Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

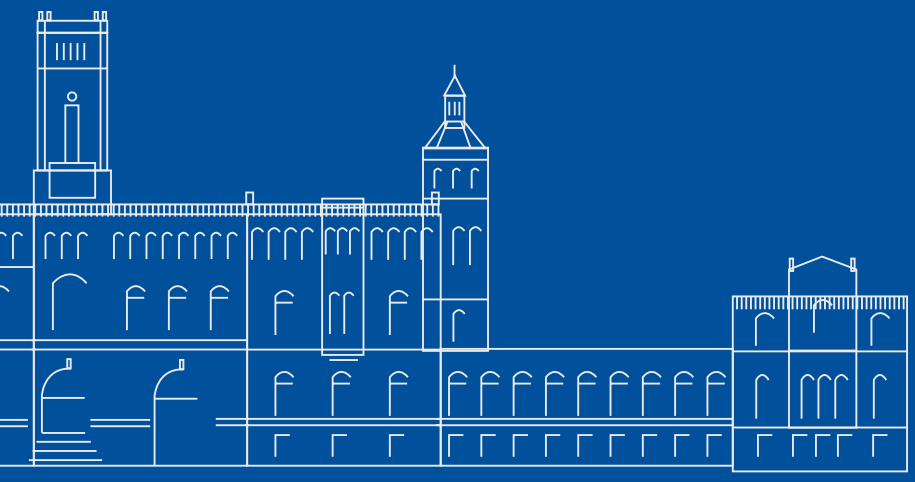
Lars Wrenger

Lo(ck|g)-free Page Allocator for Non-Volatile Memory in the Linux Kernel

Masterarbeit im Fach Informatik

2. Juni 2022

Please cite as:
Lars Wrenger, “Lo(ck|g)-free Page Allocator for Non-Volatile Memory in the Linux Kernel” Masters’s Thesis, Leibniz Universität Hannover, Systems Research and Architecture Group, June 2022.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Lo(ck|g)-free Page Allocator for Non-Volatile Memory in the Linux Kernel

Masterarbeit im Fach Informatik

vorgelegt von

Lars Wrenger

geb. am 2. August 1997
in Magdeburg

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Christian Dietrich**
Betreuer:	**Florian Rommel, M.Sc.**

Beginn der Arbeit:	**4. Oktober 2021**
Abgabe der Arbeit:	**4. April 2022**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Lars Wrenger)
Hannover, 2. Juni 2022

ABSTRACT

With the advent of new memory technologies, the focus in the operating system's research is currently shifting from processing elements towards memory. Especially byte-addressable persistent memory (NVM) is heavily discussed in the scientific community. Its applications have been theoretically explored even before Intel released capable hardware in 2019.

This work explores a new minimal, self-contained memory abstraction (*morsel*) that combines the two concepts of memory and file. A morsel is a small partial address space that can be directly integrated into the address spaces of processes or devices (DRAM, NVRAM, GPU, and Remote-DMA). Morsels build upon the (IO)MMU as the interface for memory sharing. They are lock-free to support non-OS-controlled devices that cannot participate in complicated locking protocols, like GPUs. A morsel can be stored in NVM to allow the shared memory to survive system crashes or power failures.

To persistently allocate pages for the metadata and content of morsels, a new scalable and crash-tolerant page allocator for NVM is needed. This work discusses the design and development of this new highly parallel allocator. It avoids locks for its data synchronization, as they are difficult to recover after crashes. Logging is another widely-used strategy to implement atomic transactions. However, the high amount of writes it causes makes it unsuitable for the current NVM hardware. Like SSDs, NVM can endure only a limited number of writes and thus uses costly wear-leveling techniques to increase its lifespan. Therefore, the developed allocator is both lock- and log-free. Also, different strategies for reducing fragmentation, metadata overhead, and memory sharing were implemented. They are compared with the Linux page allocator in several microbenchmarks. This work shows that these allocator variants recover from system crashes, significantly outperform the Linux allocator, and in some cases, even an allocator based on CPU-local linked lists.

KURZFASSUNG

Mit der Einführung neuer Speichertechnologien verschiebt sich der Fokus der Betriebssystemforschung aktuell von Prozessoren in Richtung Speicher. Besonders byte-addressierbarer, persistenter Speicher (NVM) wird in der wissenschaftlichen Community ausgiebig diskutiert. Seine Anwendungen wurden bereits theoretisch erforscht, bevor Intel entsprechende Hardware in 2019 veröffentlichte.

Diese Arbeit befasst sich mit einer neuen, minimalen, abgeschlossenen Speicherabstraktion (Morsel), welche die beiden Konzepte Speicher und Datei kombiniert. Eine Morsel ist ein kleiner Teil eines Adressraums, der direkt in die Adressräume von Prozessen oder Geräten eingehängt werden kann. Die Morsels bauen auf der (IO)MMU als Schnittstelle zum Teilen von Speicher zwischen verschiedenen Geräten (DRAM, NVRAM, GPU und Remote-DMA) auf. Um Geräte, die nicht vom OS kontrolliert werden und keine komplexen Synchronisationsprotokolle befolgen können, zu unterstützen, müssen Morsels lock-free sein. Eine Morsel kann zudem im NVM gespeichert werden, was es ihrem geteilten Speicher ermöglicht, Systemabstürze oder Stromausfälle zu überleben.

Um Seiten für die Metadaten und Nutzerdaten einer Morsel persistent zu allokieren, wird ein neuer, skalierbarer und absturzsicherer Seitenallokator für NVM benötigt. Diese Arbeit diskutiert das Design und die Entwicklung dieses neuen Allokators. Er vermeidet Locks zur Datensynchronisation, da diese nur schwer nach Abstürzen wiederhergestellt werden können. Logging ist eine weitere weit verbreitete Strategie, um atomare Transaktionen zu realisieren. Allerdings verursacht Logging eine hohe Anzahl an Schreibzugriffen, deshalb ist es ungeeignet für die aktuelle NVM Hardware. Ähnlich wie SSDs kann NVM nur eine begrenzte Anzahl an Schreibzugriffen aushalten und nutzt daher teure wear-leveling Methoden, um seine Lebensdauer zu erhöhen. Deshalb ist der Allokator sowohl lock-free, als auch log-free. Darüber hinaus wurden verschiedene Strategien implementiert, um Fragmentierung, Speicherverbrauch und Memory-Sharing zu verringern. Diese werden in mehreren Micro-Benchmarks mit dem Linux Kernel Allokator verglichen. Die Arbeit zeigt, dass diese Allokatorvarianten Systemabstürze überleben und performanter sind als der Linux Allokator und in einigen Fällen sogar als ein Allokator der auf CPU-lokalen verketteten Listen basiert.

CONTENTS

1 INTRODUCTION

The memory hierarchy is becoming increasingly deep for general-purpose operating systems. In addition to caches, DRAM, and hard disks, there are many new memory technologies in between. Examples of these are video memory (GPU), Remote Direct Memory Access (RDMA) over network cards, or Non-Volatile Byte-Addressable Memory (NVM), like the Intel Optane DIMMs that were released in 2019. All of these have to be managed by the operating system (OS), together with the ability to directly access it (DAX) and share it between Cores, Sockets, devices (like GPUs), or even whole systems in the case of RDMA. Current concepts and abstractions come to their limits in representing these memory types [Pel+15]. This is especially difficult for NVM because it is byte-addressable like DRAM but also persistent like filesystems on hard drives [Bai+11; Bit+21]. Therefore, it has properties of both concepts: memory and file.

This work is part of the ParPerOS (Parallel Persistency OS) project, which proposes a new minimal and self-contained memory abstraction, named *morsel*, to overcome these limitations. A morsel consists of a small page table tree, similar to the virtual address space mappings of the OS. Depending on their depth, these small page table trees can span over different sized chunks (2 MiB, 1 GiB, 512 GiB, ...). As they are compatible with (IO)MMUs, they can be directly *mounted* into a virtual address space by integrating them as subtrees into the current page table tree. This allows direct unconstrained access to the mounted morsel.

The primary goal of this new method of integrating external memory into address spaces is to improve performance due to the closer proximity to the hardware. Additionally, morsels are developed to support non-OS-controlled memory, like video memory. This requires that morsels are lock-free, as non-OS-controlled devices like GPUs cannot use complex interfaces that require locks or logs. This lock-free property also satisfies the persistency requirements of modern non-volatile memory: In the past years, additional considerations had to be made to guarantee persistency on Intel's first generations of Optane memory [IMS16b; Fu+21], including inserting manual cache line flushes [IMS16a]. This, however, changed when Intel announced the extended asynchronous DRAM refresh (eADR) "power failure protection domain" in early 2021 [Int21]. This domain guarantees that caches, which were previously lost after a power failure, are also being persisted. Consequently, any lock-free algorithm is also persistent in this new persistency model.

A fast and persistent page allocator is required to allocate the pages for the morsel's page tables and, optionally, its user data. Related work on physical page allocators for NVM is relatively rare, as most persistent allocators are designed explicitly for general-purpose userspace applications [Mor+13; Sch+15; BCB16]. Therefore, this work aims to develop a new persistent, highly scalable, and lock- and log-free (lo(ck|g)-free) page allocator. Like the morsel, it also uses page tables to store its metadata and manage its free pages. Updates on page table entries are performed using atomic compare-exchange operations. Thus locks and complex logging algorithms, which are suboptimal

for NVM due to its write wearing, are not needed. This NVM allocator can allocate small (4 KiB), huge (2 MiB), and giant (1 GiB) pages. To reduce memory sharing, the CPU cores exclusively reserve subtrees that span over 1 GiB and contain 512^2 4 KiB pages or 512 2 MiB pages. All subsequent allocations are performed in the reserved subtrees. The allocator's architecture is subdivided into a persistent lower part responsible for the inner subtree allocations and a volatile upper part that manages the subtrees the lower allocator uses. The volatile upper part is rebuilt on boot from the persistent lower allocator. Different strategies for storing the page tables of the lower allocator in NVM were developed and compared. Also, various upper allocator strategies were evaluated using additional page tables or linked lists to manage the subtrees. They have different performance characteristics and different memory overheads for their metadata. The allocators also contain mechanisms to avoid fragmentation by prioritizing partially-filled page tables for allocations. Several optimizations have been developed to reduce NVM accesses and avoid memory sharing, including false sharing. The latter is especially problematic because different concurrently-accessed page table entries often are part of the same cache line, resulting in costly cache-line invalidations. In these cases, core-local replicas were used, retaining most updates.

The benchmark results show that the allocator's performance scales very well for high core counts and large amounts of memory. Its reallocation performance is up to twice as good as the Linux Kernel Allocator. The improvement is even more significant for bulk allocations, where many pages are allocated or freed simultaneously. Additionally, the allocator variants presented in this work are persistent, survive reboots, and recover from system crashes and power losses.

This work is divided into six chapters: After this first chapter, the next one (chapter 2) continues with an introduction to the basic concepts of modern hardware and OS architectures that guided the design of the allocator. After that, the *morsel* concept is discussed together with a top-down overview of the lo(ck|g)-free allocator's architecture (chapter 3). Then chapter 4 digs deeper into its implementation and describes the different approaches and optimizations from the bottom up. These allocator approaches are evaluated and compared with the Linux allocator in chapter 5. Finally, chapter 6 concludes the findings and discusses future works in this area.

2 FUNDAMENTALS

This chapter introduces the memory architecture and management of modern computer systems in section 2.1. It describes the basic concepts of memory management (section 2.2) and parallel computing (section 2.3) essential for the design of the lo(ck|g)-free allocator. Also, the current non-volatile hardware and its capabilities are discussed, together with its integration on the operating system level (section 2.4). Finally, the chapter closes with a review of related works in the NVM research area in section 2.5.

2.1 The Memory Hierarchy

Current servers and workstations use many different memory types combined in the same machine. These memory types differ in access times and size due to production costs and power consumption, as shown in Figure 2.1. At the bottom are the hard drives that are slow but cheap and have high capacities. Towards the top, the memory technologies become faster but more expensive and thus smaller. The general concept behind this hierarchy is to speed up program execution by copying the most accessed memory, from slower, high-capacity memory to faster but smaller RAM, caches, and registers.

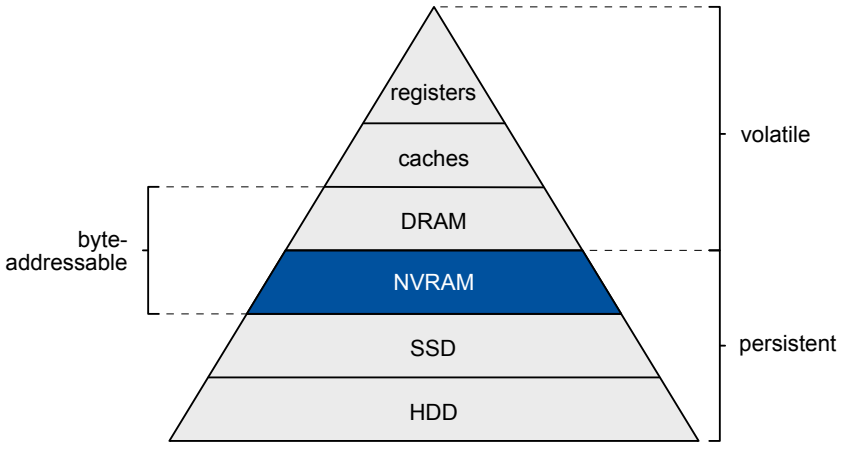


Figure 2.1 – The different memory technologies existing in a heterogeneous system, ordered by their capacities and access times.

CPU registers are the fastest available memory, while various levels of CPU caches follow close behind. This work focuses specifically on Intel x86 Cascade Lake and newer, as this is the first generation that supports Intel's non-volatile memory. This architecture has three levels of caches [HP11]. Each core has its own L1 and L2 caches, where the L1 cache is further split into separate instruction and data caches. The L3 cache (or Last Level Cache (LLC)) is shared between all cores. In these caches, recently accessed memory chunks are stored for faster access. These memory chunks or *cache lines* have a size of 64 Bytes on x86. For each memory access, it is first tried to load the data from the L1 cache. On a miss, the L2 and later the L3 caches are also checked. If the data is not in the cache, it has to be loaded from the main memory into the cache, which takes a lot more CPU cycles. On the software layer, this is abstracted away, as the software only sees the main memory. The caches cannot be accessed directly. It is up to the CPU to manage them. However, the x86 architecture has instructions to manually flush and write-back cache lines to main memory like `clflush`, `clflushopt` and `clwb`. Additionally, there are ways to circumvent caches like non-temporal stores (`MOVNTQ`) or the SSE and AVX instructions. They are beneficial when a large amount of data has to be written at once.

Main memory and byte-addressable non-volatile memory are located in the middle of the memory pyramid. The access times are more than two orders of magnitude slower than registers, but their size is much larger. They are directly connected over the memory bus with the CPU and thus have a very high bandwidth. Additionally, they are byte-addressable, which means that the CPU can directly access every byte of this memory.

SSDs and hard drives are connected over the Peripheral Component Interconnect Express (PCIe) or the IO bus with the CPU and have higher access times and lower bandwidth. These devices are accessible at the granularity of blocks (usually between 512B and 4 MiB). This means that if the CPU wants to read a single byte, it has to load the entire block into the main memory (an exception to this is reviewed in subsection 2.4.2).

Another differentiating property of memory is whether it can keep its data in the event of a power loss. Volatile memory like registers, caches, and DRAM are cleared without power, while non-volatile memory like NVRAM, flash, and hard disks can maintain their data. Traditionally, the size and latency differences between volatile and non-volatile memory were very extensive. This gap has been closing only recently with advancements in solid-state drives and byte-addressable non-volatile memory. The latter has been commercially available since 2019 as Intel Optane memory, which is described in more detail in section 2.4.

2.2 Virtual Memory Management

Most registers are directly usable from userspace, and the CPUs control the caches directly. The main memory (DRAM) and anything below is managed by the operating system, which allocates this memory and shares it between processes.

Additionally, the OS uses the concept of (*virtual*) address spaces to further abstract memory access and hide the physical memory layout. For this, the memory is split up into pages (4 KiB on x86). An address space is a mapping from virtual pages to physical pages, as shown in Figure 2.2. Every userspace process has its own address space, its distinct view of the memory. Applications use virtual addresses to access memory, which are internally translated to physical addresses that point to the corresponding physical memory.

