controller buffers from power failures using capacitors. Hence, it is safe to assume that a cache line flush guarantees persistence.

Data recovery

When a program restarts, it loses its previous address space, invalidating any stored virtual pointers. Hence, there is a need to devise ways of discovering and recovering data stored in SCM. Using a file system on top of SCM provides a way of discovering data after a restart.

Reads and writes to a file created and memory mapped by an SCM-aware file system are made with direct load and store instructions. Hence, SCM-aware file systems should not have a negative performance impact on the application. A state-of-the-art technique to recover data is using persistent pointers in the form of a file ID and an offset relative to that file [27]; we follow this approach in this paper.

Persistent memory leaks

Memory leaks pose a greater problem with persistent memory than with volatile memory: they are *persistent*. Besides, persistent memory faces a new class of memory leaks resulting from software failures. To illustrate this problem, consider the example of a linked-list insertion. If a crash occurs after a new node was allocated but before it was linked to the previous node, the persistent allocator will remember the allocation while the data structure will not, leading to a persistent memory leak. We elaborate on our approach to avoid memory leaks in Section 4.2 .

Partial writes

We define a *p-atomic* store as one that executes in a single CPU cycle; that is, a store that is immune to partial writes. Current x86 CPUs support only 8-byte p-atomic stores; larger write operations are prone to partial writes since the CPU can speculatively evict a cache line at any moment. For instance, if a thread is writing a 64-byte cache-line-aligned string, it might write 16 bytes, then get descheduled. Meanwhile the CPU might evict the cache line where the string resides, persisting the written first 16 bytes. A failure at this time will corrupt the string in SCM. A common way of addressing this problem is using flags that can be written p-atomically to indicate whether a larger write operation has completed.

Persistent memory fragmentation

Persistent memory allocations have a longer lifespan than transient ones, and therefore have more impact on the overall application. While a restart remains a valid, but last-resort way of defragmenting volatile memory, it is not effective in the case of persistent memory. This is a similar problem to that of file systems. However, file system defragmentation solutions cannot be applied to SCM, because file systems have an additional indirection step: they use virtual memory mappings and buffer pages in DRAM, which enables them to transparently move physical pages around to defragment memory. In contrast, persistent memory mappings give direct physical memory access to the application layer without buffering in DRAM. Hence, persistent memory cannot be transparently moved as it is bound to its memory

[1] https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction

mapping. As a consequence, we argue that fragmentation avoidance is a core requirement for any persistent memory allocator. Within our PAllocator solution we propose a defragmentation algorithm, detailed in Section 4.6.

Address space fragmentation

Given the existence of systems with tens of TBs of main memory, and given the currently limited amount of address space supported by both software and hardware, we anticipate that the larger main memory capacities SCM enables will pose the unprecedented challenge of address space fragmentation. Indeed, SCM enables main memory capacities of hundreds of TBs. Current Intel x86 processors use 48 bits for address space, which amounts to a maximum capacity of 256 TB. However, Linux uses only 47 bits as one bit is reserved for the kernel. Hence, Linux currently supports up to 128 TB of address space. Two challenges arise: this capacity will not be sufficient for next-generation servers that use SCM, and the amount of physical memory approaches the limits of available virtual address space making the latter prone to fragmentation. Hence, memory mapping a file that resides in SCM might fail because of the lack of a large-enough contiguous *virtual* memory region. To remedy this issue, starting from Linux kernel v4.12, the page table will be extended to 5 levels instead of 4, enabling support for 128 PB of virtual address space [9].

From all these challenges, we derive the following requirements for persistent allocators: provide data discovery and recovery mechanisms, prevent persistent memory leaks, and minimize fragmentation, or even better, offer a defragmentation mechanism, all of which are fulfilled by our PAllocator.

3. RELATED WORK

Among early works on managing SCM, Condit et al. [8] proposed BPFS, a high performance transactional file system that runs on top of SCM. Since then, many other SCM-aware file systems have been proposed, such as SCMFS [28], PMFS [11], NOVA [29], and HiNFS [20]. While these works focus on managing SCM from a storage perspective, we focus on managing SCM on top of a file system in a main-memory-like fashion.

To address SCM challenges, Volos et al. [25] proposed Mnemosyne, a collection of libraries to program SCM that require kernel modifications and compiler support. Mnemosyne uses an SCM-aware version of Hoard [4] for small block allocations, and a transactional version of *dlmalloc*[2] for large allocations. Following a similar approach, Coburn et al. [7] proposed NVHeap, a persistent heap that implements garbage collection through reference counting. Later, Chatzistergiou et al. [6] proposed REWIND, a log-based user-mode library, targeted at database systems, that manages persistent data structures in SCM in a recoverable state. While all these approaches are laudable, we argue that transactional-memory-like approaches suffer from additional overhead due to systematic logging of modified data, which is amplified by the higher latency of SCM.

Moraru et al. [19] propose NVMalloc, a general purpose SCM allocator whose design emphasizes wear-leveling and memory protection against erroneous writes. The authors propose to extend CPU caches with line counters to track which lines have been flushed. Yu et al. [30] proposed WAlloc, a persistent memory allocator optimized for wear leveling. In contrast to NVMalloc and WAlloc, we focus on performance and defragmentation, and ignore wear-leveling which we envision will be addressed at the hardware level. Indeed, several works, such as [23], have already proposed efficient, hardware-based wear-leveling techniques to increase the lifetime of SCM.

Schwalb et al. [24] propose nvm_malloc, a general purpose SCM allocator based on jemalloc[3]. It uses a three-step allocation strategy, first proposed by *libpmem* from NVML, namely reserving memory, initializing it, and then activating it. nvm_malloc creates a single, dynamically resizable pool and uses link pointers, which represent offsets within the pool, to track objects. nvm_malloc uses a segregated-fit algorithm for blocks smaller than 2 KB, and a best-fit algorithm for larger blocks.

NVML [2] is an open-source, actively developed collection of persistent memory libraries from Intel. The most relevant library to our work is *libpmemobj*, which provides, among other features, an advanced fail-safe memory allocator, which is the only one, besides our PAllocator, to account for fragmentation, a very important challenge as explained in Section 2. It uses a best-fit algorithm for memory allocations larger than 256 KB, and a segregated-fit algorithm with 35 size classes for smaller allocations. In the latter case, to minimize fragmentation, a chunk of 256 KB is divided into blocks of 8× the class size, which are then inserted into a tree. Hence, each chunk can service allocations of up to 8× their class size. Nevertheless, in contrast to our PAllocator, NVML does not have a defragmentation mechanism. NVML handles concurrency by maintaining thread local caches.

More recently, Bhandari et al. [5] presented Makalu, a fail-safe persistent memory allocator that relies on offline garbage collection to relax metadata persistence constraints, resulting in faster small-block allocations, and enabling Makalu to catch persistent memory leaks that stem from programming errors. Makalu's allocation scheme is similar to that of nvm_malloc, but differs in that it enforces the persistence of only a minimal set of metadata, and reconstructs the rest during recovery. Potential inconsistencies that might arise during a failure are cured using garbage collection during recovery. Makalu relies on normal volatile pointers, and keeps them valid across restarts by memory mapping its pool at a fixed address (using the MAP_FIXED flag of *mmap*). However, besides being a security issue, fixed-address mappings will unmap any objects that are mapped in the requested range. Finally, garbage collection can limit certain functionalities of unmanaged-memory languages, such as C++, that are usually used for building database systems.

To the best of our knowledge, only Mnemosyne, NVML, nvm_malloc, Makalu, and NVMalloc are publicly available. We compare the performance of PAllocator against NVML, Makalu, and nvm_malloc; we exclude NVMalloc since it does not provide a recovery mechanism (see Section 5 for further details), and Mnemosyne since its default allocator is outperformed by Makalu [5].

4. PALLOCATOR

In this section we present in detail our proposed PAllocator.

4.1 Design goals and decisions

We identify the following design goals for persistent memory allocators tailored for large-scale SCM-based systems:

- The ability to adapt to changes in memory resources; This is particularly important in a cloud environment.

