

nvm_malloc: Memory Allocation for NVRAM

David Schwalb[†], Tim Berning^{*}, Martin Faust[†], Markus Dreseler[†], Hasso Plattner[†]

[†]Hasso-Plattner-Institute, Germany

^{*}SAP SE, Walldorf, Germany

Contact: david.schwalb@hpi.de

ABSTRACT

Non-volatile main memory (NVRAM) has the potential to fundamentally change the persistency of software. Applications can make their state persistent by directly placing data structures on NVRAM instead of volatile DRAM. However, the persistent nature of NVRAM requires significant changes for memory allocators that are now faced with the additional tasks of data recovery and failure-atomicity. In this paper, we present `nvm_malloc`, a general-purpose memory allocator concept for the NVRAM era as a basic building block for persistent applications. We introduce concepts for managing named allocations for simplified recovery and using volatile and non-volatile memory in combination to provide both high performance and failure-atomic allocations.

1. INTRODUCTION

Today’s memory hardware is either fast but volatile (i.e., DRAM) or persistent but slow (SSDs/HDDs). As a result, applications go through great lengths to persist runtime information onto non-volatile storage media to prevent data loss. In the process, data structures in main memory regularly undergo costly transformations to reflect the different access characteristics of block storage devices and to better utilize the limited bandwidth. To ensure consistency, execution of the program must often halt until the data has successfully been pushed down the system’s I/O stack and reached the persistency domain. In a similar fashion, recovery after a crash requires loading data back into main memory and transforming it into usable data structures. Therefore, persistency can add significant architectural complexity and runtime overhead to applications.

An example of this is the database log where database tuples are transformed into undo and redo entries and commit entries are written so that a continuous log stream can be used to recover from failures such as power outages.

Non-volatile main memories (NVRAM) are a recent development in the field of memory technologies and represent a unique opportunity to simplify persistency as we know it.

In contrast to DRAM, NVRAM cells do not require periodic refreshing to maintain their state and can thus preserve contents even in the event of power loss. Performance analyses currently place NVRAM between DRAM and NAND-flash, but due to ongoing research and development its performance is predicted to eventually reach or even surpass that of DRAM. By allocating data structures on NVRAM instead of DRAM, application state could finally be made persistent in-place without sacrificing speed or increased software complexity.

Along with the changes it brings, NVRAM also requires change. Not just applications themselves, but memory allocation in particular as a fundamental building block of software faces new challenges and requirements, which are not met by today’s implementations. In this work we explore how classic memory allocation concepts and tasks are about to in the NVRAM era and present `nvm_malloc`, our own NVRAM allocator concept.

Contributions. In this work we explore how classic memory allocation concepts and tasks are about to change in the NVRAM era and make the following contributions:

- (1) We present a novel general-purpose NVRAM allocation concept with fast builtin recovery features as well as fine-grained access to NVRAM in a safe and performant fashion and present `nvm_malloc` as an implementation (Section 3).
- (2) We show how performance is affected by NVRAM memory management in various benchmarks of `nvm_malloc` and compare it to both existing volatile and non-volatile allocators (Section 4).

2. NVRAM BACKGROUND

Versatile, general-purpose memory allocators have been a fundamental element of software for decades [8] and the need for dynamic memory management translates into the NVRAM domain as well. However, the challenges imposed by NVRAM require an overhaul of the classic concepts of memory allocation.

As a precursor to our upcoming discussions, we provide an overview of upcoming NVRAM technologies and their characteristics, as well as the possible ways of integrating NVRAM into a computer system and discuss the challenges arising from the use of NVRAM from a software development perspective in general and for the task of memory allocation in particular.

Multiple different technologies are currently being developed: Ferroelectric RAM (FeRAM), Phase-Change Memory

	DRAM	FeRAM	PCM	MRAM	STT-RAM
Read	10ns	50ns	100ns	12ns	10ns
Write	10ns	75ns	20ns	12ns	10ns

Table 1: Latency comparison of DRAM and various NVRAM technologies based on [11], numbers vary in other publications [9]

(PCM), Magnetoresistive RAM (MRAM), and Spin-Torque-Transfer RAM (STT-RAM) are the main competitors in the field and differ in various aspects such as manufacturing and operating costs, density and access latency. Table 1 shows the expected read and write latencies of these technologies compared to DRAM. Besides the advantage of being persistent, NVRAM can also significantly reduce the total cost of operation by saving power that is normally required for the periodic refreshes of DRAM. Currently all NVRAM technologies still have weaknesses such as limited write cycles or high latency.

In order to use NVRAM, it must be integrated into existing system architectures, requiring changes on hardware, operating system and application level alike. One possibility of integrating NVRAM into the memory hierarchy of a system is to use it as a drop-in replacement for slower block devices. Since the I/O bus is shared among all block devices, access costs are increased. In order to use NVRAM as a persistent alternative to DRAM, it is instead favorable to attach it directly to the CPU memory bus to preserve byte-addressability and highest possible access speed.

Being directly attached to the CPU through the memory controller, NVRAM could either replace DRAM altogether or be used in parallel. The former would require NVRAM to reach or even surpass DRAM speeds, either through technological advances or the addition of volatile caches, and advanced operating systems support is required to e.g. transparently simulate volatile memory on NVRAM for backwards compatibility. The second option has both DRAM and NVRAM connected to the CPU memory controller using a shared address space. We believe that this hybrid mode provides the best of both worlds at least as long as the performance of NVRAM is lower than that of DRAM. This way, fast DRAM can be used for volatile parts of the application and data to be persisted can be stored on NVRAM. Because of the shared address space, developers need to take care which addresses are volatile and which are persisted.

Storing persistent data also requires reliable access to specific locations on NVRAM for retrieval. Allowing applications direct access to NVRAM however is likely to cause harm, different applications may wrongly access or corrupt each other’s data. Instead, current research suggests the usage of a lightweight filesystem where byte-level access is achieved by `mmap`ing a file into the virtual address space. Using the kernel’s execute in-place feature (XIP, in the future superseded by direct access, DAX) to directly map physical NVRAM pages onto virtual pages allows for zero-overhead access. Exposing NVRAM through a filesystem has a number of advantages: All files of a program can be grouped in a directory and protected using access control mechanisms. The file concept also ensures that no stray writes can happen, as all NVRAM space is contained within the file. Three

Instruction	Ordered by	Description
CLFLUSH	MFENCE	Invalidates cache line causing write back
CLFLUSHOPT	SFENCE	Invalidates cache line causing write back
CLWB	SFENCE	Issues write back of cache line without eviction

Table 2: Persistency instructions

examples of such filesystems are the Persistent Memory File System (PMFS) [5], SCMFS [22] and BPFS [4]. For the remainder of this paper we assume a system equipped with both DRAM and NVRAM attached to the memory controller and NVRAM exposed through PMFS.

2.1 General NVRAM Challenges

Without proper care, NVRAM contents are prone to enter an inconsistent state, such as when an in-place update to a data structure is interrupted by an application crash, leaving it half-updated without any information on how to roll-back or replay the update. Most use cases for NVRAM therefore require ACID¹-like guarantees for all modifying operations.

2.1.1 Persistency

Current CPUs generally employ a hierarchy of write-back caches, buffering data read from memory for faster access. Likewise, writes to memory pages are performed within the caches as well and propagated back to memory transparently to the programmer. With caches implemented in volatile SRAM, any cached write will be lost in case of a power failure.