This memory virtualization has several benefits for the userspace applications and the OS [HP11]. First of all, it greatly simplifies the development of applications. Developers do not have the burden of managing the physical memory, which might have a limited size, is non-consecutive, contains

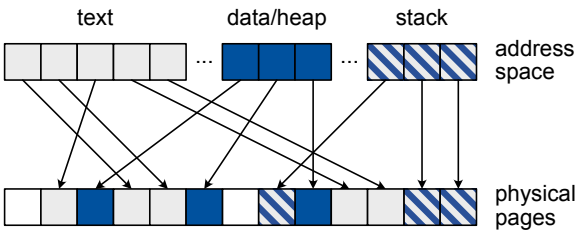


Figure 2.2 – Virtual address space of a process.
The sections *text* (containing executable code), *data/heap* and *stack* are common to most userspace applications.

areas occupied by memory-mapped devices, and is generally highly hardware-specific. On the other hand, a virtual address space provides a simpler view of the memory, hiding all of this complexity. It also allows deploying the same applications on systems with different memory configurations.

The OS also uses address spaces to *isolate* processes by their different views of memory. Thus, the address spaces include only the memory a process may access. Additionally, every memory region within the address space is protected by access properties, controlling how a process can use these different memory areas. For example, if the memory is writable or accessible from userspace. Memory isolation is a means of providing security and safety. It prevents processes from accessing the private memory of other processes and extracting sensitive data or corrupting memory and crash systems either by accident or malicious intent.

Besides these immediate benefits, memory virtualization allows the OS to save resources. The OS generally fills in the address spaces on-demand when a process accesses new memory. This saves memory, as processes seldom access all of their memory at the same time.

Memory virtualization also makes it possible to speed up userspace access to slower memory transparently by using DRAM caches. Like L1, L2, and L3 caches, the main memory is also split into chunks (4 KiB on x86), called pages. These pages can be used as caches for faster access to files on disks, similar to the CPU caches. It also allows the OS to specify more virtual memory than it has physical memory and to swap in pages from secondary storage (disks) if necessary.

2.2.1 Page Tables

The translation of virtual addresses to their physical counterparts is performed using a special mapping data structure. These maps have virtual addresses as keys and the corresponding physical addresses as their values, together with protection flags (like read-only or writable) and other metadata. On a 64 bit system, a complete table to translate a 64-bit address to a physical 64-bit address would need 2^{64} entries with a size of 64 bit (or 8 Byte). The resulting size would be about 150 Terrabytes for every address space mapping. To reduce this size, multiple hierarchical levels of smaller page tables are used. These page tables form a tree, where the page tables are the nodes, and the physical pages are the leaves. Each Page Table Entry (PTE) contains a pointer to the child page table and finally to the physical memory page. The virtual address translation is performed in multiple steps by walking the page tree down to the physical page. This tree is sparsely populated and thus takes up only a fraction of this memory.

A page table has 512 8 Byte entries and thus is 4 KiB large and fits into a single memory page. Each level corresponds to 9 bits in the virtual address ($2^9 = 512$), which are used as an index into the page table as shown in Figure 2.3. The final 12 bits of the virtual address specify the offset

within the physical memory page. Linux, for example, currently has four levels of page table that allow it to address 512^4 pages (256 TiB) in each address space. For the rest of this work, the page tables are numbered from bottom to top. The leaf pages at the bottom of the page table tree belong to level 0. Above that is the first level of page tables, that references 512 pages, and above that is the second level, referencing 512 1st-level page tables, and so on.

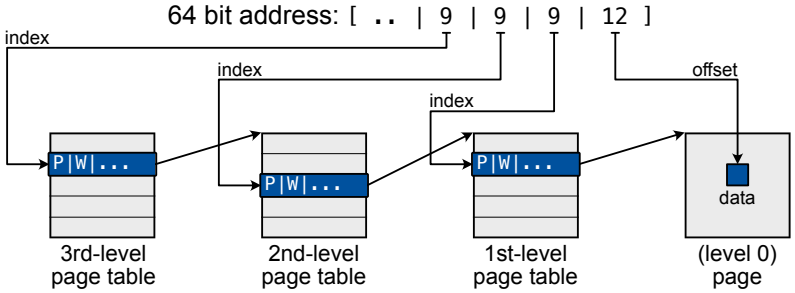


Figure 2.3 – Example of a 3-stage page table mapping.

This page table walk is relatively expensive, especially if it must be done for every memory access. Therefore, CPUs have Memory Management Units (MMUs) that perform the translations from virtual to physical addresses. They include special hardware for walking these page tables and caches to store recently accessed addresses, like the Translation Lookaside Buffer (TLB) on x86 CPUs. To also access other Direct Memory Access (DMA) devices like PCIe graphics cards, modern x86 CPUs have additional Input-Output Memory Management Units (IOMMUs). They function similar to MMUs and translate virtual addresses from the device's own address spaces to physical (main memory) addresses.

These page tables are usually filed on demand. If a process accesses memory that is not present in the page table tree, the page fault handler of the OS is executed, which allocates new memory and inserts it into the page table tree. Even the pages for the page table are lazily allocated when they are needed. The following section discusses the memory allocators responsible for these allocations.

2.2.2 Memory Allocators

In general, allocators can be separated into two categories, fulfilling different requirements. To the first category belong the general-purpose allocators typically used by userspace applications. Usually, they work entirely on virtual memory and provide functions to allocate and free variable-sized memory chunks. These heap allocators are implemented by the Standard Libraries, like the `malloc` implementation from `glibc` (GNU C Library) [Gli]. Similar to this is the general-purpose allocator of the Linux Kernel, which also supports variable-sized chunks. It can be accessed by `kmalloc` and `kfree` [Lin]. In general, these allocators request large chunks of virtual memory from the OS to split these into smaller chunks on demand when an application allocates memory.

This work focuses on the second category: special-purpose allocators, optimized for specific workloads and allocation sizes. An example of this is the physical page allocator of an OS that manages the physical memory pages. The page allocator of Linux is mainly used for allocating new physical pages on demand for the virtual address spaces. Every time a process accesses new memory that is not already included in the corresponding page table tree, the page fault handler of the OS is executed. It checks the access validity and allocates new physical pages to insert them into the corresponding page table tree. These allocations frequently happen during the startup of a process

when it accesses most of its memory for the first time. This is implicit and hidden behind simple memory accesses, almost entirely invisible to the process, except for the much longer runtime of these memory accesses that trigger page faults. Therefore these page allocators must be very fast, interrupting the process as little as possible.

2.2.3 The Linux Buddy Allocator

The design of Linux's physical page allocator is based on a buddy allocator. It can allocate blocks the size of $2^n \cdot P$ where P is the page size and $n \in \{0, ..., 11\}$ [Gor04]. These block sizes start with the page size as order 0 and duplicate on every higher order. The buddy allocator contains a free list for each of these different-sized memory blocks. For example, the order one free list manages blocks the size of $2^1 = 2$ times the page size, and order nine manages blocks of $2^9 = 512$ pages. The free lists are implemented as linked lists where only the pointer to the first free memory block has to be stored. The memory blocks themselves then contain pointers to the next free memory blocks. The allocator is initialized with the free memory stored in the highest order free lists. When a chunk of a specific order is allocated, a block is taken from the free list with the same order. If this list is empty, a block is taken from a higher-order free list and split into two halves until it has the requested size. The other halves are stored in the corresponding free lists as shown in Figure 2.4. When a block is freed, it is checked if its corresponding half (its buddy) is also free. Then both halves are merged and inserted into the higher-order free list. Otherwise, if the buddy is not free, the block is inserted into the current-order free list.

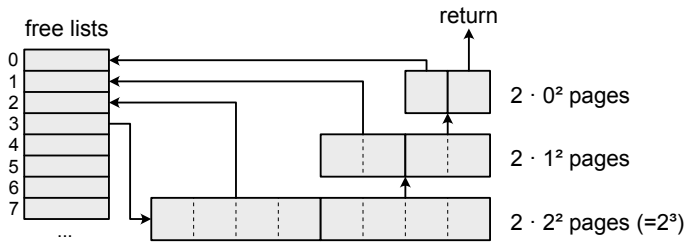


Figure 2.4 – Allocation procedure of the Linux buddy allocator.

Linux also manages memory zones, supporting Non-Uniform Memory Access (NUMA) architectures with different memory devices. It has a buddy allocator per memory zone. To optimize reallocations, the Linux allocator has additional free lists per CPU core that cache recently freed blocks for the next allocations. It further uses the least recently used (LRU) caches to speed up the merging of buddies. In general, it has been undergoing various optimization attempts in the past years: from memory layout optimizations and heuristics to find the best memory zones for allocations to various fast paths and shortcuts in the allocation and free routines. It also became increasingly complex with `page_alloc.c`, the heart of the allocator, currently having over 8 K lines of code alone.

The lo(ck|g)-free allocator has conceptual similarities with the buddy allocator. The different requirements and goals are further described in section 3.2.

2.3 Parallel Processor Architectures

The core count on modern systems increases rapidly. Operating systems and userspace applications must be optimized accordingly to use the full potential of these parallel architectures. This includes several new challenges, from synchronization and inter-process communication to the way memory is accessed and shared between cores [Gra+03]. Parallel algorithms have to be able to exchange data and wait for parallel computations on different cores. The following sections focus on concurrent memory access, starting with the CPUs' low-level ordering of memory accesses, followed by OS-level memory sharing and synchronization primitives.

2.3.1 Memory Ordering and Atomic Operations

Most modern processors support memory sharing between processing units (cores and sockets), allowing them to access data from each other directly. Concurrent write and read accesses to the same date might lead to unexpected race conditions. This has multiple reasons: An optimizing compiler may reorder instructions to optimize memory access, which can impact or break the behavior of concurrent accesses. This can be prevented with compiler-based memory fences that forbid the reordering of instructions. Also, superscalar CPUs with a high degree of instruction parallelism using pipelining and multiple execution units may reorder memory accesses during runtime. They are limited to reordering these operations so that the result remains the same. This, however, does not factor in concurrent accesses by other CPU cores. To solve this, x86 and most other architectures have well-defined memory ordering constraints or specific instructions preventing reordering.

On a single CPU core, x86 has a total memory order with the following constraints [Int]:

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Loads and stores to the same location are not reordered.
- Writes to memory are not reordered with other writes (except for non-temporal moves and string operations).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.
- Reads or writes cannot be reordered with I/O, locked, and serializing instructions.

The reordering can be prevented explicitly with memory fences that wait for previous reads (`LFENCE`) or writes (`SFENCE`) or both (`MFENCE`). Also, no read or write is reordered with `CLFLUSH` that flushes a cache line to the main memory.

For multicore systems, x86 has the concept of global visibility. All processors observe writes from other processors in the same order. Thus all processors have the same (global) visibility to data changes. The order of writes a single processor performs remains the same for global visibility. Writes from multiple processors, however, do not have any ordering guarantees and may interleave in any possible way. The architecture also provides locked load and store instructions with a total order.

However, memory ordering is heavily platform-dependent. Most programming languages, therefore, have abstractions to control the memory ordering and the access to shared variables [Stda]. These atomic functions guarantee that load and store operations are complete and that there are no partial updates. Additionally, they allow specifying memory ordering constraints for these operations. The most common ordering is the *release-acquire*. This *happens-before* relation strictly specifies the order in which operations tagged as *release* or *acquire* are executed. The *release* operations prevent other loads and stores that happened before from being reordered after these

operations. The *acquire* operations, on the contrary, prevent other loads and stores that happen afterward from being reordered before these operations. Both are generally combined to synchronize a *release* store operation of one thread with an *acquire* read on another thread. After the *acquire* operation is completed, as a side-effect, all other stores of the first thread that happened before the *release* store are guaranteed to be completed. The global visibility of x86 fulfills the *release-acquire* relation, thus all regular stores are *release* and all reads are *acquire*. On top of this is the *sequentially-consistent* ordering, the strongest memory ordering relation. It not only fulfills *release-acquire* but also guarantees a single total order in which all threads, including the executing one, observe all modifications. On x86, this is realized by using the previously described memory fences and special *lock* instructions that synchronize with other cores.

Besides the atomic loads and stores, there are several other functions that combine load and store into single atomic operations. Examples of these are fetch-and-add (FAA), fetch-and-xor, or *compare-exchange* or often called Compare-And-Swap (CAS). CAS checks if the atomic variable contains a specific value and, in this case, stores a new value in it. The lo(ck|g)-free allocator uses this function heavily to update its persistent metadata atomically.

2.3.2 Shared Memory

On the software level, there are two primary forms of data exchange between parallel tasks – exchanging messages or accessing a shared data space. Message passing is less commonly used on the OS level due to its higher abstraction and distance from the hardware. Instead, this communication strategy is generally used for distributed applications on multiple servers or clusters. Therefore, this work primarily focuses on memory sharing. Operating systems implement this by allowing address spaces of different processes to point to the same (shared) physical pages. Shared memory comes with synchronization challenges, both for the OS and the userspace applications.

If the shared data is immutable, no synchronization is needed, but if one or more threads modify the data, this concurrent access must be synchronized. Reading or modifying data that is changed simultaneously might lead to several race conditions where the data has only been partially updated. Even if the concurrent access to data should be kept to a minimum, it is necessary to share input data for computations and collect and merge results. There are synchronization and locking primitives that enable this coordination. The most commonly used ones are semaphores, mutexes, and barriers. A *semaphore* consists of a counter variable representing the number of available resources the semaphore protects. Each thread that wants access decrements the counter. If it is already zero, it has to wait until another task releases the semaphore, notifying a waiting thread or else incrementing the counter. A *mutex*, or *lock*, protects critical sections and data completely from concurrent access. Only one thread can access a protected section simultaneously, as all other threads that are trying to take the mutex have to wait until it is released. This is similar to a semaphore initialized with a counter of one. And finally, *barriers* enable multiple threads to synchronize and coordinate their execution. They are initialized with a counter of how many threads they can handle. When a thread reaches a barrier, it is suspended until the specified number of threads also reaches this barrier. Then all threads are awakened and continue execution. All of these locking primitives are commonly implemented on the OS level with support from the scheduler, which suspends waiting threads so that they do not occupy resources (passive waiting). On top of these OS specific implementations there are a number of libraries that implement these (or similar) primitives, like *pthread* [Pth] for C, the C++ *Standard Template Library* [Stdb] or *Rusts Standard Library* [Rus].

Parallel architectures not only have multiple cores per CPU, but large servers also use multiple CPUs on multiple sockets, further increasing the number of cores. These different CPUs generally have their own memory controller connecting to different memory DIMMS. Despite this, CPUs are

still capable of accessing the memory of other CPUs but with considerably higher latency. Because of this NUMA, the CPUs (and their cores) are separated into different NUMA nodes together with their corresponding memory and IO devices. The OS manages these NUMA nodes. It optimizes memory allocation to prioritize local NUMA nodes rather than remote NUMA nodes, where memory accesses are slower. This is generally done by the memory allocator returning NUMA near pages for the processes.

2.3.3 Non-blocking algorithms

The traditional approach for designing parallel algorithms on shared memory uses locking mechanisms like semaphores or mutexes to synchronize execution and data. These locks can limit the scalability of an algorithm if there is heavy contention on them. This can happen if the locks are too coarse-grained, protecting too much data used for different operations. However, if many fine-grained locks are used, the algorithm might become very complex. It could be hard to avoid deadlocks or priority inversion. Non-blocking algorithms can provide viable alternatives for their blocking counterparts in highly concurrent or real-time situations. Non-blocking algorithms guarantee that a thread cannot cause a failure or suspension of another thread. These algorithms can be further divided into three types, depending on their progress guarantees [Fra04]:

Wait-free algorithms guarantee that each thread completes an operation in a bound number of steps. This guarantees system-wide throughput, combined with starvation freedom. In theory, every algorithm can be implemented wait-free (using universal construction [Her88]), but often with worse performance than the blocking version. Since then, several improvements have been made, including strategies to integrate lock-free fast paths to improve performance [KP12].

Lock-free algorithms guarantee system-wide progress. Individual threads are allowed to starve (never completing the operation), but the remaining threads still must be able to make progress. Both lock-free and wait-free algorithms often implement *helping* mechanisms, in which a thread assists another thread in executing its operation before continuing with its own.

Obstruction-free algorithms guarantee that a single thread executed in isolation (without interference from other threads) completes the operation in a bounded number of steps [HLM03]. It is not guaranteed that threads executed in parallel might ever complete. This can happen if they end up in a live lock blocking each other from progressing. Furthermore, obstruction-freedom only requires that any partially completed operation can be aborted (rolling back any changes). No concurrent assistance (helping) is needed, simplifying the implementation.

Every wait-free algorithm is also lock-free, which itself also implies obstruction-freedom. These non-blocking algorithms are implemented using atomic hardware instructions like compare-and-swap (CAS) or fetch-and-add (FAA), discussed in subsection 2.3.1. The correctness of non-blocking data structures is derived from their sequential semantics. Commonly this is done by proving *linearizability*: The concurrent execution of a number of linearizable operations must correspond to a sequence of these operations executed sequentially with the exact same outcome. In other words, for any set of concurrent operations, a sequential execution order of these operations has to exist that is semantically equivalent.