[2] http://g.oswego.edu/dl/html/malloc.html

[3] http://jemalloc.net/

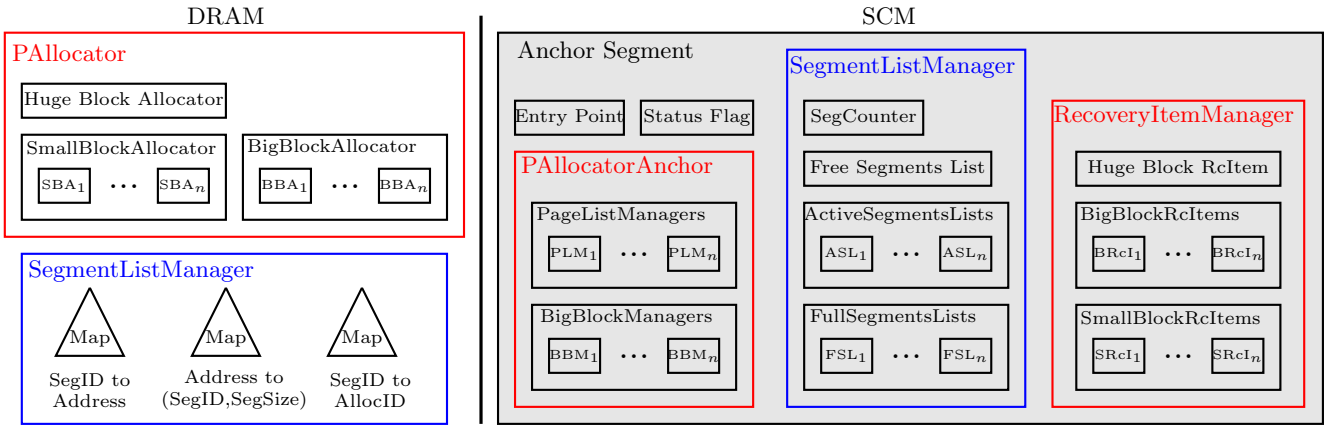

Figure 2: Architecture overview of `PAllocator`.

- High concurrency scalability; Large main-memory systems usually run on multi-socket systems with up to 1000 cores. Thus, it is important for the persistent allocator to provide robust and scalable performance.
- Provide robust performance for all sizes of allocations, as database-system allocation sizes cover a wide range, from a few bytes to hundreds of gigabytes.
- Fast recovery; Currently there are instances of single-node main-memory database systems such as SAP HANA [12] with up to 48 `TB` of main memory. With SCM this capacity will quickly exceed 100 `TB`. Thus, the persistent allocator must exhibit fast recovery and should not rely on scanning memory to recover its metadata.
- Defragmentation ability; Database systems run for a very long time, much longer than general-purpose applications, making fragmentation much more likely to happen.

So far, state-of-the-art persistent allocators, such as `NVML`, `nvm_malloc`, and `Makalu` have been engineered as general-purpose allocators, taking inspiration from existing general-purpose transient allocators. We argue that they are unfit for large-scale SCM-based database systems because:

- They all use a single pool (file), which is difficult to both grow and shrink (to the best of our knowledge, none of them can grow or shrink their pool beyond its initial size).
- They put an emphasis on the scalability of small-block allocations (from a few bytes up to a few kilobytes), and neglect that of middle-sized and large-block allocations.
- They do not provide defragmentation capabilities.
- They often rely on scanning memory to recover their metadata during recovery.

`PAllocator` is not a general-purpose allocator. It fulfills the above design goals following radically different design decisions than state-of-the-art persistent allocators:

- We use multiple files instead of a single pool, which allows us to easily grow and shrink our pool of persistent memory.
- We use large files to avoid having a large number of them which would hit the limitations of current file systems.
- We use three different allocation strategies for small, big, and huge allocation sizes, mostly independent from each other, to ensure robust performance for all allocation sizes.
- We aggressively cache free memory by not removing free files. Instead, we keep them to speed up future allocations. This is acceptable since main-memory database systems usually have dedicated resources.
- Instead of thread-local pools, we use one allocator object per physical core. Database systems can create and terminate threads at a high rate during query processing. Using thread-local pools in this case might hurt performance and complicates fail-safety management as the local pool has to be given back and integrated in the global pool upon thread termination. Using striping per physical core combined with aggressive caching provides a stable, robust, and scalable allocation and deallocation performance.
- To defragment memory, we leverage the hole punching feature of sparse files. This is an additional advantage of using multiple files.
- To provide fast recovery, we persist most of `PAllocator`'s metadata and rely on hybrid `SCM-DRAM` trees to trade off between performance and recovery time when necessary.

4.2 Programming Model

We assume that `SCM` is managed by a file system that provides direct access to the application layer through memory mapping. To retrieve data, we use *persistent pointers*, which consist of an 8-byte file ID and an 8-byte offset within that file. We devise two classes: `PPtrBase` and `PPtr`. `PPtrBase` is an untyped persistent pointer, equivalent to `void*`. `PPtr` inherits from `PPtrBase` and is a template, that is, it is aware of its type, and it also provides a cast function:

```
class alignas(16) PPtrBase {
    uint64_t fileID;
    ptrdiff_t offset;
    void* toPtrBase(); // Swizzling function
};
template<typename T>
class PPtr : PPtrBase {
    T* toPtr(); // Swizzling function
    template<typename U>
    PPtr<U>& as(); // Cast function
};
```

Persistent pointers are aligned to 16-bytes to make sure that the file ID and the offset reside in the same cache line. Since persistent pointers are 16-byte large, they cannot be assigned p-atomically. To remedy this issue, we adopt the convention that a null pointer is a pointer with fileID equal to -1. Thus, by setting the offset first, then the file ID, we ensure that the persistent pointer is moved p-atomically from a null value to a valid value. As will be explained in Section 4.3, persistent pointers can be swizzled (converted) to ordinary, virtual memory pointers and vice versa.

To prevent memory leaks, we changed the allocation interface to take as argument a reference to a `PPtrBase`:

```
allocate(PPtrBase &p, size_t allocationSize)
deallocate(PPtrBase &p)
```

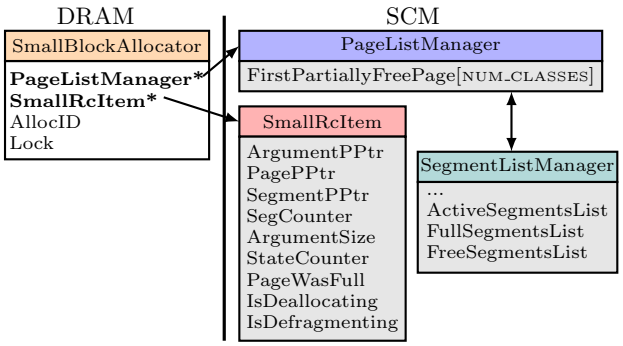

Figure 3: Architecture overview of `SmallPAllocator`.

The data structure must provide a persistent pointer that resides in `SCM`, where the allocate function can write the address of the allocated memory before returning. This ensures that the data structure has always a handle on its requested allocation, even in case of a crash. Hence, the responsibility of avoiding memory leaks is split between the allocator and the requesting data structure: If a crash occurs before the allocate function has persistently written the address of the allocated memory into the provided `PPtrBase`, it is the allocator's responsiblity to catch the memory leak during recovery; otherwise, the allocator considers that the allocation has successfully completed, and the data structure has a handle on the allocated memory. In addition, the deallocate function resets the provided persistent pointer to null in order to prevent dangling persistent pointers.

4.3 PAllocator Architecture

`PAllocator` is a highly scalable persistent allocator specifically designed for `SCM`-based databases that use large amounts of memory. `PAllocator` does not keep a single memory pool like previous works. Instead, it creates multiple files, referred to as *segments*, on a need basis.