To guarantee durability, the programmer needs a way to ensure that modifications reached physical NVRAM. Currently, the only existing option to achieve this is to explicitly evict a cache line using the `CLFLUSH` instruction. This invalidates the respective cache line, triggering a write back to physical memory. The invalidation causes future reads to miss the cache and require a new, expensive memory access. However, Intel recently released a new instruction set manual² featuring new memory instructions targeted specifically at NVRAM: `CLFLUSHOPT` and `CLWB`, Table 2 shows a basic comparison of these operations. The instruction `CLFLUSHOPT` is a slight improvement over `CLFLUSH` as it can be ordered using `SFENCE` instead of `MFENCE`. `CLWB` is much more promising, allowing to trigger a write back without cache invalidations. Depending on the hardware configuration, there may be further write buffers between the CPU and physical memory, therefore research suggests that an additional instruction is needed to guarantee the durability of memory stores [5]. The `PCOMMIT` instruction causes certain store-to-memory operations to persistent memory ranges to become persistent and power failure protected. For the remainder of this work, we assume that `CLFLUSH` is sufficient to reliably issue a write back to NVRAM from the CPU caches. Since a cache line flush evicts data from the CPU cache, it should be used only when necessary. Subsequent access to the same data requires it to be loaded from memory again and incurs high costs, which requires careful placement of flushes.

¹Atomicity, Consistency, Isolation and Durability

²<https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf>

2.1.2 Write Reordering

Compilers and CPUs optimize applications by reordering certain instructions if internal analyses deem it a performance improvement and it does not change the correctness of the program. Also, cache eviction may happen at any point in time without being explicitly triggered by the application, causing dirty cache lines to be written back to NVRAM unexpectedly.

In the context of NVRAM, this is problematic because the order of updates to NVRAM data structures is often essential to ensuring atomicity. As an example, assume a simple vector implementation comprised of an integer array and a pointer to the last element. The addition of a value requires appending the value to the array and incrementing the pointer to the last element. By incrementing the pointer *after* the insertion of the value, the vector always remains in a consistent state. Even if a crash terminates the program after the insertion, the update would be lost but the AC(ID) criteria not violated. However, the compiler and/or CPU may reorder the instructions so that the pointer is incremented before the new value is inserted. If the program crashes in between those two steps, the vector would be left in an inconsistent state where the end pointer points to a not-yet-inserted value.

The example demonstrates the importance of ensuring the correct execution path, which is done through the explicit use of compiler and memory fences. A compiler fence such as `asm volatile(":::memory")` instructs the compiler to limit reordering such that all instructions before the fence are executed before any instruction after the fence. To avoid CPU-side reordering, memory fences such as `LFENCE`, `SFENCE` or `MFENCE` can be used. These instructions force the CPU to execute all previous load instructions, store instructions or both before continuing execution.

An issue similar to the reordering problem arises from the fact that modern processors only guarantee atomic writes of up to 8 bytes. Failure atomicity demands that NVRAM contents either are in or can be transitioned into a consistent state at any given point in time. Considering a single value to be updated, this is trivial as long as the value is 8 bytes or smaller in size. The update can either succeed or fail, but the value will be in a consistent state regardless. In contrast, if more than 8 bytes need to be updated, a crash may leave the value in a partially updated state. An alternate form of this problem occurs when multiple values at different locations in memory need to be changed semantically at the same time. If applicable, the easiest solution is to perform the update on a shadow copy of the original value(s). Once the update is guaranteed to be complete, the pointer to the original value is updated atomically to point to the updated copy. A second option is to use validity flags to protect values that have not been updated properly yet. A more elaborate form of this are multiversion data structures [18], which use an internal versioning system to hide incomplete updates from being accessed. If none of these approaches can be applied, a logging mechanism must be implemented akin to database logs. The maintenance costs for the log and resulting data redundancy make this the least favorable option.

2.1.3 Recovery

Lastly, any application persisting data on NVRAM must implement a recovery strategy to recreate its former state after a crash. Again assuming the use of PMFS or a similar

filesystem, the recovery must take care of two steps: 1) locating and mapping the respective NVRAM region(s), and 2) identifying persisted objects within the mapped region(s).

To solve 1), applications can simply store their NVRAM files in a uniquely named working directory for trivial access. The details of step 2) depend on the kind and amount of data an application maintains on persistent memory. As a general concept, pointers to the beginning of data structures should be kept at a well-defined location, such as a static area at the beginning of the mapped region. Coupled with unique identifiers, the program could scan said static region for the existence of data structures and either reuse formerly persisted ones without any further work or (re)create them if they are missing or corrupt.

2.2 Memory Allocation on NVRAM

At its core, a volatile memory allocator faces one functional requirement: Delivering a pointer to a sufficiently large region of free virtual memory upon request. Since extensions of the working set of a process only happen at the granularity of pages from the operating system's point of view, allocators take on the task to further subdivide these pages to carve smaller regions for the application. Besides that, a number of non-functional requirements need to be met, such as minimizing memory usage and maximizing allocation speed [1, 12].

An NVRAM memory allocator faces similar challenges as volatile ones, however there are a number of important differences that need to be accounted for. First of all, we believe that the original functional requirement must be extended by a second one: Delivering a pointer to a previously allocated region. After a program restarts, it requires means of accessing its former allocations that were persisted on NVRAM.

Another extension is required on the list of non-functional requirements: Guaranteeing the atomicity of all allocations such that an application crash will never leave portions of the memory uninitialized or unreachable. This requirement is the direct result of our previous discussion about NVRAM challenges in Section 2.1.

2.2.1 Memory Pool Differences

To satisfy dynamic allocations from applications, volatile memory allocators build a per-process heap by requesting virtual memory from the operating system using the deprecated and less functional [13] `sbrk` or more portable `mmap` system calls. To reduce the costs incurred by these calls, memory is typically requested in larger chunks of several megabytes [6, 13]. Requesting too much memory at once may quickly exhaust physical limits, after which parts of the working set need to be paged out to make room for subsequent requests. Similarly, the memory overcommit feature allows for requesting nearly infinite amounts of virtual memory by deferring the mapping of virtual to physical pages until the memory is actually used. Neither of these features can easily be used on NVRAM. Since a file must be created and resized accordingly before mapping, any memory request will immediately (and permanently) decrease the available space on NVRAM. Therefore, an NVRAM allocator must choose the chunk size not just in regards to performance but also memory conservation.

Furthermore, volatile allocators do not need to care about where chunks are placed in virtual memory. During runtime

of the program, the address of a chunk will not change. Theoretically the same is true for NVRAM, but while volatile contents are erased upon termination and potentially rebuilt after a restart, NVRAM contents persist. To ensure pointer consistency, all chunk files must be mapped to the same (at least relative) address in virtual memory. Since virtual memory is shared with other components, there is no guarantee that a certain page will be available upon restart, jeopardizing integrity. A solution would be to track chunks and extend classic pointers to point to a specific chunk and then an offset inside it, but this would incur high costs for the translation each time an access occurs. Instead it is preferential to place all NVRAM chunks into a contiguous range in virtual memory.

2.2.2 AC(I)D-Compliant Allocation

The non-volatile memory pool must always be consistent (or in a state that can be made consistent) in order to be reusable after a program restart. For that, an NVRAM allocator must maintain the AC(I)D properties in regards to all allocator metadata. Updates to metadata require protection through flushes and memory barriers, and potentially even more elaborate logging constructs. Theoretically, all volatile allocation mechanisms and strategies can be used in the non-volatile domain as well.

Some approaches tackle this specific issue by introducing a logging system similar to database logging [3, 10, 19], in which allocator operations are wrapped in a transactional context. By maintaining a write-ahead log, the necessary information for a rollback or replay after a crash is present. While possible, we believe that it is possible to avoid the overhead of logging and still provide fast and safe allocator operations.