2.3.4 (False-) Sharing

A significant performance factor for shared memory is the behavior of caches. When data inside a cache is modified, the cache line is invalidated for the other cores. These cache lines then have to be reloaded on the next access. As a side-effect, this might even occur if CPU cores modify different but closely stored data. As long as this data is located on the same-cache line, it is also invalidated

and reloaded on each access. This effect is called *false-sharing*. Both sharing and false-sharing are common causes of performance problems of parallel applications and thus should be avoided if possible. As described in subsection 4.4.2, this also had a significant effect on the performance of the lo(ck|g)-free allocator.

2.4 Non-Volatile Memory

Research around non-volatile byte-addressable memory (NVM) is not new. Various concepts for potential applications and integrations into the operating system were proposed over the past decades. Thirty years ago Copeland et al. [Cop+89] made the argument that battery-backed-up RAM was reliable and cost-effective, including the assumption that cost-effectiveness would increase over the following years due to falling RAM prices. Later Chen et al. [Che+96] developed RIO, a reliable file system cache in NVM to reduce IO operations and increase performance. The focus of these early uses of integrated NVM laid heavily on protection mechanisms against stray writes that corrupt the non-volatile battery-backed-up RAM or separate it into external modules. These problems, however, have been partially mitigated by better memory protection techniques in current OSs.

In the past 10 years a new wave of NVM research has started, driven by advancements in NVM hardware like Phase-Change Memory (PCM) [Rao+08]: From the development of durable multiversioned data structures [Ven+11] or lock-free data structures [Dav+18; Fri+18] over file systems [XS16; Kad+21] and userspace allocators [VTS11; Cob+11; Mor+13; Sch+15; BCB16; HJW15] to whole systems based on NVM like Twizzler [Bit+17]. Most of this research is theoretical or only simulated on conventional RAM due to the lack of hardware. This recently changed with the release of the Optane Memory in 2019.

Intel Optane is the first commercially available byte-addressable non-volatile memory hardware for use in large server systems with high memory requirements. The capacity of these NVDIMMs greatly surpasses conventional DRAM DIMMs, starting at 128 GiB and going up to 512 GiB per DIMM. Its performance is similar to DRAM, with almost equal write and about 2-3 times slower read latency [Yan+20]. However, this best-case performance drops heavily for multicore accesses. Like SSDs, Optane DIMMs perform wear-leveling and bad-block management using an internal address indirection table. This wear-leveling technique migrates blocks to different areas if used heavily and tries to spread write accesses between blocks evenly. In the case of a migration, the access latency is two orders of magnitude higher because the block must be copied over.

2.4.1 (Extended) Asynchronous DRAM Refresh

Intel defines power failure protection domains for their Optane DIMMs. These domains include anything that is guaranteed to be persisted in the event of a power loss. For the first generation, only the Write Pending Queues (WPQs) were persisted as shown in Figure 2.5. Intel calls this mechanism Asynchronous DRAM Refresh (ADR). Any data that is still in the registers or caches would be lost in the case of a system crash. This ADR based NVM can be considered as a persistent data storage with a partially unreliable connection. There are no guarantees when stores are persisted and in which order that happens. Explicit flush operations (`CLFLUSH`, `CLFLUSHOPT`) and memory fences must be used to persist stores in the desired order. The performance suffers from these manual cache-flushes, especially for CPUs that do not support the optimized `CLWB` instruction that retains cache lines on write-back [Izr+19]. In this case, a CPU not only has to wait for the write-back to complete, but it also has to reload the evicted cache line on the next access.

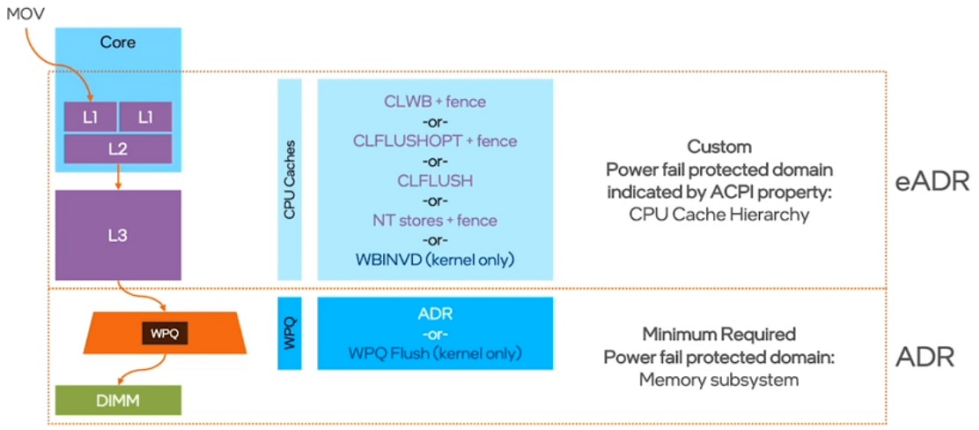


Figure 2.5 – Intel Optane "power fail protected domain" [Int21]

To optimize performance on ADR, Yang et al. [Yan+20] proposed a set of best practices: (1) Random access, smaller than 256 bytes, should be avoided as the access granularity of the NVDIMMs is 256 bytes. Thus, also smaller writes trigger 256-byte updates. Sequential access is less of a problem due to a write combining buffer (XPBuffer) that collects consecutive writes. (2) For large transfers, non-temporal stores should be used as their bandwidth is higher than normal stores and flushes. Manually controlling cache evictions could also increase performance compared to the non-determinism of automatic evictions. (3) Further, the number of concurrent threads that access the same DIMM should be reduced due to contention on the XPBuffer and integrated memory controllers (iMCs). The write bandwidth to a DIMM decreases with more than four threads. (4) Finally, remote NUMA accesses, especially read-modify-write sequences, should be avoided because they are exceptionally costly.

However, in part, these specific performance characteristics for this hardware could not apply to different platforms and probably also future revisions of Optane memory. In addition to the performance implications of ADR, this persistency strategy is considerably error-prone. Especially for multicore applications that rely on shared persistent memory, the gap between global visibility and persistency introduces a new class of race conditions. A considerable amount of research has been made to either detect these correctness bugs [Fu+21] or transform lock-free algorithms to such a full system crash model [IMS16a].

In 2021 Intel introduced Extended ADR (eADR) as part of its Optane 200 series. This power failure protection domain includes caches and registers. Therefore, no manual flushes are needed to guarantee persistency. Additionally, eADR sets persistency equal to global visibility. This means that if a write operation becomes visible to other CPUs, it is at the same time guaranteed to be persistent. eADR makes persistent memory programming much simpler as algorithms now only have to be lock-free to be persistent at the same time. The performance is expected to be up to two times faster than ADR with `CLWB` [Int21].

2.4.2 Direct Access (DAX)

As the concept of NVM is comparatively new, most OSs were never designed for it. Linux, for example, distinguishes heavily between memory and files. Thus it allows configuring NVM as either standard memory similar to DRAM or as disks with a file system to manage its access. This work focuses on the latter. Using `fsdax` or `ext4`, applications can mount the Optane Memory and create

persistent Direct Access (DAX) files. These files can be mapped into memory with a special flag that enables direct access to the persistent memory. Linux had support for DAX since 4.15.

After mapping the persistent memory, the application has to ensure persistency itself by using flushes or non-temporal stores depending on the power failure protection domain as described in subsection 2.4.1. To simplify this, Intel provides the Persistent Memory Development Kit [Pmd] for managing and safely accessing the persistent memory from the userspace. On Linux, this library builds upon the DAX capabilities the OS provides.

2.5 Related Work

Many of the earlier approaches in designing memory allocators for NVM were either closed source and not benchmarked, like *Mnesmosyne* [VTS11] or *NVHeaps* [Cob+11] or rely on special hardware or CPU instructions. Examples of these are the epoch barriers that *NVHeaps* [Cob+11] proposes or new cache line counters from Moraru et al. [Mor+13], that were never realized.

General Purpose Allocators

There are some newer promising persistent allocators designed for userspace applications: Schwalb et al. developed *nvm_malloc* [Sch+15], that is based on jemalloc [Jem; Eva06]. It follows a two-step approach of first reserving memory and then activating it to prevent memory leaks if the system crashes during an allocation. *nvm_malloc* uses `mmap` to map a single dynamically sized pool and internally addresses its data as offsets from the beginning of the pool to allow Address Space Layout Randomization (ASLR). It relies on logs to recover interrupted allocations and frees for atomic updates.

Bhandari, Chakrabarti, and Boehm presented *Makalu* [BCB16], a fail-safe allocator with recovery time garbage collection that has many similarities to *nvm_malloc*. It, however, works with a single allocation instruction that writes the memory address of the allocated memory block directly into persistent memory. Furthermore, it reduces the amount of metadata stored in NVM heavily compared to *nvm_malloc* using recovery time garbage collection to relax the persistency constraints of the metadata. Internally it uses normal volatile memory pointers and relies on a fixed memory mapping, which can be a security issue (ASLR).

In 2017 Oukid et al. developed *PAllocator* [Ouk+17] that was specifically designed for database systems that have a different allocation workload than other userspace applications [DLN19], where the focus is more shifted towards allocations larger than the page size. The allocator works on top of a DAX-capable file system and creates and adds new DAX files on-demand to its pool. Internally the allocator consists of three separate allocators for handling different sized allocation, allowing it to outscale *nvm_malloc*, *Makalu* and *libpmemobj* [Pmd] for large allocations.

Most of these allocators were developed before capable hardware was released in 2019 and thus were only simulated on DRAM with varying amounts of accuracy. Furthermore, these allocators are general-purpose userspace allocators specifically designed for allocating and managing variable-sized buffers. In this work, however, we developed an OS-level page allocator that focuses on managing physical pages of hardware-defined sizes and not variable-sized chunks (subsection 2.2.2).

Operating Systems

In the operation system space Bittman et al. presented *Twizzler* [Bit+17; Bit+21], an OS designed around NVM, that also provides specific NVM allocators. Even if *Twizzler* was initially released

before Intel Optane was available, support for it was added recently. The memory object concept is heavily influenced by *NVHeaps*. Similar to our *morsel* concept, these objects can be volatile or persistent and have unique IDs and sizes from 4 KiB up to 1 GiB. Also, they are mapped into the virtual address spaces of threads using the MMU. Each thread has a view describing its virtual address space layout in the form of a table of object-IDs and permissions. Translations between view tables and virtual addresses are trivial because the objects, which are at most 1 GiB large, are mapped directly to corresponding 3rd-level page tables entries (that manage 1 GiB subtrees). The persistent pointers *Twizzler* proposes are built upon two indirections to allow large 128-bit object IDs by keeping pointer sizes at 64-bit. Each object contains a foreign object table (FOT) with all object IDs (and permissions) referenced by pointers within that object. The persistent pointers consist of an index to the FOT and the offset within the foreign object. The FOT index of 0 refers to the own object, simplifying the translation. In general, the (persistent) memory object system of *Twizzler* scales well but comes with performance costs. It also cannot be integrated into current OSs as *Twizzler* focuses on providing first-class support for the new memory technologies. Legacy applications are only supported via extra userspace libraries and wrappers and have to be adapted to use persistent pointers. In contrast to this, we aim to integrate these new memory management technologies in modern OSs like Linux. Also, *Twizzler's* page allocator is at this moment relatively barebones and uses free lists protected by spin-locks. The lo(ck|g)-free allocator avoids these locks and scales better than similar locked-list allocators as shown in section 5.2.

Hardware-Tested

Recent research on Intel Optane hardware spans from NVM aware file systems like *WineFS* [Kad+21] to page table management strategies in NVM [Kum+21] or dynamically moving pages between DRAM and NVRAM to optimize performance [Ray+21]. *WineFS* is a hugepage-aware file system designed for NVM. It focuses heavily on limiting the effect of aging, the increase of fragmentation over time, leading to lower performance. For this, Kadekodi et al. developed a fragmentation-avoiding, alignment-aware allocator. It also prioritizes data from CPU-local pools. Additionally, the number of NUMA accesses was minimized by migrating threads to the correct NUMA nodes before writing. The lo(ck|g)-free allocator is also alignment-aware and supports the allocation of huge (2 MiB) and even giant (1 GiB) pages, and reduces fragmentation by prioritizing partially filled chunks for subsequent allocations. *HeMem* [Ray+21] proposes a strategy for switching pages between NVRAM and DRAM to optimize performance. It samples hardware events to classify hot and cold data. Also, small allocations are moved to DRAM by intercepting the allocation requests. It is implemented as a userspace library, similar to the lo(ck|g)-free allocator. However, this is subject to change in the future as we aim to integrate the allocator into the Linux Kernel.

2.6 Summary

The allocation, virtualization, and sharing of different types of memory are integral parts of operating systems. Concurrent access to shared memory can be synchronized on multiple levels, from atomic hardware instructions and memory ordering to OS-level synchronization primitives. The new non-volatile memory fills a significant gap in the conventional memory hierarchy, combining properties of byte-addressable memory with the persistency of hard drives. An increasing amount of research is being done to fully utilize this new hardware and find its conceptual and technical limitations. It reaches from userspace allocators and file systems to whole operating systems designed for NVM. Related work, specific to physical page allocators for NVM, however, is relatively rare.

3 ARCHITECTURE

This chapter focuses on the theoretical approach and higher-level concepts of *morsels*, the new minimal memory abstraction (section 3.1). They need a new allocator that persistently manages their memory. The requirements and architecture of this persistent and lock and log-free (lo(ck|g)-free) memory allocator are described in sections 3.2 and 3.3. The chapter further discusses the allocator's algorithm to find new free pages (section 3.5) and ways to store its metadata (section 3.6).

3.1 Morsels

Modern systems contain many different memory types with different properties. The OS's primary responsibility is the management of these memory resources. Paging has become the standard for access, sharing, and virtualization of main memory (section 2.2). MMUs are designed around this principle and provide (hardware) support for virtual memory based on *page table* mappings. On top of this, the memory subsystems of modern OSs use various levels of abstractions to support different memory architectures. In Linux, these abstractions have become increasingly complex. The invention of NUMA and new memory devices further aggravated this. As memory sizes increase and access times decrease, these abstractions, which were ultimately designed for former architectures, are slowly becoming a bottleneck [Bai+11; Pel+15].

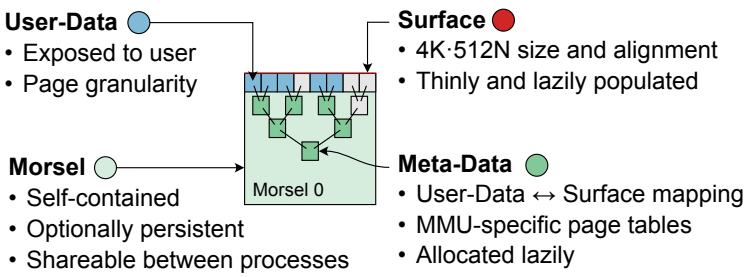


Figure 3.1 – The *morsel* as minimal self-contained address space abstraction.

The *morsel* concept (shown in Figure 3.1) aims to solve this performance bottleneck by introducing a very hardware-near memory abstraction. A morsel consists of a small page table tree that can be integrated into an existing address space by adding it as a subtree to the page table tree. Morsels can have different sizes depending on the depth of their page table trees. For example, a morsel of depth

1 contains 512 pages or 2 MiB, and depth 2 contains 512^2 pages or 1 GiB (on x86 with a page size of 4 KiB). Morsels have to be compatible with MMUs and IOMMUs to be mounted into address spaces of processes or devices. For this, they have to be lock-free because non-OS-controlled devices cannot participate in complex communication protocols that include locking mechanisms. The IOMMU itself is the only (general) interface to communicate with devices. It does not require specific drivers or capabilities some devices might not have. The lock-free nature of morsels guarantees that changes in the underlying page table trees are globally visible in a consistent manner. In addition to that, morsels are self-contained, allowing them to be stored persistently in eADR capable persistent memory. Even the user data of a morsel can be allocated from NVM, making it also persistent, similar to a DAX device.

3.2 The Lo(ck|g)-Free Page Allocator

Persistent morsels require a persistent memory allocator that can both allocate pages from persistent memory and is itself persistent and able to recover from a power loss. This section discusses these requirements further.

3.2.1 General Requirements

A persistent page allocator inherits most requirements from its volatile counterparts, like the Linux Kernel Allocator (subsection 2.2.2). It also has to manage pages of the given hardware size (e.g., 4 KiB) and preferably large pages (2 MiB and 1 GiB) with similar performance characteristics. As modern systems come with ever-increasing core counts, scalability on these multi-threaded systems is one of the primary design goals of the lo(ck|g)-free morsel allocator. Also, reducing fragmentation is important for an allocator and even more critical if it is persistent and outlives reboots. Because of its long lifespan, the available memory has to be used as efficiently as possible.