Figure 2 illustrates the architecture of `PAllocator`. It uses three different allocators: `SmallPAllocator`, `BigPAllocator`, and `HugePAllocator`. `SmallPAllocator` handles allocations in the range [64 `B`, 16 `KB`), while `BigPAllocator` handles allocations in the range [16 `KB`, 16 `MB`). Larger allocations are handled by `HugePAllocator`. For concurrency, `PAllocator` maintains one `SmallPAllocator` and one `BigPAllocator` object per core, while maintaining only a single `HugePAllocator`, as huge allocations are bound by the operating system and file system performance. Indeed, `HugePAllocator` creates one segment per allocation, and deletes that segment on deallocation. Although simple, this approach has the advantage of avoiding any memory fragmentation. `SmallPAllocator` and `BigPAllocator` objects are instantiated in transient memory and initialized with references to their persistent metadata. The internals of `SmallPAllocator` and `BigPAllocator` are detailed in Sections 4.4 and 4.5, respectively.

`PAllocator` maintains a special segment, called *Anchor Segment*, where it keeps critical metadata for recovery purposes (See Figure 2). Besides, `PAllocatorAnchor` keeps one `PageListManager` (PLM_i) and one `BigBlockManager` (BBM_i) for each `SmallPAllocator` and `BigPAllocator`, respectively. The use of these structures is detailed in Sections 4.4 and 4.5, respectively. Additionally, the different allocators rely on micro-logs to ensure the allocation and deallocation atomicity across failures. These micro-logs are referred to as *Recovery Items*. The `RecoveryItemManager` maintains one such recovery item for each allocator object.

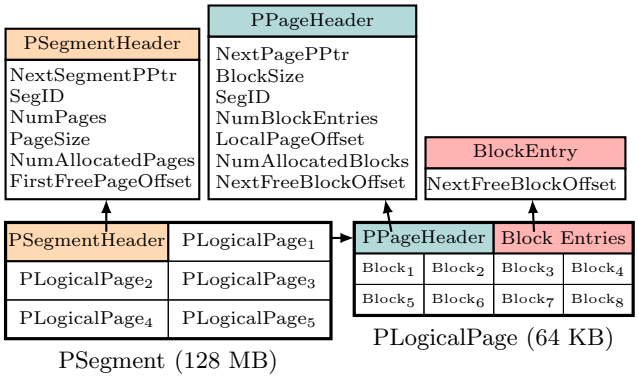

Figure 4: Data organization in `SmallPAllocator`.

`SegmentListManager` is a central component of our design and is responsible for creating segments. It has a transient part and a persistent part. The persistent part maintains a global segment counter, a shared list of free segments, and for each `SmallPAllocator`, a list of active segments (ASL_i) as well as a list of full segments (FSL_i). The use of these lists is explained in Section 4.4.

The transient part of `SegmentListManager` maintains the following segment mappings (See `DRAM` side of Figure 2):

- `SegIDToAddress`: It maps segment IDs to their corresponding virtual addresses. This map is used to swizzle persistent pointers to ordinary pointers by fetching the address of a segment and adding the offset.
- `AddressToSegInfo`: It maps the virtual address of a segment to its ID and size. This map is used to swizzle ordinary pointers to persistent pointers by getting an upper bound of the address, checking whether it is in the address range of the returned segment using its size, computing the offset and returning the corresponding persistent pointer.
- `SegIDToAllocID`: It maps a segment ID to its owner allocator ID. Segments are not shared between allocator objects, which implies that the allocator object that allocated a block is also responsible for deallocating it. Indeed, allocator objects operate independently from each other. This map is used in deallocations to identify which allocator object should perform the deallocation.

These mappings are stored in transient memory for better performance, and are easily retrieved from persistent metadata during recovery, as detailed in Section 4.7.

4.4 SmallPAllocator

`SmallPAllocator` relies on a segregated-fit allocation algorithm. It has 40 size classes, ranging from 64 `B` to 15 `KB`. We align all allocations to a cache-line-size (typically 64 `B`) in order to avoid false sharing of cache lines which could unintentionally evict adjacent data when flushed. Figure 3 gives an overview of the architecture of `SmallPAllocator`. Its two main auxiliary data structures are `PageListManager` and `SmallRcItem`. `PageListManager` consists of an array that contains a persistent pointer to the first partially free page of a class size. It interacts with `SegmentListManager` to get new free pages when needed upon allocations, and to return completely free pages upon deallocation. Each `SmallPAllocator` owns a set of segments which are represented by the `ActiveSegmentsList` and `FullSegmentsList` in `SegmentListManager`. `SmallRcItem` is a 64-byte long recovery item. It is used to log ongoing allocation or deallocation information to ensure their atomicity in case of a crash. For instance, `ArgumentPPtr` stores the persistent address of

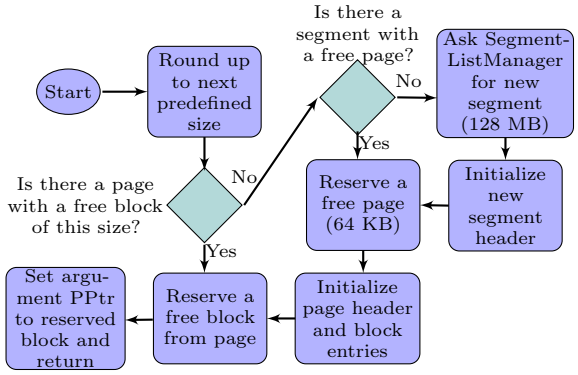

Figure 5: Allocation process of `SmallPAllocator`.

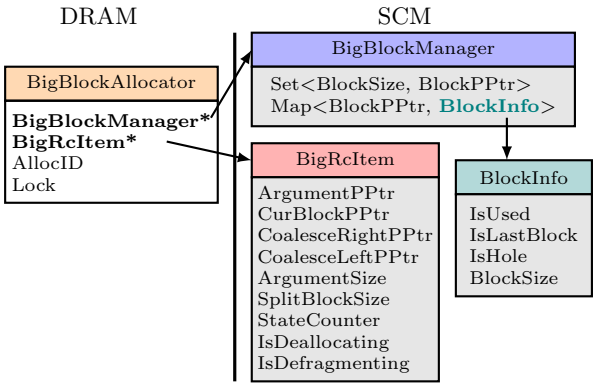

Figure 6: Architecture overview of `BigPAllocator`.

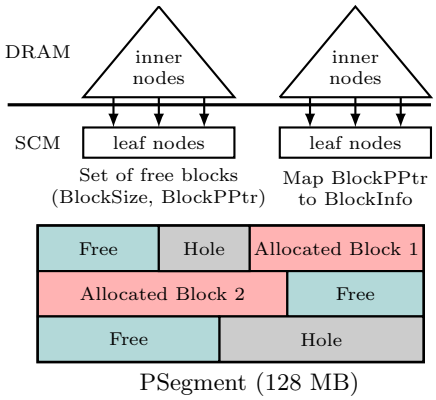

Figure 7: Data organization in `BigPAllocator`.

the persistent pointer provided by the requester, which enables `SmallPAllocator` to inspect it upon recovery to see whether it completed the allocation process.

Figure 4 illustrates data organization in `SmallPAllocator`. Each segment is 128 `MB` large, and is divided into logical pages of 64 `KB`, the first of which serves as the segment header. Segments are chained together using `NextSegmentPPtr` to form either the active or the full segments list whose heads are stored in `SegmentListManager`. Similarly, pages of the same size class are linked using `NextPagePPtr` to a list whose head is stored in `PageListManager`. Pages are in turn divided into blocks of the same size. To track its blocks, a page stores one 2-byte `BlockEntry` per block, which represents the offset of the next free block, hence forming a list whose head is referenced by `NextFreeBlockOffset` in the page header.

To illustrate how all the components of `SmallPAllocator` interact together, we depict in Figure 5 its allocation process. For the sake of simplicity and due to space constraints, we do not include operations on recovery items. First, the requested size is rounded up to the nearest predefined size class. Then, `PageListManager` checks whether there is a partially free page of this size, in which case it reserves the block pointed to by `NextFreeBlockOffset` in the page header by popping the head of the list of blocks; or if no free page is available, then `PageListManager` requests a free page from `SegmentListManager`, which returns a page from the head of the active segments list, creating a segment in the process if the latter is empty. Thereafter, the page header and block entries are initialized and a block is reserved.