3. nvm_malloc: A FAST AND SAFE NVRAM ALLOCATOR

We now present `nvm_malloc`, a novel concept for a failure-atomic and fast general-purpose NVRAM allocator that has a built-in near-constant time recovery functionality. While we assume a system readily configured to support PMFS or a similar file system, we chose to design `nvm_malloc` itself to be minimally invasive. This means that it operates in user mode and does not require any currently unavailable hardware mechanisms or changes to the operating system.

3.1 Functional Overview

We start by giving a high-level overview of the functional design of `nvm_malloc`. The overall goal was to provide flexible, safe and fast access to NVRAM through an interface familiar from volatile memory allocators. `nvm_malloc` should additionally provide means of easily locating allocated regions to reduce the overhead of crash recovery.

`nvm_malloc` is designed to be used in parallel to a volatile memory allocator instead of replacing it, whereby both allocators operate on the same virtual address space as depicted in Figure 1. By explicitly using a different allocator, the application is aware of which regions are persistent and which are not. In order to avoid collisions between the allocators, `nvm_malloc` uses a dedicated range of virtual addresses.

If an application wants to allocate data structures on persistent memory, it can do so through `nvm_malloc`'s interface shown in Table 3. As opposed to volatile allocators, allocation of a persistent region requires a two-step process we

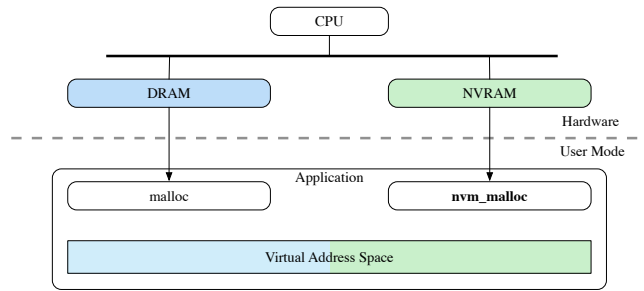


Figure 1: Conceptual architecture of an application using `nvm_malloc`

	API
1)	<code>nvm_initialize(working_dir, recover_flag)</code>
2)	<code>nvm_reserve(size)</code>
3)	<code>nvm_activate(ptr, lp1, tgt1, lp2, tgt2)</code>
4)	<code>nvm_free(ptr, lp1, tgt1, lp2, tgt2)</code>
5)	<code>nvm_reserve_id(id, size)</code>
6)	<code>nvm_activate_id(id)</code>
7)	<code>nvm_free_id(id)</code>
8)	<code>nvm_get_id(id)</code>
9)	<code>nvm_persist(addr, len)</code>

Table 3: `nvm_malloc` public interface - `lp` denotes link pointers, explained in Section 3.5

explain in Section 3.5, consisting of a reservation and an activation step. This separation is necessary to guarantee that no allocated region can be lost permanently.

Persistent allocations are preserved across program (and system) restarts and thus an application requires means of locating previously allocated objects. For this purpose, `nvm_malloc` provides methods (5) through (8), which allow specifying a string identifier (ID) for allocations. Using the `nvm_get_id` method, an application can retrieve formerly allocated regions in constant time. Both the regular methods (2) through (4) and the `*_id` methods are available, since not all allocations need to be named, such as for instance in the case of linked data structures. An application only needs to ensure that a root object is accessible via ID, from which the remaining objects can be reached via pointers. Instead of a name, these calls must be supplied link pointers, the purpose of which we discuss in Section 3.5.

Since modifications on NVRAM must reliably be made durable, `nvm_malloc` provides with (9) a convenient shortcut method to flush a given address range from the CPU caches onto NVRAM and issue a subsequent memory fence to ensure that no other modifications can be made before the call returns. This method is not used for allocations but rather on the allocated content.

Using this interface, an application can allocate its data structures on NVRAM with little overhead. Listing 1 provides a simple example of a linked list creation. For clarity's sake we excluded the conversion of relative to absolute pointers. First, `nvm_malloc` is initialized by specifying a working directory in Line 3. The root element of the list is allocated with the name "mylist" in Line 5, initialized in 6 and 7 and flushed for persistency Line 8 before it is activated in Line 9.

The next element in the list is reserved in Line 11, initialized and flushed. In Line 15, the new element is activated by

Listing 1: Exemplary usage of `nvm_malloc`- conversions from relative to absolute addresses and vice versa omitted for brevity

```

1 #include <nvm_malloc.h>
2
3 nvm_initialize("/path/to/workingdir");
4
5 elem_t *root = nvm_reserve_id("mylist",
    sizeof(elem_t));
6 root->next = NULL;
7 root->prev = NULL;
8 nvm_persist(root, sizeof(elem_t));
9 nvm_activate_id("mylist");
10
11 elem_t *e = nvm_reserve(sizeof(elem_t));
12 e->prev = root;
13 e->next = NULL;
14 nvm_persist(e, sizeof(elem_t));
15 nvm_activate(e, &root->next, e, NULL,
    NULL);

```

passing the `next` pointer of `root` as link pointer and `e` as a target. Now the simple list of two elements is persisted and can be retrieved at any point using `nvm_get_id("mylist")`.

To aid adoption, `nvm_malloc` was implemented as a user mode library and - aside from support for PMFS or similar - requires no further modifications to the system. Other approaches solved the issue of a contiguous virtual memory pool by modifying the kernel to reserve a certain range of virtual addresses specifically for NVRAM usage. While `mmap` can be supplied a target address, reliably choosing a fixed range of virtual memory from user mode is difficult, as it is unclear which regions are used to load dynamic libraries due to address space layout randomization (ASLR, [17]). We chose a different approach for `nvm_malloc` by reserving a range of virtual addresses dynamically and letting `mmap` decide the placement. As a downside, the start address of the range is non-deterministic and therefore all pointers stored on NVRAM must use an addressing scheme relative to the beginning of the range.

Keeping in mind the expected higher access latencies of NVRAM compared to DRAM, we limit accesses to NVRAM as far as possible by only keeping minimal metadata on NVRAM in the form of header structures for regions. This information is replicated in DRAM using the newly developed VHeader concept explained in Section 3.3.

Besides the costs of a cache line flush itself, the side effects can actually have an even larger impact on performance, namely if unrelated data is evicted from caches in order to persist data occupying less than a cache line. To avoid unnecessary cache evictions, we therefore chose to impose a strict 64-byte alignment requirement on all data on NVRAM, including both allocator metadata and user allocated regions.

In the remainder of this section we explain the various components and mechanisms in `nvm_malloc`. First, we describe the basic memory management concept and how the information is replicated in DRAM for better access. Next, we explain the allocation process and the arena structure used to increase scalability across multiple threads that allocate memory. Finally we show how the named allocation

Term	Definition
Chunk	Unit at which NVRAM is requested from the system via <code>mmap</code> , default size 4MiB
Block	Logical subdivision of chunks, blocksize = 4KiB
Run	A single block used to store multiple smaller allocations

Table 4: Memory management units

Class	Size (bytes)	Container
Small	[64, 128, 192, ..., 1984]	1 run
Large	$2048 \leq size < \frac{chunksize}{2}$	1.. <i>n</i> blocks
Huge	$\frac{chunksize}{2} \leq x$	1.. <i>n</i> chunks

Table 5: Allocation size classes

mechanism is implemented and explain the internal recovery process of `nvm_malloc`.