One requirement that is ignored in this work is the management of memory zones for different memory devices and NUMA nodes. Our allocator is initialized with a single contiguous memory range to be managed. This, however, could be supported by instantiating the allocator for each NUMA device. An additional small abstraction layer would be required that chooses the best allocator instance of the nearest NUMA node for each allocation (similar to the Linux kernel memory zones). However, managing different zones is outside the scope of this work.

3.2.2 Persistency Related Requirements

There are also a set of new requirements specific to persistency: Mainly, the allocator's state has to be recoverable at any point in time without losing any pages or at least only a limited number of them. Pages could be lost if a crash happens during an allocation or shortly after it, before the allocating process can persistently store the allocated page. *nvm_malloc* [Sch+15] solved this issue with a two step-approach of reserving a page and committing (activating) the allocation after the page has been inserted into a persistent data structure. However, the downside of this approach is the additional complexity that comes with two functions instead of one, leading to a more error-prone interface.

Instead, the lo(ck|g)-free morsel allocator only fulfills the laxer requirement of losing only the pages currently being allocated or freed when a crash happens. This provides an upper bound to the number of pages potentially lost during a crash, equal to the number of cores of the system. Assuming that the probability of a crash is reasonably low and the probability of crashing during allocations is even lower, the amount of lost memory is expected to be reasonably low in practice.

Additionally, the morsels themselves are persistent and store all their allocated pages. Thus, they could be used to determine which pages have been leaked when the allocator recovers after a crash. These pages are not present in any of the morsels and can be freed and reused later. Other implications for the allocator coming from the persistency of morsels are discussed in section 5.6.

The second goal, specific to the persistency properties of NVM, is the avoidance of locks and logs. The first, locks, are obstructive to multicore scaling and make persistency and crash-recovery inherently more complex. Logs, on the other hand, are problematic for the current Optane memory because of its limited number of write cycles. Logging results in a higher number of writes, decreasing the lifespan of the NVDIMMs. The current hardware also uses wear-leveling techniques to spread write accesses evenly between blocks. If a log is frequently written to a specific block, the wear-leveling algorithm swaps it more often with other blocks. This is costly because its data must be copied over.

3.3 Page Table-Based Architecture

The core approach of the lo(ck|g)-free morsel allocator relies on page tables to persistently store its metadata. The idea is to build an identity mapping over the physical pages the allocator manages and store the information about which pages are free within their corresponding page table entries. Updates to the page table entries are performed using atomic compare-exchange operations. Therefore no locks or logs are necessary to synchronize and persist updates. This simplifies the recovery step as no locks have to be recovered and checked. Especially for locks, it is challenging to recover their protected data because changes to it might be only partially written to NVRAM.

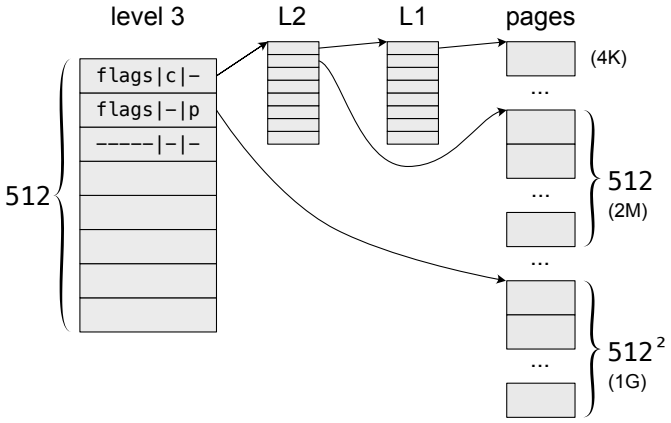


Figure 3.2 – The page table mapping that represents the allocator state.
The *P* flag marks allocated pages and *c* is a free page counter. Depending on the level of the page table where an entry is allocated the corresponding size can reach from a single page (4 KiB) to 2 MiB or 1 GiB large blocks.

Furthermore, this table-based approach makes it easy to support huge pages, with sizes of 512 or 512^2 times the page size (2 MiB or 1 GiB). These huge/giant pages are effectively the blocks that the 1st-/2nd-level page tables are addressing, as shown in Figure 3.2. Instead of allocating a leaf page (referenced by a 1st-level page table entry), the whole subtree can be allocated for a huge page.

On the conceptual side, this allocator thus can be understood as a buddy allocator (subsection 2.2.2) with 512 buddies instead of two.

The lo(ck|g)-free allocator is separated into two smaller allocators. The lower allocator manages smaller allocations of 4 KiB and 2 MiB pages, and the upper allocator manages the 1 GiB allocations and the 1 GiB subtrees in which the lower allocator operates. These subtrees have the 2nd-level page tables as roots and address 1 GiB of memory. A subtree thus can be allocated either as a single 1 GiB page or split into multiple 2 MiB or 4 KiB pages. This work compares multiple lower and upper allocator strategies with each other. These strategies are further described in chapter 4.

3.4 Heterogeneous Memory Systems

Even if NVRAM has similar access times as DRAM, enabled mainly through caches, they become increasingly bad if large amounts of data are accessed regularly, which does not fit into these caches. Also, the ADR guarantees, which require frequent flushing to gain persistency, do come with a heavy burden on performance. Currently, NVRAM and DRAM are generally combined in a single heterogeneous system. Thus the allocator also combines them in order to increase performance. Only essential data that cannot be recovered is stored persistently to reduce NVRAM accesses. This persistent data includes the page tables of the lower allocator, together with metadata defining the allocator's memory range. The volatile DRAM contains the page tables of the higher levels, CPU local data, indices, and free lists that can be rebuilt after a crash. In general, this includes all the metadata of the upper allocators.

3.5 Search Strategies

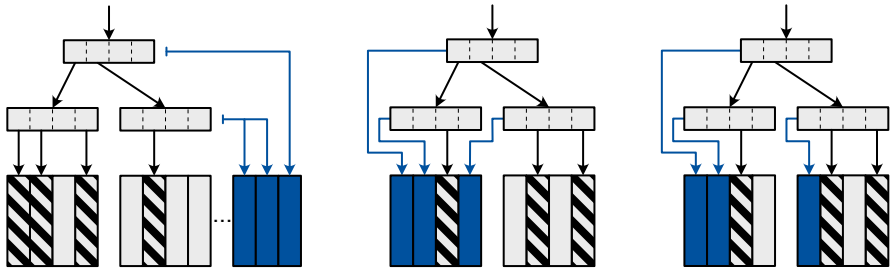
The search for new free pages is an essential part of the allocator, as it heavily affects its allocation performance. For a free operation, the caller provides a pointer for the page to be freed, but this is not the case for allocations. It is up to the allocator to find the next free pages. The lo(ck|g)-free allocator does this by iterating through the page table mapping until a free entry is found. The higher levels of page tables include counters of free pages within their subtrees to speed up this iteration. These counters allow the search algorithm to skip empty subtrees entirely and, in addition to that, prioritize partially full subtrees to reduce fragmentation. The downside is that these counters have to be updated on every allocation and free operation. Subsection 4.4.2 discusses the memory-sharing-related problems that come with this further.

To speed up subsequent allocations, the allocator saves the address of the last allocation per CPU core. On the next allocation, the search starts with this saved address. Consequently, for a sequence of allocations, only the last page, which already has been allocated last time, and the page after that, which is usually free, must be checked. For an alternating sequence of allocations and frees, where the allocated page is freed immediately, this page can be reused for each allocation. In these cases, only a single entry of the lowest page table has to checked and updated. In both examples, caching the last allocated addresses reduces the number of NVRAM accesses drastically.

3.6 Page Table Pages

The question of how and where to store the allocator's page tables is deceptive, as it is more complicated than it seems at first. The page tables of the lower levels have to be stored in non-volatile memory to guarantee persistency. Thus, they must be located in the NVRAM region passed

to the allocator. This is the same region that is used for the allocations. All accesses to these page tables have to be lock-free to be recoverable. Also, multiple threads must be able to access these page tables concurrently, as locks are avoided. In general, there are three strategies for managing page tables:



(a) In Separate Memory Area **(b)** Inside Previous Page Tables **(c)** Inside Their Own Memory

Figure 3.3 – The different strategies for storing page table.
The pages are at the bottom, and page tables managing them are shown at the top. The white pages are allocated, and the dashed ones are still free. The page tables have only four entries in this visualization. In practice, they have 512 entries.

In Separate Memory Area

The first approach is to reserve a fixed memory area for the page tables, as shown in Figure 3.3a. This fixed area is not available for allocation. The size of this area depends on the size of the memory range the allocator manages. For an identity mapping with l levels of page tables and n entries per page table are $\sum_{k=0}^{l-1} n^k$ pages needed. For example, for $l = 3$, $n = 512$ and 4 KiB pages, the page tables take up more than 1 GiB to manage a 512 GiB area or 0.2% of this space. This amount of metadata overhead is far from ideal. On the other hand, this static placement of page tables makes it trivial to find them as their location is fixed. Thus they can be directly accessed. Also, the page table entries do not have to store any pointers to the child page tables and can be used entirely to store other metadata, like counters or flags.

Inside Previous Page Tables

Page tables are only needed if the corresponding subtree is not entirely free or allocated. In both cases, this information can be stored as part of a free page counter in the parent's page table entry. If it is zero, the subtree is wholly allocated, and if it has the maximum value, no page is allocated. The second strategy is to dynamically allocate and free the page tables to save space (Figure 3.3b). Here, the allocator starts with fixed page tables for the first chunk of memory. After that, every new page table can be allocated within the last chunk of memory before its page tables are full. If a page table is full, its page could also be freed and reused for subsequent allocations.

However, this on-demand page table allocation has the downside that it quickly might reach deadlocks. If all page tables are full, no page for a new page table can be allocated because a free page table entry is required to mark this page as allocated. This could even occur if the allocator still has memory left. Also, if the allocator's memory is entirely allocated, free operations might become difficult. A free operation requires multiple page tables to mark its page as free. One page table is needed for the lowest (1st) level and one for every upper level to store the pointers to the child page

tables (including flags and counters). Suppose all of these page tables have been removed as they were are not needed for fully allocated areas. It might become impossible to allocate pages for new page tables when the allocator has no free pages left. This dynamic approach quickly becomes very complex, as it has to decide when page tables should be allocated or freed to circumvent deadlocks. Especially concurrent access is challenging as there are several possibilities for race conditions.

Also, if the page tables can be allocated everywhere, the entries have to store entire pointers to their child page tables. This shrinks the usable space for counters and flags to only 12 bits, as 52 bits are needed for a pointer to an aligned page.

Inside Their Own Memory

The third strategy is to store the page tables in their own memory chunks, as shown in Figure 3.3c. They are stored dynamically in one of their own pages. The parent's page table entry has to store the index to the page with the child page table. This index can be much smaller than a full pointer. Consequently, some pages of the page tables are present from the beginning, the ones where the page tables themselves are located. The 1st-level page tables can also be freed if their area is fully allocated. Again, this information can be stored in the free counter of the parent's page table entry. Problematic are then the level two and higher page tables. They cannot be freed for the same reasons as in the previous strategy. This would lead to deadlocks where multiple pages are needed for the next free operation. Also, in this strategy, the support for huge and giant pages becomes more challenging. It quickly becomes difficult to decide whether the 1st-level page tables are needed or not, especially if small and huge pages are allocated in the same subtree.

All of these strategies have downsides. This work compares the first approach of *fixed page tables* with a hybrid approach of *dynamic page tables*. The latter reduces the memory overhead by combining the first and last strategies. The idea is to store the 1st-level page tables as a page in their own area. These page table pages are allocated as the last page if the rest of the pages are already allocated. The first freed page in a full area then becomes the new 1st-level page table. This allocation of the page table pages dramatically reduces the memory overhead of the allocator. In section 4.2 this hybrid approach and the strategies for updating page tables are described in more detail.

3.7 Summary

This chapter discussed the concept of *morsels* as minimal address-space abstractions, providing new ways of sharing memory with (IO)MMU devices. The lo(ck|g)-free allocator is developed to manage the memory of these morsels. Its primary goals are multicore scalability and persistency. Its architecture is based on page tables that are atomically updated, altogether avoiding locks or logging algorithms. The lo(ck|g)-free allocator is designed for heterogeneous systems, storing as few metadata in NVM as possible to reduce the number of costly NVM accesses. Finally, the problem of storing the 1st- and 2nd-level page tables in NVM has been discussed. Two suitable page-table strategies are further described and compared in the following chapters.

4 IMPLEMENTATION

In this work, multiple allocators were developed to compare different strategies. This chapter describes the critical parts of these lo(ck|g)-free allocator approaches in more detail. The first section (4.1) describes the general API that the allocators provide. After that, the lower allocator, which manages the 1st- and 2nd-level page tables, is presented together with its persistent page table updates (section 4.2). Based on this lower allocator, multiple upper allocators were developed that implement different strategies for managing the 1 GiB pages and subtrees (section 4.3). Finally, several critical performance optimizations are described in section 4.4 before concluding the chapter with a set of simple allocators that were developed as a baseline for the benchmarks in the following chapter.

4.1 General API

```rust
pub trait Alloc: Send + Sync {
    /// Initialize the allocator for this `memory` range.
    fn init(&mut self, cores: usize, memory: &mut [Page], overwrite: bool)
        -> Result<()>;
    /// Allocate a new page.
    fn get(&self, core: usize, size: Size) -> Result<u64>;
    /// Free the given page, returning its size.
    fn put(&self, core: usize, addr: u64) -> Result<Size>;
    // ...
}
```

Listing 4.1 – Basic API of the lo(ck|g)-free allocator.

The API of the allocator is shown in Listing 4.1. Every of the allocator variants discussed in this chapter implements this `Alloc` interface. The allocators are initialized once during system startup with a given contiguous memory range and maximum core count. Internally any addresses to this memory range are converted to page offsets from the beginning, thus allowing this range to be mapped elsewhere on the next boot, enabling ASLR. Also, an internal dirty flag is set during initialization. On a regular system shut down, the allocator clears this dirty flag. However, when a crash occurs, this flag is still dirty on the next boot. In this case, the allocator initiates a full recovery and checks for interrupted allocations and incorrect counters.

The `get` and `put` functions allocate or respectively deallocate blocks of the provided size. They also expect the id of the current core, allowing them to access core local data. We assume that the threads are not migrated during the get and put functions, and no concurrent calls to these functions happen for the same core. An additional `get_cas` helper function combines the `get` function with a compare exchange operation (CAS) storing the result of the allocation directly at a given location. The function also accepts a callback that translates the address returned by `get` before the CAS operation is executed. If the CAS operation fails, the allocated page is freed, and the function returns an error.

4.2 Lower Allocator

The page tables of levels three and upwards and other data structures are allocated in the volatile DRAM, but the 1st- and 2nd-level page tables are stored in NVM. The lower allocator manages these lower-level page tables, which are persistent and recoverable at any time. Its purpose is to allocate 4 KiB and 2 MiB pages within these lower-level page tables. As described in section 3.6, the decision of where to place these page table pages is a tradeoff between complexity, performance, and memory overhead. This section describes the two most promising approaches for the lower allocator.

4.2.1 Fixed Page Tables

In this approach, both the 1st- and 2nd-level page tables are stored in arrays at the end of the NVM region, as shown in Figure 4.1. As mentioned in section 3.6, this has the advantage that the page table entries do not have to store any pointers and can be fully used for other metadata, like flags and counters. However, the downside of this approach is that these page table pages cannot be used for allocations anymore. The allocator loses about 0.2% of its memory to these page tables. However, this could be further reduced as discussed in section 5.5.

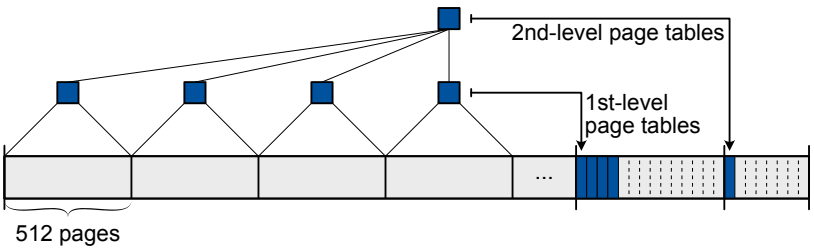


Figure 4.1 – NVM layout with fixed page tables.