Note that if a segment becomes empty after a deallocation, it is not returned to the system. Rather, free segments are chained in the `FreeSegmentsList` of `SegmentListManager` to avoid the cost of segment creation and page faults. This is a commonly used technique for transient memory in main-memory databases which implement their own internal memory management [12, 17].

4.5 BigPAllocator

`BigPAllocator` is a tree-based allocator that uses a best-fit algorithm. It is responsible for allocating blocks ranging from 16 `KB` to 16 `MB`. To enable defragmentation (see Section 4.6), all blocks are aligned to a system page size (usually 4 `KB`). Figure 6 gives an overview of `BigPAllocator`. It comprises two main auxiliary data structures: `BigBlockManager` and `BigRcItem`, which is a 128-byte recovery item used to ensure fail-safe atomicity of `BigPAllocator`'s operations. `BigBlockManager` is responsible for keeping the topology of allocated and free blocks. To this aim, it comprises two trees:

- *Free blocks tree*: It is a set of pairs of a block size and its persistent pointer. It keeps track of free blocks.
- *`BlockInfo` tree*: It maps the persistent pointers of all existing blocks to their corresponding `BlockInfo`.

A `BlockInfo` is 8-bytes long and consists of the block size and three flags: `IsUsed` indicates whether a block is allocated; `IsLastBlock` indicates whether a block is at the end of a segment; and `IsHole` indicates whether a block is a hole.

Figure 7 illustrates data organization in `BigPAllocator`. Similarly to `SmallPAllocator`, it uses segments of 128 `MB` that are then divided with a best-fit strategy into blocks of different sizes. We purposefully make `SmallPAllocator` and `BigPAllocator` segments of the same size to enable both to fetch segments from `FreeSegmentsList`. The trees of `BigBlockManager` are implemented using the `FPTree` [21], a hybrid `SCM-DRAM` persistent tree, which store a linked list of leaf nodes in `SCM` and keep inner nodes in `DRAM` for better performance. Upon recovery, the leaf nodes are scanned to rebuild the inner nodes. These hybrid trees use the small block allocators to allocate their leaf nodes. We chose to use the FPTree because it provides near-DRAM performance while recovering up to two orders of magnitude than a full rebuild of a transient counterpart.

To explain how the different components of `BigPAllocator` interact together, we depict its allocation process in Figure 8. For the sake of simplicity and due to space constraints, we do not include operations on recovery items. First, with a lower bound operation on the free blocks tree, using as search key the requested size and a null pointer, the smallest free block that is larger than the requested block is retrieved. If no block was found, then a request for a new segment is made to `SegmentListManager`, and the returned segment is inserted in the free blocks tree. Thereafter, if the block is larger than

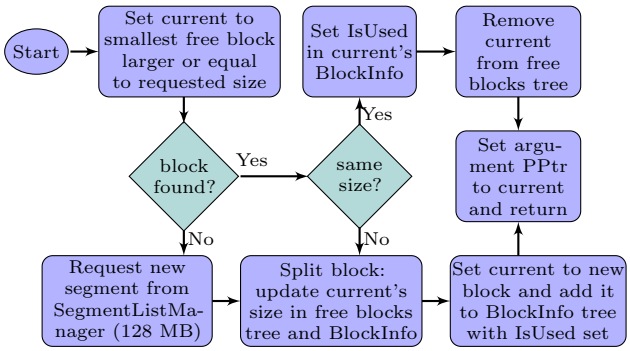

Figure 8: Allocation process of `BigPAllocator`.

the requested size, it is split into two blocks: the left one is kept free and its size updated both in the free blocks tree and its `BlockInfo`, while the right one, whose size is equal to the requested size, is inserted as a new allocated block in the `BlockInfo` tree.

4.6 Defragmentation

`NVML` and `nvm_malloc` do not have any mechanism to defragment persistent memory. Both create a single memory pool which exacerbates fragmentation in the presence of frequent large allocations and deallocations. The segregated-fit approach of `PAllocator`, `NVML`, and `nvm_malloc` already limits fragmentation for small blocks. In contrast, the tree-based big block allocators of the aforementioned allocators are more prone to fragmentation. To make matters worse, `NVML` and `NVMalloc` service huge allocation requests through their respective pool, which makes them fail when no matching free contiguous block exists in the pool. `PAllocator`, however, creates one segment per huge allocation, enabling it to avoid any fragmentation that might arise from huge allocations. This highlights the benefits of having multiple segments instead of a single large pool. Additionally, `PAllocator` implements an efficient defragmentation mechanism that we detail in this section.

To defragment memory, `PAllocator` relies on the hole punching feature of sparse files. When an allocation fails because `PAllocator` was unable to create a segment, the allocation request is first redirected to the other allocator objects in a round-robin fashion, until one of them succeeds, because one of them might have a partially free segment with a matching free block. If all of them fail, then a defragmentation request of the requested segment size is made. The defragmentation process first checks the free segments list in `SegmentListManager` and removes as many of them as needed. If that is not enough, then the segments of `BigPAllocator` are defragmented. The latter process is illustrated in Figure 9.

The defragmentation process starts by getting the largest free block available in the free blocks tree. Then, it updates its `BlockInfo` by setting `IsHole` to true, and punches a hole of the size of the block in the corresponding segment. We use *fallocate*[4] with the flags `FALLOC_FL_PUNCH_HOLE` and `FALLOC_FL_KEEP_SIZE` to perform the hole punch. Note that the latter flag enables the segment to keep its apparent size, which is required since collapsing the punched hole could invalidate the offsets of existing persistent pointers. Punching holes in free blocks is safe because they are aligned to a system page size. Thereafter, the defragmentation process erases

[4] `http://man7.org/linux/man-pages/man2/fallocate.2.html`

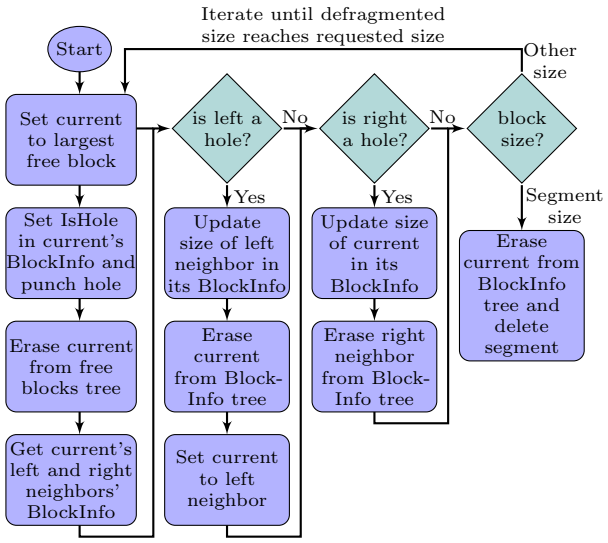

Figure 9: Defragmentation process of `BigPAllocator`.

the current block from the free blocks trees, and gets handles on its left and right neighbors from the BlockInfo tree. If the left neighbor is a hole, then it is coalesced with the current block by updating its size and erasing the current block from the `BlockInfo` tree. The current block is then set to the left neighbor. If the right neighbor is a hole, then it is coalesced with the current block by updating the size of the current block and erasing the right neighbor from the `BlockInfo` tree. Finally, if the size of the current block is equal to that of a segment, then the whole segment is a hole, in which case it is deleted both from the `BlockInfo` tree and from the file system. This process is repeated until either we reach the requested size, or no more free blocks exist.

Figure 9 does not show the use of the recovery items. Nevertheless, the defragmentation process uses them to achieve fail-safe atomicity. In case of a failure during defragmentation, `PAllocator` rolls forward during recovery only the defragmentation iteration that was ongoing at failure time. The recovery is greatly simplified since removing from a tree and punching a hole in a file are both idempotent operations.