3.2 Basic Memory Management

The general design of `nvm_malloc` is based on the popular `jemalloc` [6] allocator, which we chose for its excellent scalability and overall performance. We will use the terms shown in Table 4 to refer to different memory management units. Memory is requested from the system in chunks with a default size of 4MiB, which can be broken down logically into blocks of 4KiB. To ensure that the separate chunks can be mapped into a contiguous region, a range of virtual addresses is “reserved” in the initialization phase of `nvm_malloc` using `mmap` and the anonymous mapping mode. Respective protection flags in the `mmap` call ensure that the reserved region can neither be written to nor read from. By default, `nvm_malloc` reserves a 10TiB virtual address range for this purpose. Now, `mmap` can safely be used to map files on NVRAM into the reserved range.

Allocations are split into three major size classes as shown in Table 5: small (under 2KiB), large (between 2KiB and half a chunk) and huge (larger than half a chunk). Huge allocations directly use one or more contiguous chunks, large allocations are placed into one or more contiguous blocks. Small allocations are subdivided into subclasses that are multiples of the 64 byte alignment requirement. Multiple small allocations of the same subclass are grouped into a so called run and stored within a single block. For all allocations, a small header structure is placed within the first 64 bytes, indicating type and size. A run header also contains a usage bitmap. The remaining space in a run block is logically split into equally sized slots of the respective size subclass, the *i*th bit in the bitmap indicates whether slot *i* is used or free. In contrast to `jemalloc`, runs are always a single block in size. As a result, given a pointer to an allocated region, the according header structure can always be found in constant time by rounding the address down to the nearest multiple of the block size. Depending on the subclass, runs may introduce some wasted space at the end of the block. Through the strict 64 byte alignment it is guaranteed that data is never inadvertently flushed from caches.

The alignment constraint also allows all headers to be 64 bytes in size, since any subsequent data would need to be placed at a 64 byte offset regardless. A block or chunk header is also used for unused regions, a flag in the header

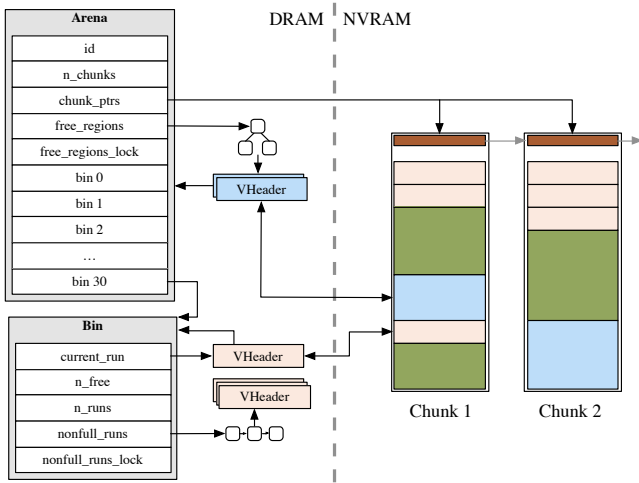


Figure 2: Overview of the arena architecture

indicates the free state. That way, all regions in the memory pool are accounted for at all times and each header contains the required information to locate the next one, making a full scan of NVRAM contents possible without further metadata.

3.3 VHeader Concept

To reduce access to the NVRAM headers as much as possible, `nvm_malloc` maintains volatile copies, VHeaders, in DRAM. VHeaders maintain a pointer to their NVRAM equivalent and are used to track regions in trees or hashmaps in DRAM without requiring NVRAM access. VHeaders are deallocated while a region is in use by the application and will only be recreated when freeing the region since no further tracking is required until then. The exception are runs which group multiple regions under the same header, thus the VHeader must be kept at least until all slots in a run are used. `nvm_malloc` does not deallocate `run` VHeaders at all and maintains a connection from the NVRAM header to the VHeader so it can be linked back into the free list once at least once slot becomes available again.

Instead of direct NVRAM accesses, all read-only operations can now be performed on the volatile copies. This reduces the number of accesses to NVRAM which, with the expected higher access latencies, improves the performance. Simultaneously, flushing NVRAM metadata for persistency reasons will not evict the VHeader from the cache hierarchy.

The increased memory consumption for this redundant storage of information furthermore does not alter memory footprint on NVRAM, and with VHeaders being small in size only comes at minimal costs.

3.4 Arenas

Huge allocations are served from a global tree storing free chunk VHeaders. The logic necessary to apply the block and run concepts is contained within an arena, a purely volatile structure that manages small and large allocations. Figure 2 gives an overview of the arena concept in `nvm_malloc`. Arenas are assigned initially one chunk from which they can carve regions in the form of one or more consecutive blocks. If an arena runs out of space, it may also request additional

chunks from the global chunk tree or request explicit chunk creation if no more free chunks are left. The ad-hoc addition of chunks means that they cannot be guaranteed to be contiguous, making another case for the large class cap at half a chunk since allocations in arenas cannot go beyond chunk boundaries.

To distinguish chunks used for arenas from chunks used directly for huge allocations, a usage flag in the header is used. Since the chunk is to be split into equally sized blocks, arena chunk headers are also larger due to padding necessary to align the first block at a 4096 byte offset. At 4MiB total size, an arena can thus create $\frac{4MiB - 4096B}{4096B} = 1023$ usable blocks per chunk. The remaining space in the first block is used for the object table explained in Section 3.6.

An arena maintains a separate volatile bin for each of the 31 subclasses, which stores the bin’s size class and a linked list of all VHeaders of non-full runs of the respective size. A separate `current_run` pointer is used to provide direct access to the last used non-full run. Full runs are not tracked, instead they will be added back into the non-full list once an element in the run is deallocated. The run list in a bin is protected by a mutex specific to the bin.

Initially, the usable content of an arena chunk is represented by a single free range spanning 1023 blocks. Free ranges use the same headers as regular blocks, but indicate their free state through a special flag. Arenas in `nvm_malloc` use a tree to store references to free block ranges sorted by size. When n free blocks are required, an upper-bound search on the tree will return a free range covering $n' \geq n$ blocks. The remaining range of $r = n' - n$ blocks is split off and reinserted as a new free block into the tree.

Multithreaded allocations are supported in all cases with the help of locks. However, especially in highly parallel applications this often leads to high lock contention and decreases performance. Like `jemalloc`, `nvm_malloc` therefore uses multiple parallel arenas which are assigned to threads in a round-robin fashion. By default, `nvm_malloc` will create 20 parallel arenas, but the number can be adjusted as a compile time constant. Instead of choosing a fixed number at compile time, it would be possible to dynamically create an arena per thread, but this would involve creating a chunk for every thread accessing `nvm_malloc`. It is neither guaranteed nor likely that each thread will perform enough allocations to utilize the entire chunk, which would result in unnecessarily wasted space. For maximum scalability in environments with sufficient amounts of NVRAM, it could however be considered as an optimization.

With multiple parallel arenas, all small and large allocations can be performed without interference in an application that uses 20 threads or less, since any locking required for arena tasks uses arena-specific mutexes. For small allocations this goes even further, as only the respective bin mutex must be acquired.

A DRAM hashmap is used to store the assignment of thread identifiers to arenas for constant time lookup. The association does not need to be persisted as it happens on an ad-hoc basis. For mere allocations in a scenario with fewer concurrently allocating threads than arenas, no locking aside from the global chunk lock would be required. However, deallocations require this degree of locking, as it is not guaranteed that the deallocation is performed by the thread assigned to the respective arena.