The implementation of this approach is reasonably straightforward. The 1st-level page table entries only contain a single flag storing if the corresponding page is free or allocated (Figure 4.2). The 2nd-level entries contain an additional free page counter and other flags described later. The free page counter is initially set to its maximum value. It is decremented *before* every allocation in the corresponding 1st-level page table and incremented *after* every free. Thus the counter represents the lower limit of free pages present in the 1st-level page table at any point in time. Like a semaphore, this counter synchronizes concurrent access to the corresponding subtree. If the counter reaches zero, the decrement fails, and the subtree is not accessed; instead, it is continued with the next

one. Both updates on the 1st- and 2nd-level entries are performed with atomic CAS operations. Therefore, the only possible race condition is that the free page counter is temporarily smaller than the actual number of free entries in the 1st-level page table. This, however, does not affect the functionality of the allocator and is considered a valid state. Also, after a system crash, any false counters are corrected at reboot.

For the allocation and free of 2 MiB huge pages, only the 2nd-level page tables are accessed and searched for an entry where its counter is at the maximum value, indicating that the chunk is entirely available. With a single atomic CAS operation, an entry can be checked and allocated at the same time by setting the page flag (marking this huge page as allocated).

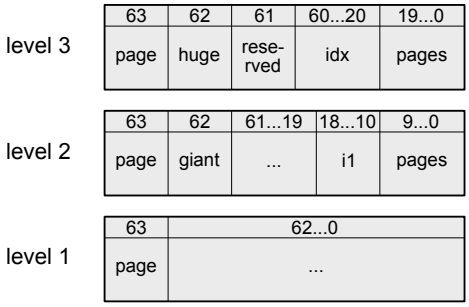


Figure 4.2 – Layout of the different page table entries.

4.2.2 Dynamic Page Tables

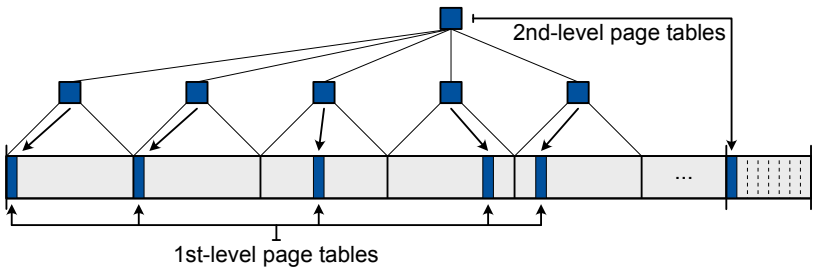


Figure 4.3 – NVM layout with dynamic 1st-level page tables.

Due to the high memory overhead of the fixed page table approach, another strategy has been developed that reduces the memory overhead dramatically. In this approach, the 1st-level page tables are dynamically stored to reuse the pages they occupy for allocations if the page tables are not needed anymore. To achieve this, the 1st-level page tables are stored in one of their own pages, as shown in Figure 4.3. The 2nd-level page tables are still stored in a fixed area at the end of the NVM memory. Their entries contain an index to the page of the corresponding 1st-level page table (`i1` shown in Figure 4.2) in addition to a free page counter. In the beginning, the 1st-level page tables are initialized at index zero. The first allocations update the 1st-level page table and upper counters similar to the fixed page table approach. However, the allocation of the last page of a page table is different. At this point, all pages are allocated, except the page of the page table itself. In this case,

this page table's page is returned from the allocator, which is possible because the 1st-level page table is not strictly needed for a fully allocated area. The information that the area is fully allocated and devoid of free pages is encoded in the free page counter of the corresponding 2nd-level PTE. When the first page is freed of a fully allocated area, this freed page is initialized as the new page table. Then the index of the 2nd-level PTE is updated accordingly. Two or more concurrent frees in a full page table might try to initialize their own pages as the new page tables. However, the corresponding 2nd-level PTE update is only successful for one of these concurrent frees. For the others, the CAS operation fails. These free operations are then repeated, modifying the now existing new page table.

This reuse of page tables for allocations reduces the persistent metadata overhead drastically. Now the fixed metadata overhead only consists of the 2nd-level page tables. This amounts to about 0.00038% of the allocator's memory or one 4 KiB page per 1 GiB of memory (instead of 0.2% from the previous approach).

However, this dynamic reinitialization of the 1st-level page tables adds additional complexity and a window for new race conditions. The biggest challenge is to synchronize the allocation of the last page (the page table itself) with other allocations affecting this page table. If not done correctly, this could result in writes to memory already given to the user or updates that are lost entirely. This can occur if the last page containing the page table is given to the user too early. Then any concurrent allocation in this table operates on a page that the user now owns. To avoid these stale writes, the allocation of the last page has to wait until all other operations on this page table are completed. This is achieved by storing the address of the current page table that a CPU core updates in a global array. When the last page is allocated (after decrementing the 2nd-level PTE counter), the core iterates through this array, checking that no other core accesses this page table. If an entry is found with the current page table address, it actively waits until this entry is changed. This strategy is similar to the hazard pointers of Michael et al. [Mic04], which were developed to solve the similar but more general memory reclamation problem of lock-free objects. These per-core page table pointers are stored in volatile memory and do not have to be recovered. Also, they are only accessed by their own cores most of the time, largely amortizing any memory-sharing effects.

Concurrent frees are still possible while a CPU core waits to allocate the last page. These frees behave identically to the first free in a fully allocated area, meaning that their own page becomes the new page table. Any operation on this new page table can be executed without interfering with the running last page free operation.

This approach generally enables highly concurrent access without locks and temporary inconsistent states. The latter also simplifies the recovery process. Like the first approach, only the diverging 2nd-level PTE counters have to be corrected during a recovery. Incorrect page table counters occur only if the crash interrupted an allocation or free. Their occurrence is limited to the number of CPU cores, as only this number of parallel operations can be interrupted by the crash.

4.3 Upper Allocators for Subtree Management

The following sections describe several concepts developed on top of the lower allocator to manage the 2nd-level page tables and distribute them to CPU cores, reducing contention and fragmentation. The lower allocator manages the access to the level two and lower page tables. These 2nd-level page tables address 1 GiB of memory. For the following sections, they are referred to as 1 GiB subtrees. For each 1 GiB subtree, a 3rd-level entry exists, containing a counter of free pages and a flag if the underlying memory chunk was allocated as a 1 GiB page (Figure 4.2). The following two upper

allocators implement different strategies for managing these subtrees. Both are built upon the lower allocator, mainly selecting the subtrees where the lower allocator should allocate pages.

In both apporaches, each CPU core reserves a 1 GiB subtree for small 4 KiB page allocations and one for large 2 MiB page allocations. The cores then primarily operate on their reserved subtrees. These per-core subtrees reservation aims to reduce memory sharing. This is important because congestion and memory-sharing penalties often heavily limit multicore scalability.

In addition to that, a 1 GiB subtree is limited to contain either small (4 KiB) or huge (2 MiB) pages, never both. The support for small and huge allocations in the same subtree comes with several downsides. The 3rd-level entry would require an additional counter for the number of free huge pages. Because the same memory can either be allocated as multiple small or a single huge page, this counter is necessary to keep track of huge pages that are entirely free of small pages. It would also be more prone to fragmentation. The allocation of a huge page could lead to the reservation of a new subtree even if the old subtree still has many free small pages.

Therefore, the allocator requires at least two 1 GiB subtrees for each CPU core to be available for reservation (one for small and one for huge pages). Thus, the memory range must be at least twice the core count gibibytes large. Given the enormous size of the Intel Optane DIMMs, it can be assumed that current NVM systems generally fulfill this requirement. However, it might be necessary for systems with smaller memory sizes or very high core counts to adjust the allocator to reserve the same subtrees for multiple CPU cores.

4.3.1 Table Allocator

The *Table* allocator uses a four-level identity mapping with page tables (subsection 2.2.1), to store its metadata. The Figure 4.4 shows the 1 GiB subtrees of the lower allocator at the bottom. They are stored in NVRAM. Above are the 3rd-level page tables, with entries containing a free page counter and additional flags (Figure 4.2). They are stored in DRAM, together with the 4th-level page table. The 4th-level entries contain three counters: One for empty 1 GiB subtrees and two for partially filled ones with either small or huge pages. The *Table* allocator can manage up to 256 TiB with this four-level page table tree.

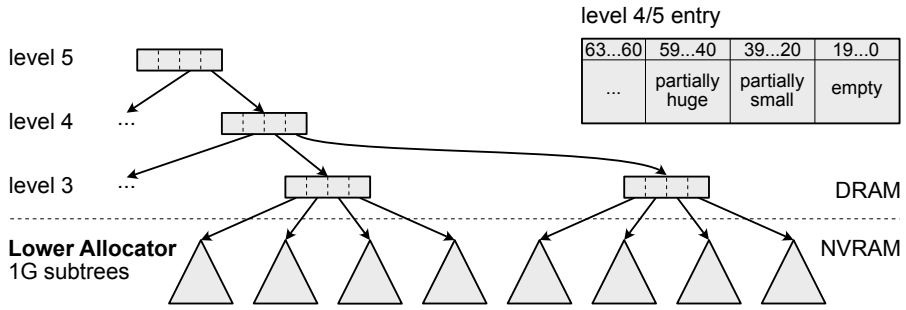


Figure 4.4 – The idendity mapping of the *Table* allocator.
In this visualization, the page tables have only four entries. However, the actual allocator uses page tables with 512 entries.

If a CPU core wants to reserve a new subtree, the allocator checks the page table entries sequentially for a not yet reserved subtree that is partially filled (with the requested size). The iteration starts at the previously reserved 3rd-level entry, iterates over the following entries, and wraps around the end of the mapping. Page tables that contain no partially free subtrees are skipped.

If no partially allocated subtree was found, an empty subtree is searched. This preference for partially filled subtrees reduces fragmentation. For giant pages, empty subtrees are searched and allocated entirely. This operation sets the page flag of the 3rd-level entry. However, the 3rd-level entries are volatile and stored in DRAM. Therefore this giant allocation is persisted by also setting the `giant` flag in the first entry of the corresponding 2nd-level page table (Figure 4.2). The recovery algorithm checks for this flag after reboot.

Similar to the lower allocator, the counters of the parent page table entries are decreased before the child page table is searched. For unreserve, the counters are incremented in reverse order. Again these counters represent the lower bounds of subtrees guaranteed to be available in the children's page table.

A reserved subtree gives the core exclusive access to it for allocations, reducing the amount of memory sharing. However, concurrent free operations from other CPU cores are still possible. In general, this is hard to circumvent, especially if threads are migrated to other cores.

This approach supports large memory areas (up to 256 TiB). It can be expanded to 131 072 TiB by adding a fifth level of page tables. Holes in the memory area can also be handled by marking them as allocated from the beginning. However, there has to be enough contiguous space at the end of the region for the 2nd-level page tables. Alternatively, they could also be stored in a completely different memory region. The downside of this table allocator is the fairly complex and recursive searching algorithm for the next (partially) empty subtrees. The page tables have to be traversed sequentially, and multiple CAS operations have to be performed to update the counters. This complicates the search algorithm and makes it more fragile and prone to race conditions.

4.3.2 Array Allocator

The goal of the *ArrayLocked* allocator is to simplify the reservation of new subtrees. Instead of iterating through the whole identity mapping, this approach uses three array-based stacks to keep track of empty and partially empty subtrees, as shown in Figure 4.5. The 3rd-level entries are all stored in a large array instead of separate page tables. The three stacks store the indices to the 3rd-level entry array. As these stacks are shared between the CPU cores, they are protected by mutexes (ticket locks). Again the partially empty stack is prioritized for subtree reservations. A simple pop operation is performed, and the entry behind the returned index is reserved. If this stack is empty, the operation falls back to the stack with the empty subtrees. For giant allocations, the allocator directly takes a subtree from the empty stack to allocate it as a single page.

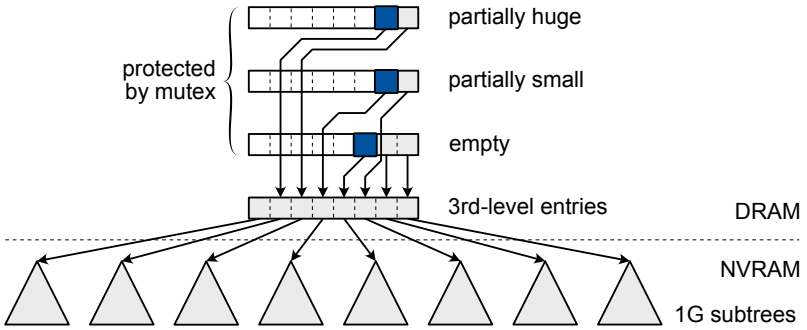


Figure 4.5 – Architecture of the *ArrayLocked* allocator.

The implementation of this approach is much more straightforward than the table allocator. It needs about 40% fewer lines of code, requires no recursive table walks, and is less prone to race conditions, as operations on the stacks are protected by locks. These locks are stored in DRAM and do not have to be recovered at reboot. However, in benchmarks with higher contentions on shared data structures, this approach does not perform very well (section 5.2).

4.4 Optimizations

The following sections describe further optimization in addition to the caching in the search algorithm (section 3.5). They reduce the allocator's memory overhead and resolve false sharing to increase throughput.

4.4.1 Lock-Free Linked Lists

In the first version of the array allocator, the empty and partially filled lists are implemented as vectors, protected by mutexes. Besides violating the goal of developing an allocator without locks, this approach takes up a fair amount of memory. These empty and partially empty vectors have to have a maximal capacity of the size of the PTE array. Thus the memory usage of the allocator increases with every gibibyte by $3 \cdot 8 = 24\,\mathrm{B}$ for these arrays alone. The second version of the array allocator (the *ArrayAtomic* allocator) instead uses atomic linked lists, as shown in Figure 4.6. The 3rd-level entries themselves store the next pointer in the `idx` fields (Figure 4.2). These next pointers are indices to the 3rd-level entry array. The push and pop operations of these linked list-based stacks are implemented in a lock-free manner, using atomic CAS instructions. Therefore no locks are required to synchronize concurrent access. The memory overhead is also reduced because now only the array of 3rd-level entries is needed.

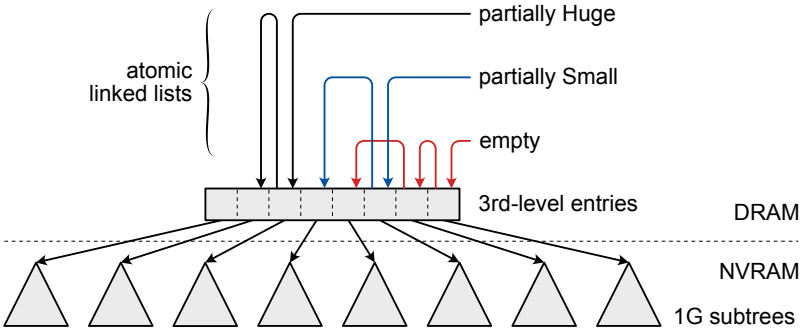


Figure 4.6 – Architecture of the *ArrayAtomic* allocator.

4.4.2 Reducing False Sharing

Unfortunately, the per-core reservation of subtrees is not enough to solve memory sharing completely. Our first benchmarks showed that both allocator variants suffered from high false sharing costs. These were caused by concurrently updating the 3rd-level entries, which are located directly side by side. These 8 B large entries are stored subsequently in page tables or arrays and thus often

share the same 64 B cache lines. Concurrently changing adjacent entries leads to frequent cache-line invalidations with heavy performance penalties (subsection 2.3.2).

A trivial countermeasure is to align the 3rd-level entries to cache-line size. This solves false sharing completely, as shown in Figure 4.7. However, it multiplies the memory used by the entries by eight. In the same manner, the number of cache lines increases. If their number exceeds the cache size, they must be displaced more often. This results in more frequent cache misses and lower performance.

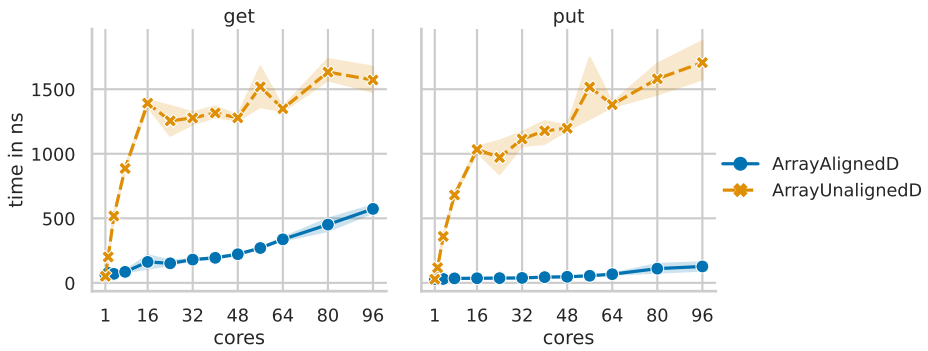


Figure 4.7 – The impact of false sharing on the *ArrayAtomic* allocator, depending on cache-line alignment.
This was measured on DRAM using the bulk benchmark from subsection 5.2.2.