One limitation of our defragmentation algorithm is that holes are still memory mapped, hence blocking address space. To mitigate this shortcoming, we propose to collapse holes that are located at the end of a segment and remap the segment by shrinking the existing memory mapping.

Additionally, `SmallPAllocator` can also be defragmented by punching holes in free logical pages, and subtracting them from the number of pages in the segment header, which allows us to account for logical page holes when deciding if a segment if empty.

4.7 Recovery

`PAllocator` uses the status flag in the anchor segment to check whether a failure occurred before the end of the initialization process, in which case the initialization is restarted from scratch. If a failure occurs after the initialization completed, then `PAllocator` starts recovery by memory mapping all existing segments and reconstructing `SegIDToAddress` and `AddressToSegInfo` of `SegmentListManager`. Thereafter, `PAllocator` instantiates the allocator objects and calls their respective recovery functions, which check the recovery items and restore the allocator to a consistent state in case the failure happened during an allocation or a deallocation. Note

that the recovery of BigPAllocator involves reconstructing the inner nodes of its hybrid trees from the persistent leaf nodes. Finally, PAllocator recovers the SegIDToAllocID map in the following steps:

1. All segments are initially assigned to HugePAllocator.
2. The active segments list and full segments list, located in SegmentListManager, of each SmallPAllocator are scanned. This enables to restore segment ownership information of all SmallPAllocators.
3. The recovery of a hybrid tree involves scanning the persistent leaf nodes to retrieve the max key in each one of them, which are in turn used to reconstruct the inner nodes. We extend the recovery of the BlockInfo tree to extract a list of encountered segments while scanning the leaf nodes, which is sufficient to restore segment ownership information of all BigPAllocators.
4. The segments present in the FreeSegmentsList are not assigned to any allocator.
5. The remaining segments whose ownership has not been updated remain assigned to HugePAllocator.

Recovery time is dominated by the recovery of the trees of BigPAllocator when these are large. Nevertheless, hybrid trees recovery is one to two orders of magnitude faster than that of transient counterparts. Note that the recovery of segments that contain holes does not pose any challenge and are recovered in the same way as segments without holes.

5. EVALUATION

Experimental setup. We run our experiments on an SCM emulation system provided by Intel. It is equipped with 4 8-core Intel Xeon E5-4620 v2 processors clocked at 2.60 GHz. Each core has 32 KB L1 data and 32 KB L1 instruction cache and 256 KB L2 cache. The cores of one processor share a 20 MB last level cache. To emulate SCM, two sockets are disabled and their memory is interleaved at a cache-line granularity and reserved as a separate persistent memory region. Thanks to a special BIOS, the latency of this memory region can be configured to a specified value. A detailed description of this system is publicly available [10]. The system has 64 GB of DRAM and 359 GB of emulated SCM on which we mount *ext4* with DAX. The system runs Linux kernel *4.8.0-rc4*.

In addition to PAllocator, we evaluate the following state-of-the-art persistent allocators:

- NVML v1.1[5]: We use libpmemobj's POBJ_ALLOC to allocate and POBJ_FREE to deallocate.
- nvm_malloc[6]: We use nvm_reserve + nvm_activate to allocate, and nvm_free to deallocate. To keep track of allocated objects, we provide the allocation and deallocation functions with a single link pointer.
- Makalu[7]: We use MAK_malloc/MAK_free to allocate/deallocate. We track allocated objects using a linked list.

We intentionally exclude NVMalloc[8] from our experiments because it uses anonymous memory mapping as a means to emulate SCM, and does not provide any recovery mechanism. As a baseline, we also include in our experiments the following state-of-the-art transient memory allocators:

- *ptmalloc*: The GNU C library allocator. We use *glibc* v2.19.
- *jemalloc* v4.2.1: An allocator designed for fragmentation avoidance and high concurrency scalability.
- *tcmalloc (gperftools v2.5)*: An allocator designed for high concurrency scalability through thread-caching.

We compile all tests with *gcc-4.8.5* with full optimizations. We let the operating system schedule the threads but we forbid thread migration between sockets in order to get more stable results. In all experiments where we vary the number of threads, we set the latency of SCM to the minimum latency, namely 160 ns, which corresponds to the normal remote-socket latency. This gives us the best performance evaluation since higher latencies are emulated using microcode and do not account for out-of-order execution and instruction prefetchers. All reported results are the average of 5 runs.

Standalone allocation and deallocation

To evaluate standalone allocation and deallocation performance, we designed a micro-benchmark that takes as parameters a fixed object size, a number of threads, and the number of allocations per thread N. The program first executes the requested allocations as a warm-up, then it deallocates all objects and allocates them again. We measure the time of the latter deallocation and allocation phases separately. We set $N = 500k$ for object sizes 128 B, 1 KB, and 8 KB, $N = 80k$ for 64 KB, and $N = 10k$ for 512 KB. The goal of the warm-up phase is to trigger all page faults to decouple the performance of the allocators from kernel operations. This is especially important in concurrent environments where kernel locking can be an important performance factor. We report the results in Figure 10 and Figure 11. Note that we depict results of nvm_malloc only for object sizes 128 B and 1 KB, because nvm_malloc uses a non-balanced binary tree to index allocated and free blocks greater than 2 KB. Since the object size is fixed in this experiment, the binary trees degenerate into linked-lists, thus incurring severe performance issues.

Figures 10a-10e show allocation throughput per thread for different allocation sizes. For small sizes (128 B and 1 KB), we observe that PAllocator scales linearly; it retains nearly the same throughput per thread with 16 threads as with one thread. NVML scales nearly linearly as well, but with a drop from 8 to 16 threads. In contrast, Makalu and nvm_malloc scale less than the aforementioned counterparts. Indeed, Makalu and nvm_malloc are the fastest single-threaded but the slowest with 16 threads. Worse, Makalu's performance degrades linearly for sizes 1 KB and higher, because it uses thread-local allocation only for small blocks, and relies on a global structure for larger blocks, which results in poor scalability. Overall, PAllocator scales better and significantly outperforms NVML and Makalu, and nvm_malloc with 16 threads. For 128 B allocations, transient allocators are one to two orders of magnitude faster than the persistent ones. This is expected since persistent allocators use expensive flushing instructions. Yet, the performance gap narrows significantly at 16 threads, except for jemalloc which remains one order of magnitude faster. For 1 KB allocations however, only tcmalloc is one to two orders of magnitude faster than persistent counterparts with one thread. With 16 threads, PAllocator outperforms ptmalloc and tcmalloc, and performs in the same order of magnitude than jemalloc.

For medium sizes (8 KB and 64 KB), we observe that PAllocator still scales linearly, while NVML scales less compared with the small objects case. This is because the design of NVML emphasizes fragmentation avoidance and thus grants chunks of only 256 KB at once to local thread caches. This leads to lock contention as thread local caches request more often chunks with 1 KB allocations than with 128 B ones.

[5] https://github.com/pmem/nvml/releases
[6] https://github.com/IMCG/nvm-malloc
[7] https://github.com/HewlettPackard/Atlas/tree/makalu
[8] https://github.com/efficient/nvram/

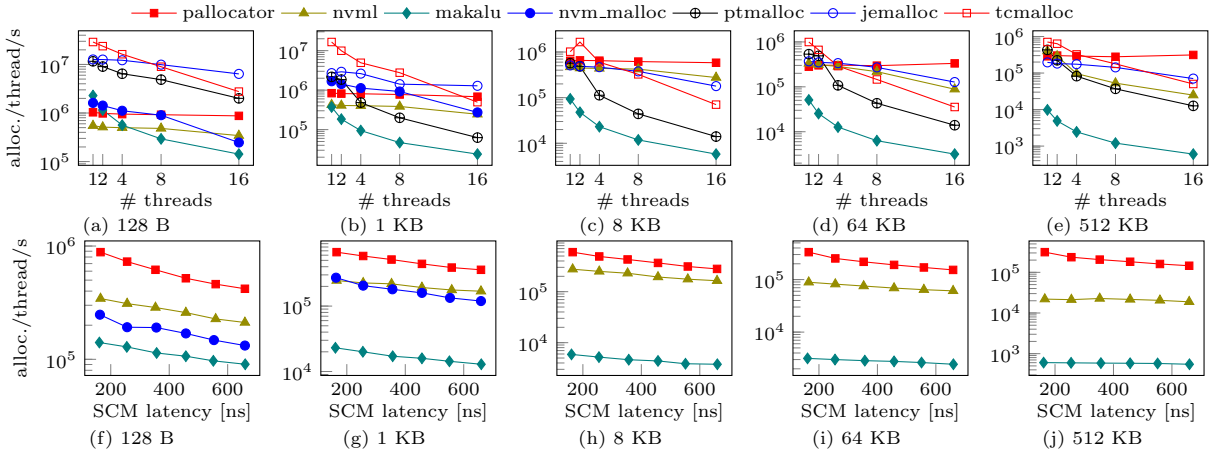

Figure 10: Allocation performance for different object sizes and SCM latencies.