3.5 Safe Allocation Strategy

When allocating memory on NVRAM, the application must create persistent links to the allocated region before it is marked as in use to avoid permanent memory leaks. Simultaneously, the region should be initialized with values before persistently being linked into an existing data structure to avoid costly sanity checks of all data structures after a crash. Therefore, an allocation on NVRAM requires repeated interaction with the application.

To solve this issue, `nvm_malloc` relies on a concept proposed in `pmemalloc` [16], in which allocations are split into two distinct steps: reserve and activate. After the reserve step, the allocated region is returned to the application, which can then initialize the contents. `pmemalloc` uses a distinct state in the region header to indicate that the region was not activated yet. After the contents are initialized, the activation method is called, which changes the state again to indicate that the allocation was successful. This three-state system however requires two subsequent flushes for switching from free to initializing to initialized.

In `nvm_malloc` we exploit the existing VHeaders: Instead of introducing an intermediary state, the reservation step is performed only on the volatile structures. Since all algorithms rely on the information in the volatile data structures, this is sufficient to hide the region from concurrent allocations. Only once the application issues the activation of a region, the changes are persisted on NVRAM. Besides a pointer to the region, the activation method also takes up to two *link pointer* addresses with according *target* addresses, guaranteeing the pointers to point at their specified targets in a failure-atomic fashion once the activation returns.

3.5.1 Reserve

In the reserve step, the actual allocation task is executed by finding or creating a free region large enough for the request. For huge allocations, a free chunk range is used from the global tree or a new one created, whereas small and large requests are passed on to the allocating thread's arena.

Large requests search the arena's free block tree for a sufficiently sized range, potentially splitting off and reinserting the remainder. For a small request, the according subclass is determined and its bin consulted. If the current run is full, another run is selected from the non-full list as the current run or, if no non-full run exists, a new run is created. A sequential scan of the run's bitmap determines the slot in the run in a first-fit manner. At only 8 bytes in size the bitmap can be loaded into the CPU register as a whole and, depending on the bin's size class, not even the entire bitmap must be scanned, making this a fast operation.

The worst case complexity of the reservation algorithm is $O(\log n)$, where n is the number of elements in either the free chunk tree for huge allocations, or the arena free block tree for large and small allocations.

Another interesting aspect is the number of flushes involved. In the best case for small allocations, the current run in a bin is non-full and reserving a slot requires no flushes at all. If a new run must be created or for large allocations, as long as a free range of the desired size exists in the tree, no flush must be performed either. If a larger free range must be split is involved, two flushes are necessary to persist the inserted new and the adjusted old header. The same applies to huge allocations if a chunk range must be split.

3.5.2 Activate

The `pmemalloc` interface offers a separate `on_activate` method, which takes a persistent pointer and a target address and stores them in a special section in the header for a region. For `nvm_malloc`, this mechanism had to be altered slightly. Firstly, due to the 64 byte size restriction and 8 bytes for a link pointer and its target value each, only two pointer-value-pairs are supported. Secondly, while the link pointer fields in block and chunk headers are specific to a single allocation, run headers are shared among all allocations in the run. Two threads assigned to the same arena trying to activate two slots in the same run could therefore collide by overwriting each other's link pointers if they were written in a separate step. In `nvm_malloc`, the link pointers are specified in the activation call, which can lock the bin to prevent concurrent access.

The activation call takes the address of the allocated region and retrieves the according header section, which is located at the previous block boundary. Next, for large and huge allocations, the link pointers are written into the respective header on NVRAM and the state is changed to indicate the pending activation. Since both are located on the same cache line, a memory fence between the two instructions followed by a flush after the state change is sufficient. A crash before the changed state would cause the region to be recovered as free. Otherwise, the fence ensures that the state alteration was preceded by writing the link pointers and a replay is possible.

In case of small allocations, changing the state itself is not sufficient because a run header is shared among all elements in the run and the slot of interest must be marked in the bitmap. An additional field in the header is therefore used to store the bit index of the slot to be activated. Considering that runs cannot contain more than 63 elements, a single byte is sufficient for this value. The activation of a small allocation thus writes the link pointers *and* the bit index, followed by a memory fence, state change and flush.

In all cases, the link pointers are now set to their specified target values and flushed, and for small allocations the specified bit in the bitmap is set. Next, the state is changed to reflect the activation being successful and, after a memory fence, the link pointers (and bit index) are cleared before the header is flushed. After the state change the allocation is successful. The recovery algorithm may redo setting the link pointers (and updating the bitmap) without any collisions.

The activation algorithm as described has constant time complexity, since it only requires setting and flushing the header state. As far as flushes are concerned, a minimum of two are necessary for atomic changes of the header, with additionally up to two further flushes to persist the link pointers.

3.5.3 Deallocate

Deallocation face a similar problem as allocations in that they require working with link pointers to ensure consistency, except link pointers here are used to safely remove links to the region. The other difference is that deallocations for small and large regions are not necessarily performed by the thread assigned to the responsible arena. Retrieving the header for the allocated region is trivial as explained, but the freed range must be returned to the correct arena. Each header therefore includes the ID of its arena, to which the deallocation request is then passed on.

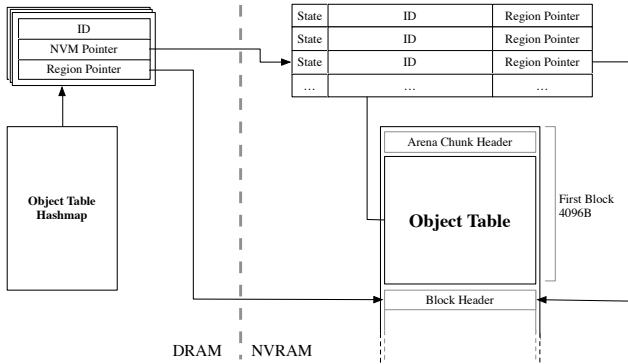


Figure 3: Overview of the object table in nvm_malloc

Deallocation of a region follows the same pattern as the previously described activation step, with the exception that a different state is used to indicate the freeing process. Afterwards, the now free region is returned to the respective free tree by recreating its VHeader and linking it into the tree. Runs again are treated differently in that a run will not be destroyed and converted back to a free block after all slots were freed. Only a run that was formerly full is inserted into the according bin’s non-full run tree, but nothing happens if the deallocation completely empties the run. This is based on the assumption that an application is likely to make subsequent requests for the same size class.

3.6 Allocation by ID

Persisted memory contents are not sufficient by themselves, an application must still be able to locate formerly created data structures to reuse them. The issue is very similar to data stored on hard disks, where some form of index is needed to locate files. Instead of burdening applications with implementing their own storage management concept, filesystems were developed to provide a consistent abstraction and file names can be used to reference “allocated” data. The same reasoning can be applied to NVRAM and nvm_malloc therefore implements a secondary allocation mode, in which the application can specify a string ID to be associated with an allocation.

nvm_malloc provides an object table to associate single allocations with a unique identifier in order to facilitate data organization and recovery. IDs are stored in a volatile hashmaps shown in Figure 3, allowing for constant time lookups of named regions. The core idea of the object table is to allow the user to “name” allocated regions for easier access and recovery. Reservation and activation of named regions also do not require link pointers to be provided, as the object table entry implicitly ensures their reachability.

Named allocations are meant for “root” objects, i.e. data structures that are not part of (and thus referenced by) larger structures but serve as an entry point to the data structure. Upon restart, the application may query the object table for named structures and whether they were created previously. Insertion and maintenance of object table entries come at increased allocation costs and therefore should not always be used. Instead, especially for linked data structures, the idea is to name the root object (e.g. root node in a tree, head element of a linked list) and allo-

cate the remaining elements using the standard method and linking them into the data structure in the activation step.