Another strategy is to use core-local copies of the reserved 3rd-level entries. When a subtree is reserved, the core copies the 3rd-level entry to a local replica. All subsequent updates for allocations and frees are performed on this local copy. At the same time, the 3rd-level entry in the global page table/array is changed to be fully allocated. Any free operation on a subtree not reserved by the current core still updates the global entry. This is also the case if a subtree has been reserved by another CPU core. When a core later unreserves its subtree, it adds the free pages counter of the local entry back to the global entry and unsets its reserved flag. This reduces false sharing for allocations substantially. However, frees that do not modify the reserved subtrees still update the global page tables/arrays. If adjacent entries are modified, this still causes false sharing. This strategy has been implemented for the *Table*, *ArrayLocked* and *ArrayAtomic* allocators.

4.4.3 Utilizing Locality for Free Operations

The previous optimization successfully reduced false sharing for allocations. Frees, however, are a completely different problem. For allocations, the allocator can utilize locality to optimize memory access. Depending on its use case, this might also be the case for frees. For example, if subsequent allocations were stored in a data structure that is late freed at once. In this case, the pages that were previously allocated at the same time are also freed successively. These pages often share the same subtrees because subsequent allocations are executed in the same subtree. This spatial locality could be utilized to counteract the false-sharing problem of the free operation.

The following optimization is based on the assumption that subsequent frees are often part of the same subtree. The idea is to also reserve subtrees on free operations if the last N frees also modified this subtree. To avoid reserving almost full subtrees, they also must have at least a certain amount of free pages. The check for the last N allocations is implemented using small core-local N-sized ring buffers. On every free, the corresponding address is added to the ring buffer (overwriting the

old values). Every free operation in a non-reserved subtree checks if all addresses in the ring buffer belong to this subtree. In this case, the subtree is reserved. Any following free in this subtree updates the local copy of the 3rd-level entry. Consequently, updates on these entries are more often retained in their core-local replicas. This significantly reduced the false-sharing problem of free operations for the *Table*, *ArrayLocked* and *ArrayAtomic* allocators. It especially improves the bulk-free performance (subsection 5.2.2). The best performance was achieved with $N = 4$ and a free-page threshold of $512 \cdot 8$.

However, because morsels are allocated lazily, the allocations might have occurred at entirely different times and in any order. They could have no temporal connection at all. If a morsel is freed, freeing its pages from start to end could render the free-reserve optimization mostly ineffective. Therefore, it might be beneficial to sort the morsel's pages first before freeing them. Combined with the free reserve optimization, this would again reduce false sharing. However, this is only effective if the performance gains of the reduced false sharing outweigh the additional overhead for sorting.

4.5 Baseline Allocators

Two other allocators were implemented to better assess the performance of the lo(ck|g)-free allocators. They do not fulfill any persistency and consistency guarantees. Their sole purpose is to provide a performance baseline for the benchmarks. The first one is the *ListLocal* allocator, which is based on core-local free lists. During initialization, the available pages are split evenly between the CPU cores and inserted in their corresponding free lists. For every allocation, the core takes an element from its free lists. This page is later put back to the core-local free list for every free. The allocator uses singly linked lists with their next pointers stored within the free pages themselves. Thus these lists do not require memory allocations to grow and shrink. Also, this allocator only manages a single size of pages to keep it simple.

The *ListLocked* allocator uses a similar linked list. However, it is global and locked by a mutex (ticket lock). First, each allocation and free has to take this lock before updating the list. This protects the list against race conditions. However, it causes heavy contention on the lock. Therefore it serves as a negative example in our benchmarks. The *ListLocal* allocator, which is entirely devoid of sharing and lock contention, represents the best case.

4.6 Summary

This chapter addressed the different allocator variants developed in this work. From the bottom up, the lower allocator was described, followed by the different upper allocators. These included the *TableAllocator*, and the *ArrayLocked*, *ArrayAtomic*, and *ArrayAligned* approaches. These either use page tables or arrays and free lists for managing their 1 GiB subtrees. The suffixes *F* and *D* used in the following chapter indicate whether the lower allocator with *F*ixed page tables or *D*ynamic page tables is used. Also, the most critical performance optimizations for the allocators were described before concluding with the *ListLocal* and *ListLocked* allocators that provide baselines for the benchmarks in the following chapter.

5 EVALUATION

The previous chapters introduced the different lo(ck|g)-free allocator approaches and described their functionality in depth. In this chapter, they are benchmarked, analyzed, and compared against the Linux buddy allocator. First, the correctness tests are described, which have guided the allocators' development. Then the hardware and different benchmarks are introduced, together with the kernel module for benchmarking Linux's allocator. After that, the benchmark results are presented and analyzed before evaluating the allocators' crash consistency and recovery capabilities. Finally, the allocators' memory overheads and persistency requirements are discussed.

5.1 Race Condition Tests with Stopping Points

The allocator is extensively tested for race conditions by explicitly ordering the execution of parallel atomic operations. In debug builds, a stopping point is inserted before each atomic load or update. It is similar to a barrier. All threads that reach a stopping point are suspended, and only a single one is executed until it reaches the next stopping point. The order in which threads are chosen and executed is defined at the start. This allows the creation of tests for specific race conditions and updating orders. It is used to test various combinations of concurrent allocate and free operations. Also, a number of stress tests were developed that use random updating orders. They check that all operations are executed successfully and that the allocator's state is consistent. Especially during the development, these tests were helpful to find design and implementation errors and verify the correctness of the algorithms.

However, this technique cannot guarantee that the allocators are entirely devoid of race conditions. This is very difficult [Fu+21]. The probability of their occurrence could be so low that none of the tests was able to catch them. These stopping points are helpful, as they can enforce the occurrence of a race condition. However, they also have limits. Checking every permutation of every atomic operation is practically impossible. Therefore, this technique was only applied to the lower allocators. The upper allocators were tested using more conventional stress tests.

5.2 Microbenchmarks

In this section, the allocator approaches discussed in the previous chapter are benchmarked and compared with the Linux page allocator. To summarize, these are:

Table:	Uses additional higher-level page tables (subsection 4.3.1)
ArrayLocked:	Stores the 3rd-level entries in an array and uses free lists protected by locks (subsection 4.3.2)
ArrayAtomic:	Uses lock-free linked lists avoiding locks (subsection 4.4.1)
ArrayAligned:	Alignes the 3rd-level entries to cache-lines (subsection 4.4.2)
ListLocal:	Uses separate core-local linked lists to store free pages (section 4.5)
ListLocked:	Uses a shared linked list, protected by a ticket lock (section 4.5)
Kernel:	The Linux page allocator (subsection 2.2.3)

The suffixes **F** and **D** indicated which lower allocator implementation is used (**F**ixed page tables 4.2.1 or **D**ynamic page tables 4.2.2).

Experimental Setup

The following benchmarks were executed on a system equipped with two 24-core Intel Xeon Gold 6252 CPUs at 3.7 GHz with hyperthreading. In total, these are 96 virtual cores on two NUMA nodes. Each NUMA node is directly connected to six 32 GiB DRAM DIMMs and six 128 GiB Intel Optane 100 DIMMs. The Optane DIMMs are configured in interleaved mode to get the best performance. The benchmarks are both executed on DRAM, using an anonymous mapping, and on NVRAM by creating and mapping a `fsdax` device on NUMA node 0. This DAX device is configured as a `PMEM` namespace with its metadata (*struct page*) stored in DRAM. All allocators except the Linux allocator are benchmarked in the userspace but with real-time priority (`chrt 99`) to get more stable results. The Ubuntu 20.04.4-based system uses a 5.4.0-97 Kernel with enabled NVDIMM, PMEM, and DAX modules [Nvk].

5.2.1 Benchmarking the Linux Page Allocator

Even if not persistent, the Linux page allocator is a good competitor for the lo(ck|g)-free allocator, as it has similar requirements (section 3.2). As described in subsection 2.2.3, the Linux page allocator's design is based on a buddy allocator. Benchmarking it from the userspace is, without further ado, not possible. Thus a kernel module was developed that directly calls `alloc_page` and `__free_page` of the allocator. The benchmarks and the random number generator, which is based on *wyhash* [Wyh], were reimplemented as part of the kernel module. The benchmarks are directly executed when the module is loaded. After that, the measurements are accessible via an "alloc" *kobject* in the *sysfs*. This read-only *sysfs* file at `/sys/kernel/alloc/out` outputs the benchmark results in the same CSV format as other userspace benchmarks. The *Kernel* allocator was only benchmarked on DRAM because it is currently difficult to execute it on NVRAM.

5.2.2 Bulk Allocations

The bulk allocation benchmark allocates half of the allocators P pages on T threads. Only half the mapped memory is allocated because the allocators have different memory overheads and thus a slightly different number of available pages from the start. Each thread allocates the same number of pages concurrently ($P/(2 \cdot T_{max})$). The time between the first and last allocations is measured and divided by the number of allocations performed. In the next step, these pages are freed in reverse order, and the time is measured similarly. For the DRAM test, 192 GiB of memory ($P = 192 \cdot 512^2$) is given to the allocator. The NVRAM test uses a DAX file of 512 GiB ($P = 512 \cdot 512^2$).

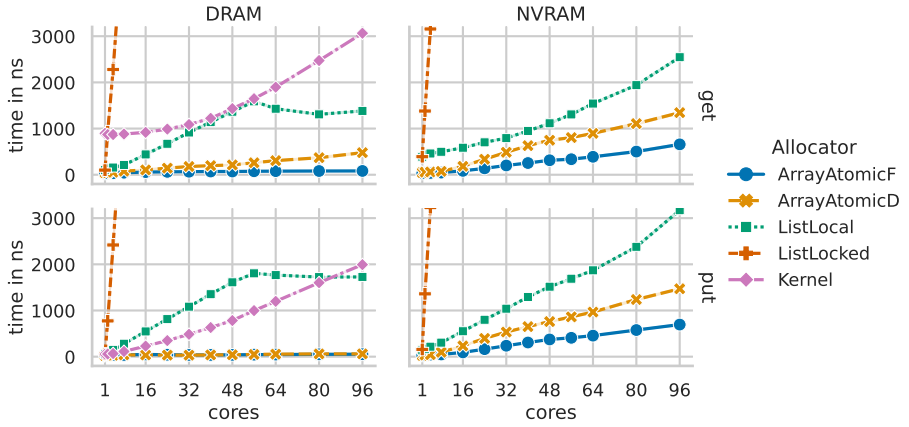


Figure 5.1 – Average time for allocating (get) and freeing (put) numerous 4 KiB pages for different core counts.

Figure 5.1 shows the average allocation and free times for different core counts. The best performing lo(ck|g)-free allocators are displayed together with the *Kernel*, *ListLocal* and *ListLocked* allocators. In this benchmark, all lo(ck|g)-free allocators outperform the other allocators significantly.

The *ListLocked* allocator has by far the worst performance. It has a high contention on the ticket lock protecting its global free list. This allocator does not benefit from multiple cores; on the contrary, it becomes slower at allocating the same number of pages on more cores. Therefore, it represents the worst-case allocation performance in this and the following benchmarks.

The *ListLocal* allocator also performs surprisingly poorly. The allocator's linked lists span over its whole memory because the next pointers are stored directly on its free pages. Consequently, these pointers are on different cache lines. Adding or removing pages from the free list results in frequent cache misses. The Linux profiler (*perf*) shows that in the *ListLocal* benchmark, over 85% of all cache references were misses. In comparison, the *ArrayAtomic* allocator has a miss rate of about 20%. This is even more pronounced at LLC misses with 95% compared to 7.5%. As every cache miss takes orders of magnitude longer, the performance deteriorates sharply. However, the performance improves slightly on DRAM above 48 cores when the cores of the second NUMA node are also used. At this point, both the available cache size and memory bus throughput double.

Also noteworthy is the *Kernel* allocator's less than stellar bulk allocation performance. It starts at over 800 ns for a single core and increases exponentially on higher core counts. Most of the allocator's optimizations, like local free lists, are ineffective in this benchmark due to the sheer number of allocations. Its free times are substantially lower, starting almost as good as the lo(ck|g)-free allocators. However, they increase significantly faster for rising core counts.

The different upper allocators (*ArrayAtomic*, *ArrayLocked*, *ArrayAligned*, and *Table*) have very similar performance characteristics, as shown in Figure 5.2. Only the times of *ArrayAligned* allocators are slightly worse than the other lo(ck|g)-free allocators. Also, their times vary more strongly between executions. This performance degradation is most likely caused by the cache-line alignment of the 3rd-level entries. These allocators occupy more cache lines and also have a slightly higher number of LLC misses than the *ArrayAtomic* implementations. However, this becomes only noticeable when they manage large amounts of memory.

Apart from this, the performance of 4 KiB allocations is hardly affected by the upper allocator approach. For small pages, the subtree reservations are rare compared to the number of allocations

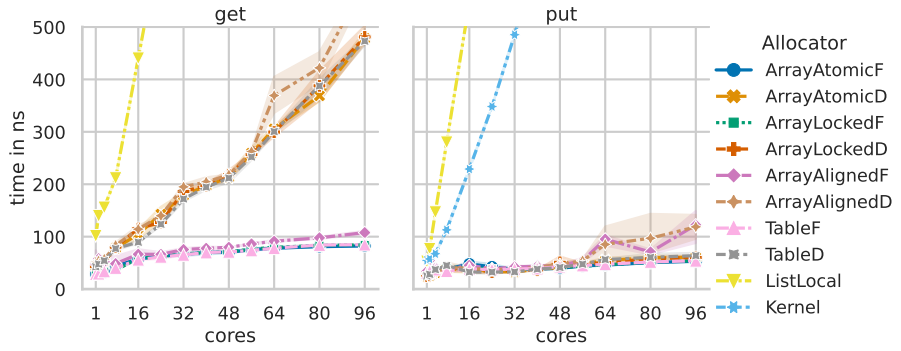


Figure 5.2 – Average time for allocating (get) and freeing (put) numerous 4 KiB pages in DRAM.

within these subtrees. Thus, the performance impact of these reservations vanishes almost completely. Therefore, the different upper implementations have almost equal performance when allocating small pages. The most significant difference comes from the lower allocator implementations. The fixed page table implementation (suffix **F**) is significantly faster than the dynamic page table implementation (suffix **D**). The dynamic version has to perform more checks and periodically reinitialize the 1st-level page tables as described in subsection 4.2.2. The low free times of all the lo(ck|g)-free allocators are achieved by the free-reserve optimization (subsection 4.4.3). It prevents memory sharing very effectively in this benchmark (Figure 4.7).

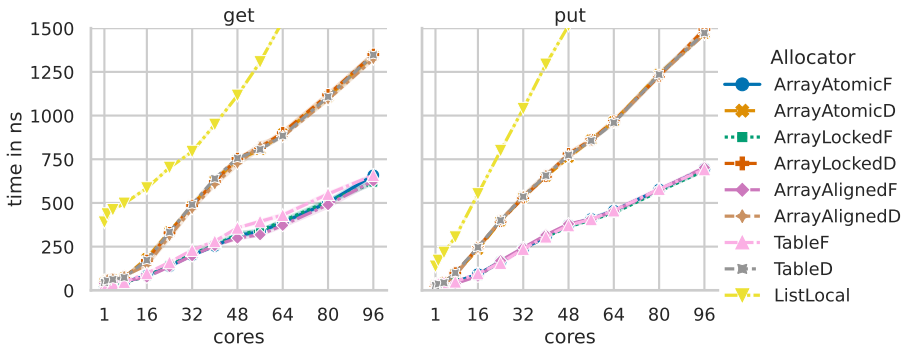


Figure 5.3 – Average time for allocating (get) and freeing (put) numerous 4 KiB pages in NVRAM.

In NVRAM, the fixed version is about twice as fast as the dynamic approach for both allocations and frees (Figure 5.3). In general, the performance of the lo(ck|g)-free allocators is noticeably poorer in NVRAM compared to DRAM. Especially the free times, which are almost constant on DRAM, increase on NVRAM about as fast as the allocation times. This closely resembles the observations made by Yang et al. [Yan+20]. The NVM performance increasingly degrades when the number of threads surpasses the number of interleaved NVDIMMs (6 for these benchmarks). The researchers state that this effect is caused by contention on the integrated memory controllers (iMC) and the XPBuffers of the Optane DIMMs.