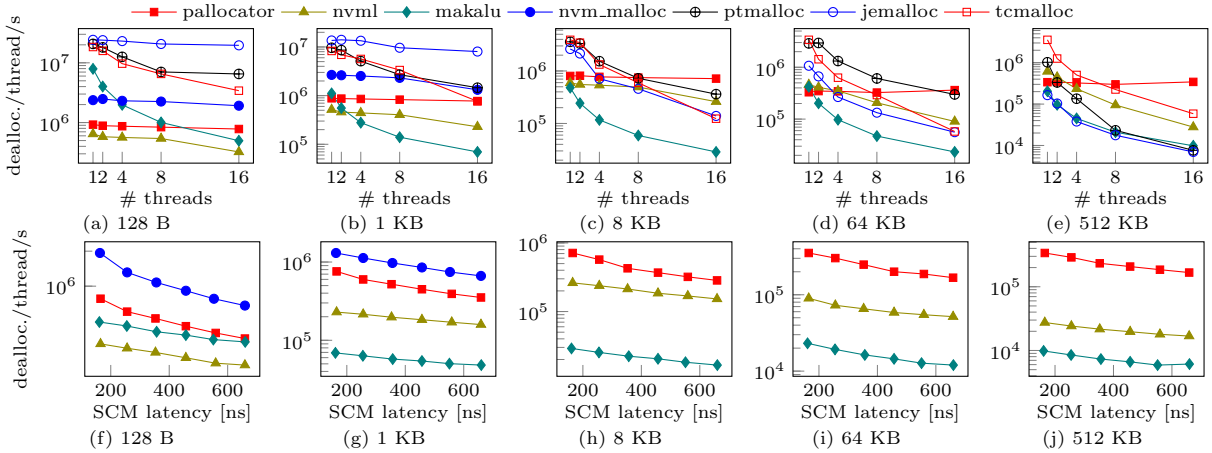

Figure 11: Deallocation performance for different object sizes and SCM latencies.

PAllocator is able to significantly outperform all transient allocators with 16 threads. In fact, the larger the allocation size, the less the transient allocators scale, especially ptmalloc which is outperformed by both PAllocator and NVML with only two threads.

For large sizes (512 KB), we observe that PAllocator is the only allocator that scales linearly, and it outperforms NVML and all transient allocators with only 2 threads. We note that NVML is faster than PAllocator with one thread: NVML uses transient trees to index its large blocks, while we use hybrid SCM-DRAM trees which incur a small additional overhead but enable much faster recovery (see recovery experiments below).

Figures 10f-10j show allocation throughput per thread for different allocation sizes, varying SCM latencies, and 16 threads. We observe that performance degrades with higher latencies by up to 54%, 43%, 44%, and 57% for PAllocator, NVML, Makalu, and nvm_malloc, respectively, with an SCM latency of 650 ns compared with 160 ns. We also observe that for allocation sizes 64 KB and 512 KB, PAllocator seems to suffer more from higher SCM latencies than NVML. This is explained by two factors: (1) the bottleneck for NVML with 16 threads is synchronization rather than persistence; and (2) accessing the leaf nodes of PAllocator's hybrid trees becomes more expensive with higher SCM latencies.

Figures 11a-11e show deallocation throughput per thread for different allocation sizes. Similar to allocations, PAllocator scales linearly for all allocation sizes. NVML exhibits similar behaviour to its allocations as well and has lower throughput than PAllocator except for sizes 64 KB and 512 KB with up to 4 threads. This is again explained by the fact that PAllocator uses its big block allocator for these allocation sizes, which involves operations on its hybrid trees that incur an extra overhead compared with NVML's transient counterparts. We note that nvm_malloc has 2× higher throughput for deallocations than for allocations for sizes 128 B and 1 KB, and outperforms PAllocator even with 16 threads. Indeed, in contrast to its allocations, nvm_malloc deallocations scale linearly. As for Makalu, it is the fastest for 128 B, but its performance drops linearly in all cases. Interestingly, with 16 threads and for sizes 8 KB and larger, PAllocator outperforms again all allocators including transient ones.

Figures 11f-11j show deallocation throughput per thread for different allocation sizes, varying SCM latencies, and 16 threads. We observe the same patterns as with allocations. We note that the performance of all persistent allocators degrades gradually with increasing SCM latencies up to 60%, 46%, 48%, and 64% for PAllocator, NVML, Makalu, and nvm_malloc, respectively, with an SCM latency of 650 ns compared with 160 ns. We argue that this drop is still acceptable given the 4× higher latency.

Random allocation and deallocation

In this experiment we evaluate the performance of mixed random allocations and deallocations. The experiment consists in executing 10 iterations of N allocations followed by N deallocations of random size per thread. We fix the range of object sizes between 64 B and 128 KB, and we consider three different object size distributions: (1) Uniform distri-

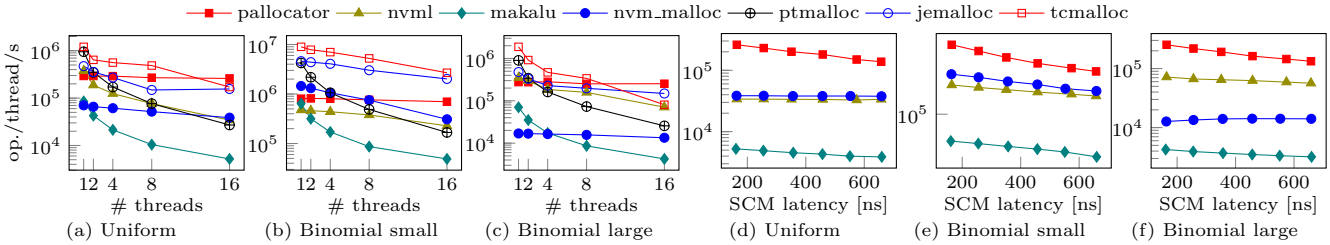

Figure 12: Random allocation/deallocation benchmark with different distributions.

bution; (2) Binomial distribution with skew factor 0.01 to emphasize smaller sizes; (3) Binomial distribution with skew factor 0.7 to emphasize larger sizes. We set $N = 500k$ for Binomial small, and $N = 50k$ for Uniform and Binomial large. Figure 12 depicts the results.

Figures 12a-12c show operation throughput per thread. We observe that `PAllocator` scales linearly for all distributions. Besides, it manages to outperform all transient allocators in the Uniform and Binomial large cases, while outperforming only ptmalloc in the Binomial small case. In the Uniform case, `PAllocator` outperforms `NVML`, `Makalu`, and `nvm_malloc` by up to 7.5×, 49.7×, and 6.7×, respectively. `nvm_malloc`'s performance degrades significantly in the case of Binomial large because of its binary trees and is 19× slower than `PAllocator`. `Makalu`'s behavior is similar to the previous experiment as its performance degrades linearly with increasing threads.