Object table entries must be persisted, which obviously requires space. As described in Section 3.4, arena chunk headers must include a padding after the 64 byte header for block alignment. The remaining 4032 bytes are therefore used to store object table entries, which consist of a state flag, a pointer to the allocated region and the ID. Each entry is exactly 64 bytes in size, limiting the ID to 55 characters. When using an ID, both reservation and activation, as well as deallocation require additional steps. Akin to the VHeader concept, object table entries are replicated in DRAM for faster access.

3.7 Recovery Mechanism

Before an application can make use of its persisted data, nvm_malloc must first rebuild its own data structures in case of a recovery. This process must include a sanity check of NVRAM headers in case activation or deallocation was interrupted, and a recreation of the object table so former allocations can be retrieved by ID. While a full scan of the memory regions is necessary to find free blocks, this is not required for correct operation of nvm_malloc. Instead each arena is recreated with initially no free regions and allocation requests will simply allocate a new chunk. VHeaders for used regions will be recreated once a deallocation happens.

This allows for a nearly constant time recovery process. Nearly, because the object table must be scanned in its entirety, but with its size likely to be small the overhead is limited. To put an upper bound on sanity checks, a ring buffer of region pointers is maintained and used as a form of log for activations and deallocations. If a header is left in an inconsistent state after a crash, a pointer to it is guaranteed to be in the ring buffer. The recovery process must therefore only check the headers in this “log” and rebuild the object table before execution may return to the actual program. A separate detached thread meanwhile starts scanning the previously used chunks for free regions, adding them to their respective arenas. No conflicts can arise here as nvm_malloc has no information about them until they are added.

The detached recovery thread also recreates VHeaders for all runs it encounters, which could lead to problems if a VHeader is recreated in parallel by freeing a region in a run before the thread is finished or vice versa. To solve this, a global version number is maintained and incremented every time nvm_malloc is initialized in recovery mode. When a run header is modified, the current version number is stored. Thus, if either a deallocation or the recovery thread encounter a run header with an outdated version, a VHeader must be created and otherwise can be assumed to exist.

4. EVALUATION

We now evaluate nvm_malloc in several benchmarks and compare it against the default glibc allocator and jemalloc. In microbenchmarks we analyze the runtime performance in various scenarios, as well as the impact of nvm_malloc parameters on its performance.

4.1 Benchmark Setup

All evaluations were performed on a dual-socket Intel64® Xeon-EP platform, each processor running at 2.6GHz with 8 cores, and support for up to 4 DDR3 Channels. As the evaluation of software for NVRAM is currently challenging

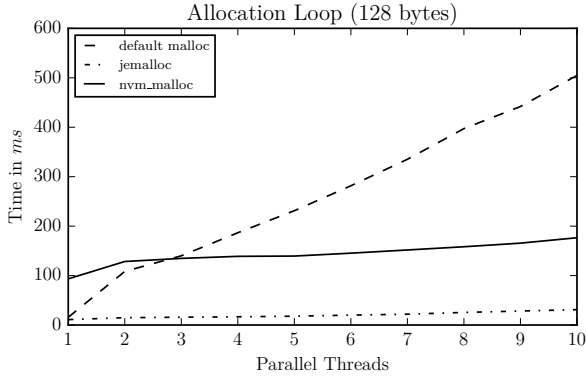


Figure 4: 128 byte allocations in a loop

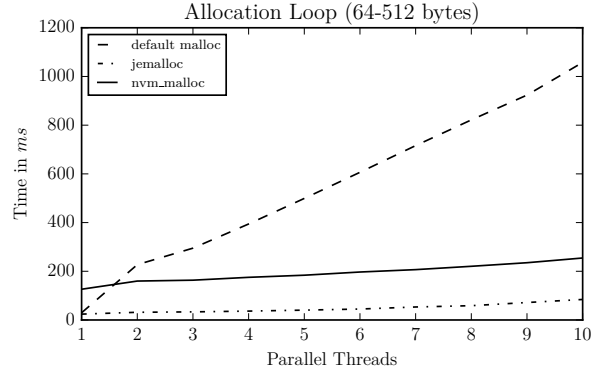


Figure 5: Random allocations between 64 and 512 bytes in a loop

due to a lack of hardware, we used the Persistent Memory Emulation Platform (PMEP) [5] to emulate the performance study of software for a range of latency and bandwidth points interesting to emerging NVRAM technologies. 256 GiB of memory were reserved using the `mmap` kernel boot parameter, onto which PMFS was mounted. The platform is capable of simulating increased latencies by applying last-level cache miss penalties to the PMFS memory regions and provides simulated `CLFLUSHOPT` and `CLWB` instructions. In order to avoid distortions of benchmark results through NUMA effects which are beyond the scope of this work, we used the `hwloc` framework [2] to restrict execution to a single node of 10 cores. All code was compiled using gcc version 4.8.2 and full optimizations.

4.2 Microbenchmarks

We decided to evaluate the performance of `nvm_malloc` against the standard glibc memory allocator and `jemalloc` as contenders. The only two non-volatile allocator implementations publicly available at the time of writing were `NVMalloc` and `pmemalloc`. The latter employs a simple first-fit algorithm and thus scored very poorly in our tests as runtime costs increase with every allocation. `NVMalloc` in turn did not actually use non-volatile memory and provided no recovery mechanism, therefore we excluded it as well for fairness reasons.

4.2.1 Allocation Performance Comparison

The first micro-benchmark performs allocations in a loop without deallocations. The loops were executed by an increasing number of parallel threads to stress test scalability of the allocator implementations. Both the default allocator and `jemalloc` just use `malloc`, whereas `nvm_malloc` performs both the reserve and activate step for each allocation. The benchmark was run in different configurations, one with uniform allocations of 128 bytes each and another with random allocation sizes uniformly distributed between 64 and 512 bytes.

Figures 4 and 5 show the results of the benchmarks. As expected, `nvm_malloc` incurs higher costs compared to its volatile contenders. This is explained by the increased overhead due to the reserve/activate mechanism, as well as the fact that `nvm_malloc` as a research prototype lacks the maturity in optimization of the others.

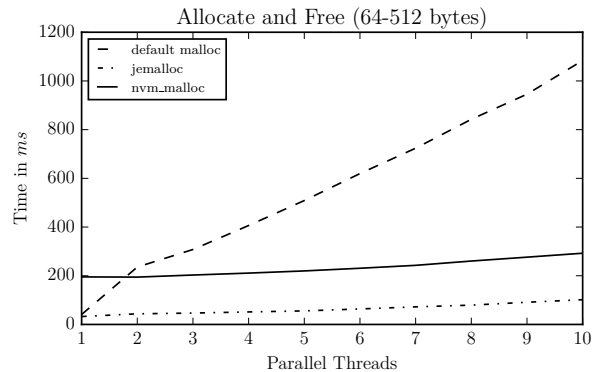


Figure 6: Performance comparison when freeing memory after allocations

However, it is notable that `nvm_malloc` shows scalability similar to `jemalloc` and outperforms the glibc allocator once several threads allocate memory in parallel.

Next, a second benchmark was run in which the allocation loop was extended to free all allocated memory in the end, the runtime of which was included in the measurement. In addition to the allocation mechanism itself, this also puts stress on the deallocation, which can be more difficult in a multithreaded environment as deallocations might occur in arenas not assigned to the current thread. The results are shown in Figure 6. Figure 7 shows the results of another experiment in which, after allocating and freeing, the allocation loop is performed again.