5.2.3 Repeat Allocations

This benchmark tests the reallocation performance of the allocators. First, half of the allocator's pages are allocated to initialize and warm up the allocator. Then a page is allocated and directly freed repeatedly. Each core concurrently frees and reallocates a different page. The time is measured between the first and last iteration and divided by the number of iterations. Figure 5.4 shows the average times of these allocation and free sequences for different core counts. Similar to the bulk benchmark, 192 GiB of DRAM memory and 512 GiB of NVRAM memory are used.

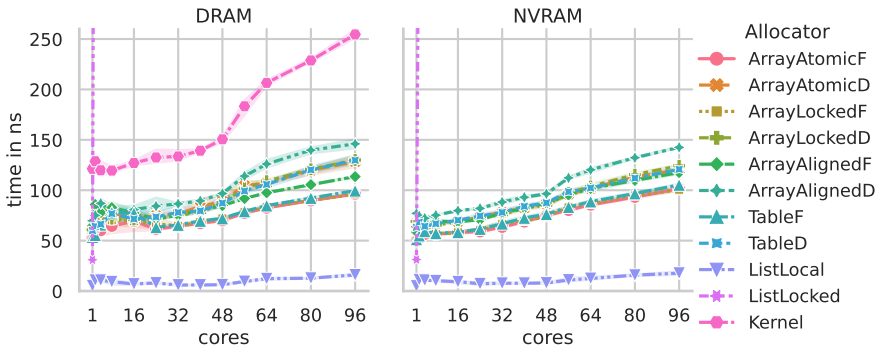


Figure 5.4 – Average time for repeatedly allocating and freeing the same 4 KiB page.

All allocators except the *ListLocked* allocators have a similar performance profile. The runtime slightly increases until 48 virtual cores, after which the slope increases, before it again decreases above 64 cores. At core counts over 48, the cores of the second socket are also used. This performance decrease is caused by remote NUMA accesses to the memory mapped on the first NUMA node. In this benchmark, the *ListLocal* benchmark has by far the best performance. Because the same page is allocated and freed, the allocator can entirely rely on the CPU caches.

The lo(ck|g)-free allocators are about twice as fast as the Kernel allocator. They are faster despite having no specific optimizations for reallocations, except for the per-core subtrees. In contrast, the Kernel allocator has fast paths for these use cases and core-local free lists that cache recently freed pages. These should by themselves have a similar performance as the *ListLocal* allocator. However, the Kernel allocator also manages different memory zones and block sizes. It must first find the appropriate free lists before each allocation.

5.2.4 Random Allocations

This benchmark evaluates the allocator's performance for randomly freeing previously allocated pages and reallocating them. Half of the allocator's pages are initially allocated, similar to the repeat benchmark. Then repeatedly, a previously allocated page is freed and replaced by a newly allocated one. The pages to be freed are chosen randomly from the set of previously allocated pages. The average times for these random free and reallocate sequences are shown in Figure 5.5. Again, the *ListLocked* allocator's performance is terrible due to the high contention on the ticket lock protecting its free list. Therefore, it was only executed for lower core counts. On DRAM, the fixed page table approach (*ArrayAtomicF*) is as fast as the *Kernel* allocator.

The dynamic approach (*ArrayAtomicD*) is far slower in both NVRAM and DRAM than the fixed page table implementations. This difference in performance is even larger in this benchmark than in the bulk benchmark. The random frees are a weak point of the dynamic page table strategy. In this

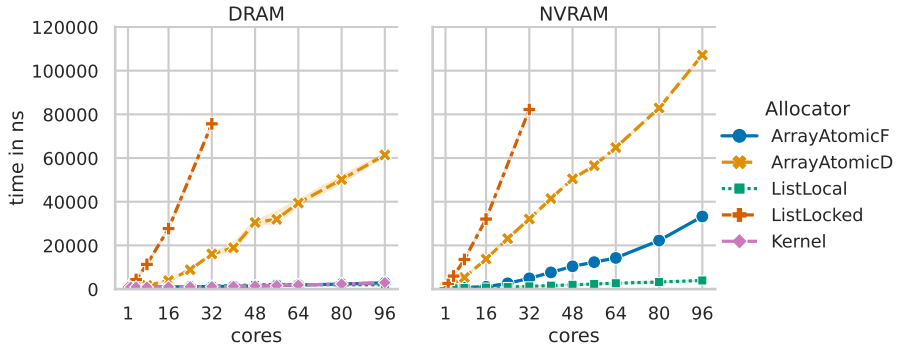


Figure 5.5 – Average time for repeatedly freeing and allocating random 4 KiB pages.

benchmark, the frees mostly affect fully allocated areas. In these cases, the dynamic approach has to initialize the first freed pages as new page tables (subsection 4.2.2). This initialization overwrites the whole page, resulting in a much longer runtime. In the bulk benchmark, this effect is amortized by subsequent faster frees in the same subtree. However, this is mostly not the case if the frees affect random areas.

The fixed implementations also perform significantly worse in NVRAM than in the bulk benchmark. These results can be explained by the higher amount of cache misses because random page tables entries have to be updated. Additionally, these random read-modify-write sequences perform even worse on Optane due to higher latencies and contention on the XPBuffers [Yan+20].

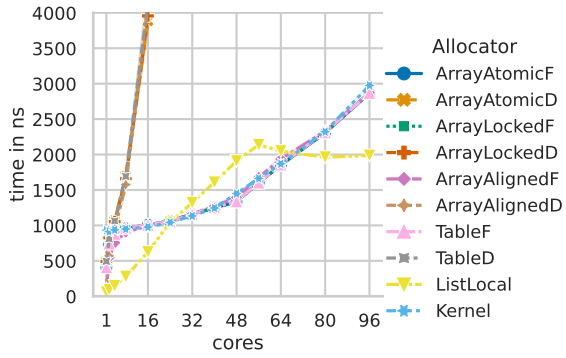


Figure 5.6 – Average times for repeatedly freeing and allocating random 4 KiB pages in DRAM.

Figure 5.6 shows the benchmark results for DRAM. All lo(ck|g)-free allocators are comparable with the Kernel allocator for up to four cores. After that, the runtimes of the dynamic versions increase dramatically. However, the fixed implementations have almost identical runtimes as the *Kernel* allocator. The performance of the *ListLocal* allocator is similar to the bulk benchmark. Its performance is again limited by its high number of cache misses.

This benchmark shows how heavily the lo(ck|g)-free allocators' performances depend on the locality of the freed pages. The runtimes differ in more than one order of magnitude from the previous benchmarks. Especially the dynamic page table implementations perform less than stellar.

5.2.5 Different Filling Levels

As the performance of allocators often varies on different filling levels, this also has been tested for the *Table* and *Array* allocators. For example, depending on how the allocator searches free pages, its performance could significantly decrease as it runs out of memory.

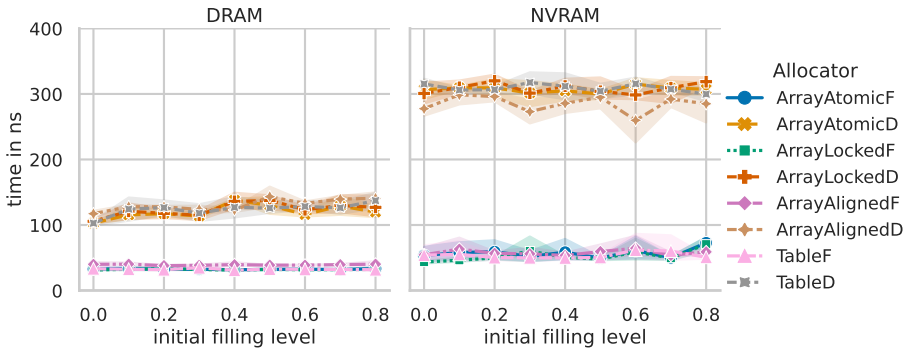


Figure 5.7 – Average time for allocating pages on different initial filling levels.

Figure 5.7 shows that the allocation times of the lo(ck|g)-free allocators remain constant for different filling levels. This is the case because the cores mainly access their reserved subtrees. As long as every core can reserve a subtree, the total filling level does not matter. Also, the reservation of new subtrees does not depend on the filling level for the *Array* allocators. The performance of the linked lists or stacks containing the partially or entirely empty subtrees remains constant, regardless of how many elements they contain. The *Table* allocators use counters to speed up page table traversal and thus are also barely affected by the filling level.

However, the drawback is that if the allocator only has a few pages left, the reservation of a new subtree could fail entirely. This happens if all not reserved subtrees are fully allocated, and the only free pages are part of subtrees reserved by other cores. It is currently impossible to access a subtree reserved by another core and the free pages within. Consequently, in this particular case, the allocation fails even if some pages are still free. As a fallback, an additional allocation procedure could be implemented, either allocating remotely in reserved subtrees or migrating the allocation to another core. However, this is beyond the scope of this work.

5.2.6 Huge and Giant Pages

The previous benchmarks measured the allocator's performance for small 4 KiB pages. This section covers the management of huge and giant pages. Again the bulk, repeat, and random benchmarks were used to evaluate the allocators' performances for these page sizes.

Figure 5.8 shows the average allocation (get) and free (put) time of the bulk benchmark for 2 MiB pages. The allocation times double but mostly remain similar to the 4 KiB pages. This can be explained by a higher rate of subtree reservations. A subtree contains 512^2 small pages but only 512 huge pages. Therefore subtrees have to be reserved more frequently for huge pages. This reservation takes additional time as it operates on shared data. Noticeable outliers are the dynamic and fixed implementations of the *ArrayLocked* allocator. Their runtimes increase faster on higher core counts, which can be attributed to lock contention.

For frees, the runtimes of the fixed and dynamic approaches are entirely different from each other. The free times of the fixed allocators are almost equal to their allocation times. However, the

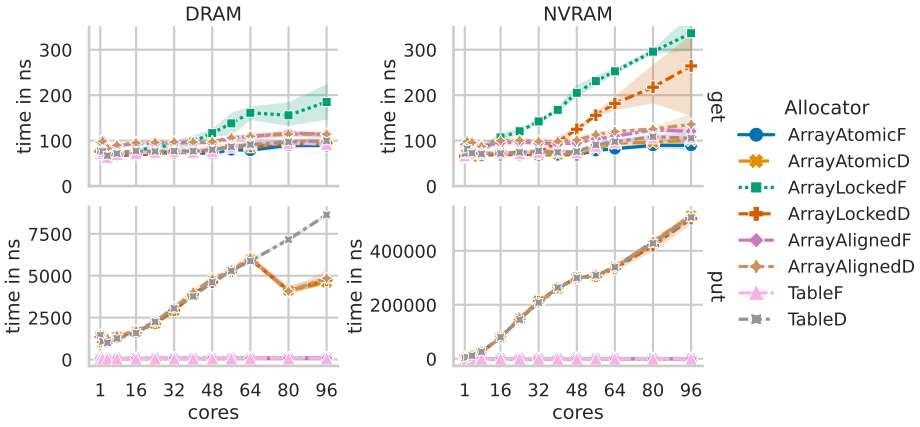


Figure 5.8 – Average time for "bulk" allocating/freeing 2 MiB pages.

free times of the dynamic implementations are orders of magnitude slower. The dynamic approach stores the 1st-level page tables as a page inside their 2 MiB area. This makes it possible to allocate these page table pages and thus reduce the memory overhead (subsection 4.2.2). The downside of this strategy is that frees periodically have to reinitialize page tables if they were previously given to the user as part of an allocation. This requires overwriting a whole 4 KiB page. For small 4 KiB pages, the page table only must be reinitialized if a free occurs in a fully allocated area, which happens only every 512th free on average. However, for huge 2 KiB pages, this has to be done on every free, negatively impacting the performance. Interestingly, the free times for the dynamic *Array* allocators drop significantly on DRAM for over 64 cores. This is most likely again a NUMA effect, as above 48 cores, the second socked is also used.

The results of the repeat benchmark for huge pages are similar to the 4 KiB pages. The times also increase by factor two. The random benchmark produces similar results as the bulk frees. The performance is also affected primarily by the more frequent subtree reservations and the dynamic allocators' 1st-level page table reinitializations.

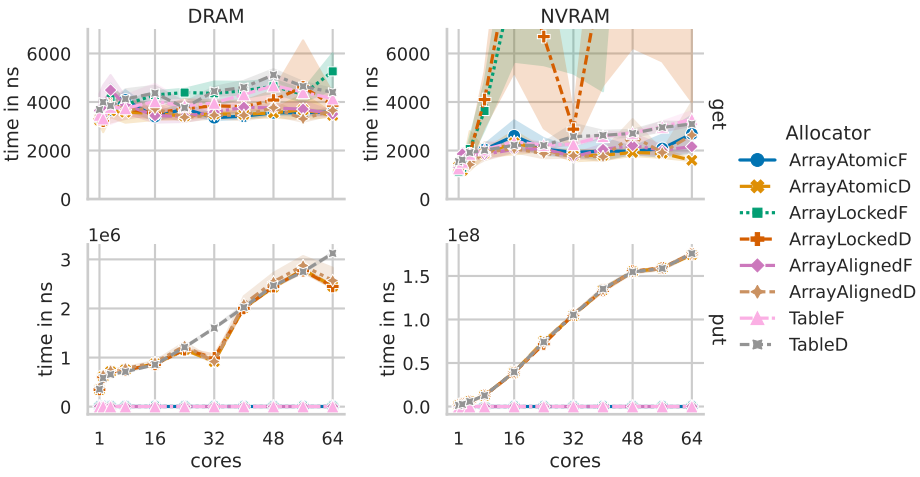


Figure 5.9 – Average time for "bulk" allocating/freeing 1 GiB pages.

Figure 5.8 shows the average allocation and free time of the bulk benchmark for 1 GiB pages. These times are more than two orders of magnitude higher than for huge pages. Every allocation now reserves a new subtree that is allocated as a giant page. These reservations operate on shared data and thus take much more time. Also, the dynamic allocators have to reinitialize 512 1st-level page tables for every freed 1 GiB page. This further decreases their performance.

To summarize, the runtimes of the fixed allocators remain reasonably low for huge and giant pages. However, the dynamic approach has significantly worse performance in the bulk-free and random benchmarks with larger pages.

5.3 Memory Access

All accesses to the NVM are performed using atomic operations with *sequentially-consistent* memory ordering (subsection 2.3.1) to guarantee consistency and persistency. Because these operations are costly on NVM [Yan+20], the number of atomic updates the allocator performs was reduced as far as possible. For regular allocations, in the best case, only two CAS operations have to be executed: One in the 2nd-level page table for decrementing the page counter and another one in the 1st-level page table for setting the entry to present. If one of them fails, it is retried for the following entries until it succeeds. In these cases, the number of executed CAS operations is slightly higher. However, this usually only happens in very few cases because the address of the last allocation is used as starting point for the next search (section 3.5). In addition to that, the search is sequential, hence ideal for the CPU caches. In most cases, the following entries are already in the cache because they are part of the same cache line the previous CAS operation tried to update. For free operations, the corresponding 2nd-level entry is read first, checking if the whole 2 MiB subtree was allocated as a huge page or a 4 KiB page was allocated. In the second case, a CAS operation in the 1st-level page table is performed, marking the entry as free, and then the counter of the 2nd-level entry is increased. The other updates in the data structures of the upper implementations only affect DRAM.

For the dynamic page table implementation, the 1st-level page tables have to be reinitialized in certain cases (by setting all entries to zero). Depending on the platform, SIMD instructions (SSE2 or AVX512) are used to zeroize these pages. After that, the corresponding 2nd-level entry is updated. To guarantee consistency, this is followed by a memory barrier. The benchmarks, however, show that even this optimized overwriting of a page still negatively affects performance.

5.4 Recovery and Crash Consistency

Because the allocator was designed for eADR, no explicit flushing is used. Therefore it is not guaranteed to be recoverable on the 100 series Optane memory of the test system. Only series 200 and newer support the new eADR power failure protection domain. As this hardware was not available to us, actual recovery tests from power losses could unfortunately not be performed.

Instead, a special test case was created to assess the allocator's recovery capabilities as accurately as possible. It is based on shared memory mappings between processes. Two shared memory mappings are created and then shared with a child process using `fork`. One of the mappings is used for the allocator. The other mapping is used for verifying the recovery. It has the layout:

```
[ ( idx | init | pages... ) foreach thread ]
```

where `idx` is the index of the currently allocated or freed page, and `init` is a status flag indicating whether the corresponding thread has finished initialization.

The parent process waits for the child to initialize the allocator and to start allocating and freeing random pages on multiple threads (similar to the random benchmark). After a random delay, it kills the child process and the allocator. Then it initiates the allocator's recovery procedure and checks that only a limited number of pages were lost. After that, it tries to free all allocated pages, verifying that they were correctly allocated in the first place. This ignores all pages that were allocated or freed during the crash (referenced by `idx`). These interrupted operations are recovered as either completely finished or not performed at all, but neither is guaranteed to be the case. It is only checked that the number of interrupted operations is limited by the number of threads.