Figures 12d-12f show operation throughput per thread for varying `SCM` latencies with 16 threads. The drop in the performance of `PAllocator` is limited to 47% with an `SCM` latency of 650 ns compared with 160 ns. Nevertheless, `PAllocator` still outperforms `Makalu`, `NVML`, and `nvm_malloc`. The latter two show little performance degradation with higher `SCM` latencies since with 16 threads, their bottleneck is synchronization in addition to the binary trees for `nvm_malloc`.

Larson benchmark

We adapt the Larson benchmark from the Hoard[9] allocator benchmark suite [4] to the evaluated persistent allocators. The Larson benchmark simulates a server. In its warm-up phase, it creates N objects of random size per thread and shuffles the ownership of these objects between the threads. Thereafter, in a loop, it randomly selects a victim object to deallocate and replaces it with a newly allocated one of random size. After the warm-up phase, each thread executes in a loop $N \times L$ times the last sequence of the warm-up phase, where $N = 1k$ and $L = 10k$ in our experiments. We experiment with three object size ranges: 64 `B`–256 `B` (small), 1 `KB`–4 `KB` (medium), and 64 `KB`–256 `KB` (large). Results are depicted in Figure 13.

Figures 13a-13c show operation throughput per thread for different object size ranges. We observe that `PAllocator` is the only allocator to scale linearly in all three object size ranges. It outperforms `NVML`, `Makalu`, and `nvm_malloc` by up to 4.5×, 1.6×, and 3.9×, respectively, and is able to outperform jemalloc and tcmalloc in size range 64 `KB`–256 `KB` with 16 threads. We note, however, that `Makalu` outperforms `PAllocator` in size range 64 `B`– 256`B` until 4 threads, partially thanks to the fact that the total size of allocated objects does not grow. Also, we note that `nvm_malloc` outperforms `PAllocator` in size range 64 `B`– 256`B` until 8 threads, and in size range 1 `KB`– 4 `KB` with one thread. Surprisingly, `nvm_malloc` scales well in size range 64 `KB`–256 `KB`, in contrast to size range 1 `KB`–4 `KB`, and outperforms `PAllocator` until 16 threads where the performances of the two allocators meet. This is explained by two factors: (1) the small number of allocated objects in this experiment maintains the binary trees of `nvm_malloc` very small; and (2) `nvm_malloc` rounds up allocation sizes to multiples of 4 `KB` starting from 2 `KB` requests. Consequently, in the range 1 `KB`–4 `KB`, half of the allocations are rounded up to 4 `KB` allocations, making the binary trees degenerate into linked-lists, while these are relatively balanced for size range 64 `KB`–256 `KB`.

Figures 13d-13f show operation throughput per thread for different object size ranges, different `SCM` latencies, and 16 threads. We observe that for size ranges 64 `B`–256 `B` and 64 `KB`–256 `KB`, the performance of the persistent allocators degrades in a similar way: up to 57%, 45%, 54%, and 56% for `PAllocator`, `NVML`, `Makalu`, and `nvm_malloc`, respectively, for an `SCM` latency of 160 ns compared with 160 ns. For size range 1 `KB`–4 `KB` however, we note that `PAllocator` is more sensitive to higher `SCM` latencies than `NVML`, `Makalu`, and `nvm_malloc`. This is partly explained by the fact that `NVML`, `Makalu`, and `nvm_malloc` suffer from other bottlenecks (synchronization) than persistence with 16 threads. Nevertheless, we found that `NVML` was consistently the least sensitive allocator to higher `SCM` latencies throughout all previous experiments.

We expect significant performance improvements with the new `CLWB` instruction, because `PAllocator` reads and flushes recovery items multiple times per operation. `CLFLUSH` evicts the recovery items from the `CPU` caches, leading to repeated cache misses as recovery items are accessed again shortly after. `CLWB` would remedy this issue and improve performance.

Recovery time

In this experiment we measure recovery time of `PAllocator`, `NVML`, and `nvm_malloc`– we exclude transient allocators. To do so, we first execute a fixed amount of allocation requests of sizes uniformly distributed between 64 `B` and 128 `KB` using 16 threads. Then, in a second run we measure the recovery time of the persistent allocator.

`nvm_malloc` re-creates upon recovery its arenas as if they were full, that is, all their chunks have no free blocks. Chunks can be recovered in two ways: (1) either `nvm_malloc` receives a deallocation request of a block that resides in a not-yet-recovered chunk, in which case the chunk is discovered and recovered; (2) or by a background thread created by the recovery process. If `nvm_malloc` receives an allocation request before its chunks have been recovered, it will create new ones, even if the non-recovered ones contain enough free space to service the allocation request. This has the effect of exacerbating (permanent) fragmentation, which is not acceptable, especially since `nvm_malloc` cannot defragment its memory. Hence, to avoid this, and to measure the total recovery time of `nvm_malloc`, we changed its recovery function to recover fully existing chunks before returning.

[9] `https://github.com/emeryberger/Hoard/`

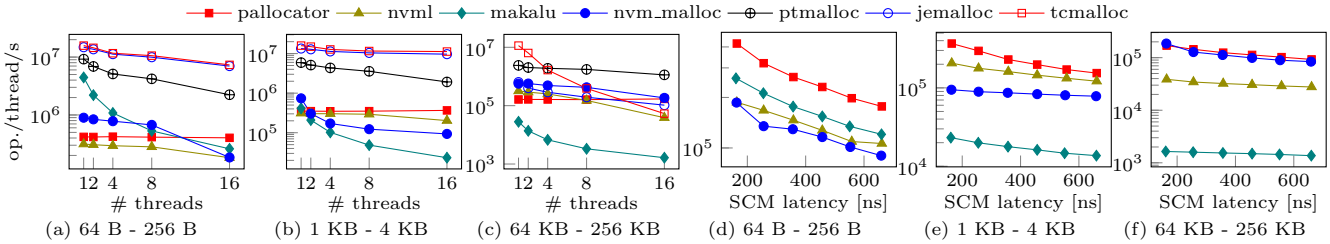

Figure 13: Larson benchmark with different random allocation ranges.

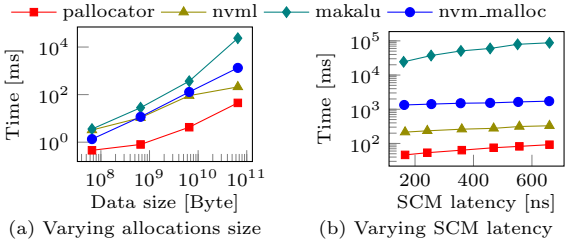

Figure 14: Recovery time for different total allocation sizes and SCM latencies (lower is better).

Makalu has two recovery modes: Upon normal process shutdown, Makalu flushes its metada to SCM and considers that garbage collection is not needed upon restart. If, however, the process was terminated due to a failure, Makalu runs an offline garbage collection process during recovery. In this experiment, we consider the second case.

Figure 14 depicts the results. In Figure 14a we fix the latency of SCM to 160 ns and vary the total size of allocations. We observe that PAllocator recovers significantly faster than NVML, Makalu, and nvm_malloc: with a total size of allocations of 61 GB, PAllocator recovers in 45 ms, while NVML, Makalu, and nvm_malloc recover in 210 ms, 23.5 s and 1.3 s, respectively. Makalu is the slowest to recover because it needs to execute an offline garbage collection, in addition to scanning its persistent metadata to rebuild its transient metadata. nvm_malloc is the second slowest to recover because it needs to scan persistent memory for existing blocks, while PAllocator and NVML keep enough metadata persisted to swiftly recover their allocation topology. The difference between PAllocator and NVML comes from the fact that NVML indexes metadata in transient trees, while PAllocator employs hybrid SCM-DRAM trees that are up to two orders of magnitude faster to recover than transient ones. Note that all four allocators implement only single-threaded recovery and could benefit from parallelizing their recovery process.