Freeing the memory in the end comes at minimal cost for all three allocator variants compared to the pure allocation loop. When performing additional allocations after freeing, both volatile allocators make use of caches to speed up allocations since the pattern remains the same. Unsurprisingly `nvm_malloc` does almost double its runtime for a second round of allocations due to the absence of such caches. However, scalability is hardly affected and the multithreaded performance remains better than the glibc allocator.

4.2.2 Recovery Performance

As explained in Section 3.7, the time needed for the internal recovery of `nvm_malloc` is only dependent on the num-

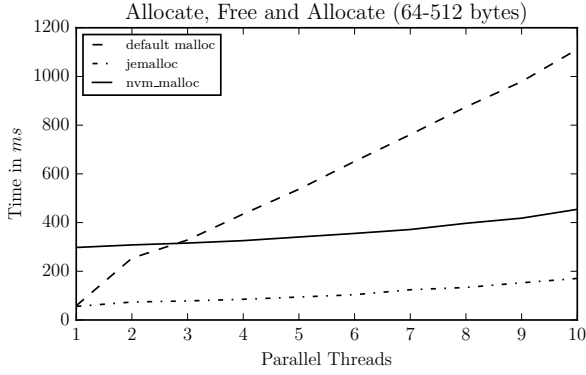


Figure 7: Performance of allocations on pre-used arenas

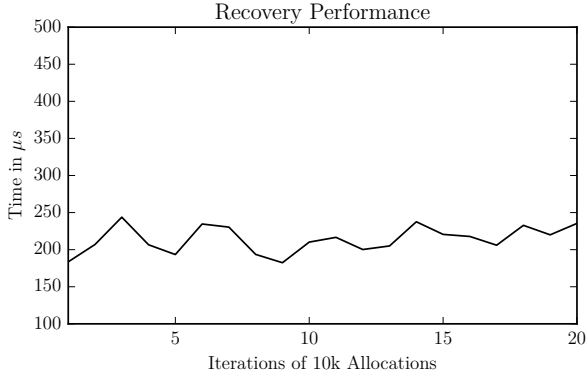


Figure 8: nvm_malloc internal recovery

ber of named allocations on an algorithmic level. To determine the actual recovery performance, a benchmark was executed in which linked lists of 10k elements were allocated on NVRAM, with the root element being a named allocation. Varying numbers of these lists were allocated, the allocator reset internally and a full recovery from NVRAM was performed. The time was taken until execution is returned to the application, at which point nvm_malloc is fully usable again and access to all formerly created regions is granted.

Without a required full scan of the memory region we expect not to see big differences in the recovery time in relation to the number of allocations. These expectations are confirmed by the outcome of the experiment shown in Figure 8.

Throughout the tests, internal recovery remained in the range of a few hundred microseconds. Only a significant number of object table entries makes a slight measurable difference. However, we believe it is fair to argue that on average the number of object table entries will remain small.

4.2.3 Persistency Costs

To determine the impact of various aspects of the persistency concept on the allocation performance, we ran the allocation loop benchmark in different configurations. We included two versions of the library that use the `CLFLUSHOPT` and `CLWB` instructions respectively, as well as the standard nvm_malloc with deactivated memory fences and/or `CLFLUSHes`. Out of these, we expect fences to have the least impact, followed by `CLFLUSHOPT` which is just a slight im-

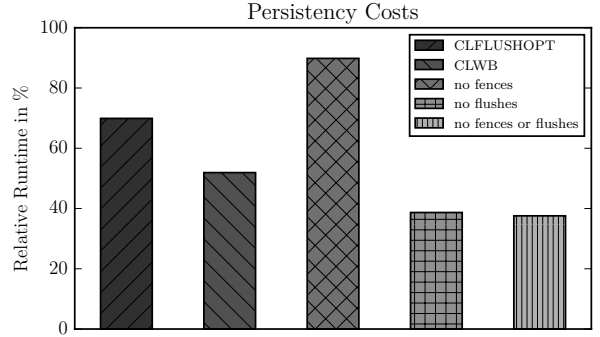


Figure 9: Runtime of nvm_malloc with different optimizations relative to baseline performance in an allocation loop

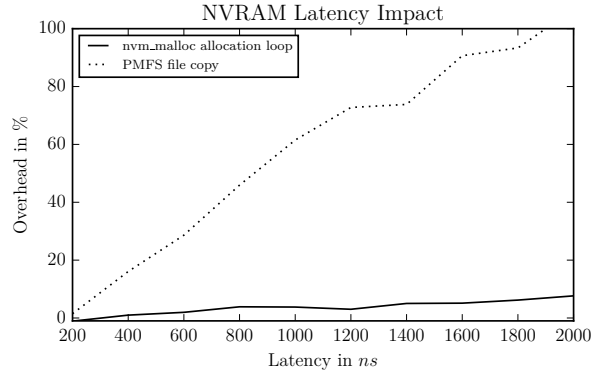


Figure 10: Comparison of runtime impact of increased NVRAM latency on nvm_malloc and file copy on PMFS

provement over `CLFLUSH`. Both `CLWB` and deactivation of flushes should, however, have a significant impact.

Figure 9 confirms these expectations. Fences are only accountable for a few percent of the runtime, but the switch to `CLFLUSHOPT` actually has a larger impact than expected. The benefit arises from not evicting non-dirty cache lines and the fact that the less expensive `SFENCE` can be used instead of a full `MFENCE`. The use of `CLWB` for persistency essentially reduces the runtime by 50% and is only beaten by completely disabling persistency guarantees provided by fences and flushes. Overall, these findings show that up to 60% of nvm_malloc’s runtime costs arise from necessary means to achieve guaranteed persistency.

4.2.4 NVRAM Latency Impact

With the testing platform capable of simulating increased access latencies, we used the allocation loop benchmark with nvm_malloc in its standard configuration to determine the impact of increased latencies on the allocation performance. As a baseline we included a benchmark that copies a 500 MiB file from one location on PMFS to another. We expect the copy operation to scale about linearly with the latency increase, as it is composed of direct sequential read and write operations. nvm_malloc in turn is optimized to avoid costly NVRAM accesses as far as possible, therefore we expect a much lesser impact in the allocation benchmark.

These assumptions are confirmed by the results shown in Figure 10, which shows the adjusted latency on the x axis

and the relative performance overhead compared to non-modified timings of 150 nanoseconds on the y axis. From 200 to 2000 ns, the file copy operation incurs an overhead of around 100% which was to be expected. `nvm_malloc` in turn proves to be very robust with under 10% overhead with latencies increased by an order of magnitude.

4.3 Evaluation Summary

As demonstrated, `nvm_malloc`'s single-threaded allocation performance is not fully on par with common volatile allocators, but can actually outperform the standard system allocator in a multi-threaded context. The overhead is not just due to the fact that `nvm_malloc` is not fully optimized, but to a large extent owed to the necessary means to achieve persistency.

We showed that fences and flushes make up as much as 60% of the overall runtime costs of `nvm_malloc`. Intel's upcoming persistent memory instructions `CLFLUSHOPT` and `CLWB` however can be used to achieve persistency at a fraction of the cost of the standard cache line flush.

Due to the smart placement of data in memory and the extensive usage of volatile copies of NVRAM data, `nvm_malloc` incurs little extra costs from increased NVRAM latencies.

In case of recovery, `nvm_malloc` requires only a few hundred microseconds to recreate its initial state and return execution to the calling application. As we showed this process is independent of number and size of allocations but only depends on the number of object table entries. We believe, however, that it is reasonable to argue that the latter will remain fairly small in number so that no massive performance decrease is to be expected here.