As described in section 3.2, the possibility of losing certain pages during a crash is not optimal. However, depending on the use case, this can be further mitigated: Either by a per-core undo log, stored in NVM, or on a higher level by the application that uses the allocator. Morsels, for example, persistently store all allocated pages. They could be used to relax the persistency requirements of the allocator further, as discussed in section 5.6.

5.5 Metadata Overhead

The Table 5.1 shows the amount of memory each allocator implementation takes up when it is fully allocated. The upper allocators' metadata is volatile. The metadata of the lower *Fixed* and *Dynamic* implementations is stored persistently. Each lo(ck|g)-free allocator consists of a combination of the lower and upper implementations. The resulting memory overhead is the sum of these parts. The memory overhead of the upper implementations is negligible compared to the lower implementations. Also, the most significant difference is between the lower *Fixed* and *Dynamic* approaches.

Allocator	Metadata Size for P pages	$P = 512^3$ (512 GiB)
Table	$4\,\text{KiB} \cdot (\lceil P/512^3 \rceil + 1)$	8 KiB
ArrayLocked	$24\,\text{B} \cdot \lceil P/512^2 \rceil$	12 KiB
ArrayAtomic	$8\,\text{B} \cdot \lceil P/512^2 \rceil$	4 KiB
ArrayAligned	$64\,\text{B} \cdot \lceil P/512^2 \rceil$	32 KiB
Fixed	$4\,\text{KiB} \cdot (\lceil P/512 \rceil + \lceil P/512^2 \rceil)$	1026 MiB
Dynamic	$4\,\text{KiB} \cdot \lceil P/512^2 \rceil$	2 MiB
Linux `struct page`	$64\,\text{B} \cdot P$	8192 MiB

Table 5.1 – Memory overhead of the allocator implementations.
The upper allocators have an additional overhead of $(2 + 2C) \cdot 64\,\text{B}$ for C cores, to store shared and per-core data. Also, an additional NVM page for metadata is reserved for the recovery.

The previous benchmarks show that the fixed page table implementation scales at least twice as good on multiple cores as the dynamic one. However, the downside is its high memory overhead. It takes up more than 512 times more space than the dynamic approach, as shown in Table 5.1. The fixed approach needs a 1st-level page table entry for every 4 KiB page. Only a single bit of this entry is used to mark the page as allocated. The allocator does not use the remaining 63-bits. Access to these page table entries could be handed over to the OS, allowing it to (persistently) store any data related to these pages. This could include reference counters or any other metadata that Linux stores in its `struct page`. If needed, their size could also be increased to store more data. These page table entries can be accessed directly, as they are stored in a known location at the end of the NVM area. This could be done similarly for huge and giant pages.

Alternatively, the size of these 1st-level page tables could be reduced. Only a single bit per entry is needed to store if a page is allocated or not. Thus a bitfield with 512 bits would be enough to store

this information, reducing the table's size from 4 KiB to 64 B. This lower allocator would only need 18 MiB metadata for 512^3 pages. However, the 1st-level entries could not be updated independently anymore as the smallest atomic operation modifies 8 bit. Consequently, two atomic operations are needed to load the current value and write it back after modification using CAS. If a race condition occurs in between them it has to be tried again. Despite this, initial tests showed an improved performance from the reduced memory size. However, this has to be further investigated in future iterations of the allocators.

5.6 A Volatile Morsel Allocator

The morsels are persistent and self-contained. If they store all pages they use for metadata and content, the allocator might not have to be persistent. However, the memory pages it manages still have to be taken from NVM. The allocator also would have to have a function to mark pages as already allocated. During boot, this function could be used to mark all pages of all morsels as allocated. In order to find all morsels on reboot, their root page tables (or pointers to them) have to be stored persistently in a known location.

However, such a volatile allocator can only be used if all pages are stored in persistent data structures, like morsels, and are easy to find. All not persistently stored pages would be lost after a crash. Also, iterating through all morsels could significantly slow down the boot process, depending on their number and size.

The benchmarks showed that even on volatile memory, the lo(ck|g)-free allocators perform better than the *Kernel* allocator. Additionally, these allocators can easily be extended with a function to mark arbitrary pages as already allocated. Their page table-based architecture makes it trivial to find the corresponding page table entries and update them. Therefore, the lo(ck|g)-free allocators are an excellent option for the morsels even when ignoring their recovery capabilities.

5.7 Summary

In this chapter, the different allocator approaches were compared against each other and the Linux *Kernel* allocator. In most cases, they were faster than the *Kernel* allocator. However, it became clear that the lo(ck|g)-free allocator's free performance heavily depends on the locality of the freed pages. Current NVM hardware further amplifies this by similarly relying on the locality of memory accesses. The lo(ck|g)-free allocator fulfills the persistency and recovery requirements specified in subsection 3.2.2. The dynamic lower allocators have a very low memory overhead but are significantly slower than their fixed counterparts. The higher memory overhead of the fixed page table implementations can also be counteracted by allowing the OS to store page-related data in the unused page table entries or replace them with smaller bitfields.

6 CONCLUSION

In this work, I revisited the design of page allocators for modern CPU and memory architectures. The developed lo(ck|g)-free allocator variants, true to their name, avoid locks and logs to achieve persistency on current eADR enabled Optane memory. They are recoverable from system crashes and guarantee that the number of potentially leaked pages is bound to the system's core count. The use of hardware-near data structures like page tables proved to be very beneficial to the allocator's performance. Together with several optimizations to reduce memory sharing, these allocators out-scale the Linux page allocator on high core counts. This further underpins the hypothesis that reducing abstractions could significantly increase the performance of the Linux memory management subsystem. The new minimal address space abstractions (*morsels*) briefly introduced in this work aim to achieve precisely this. They directly use MMU-compatible page tables for their own metadata. This work both developed allocator concepts that can be used to lazily allocate pages for the morsels, fulfilling their persistency requirements, and further underlines the performance gains such hardware-near implementations can provide. Furthermore, the allocator could be excellent for specific userspace applications working with large, evenly-sized memory chunks, like database systems.

Despite the promising benchmarking results, some improvements can still be made for the allocator. The poor performance of random frees leaves much room in this regard. Using an extra free list to cache recent frees or implementing a bulk-free mechanism that sorts addresses before freeing them could make a huge difference. The allocator also needs a fallback strategy for remotely allocating pages from subtrees reserved by different cores when no other pages are left. This is especially important because the chunks, CPUs reserve, are pretty extensive, containing 512^2 4 KiB pages. Also, the allocator was currently only tested on hardware not supporting eADR. As the allocator was specifically designed for this, it limits the way its persistency capabilities can be verified. Thus it might be interesting to check if the allocator is genuinely able to recover from a power loss on real hardware.

Without additional work, the lo(ck|g)-free allocator also cannot be evaluated accurately in real-world scenarios. This mainly requires two things: (1) The allocator must be integrated into the Linux kernel to run in kernel space. Similar to its counterpart, it must be able to prevent CPU migrations and interrupts for the duration of an allocation or free. (2) To be accessible from userspace, the morsels themselves have to be implemented as kernel objects, together with the means for creating and mounting them into address spaces. Then the allocator and morsel can be directly compared with the traditional abstractions for mapping and sharing memory. This would also allow comparing different morsel allocators with one another. After that, the persistency guarantees of the allocator could be further strengthened to lose no pages at all. The addresses of the allocated pages could be atomically inserted into persistent data structures, like morsels or persistent pointers.

LIST OF ACRONYMS

ADR	Asynchronous DRAM Refresh
ASLR	Address Space Layout Randomization
CAS	Compare-And-Swap
DAX	Direct Access of Files
DMA	Direct Memory Access
eADR	Extended ADR
IOMMU	Input-Output Memory Management Unit
LLC	Last Level Cache
MMU	Memory Management Unit
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Byte-Addressable Memory
PCIe	Peripheral Component Interconnect Express
PCM	Phase-Change Memory
PTE	Page Table Entry
RDMA	Remote Direct Memory Access
TLB	Translation Lookaside Buffer
WPQ	Write Pending Queue

LIST OF FIGURES

REFERENCES

[Bai+11] Katelin Bailey et al. "Operating System Implications of Fast, Cheap, Non-Volatile Memory." In: *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. 2011.

[BCB16] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. "Makalu: Fast recoverable allocation of non-volatile memory." In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 677–694.

[Bit+17] Daniel Bittman et al. *Twizzler: An operating system for next-generation memory hierarchies*. Tech. rep. Technical Report UCSC-SSRC-17-01, University of California, Santa Cruz, 2017.

[Bit+21] Daniel Bittman et al. "Twizzler: A Data-Centric OS for Non-Volatile Memory." In: *ACM Trans. Storage* 17.2 (2021). ISSN: 1553-3077. DOI: 10.1145/3454129. URL: https://doi.org/10.1145/3454129.

[Che+96] Peter M. Chen et al. "The Rio File Cache: Surviving Operating System Crashes." In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1996, 74–83. ISBN: 0897917677. DOI: 10.1145/237090.237154. URL: https://doi.org/10.1145/237090.237154.

[Cob+11] Joel Coburn et al. "NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories." In: *SIGARCH Comput. Archit. News* 39.1 (Mar. 2011), 105–118. ISSN: 0163-5964. DOI: 10.1145/1961295.1950380. URL: https://doi.org/10.1145/1961295.1950380.

[Cop+89] G. Copeland et al. "The Case for Safe RAM." In: *Proceedings of the 15th International Conference on Very Large Data Bases*. VLDB '89. Amsterdam, The Netherlands: Morgan Kaufmann Publishers Inc., 1989, 327–335. ISBN: 1558601015.

[Dav+18] Tudor David et al. "Log-Free Concurrent Data Structures." In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, 373–385. ISBN: 9781931971447.

[DLN19] Dominik Durner, Viktor Leis, and Thomas Neumann. "Experimental Study of Memory Allocation for High-Performance Query Processing." In: *ADMS@ VLDB*. 2019, pp. 1–9.

[Eva06] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD." In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[Fra04] Keir Fraser. *Practical lock-freedom*. Tech. rep. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory, Feb. 2004. DOI: 10.48456/tr-579. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[Fri+18] Michal Friedman et al. "A Persistent Lock-Free Queue for Non-Volatile Memory." In: *SIGPLAN Not.* 53.1 (2018), 28–40. ISSN: 0362-1340. DOI: 10.1145/3200691.3178490. URL: https://doi.org/10.1145/3200691.3178490.

[Fu+21] Xinwei Fu et al. "Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, 100–115. ISBN: 9781450387095. DOI: 10.1145/3477132.3483556. URL: https://doi.org/10.1145/3477132.3483556.

[Gli] *The GNU Allocator - The GNU C Library*. 2021. URL: https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html (visited on 01/19/2022).

[Gor04] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[Gra+03] Ananth Grama et al. *Introduction to parallel computing*. Pearson Education, 2003.

[Her88] Maurice P. Herlihy. "Impossibility and Universality Results for Wait-Free Synchronization." In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '88. Toronto, Ontario, Canada: Association for Computing Machinery, 1988, 276–290. ISBN: 0897912772. DOI: 10.1145/62546.62593. URL: https://doi.org/10.1145/62546.62593.

[HJW15] Taeho Hwang, Jaemin Jung, and Youjip Won. "HEAPO: Heap-Based Persistent Object Store." In: *ACM Trans. Storage* 11.1 (Dec. 2015). ISSN: 1553-3077. DOI: 10.1145/2629619. URL: https://doi.org/10.1145/2629619.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. "Obstruction-free synchronization: double-ended queues as an example." In: *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* 2003, pp. 522–529. DOI: 10.1109/ICDCS.2003.1203503.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[IMS16a] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. "Brief Announcement: Preserving Happens-before in Persistent Memory." In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. Pacific Grove, California, USA: Association for Computing Machinery, 2016, 157–159. ISBN: 9781450342100. DOI: 10.1145/2935764.2935810. URL: https://doi.org/10.1145/2935764.2935810.

[IMS16b] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model." In: *Distributed Computing*. Ed. by Cyril Gavoille and David Ilcinkas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327. ISBN: 978-3-662-53426-7.

[Int] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Intel®, 2021.

[Int21] Intel. *eADR: New Opportunities for Persistent Memory Applications*. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html (visited on 12/10/2021).

[Izr+19] Joseph Izraelevitz et al. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module." In: (2019). DOI: 10.48550/ARXIV.1903.05714. URL: https://arxiv.org/abs/1903.05714.

[Jem] *jemalloc Memory Allocator*. 2022. URL: http://jemalloc.net/ (visited on 01/25/2022).

[Kad+21] Rohan Kadekodi et al. "WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, 804–818. ISBN: 9781450387095. DOI: 10.1145/3477132.3483567. URL: https://doi.org/10.1145/3477132.3483567.

[KP12] Alex Kogan and Erez Petrank. "A Methodology for Creating Fast Wait-Free Data Structures." In: *SIGPLAN Not.* 47.8 (2012), 141–150. ISSN: 0362-1340. DOI: 10.1145/2370036.2145835. URL: https://doi.org/10.1145/2370036.2145835.

[Kum+21] Sandeep Kumar et al. "Page Table Management for Heterogeneous Memory Systems." In: *CoRR* abs/2103.10779 (2021). arXiv: 2103.10779. URL: https://arxiv.org/abs/2103.10779.

[Lin] *Memory Allocation Guide - The Linux Kernel Documentation*. 2021. URL: https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html (visited on 01/19/2022).

[Mic04] M.M. Michael. "Hazard pointers: safe memory reclamation for lock-free objects." In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004), pp. 491–504. DOI: 10.1109/TPDS.2004.8.

[Mor+13] Iulian Moraru et al. "Consistent, Durable, and Safe Memory Management for Byte-Addressable Non Volatile Main Memory." In: *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. TRIOS '13. Farmington, Pennsylvania: Association for Computing Machinery, 2013. ISBN: 9781450324632. DOI: 10.1145/2524211.2524216. URL: https://doi.org/10.1145/2524211.2524216.

[Nvk] *Persistent Memory Wiki*. 2022. URL: https://nvdimm.wiki.kernel.org/ (visited on 03/10/2022).

[Ouk+17] Ismail Oukid et al. "Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems." In: *Proc. VLDB Endow.* 10.11 (2017), 1166–1177. ISSN: 2150-8097. DOI: 10.14778/3137628.3137629. URL: https://doi.org/10.14778/3137628.3137629.

[Pel+15] Omer Peleg et al. "Utilizing the IOMMU Scalably." In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 549–562. ISBN: 978-1-931971-225.

[Pmd] *pmem.io: Persistent Memory Development Kit*. 2021. URL: https://pmem.io/pmdk/ (visited on 12/10/2021).

[Pth] *pthreads(7) – Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html (visited on 12/10/2021).

[Rao+08] S. Raoux et al. "Phase-change random access memory: A scalable technology." In: *IBM Journal of Research and Development* 52.4.5 (2008), pp. 465–479. DOI: 10.1147/rd.524.0465.

[Ray+21] Amanda Raybuck et al. "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, 392–407. ISBN: 9781450387095. DOI: 10.1145/3477132.3483550. URL: https://doi.org/10.1145/3477132.3483550.

[Rus] *Rust Standard Library – std::sync*. 2021. URL: https://doc.rust-lang.org/std/sync/index.html (visited on 12/10/2021).

[Sch+15] David Schwalb et al. "nvm malloc: Memory Allocation for NVRAM." In: *ADMS@ VLDB* 15 (2015), pp. 61–72.

[Stda] *C++ Atomic operations library - std::memory_order*. 2021. URL: https://en.cppreference.com/w/cpp/atomic/memory_order (visited on 12/10/2021).

[Stdb] *C++ Thread support library*. 2021. URL: https://en.cppreference.com/w/cpp/thread (visited on 12/10/2021).

[Ven+11] Shivaram Venkataraman et al. "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory." In: *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. FAST'11. San Jose, California: USENIX Association, 2011, p. 5. ISBN: 9781931971829.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: Lightweight Persistent Memory." In: *SIGARCH Comput. Archit. News* 39.1 (2011), 91–104. ISSN: 0163-5964. DOI: 10.1145/1961295.1950379. URL: https://doi.org/10.1145/1961295.1950379.

[Wyh] *wyhash: The FASTEST QUALITY hash function, random number generators (PRNG) and hash map*. 2022. URL: https://github.com/wangyi-fudan/wyhash (visited on 02/22/2022).

[XS16] Jian Xu and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338. ISBN: 978-1-931971-28-7.

[Yan+20] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory." In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. ISBN: 978-1-939133-12-0.