In Figure 14b we fix the total size of allocations to 61 GB, and vary the latency of SCM between 160 ns and 650 ns. We note that recovery time increases slowly with higher SCM latencies. Indeed, at an SCM latency of 650 ns, recovery times increased compared to those at an SCM latency of 160 ns by 100%, 54%, 260%, and 1.29% for PAllocator, NVML, and nvm_malloc, respectively. These increases are reasonable for an SCM latency increase of 4×. This is explained by the fact that the recovery processes of all three allocators involve mainly sequential reads which are more resilient to higher memory latencies thanks to hardware prefetching. The higher increase for PAllocator is explained by the recovery of the hybrid trees which dominates total recovery time. Since their leaf nodes are persisted in SCM as a list, scanning them during recovery involves sequential reads within a leaf node, but breaks the prefetching pipeline when accessing the next leaf node, hence resulting in a higher sensitivity to SCM latencies. Extrapolated to a total allocations size of 1 TB, recovery times would be 0.75 s, 3.5 s, 394.5 s, and 22.5 s for PAllocator, NVML, Makalu, and nvm_malloc, respectively.

Table 2: Fragmentation stress test.

Allocator	#allocs	Defrag.	Runtime	Max block
Pallocator	2.1m	512 GB	358 s	2 GB
NVML	55.6k	na	8 s	128 KB
Makalu	303k	na	172 s	128 KB
nvm_malloc	0	na	na	64 KB

Defragmentation

In this experiment we focus on testing resilience to memory fragmentation of PAllocator, NVML, and nvm_malloc– we exclude transient allocators. NVML's pool size is set to the maximum available emulated SCM (359 GB). The warm-up phase consists of allocating 256 GB of 64 KB objects. Thereafter, we perform in a loop: (1) deallocate every other object until half of the allocated memory (i.e. 128 GB) is freed; and (2) allocate 128 GB of objects of double the previous allocation size. We deallocate objects in a sparse way to exacerbate fragmentation. We repeat this process until either the first allocation fails or we reach object size of 2 GB. The experiment is run single-threaded. We report the total number of allocated objects, the amount of defragmented memory for PAllocator, and the total runtime (excluding the warm-up phase) in Table 2. We notice that NVML and Makalu fail early and manage to execute only a small number of allocations, while nvm_malloc does not even complete the warm-up phase. Note that for Makalu, we had to increase the maximum number of sections in its configuration file from 1 K to 1 M for the warm-up phase to run through. PAllocator manages to reach the 2 GB object size threshold by defragmenting 512 GB of memory through several iterations, managing to execute 2.1M allocations. This demonstrates the ability of PAllocator to efficiently defragment memory. Note that for sizes larger than 16 MB, in contrast to NVML and nvm_malloc, PAllocator does not create any fragmentation as it creates one segment per allocation.

Allocator impact on a persistent B-Tree performance

In this experiment we study the impact of allocator performance on a persistent B-Tree, a popular data structure in database systems. To do so, we use the concurrent, variable-size key version of the FPTree [21] (in contrast to the single-threaded, fixed-size key version used in BigPAllocator), which stores keys in the form of a persistent pointer. Thus, every insertion and deletion operation involves a key allocation or deallocation, respectively. The tested FPTree version employs Hardware Transactional Memory (HTM) in its concurrency scheme. Unfortunately, the emulation system does not support HTM. Hence, we use for this experiment a system equipped with two Intel Xeon E5-2699 v4 processors that support HTM. Each one has 22 cores (44 with HyperThreading) running at 2.1GHz. The local-socket and remote-socket DRAM latencies are respectively 85 ns and 145 ns. We mount *ext4* with DAX on a reserved

DRAM region belonging to the second socket, and bind our experiments to the first socket to emulate a higher SCM latency.

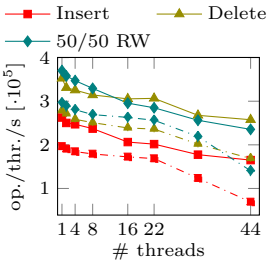

Figure 15: Allocator impact on the FPTree. Solid (dash-dotted) lines depict PAllocator (NVML).

We measure the performance of FPTree insertions, deletions, and a mix of 50% insertions and 50% find operations, using PAllocator and NVML. In all experiments, we first warm up the tree with 50M key-values, then execute 50M operation for a varying number of threads. Keys are 128-byte strings while values are 8-byte integers. We report the normalized throughput per thread in Figure 15. The solid lines represent PAllocator results while the dash-dotted lines represent NVML results. We observe that with one thread, using PAllocator yields 1.34x, 1.26x, and 1.25x better throughput than using NVML for insertions, deletions, and mixed operations, respectively. These speedups surge up to 2.39x, 1.52x, and 1.67, respectively, with 44 threads. This shows that the FPTree scales better with PAllocator than with NVML, especially when using hyperthreads. We conclude that the performance of a persistent allocator impacts that of SCM-based data structures.

6. CONCLUSION

In this paper we tackle the problem of SCM allocation as a fundamental building block for SCM-based database systems. We presented PAllocator, a highly scalable fail-safe persistent allocator for SCM that comprises two allocators, one for small block allocations, and the other one for big block allocations, where metadata is persisted in hybrid SCM-DRAM trees. Additionally, we highlighted the importance of persistent memory fragmentation, proposing an efficient defragmentation algorithm. Through an experimental analysis, we showed that PAllocator scales linearly for allocations, deallocations, mixed operations with different size distributions, and server-like workloads. Overall, it significantly outperforms NVML, Makalu, and nvm_malloc, both in operation performance and recovery time. Throughout all experiments, PAllocator showed robust and predictable performance, which we argue is an important feature for an allocator. Finally, we showed that a persistent B-Tree performs up to 2.39x better with PAllocator than with NVML, demonstrating the importance of efficient memory allocation for SCM-based data structures.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback which helped greatly improve the paper. This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence cfaed (Orchestration Path, Resilience Path) and in the context of the project "Self-Recoverable and Highly Available Data Structures for NVRAM-centric Database Systems" (LE-1416/27-1).

8. REFERENCES

[1] Intel 64 and IA-32 Architectures Software Developer Manuals. http://software.intel.com/en-us/intel-isa-extensions.

[2] NVML Library. http://pmem.io/nvml/.

[3] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[5] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA 2016*, pages 677–694. ACM, 2016.

[6] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP*, pages 133–146. ACM, 2009.

[9] J. Corbet. Linux 5-Level Page Table. https://lwn.net/Articles/717293/.

[10] S. R. Dulloor. *Systems and Applications for Persistent Memory*. PhD thesis, Georgia Institute of Technology, 2016.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, page 15. ACM, 2014.

[12] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[13] B. Govoreanu, G. Kar, Y. Chen, V. Paraschiv, S. Kubicek, , et al. 10× 10nm 2 hf/hfo x crossbar resistive ram with excellent performance, reliability and low-energy operation. In *Electron Devices Meeting (IEDM)*, pages 31–6. IEEE, 2011.

[14] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, et al. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *Electron Devices Meeting (IEDM)*, pages 459–462. IEEE, 2005.

[15] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, pages 691–706. ACM, 2015.

[16] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE micro*, 30(1):143, 2010.

[17] R. Mistry and S. Misner. *Introducing Microsoft SQL Server 2014*. Microsoft Press, 2014.

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

[19] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, page 1. ACM, 2013.

[20] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *EuroSys*, page 12. ACM, 2016.

[21] 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. ACM, 2016.

[22] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. In *CIDR*, 2015.

[23] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Micro*, pages 14–23. IEEE/ACM, 2009.

[24] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@ VLDB*, pages 61–72, 2015.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[26] R. S. Williams. How we found the missing memristor. *IEEE spectrum*, 45(12):28–35, 2008.

[27] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *SIGARCH Comput. Archit. News*, 19(4):6–13, July 1991.

[28] X. Wu and A. Reddy. SCMFS: a file system for storage class memory. In *SC*, page 39. ACM, 2011.

[29] J. Xu and S. Swanson. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX FAST 16*, pages 323–338, 2016.

[30] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen. Walloc: An efficient wear-aware allocator for non-volatile main memory. In *IPCCC*, pages 1–8. IEEE, 2015.