5. RELATED WORK

The possible applications of the emerging NVRAM technologies have gained increasing attention in the research community. Propositions were made from simpler use cases such as using NVRAM as fast buffer for flash-based storage media [9] or using it to store filesystem metadata for increased performance and reliability [7] to fully replacing DRAM altogether [15].

The previous work on volatile allocators is exhaustive [21] and nearly every aspect of memory allocation has been discussed thoroughly at this point, resulting in countless concepts and implementations of both general- and special-purpose memory allocators.

Flash-based non-volatile allocator concepts were developed such as NVMMalloc by Wang et al. [20]. By exposing an aggregated SSD array through a byte-level interface directly to the CPU's memory controller, the authors present an allocator concept that allows explicit allocation from persistent memory in a malloc-like fashion. Instead of creating a non-volatile pool, `mmap` is used directly to memory map a separate file per allocation request. Further subdivision of the allocated space is left to the user.

As one of the first NVRAM toolboxes, Swift et al. present Mnemosyne [19] as a set of tools for handling NVRAM, consisting of kernel modifications, user mode libraries and a custom compiler and linker, which are required for many of Mnemosyne's features. The authors implemented a persistent heap which is dynamically resizable, however instead of a contiguous pool, a master table is used to track scattered chunks. Pointer consistency is assured as Mnemosyne

operates on a separate range of virtual addresses implemented through their region manager. For the actual task of memory allocation, Mnemosyne furthermore uses two separate allocators, an NVRAM-adapted version of Hoard [1] for allocations below 8KiB and an unaltered `dlmalloc` [12], which is wrapped in a transactional logging system to ensure AC(I)D properties. The modified version of Hoard is not explicitly benchmarked, nor is the source code available and thus no performance comparison can be made. However, the logging system used for `dlmalloc` and other components of Mnemosyne incurs the overhead of redundant storage of data, which leads us to assume degraded performance.

NV-Heaps by Coburn et al. [3] is similar to Mnemosyne. They also provides a complete suite to simplify working with NVRAM, but use a different approach towards allocation using reference counting and garbage collection. They also facilitate the concept of atomic sections like Mnemosyne to ensure atomicity via logs. In contrast to Mnemosyne, NV-Heaps offers the concept of persistent objects; the allocator used is not exposed to the calling application. Again, no details about allocator performance are discussed. NV-Heaps requires hardware changes in the form of atomic 8-byte writes and epoch barriers as developed closed source for BPFs [4]. Like Mnemosyne, NV-Heaps is not publicly available and was thus unavailable for our evaluations.

A more specific approach for an NVRAM allocator is NVMMalloc (not to be confused with above flash-based allocator) by Moraru et al. [14], intended to be a fast, general purpose NVRAM allocator. Instead of relying on explicit flushes, the authors suggest a CPU extension in the form of cache line counter to track if regions were flushed already. The focus of NVMMalloc however is largely on wear-aware allocation algorithms to increase the lifetime of hardware and protecting memory regions from unauthorized writes. Furthermore, in the publicly available source code³, the authors construct a memory pool using the regular `mmap` system call, effectively requesting DRAM instead of working on NVRAM. The performance evaluation also shows that performance lags behind `nvm_malloc` and NVMMalloc does not provide a built-in concept for recovery.

Kannan et al. [10] recognize the inline storage of metadata in NVMMalloc to be a problem. In their approach, minimal metadata in combination with a hybrid logging system is used for increased performance. Targeting end-user devices with low-end hardware, the focus is on reducing overhead in a confined space, although the authors claim performance improvements over NVMMalloc.

The rather rudimentary `pmemalloc` [16] is another allocator for NVRAM, concepts of which are integrated into `nvm_malloc`. Its simplistic architecture provides fast allocations in single-threaded applications, but uses a fixed size memory pool and requires extensive locking for multi-threaded environments. A follow-up comparison to its successor `libpmemobj` is subject to future work.

To our knowledge, `nvm_malloc` is the first non-volatile memory allocator specifically developed for general purpose usage with performance in mind, that supports AC(I)D-compliant allocation, a centralized object recovery and discovery mechanism and provides allocation performance on par with volatile memory allocators.

³<https://github.com/efficient/nvram/tree/master/nvmalloc>

6. CONCLUSION AND FUTURE WORK

NVRAM presents a new and unique opportunity to revolutionize persistency concepts of software systems in general and main memory databases in particular. In order to take full advantage of this new technology however, hardware and software alike must be adapted. In this work we presented upcoming NVRAM technologies and discussed their potential methods of integration into current computer systems, along with the arising challenges that software programmers face in order to properly handle NVRAM. Concepts developed for the volatile world such as CPU caches become a problem in the non-volatile domain and actual solutions to these problems must still be found. The need for dynamic and fast memory allocation continues to exist in the NVRAM era and well-known allocator concepts must undergo non-trivial changes to make the transition.

We developed `nvm_malloc`, a novel NVRAM memory allocator concept for a fast, flexible and safe general-purpose memory allocator on NVRAM, which exploits the expected coexistence of volatile and non-volatile memory in a unified virtual address space for both atomicity of operations and performance alike. The allocator is the central point of recovery by integrating a filesystem-like naming concept for allocations that alleviates applications from the burden of implementing their own concepts. In various benchmarks `nvm_malloc` was compared against both volatile and non-volatile allocators and analyzed different parameters towards their impact on allocation performance. We demonstrated that, in certain scenarios, `nvm_malloc` can even outperform standard volatile allocator implementations, which solely work on DRAM and provide no persistency at all. We furthermore showed that flushes are responsible for about 25 percent of `nvm_malloc`'s runtime costs.

While currently still subject to limitations such as cache eviction as a necessary means for persistency, we believe that NVRAM is a highly promising technology and are excited to see how it will change our understanding of persistency in the long run.

Acknowledgments. The authors thank Subramanya R. Dulloor, Jeff Jackson, Detlef Poth, Brad Bickford and Andy Rudoff from Intel for their support, feedback and hardware access.

7. REFERENCES

- [1] 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.
- [2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.
- [3] 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. 39(1):105–118, 2011.
- [4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through Byte-Addressable, Persistent Memory. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP*, 2009.
- [5] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. *Proceedings of the Ninth European Conference on Computer Systems - EuroSys*, 2014.
- [6] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*. Citeseer, 2006.
- [7] K. M. Greenan and E. L. Miller. Prims: Making nvram suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07)*, 2007.
- [8] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? In *ACM SIGPLAN Notices*, volume 34, pages 26–36. ACM, 1998.
- [9] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *Computers, IEEE Transactions on*, 58(6):744–758, 2009.
- [10] S. Kannan, A. Gavriloška, and K. Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture*, 2014.
- [11] M. H. Kryder and C. S. Kim. After hard drives-what comes next? *IEEE Transactions on Magnetics*, 45(10):3406–3413, 2009.
- [12] D. Lea and W. Gloger. A memory allocator, 1996.
- [13] C. Lever and D. Boreham. Malloc performance in a multithreaded linux environment. 2000.
- [14] I. Moraru, D. G. Andersen, N. Tolia, M. Kaminsky, P. Ranganathan, N. Binkert, and R. Munz. Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores. 2012.
- [15] D. Narayanan and O. Hodson. Whole-system persistence. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 401–410, 2012.
- [16] A. Rudoff. pmemalloc, 2013.
- [17] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.
- [18] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.
- [19] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. 39(1):91–104, 2011.
- [20] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*, pages 957–968, 2012.
- [21] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.
- [22] X. Wu and A. Reddy. Scmfs: a file system for storage class memory. page 39, 2011.