LLFree: Scalable and Optionally-Persistent Page-Frame Allocation

Lars Wrenger, Florian Rommel, and Alexander Halbuer, *Leibniz Universität Hannover;* Christian Dietrich, *Hamburg University of Technology;* Daniel Lohmann, *Leibniz Universität Hannover*

https://www.usenix.org/conference/atc23/presentation/wrenger

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

LLFree: Scalable and Optionally-Persistent Page-Frame Allocation

Lars Wrenger
Leibniz Universität Hannover

Florian Rommel
Leibniz Universität Hannover

Alexander Halbuer
Leibniz Universität Hannover

Christian Dietrich
Hamburg University of Technology

Daniel Lohmann
Leibniz Universität Hannover

Abstract

Within the operating-system's memory-management subsystem, the page-frame allocator is the most fundamental component. It administers the physical-memory frames, which are required to populate the page-table tree. Although the appearance of heterogeneous, nonvolatile, and huge memories has drastically changed the memory hierarchy, we still manage our physical memory with the seminal methods from the 1960s.

With this paper, we argue that it is time to revisit the design of page-frame allocators. We demonstrate that the Linux frame allocator not only scales poorly on multi-core systems, but it also comes with a high memory overhead, suffers from huge-frame fragmentation, and uses scattered data structures that hinder its usage as a persistent-memory allocator. With LLFree, we provide a new lock- and log-free allocator design that scales well, has a small memory footprint, and is readily applicable to nonvolatile memory. LLFree uses cache-friendly data structures and exhibits antifragmentation behavior without inducing additional performance overheads. Compared to the Linux frame allocator, LLFree reduces the allocation time for concurrent 4 KiB allocations by up to 88 percent and for 2 MiB allocations by up to 98 percent. For memory compaction, LLFree decreases the number of required page movements by 64 percent.

1 Introduction

In any virtual-memory subsystem, the allocation of *physical memory* is a vital base primitive. Classically, the OS hands out physical memory in *page frames* of MMU-imposed sizes and uses simple free lists [42, 50] (Windows, Darwin) or specialized buddy-allocators [28] (Linux, FreeBSD) to manage multiple frame sizes. However, recent hardware trends (i.e., high core counts and NVRAM) challenge these *frame-allocator* designs.

One significant trend is the appearance of fast [49] byte-addressable *nonvolatile RAM (NVRAM)* in the form of Intel Optane DIMMs. Inspired by their persistence property, new programming models [5, 39, 45, 52], file systems [10, 40, 48], and crash-tolerant data structures [8, 13, 46] were proposed and evaluated. And while Intel's announcement [25] to wind down its Optane business will make NVRAM harder to obtain in the next years, it has been shown that low-cost, high-capacity NVRAM is feasible and has significant potential [1, 23, 31]. Furthermore, multiple researchers realized that persistence and scalability are deeply entangled properties [26, 29] that benefit both from lock-free algorithms [8, 16, 46] and constructive avoidance of inconsistent intermediate states.

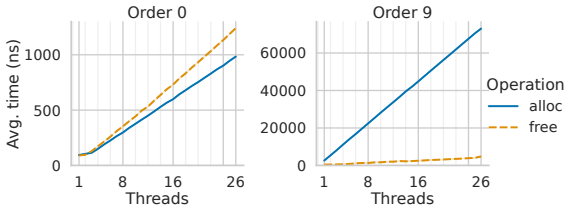

Figure 1: Linux frame allocator performance for concurrent allocations of 4 KiB (order 0) and 2 MiB (order 9) huge frames.

Together with many cores competing for resources, large-capacity NVRAM modules provoke the question of how, at which costs, and with what guarantees the OS hands out the available memory, which might be used for persistent data. For example, for databases, current virtual-memory subsystems can have a significant impact on their design [12, 35], query-processing speeds [14] and buffer management [12, 32]. Therefore, we believe it is time to revisit the whole virtual-memory stack, starting from the bottom; the frame allocator.

First, we investigated whether the Linux frame allocator [18] and its underlying buddy system [28] still match today's requirements, not only for safely allocating frames from NVRAM but also for the scalable management of DRAM. Fig. 1 shows the multi-core scalability of bulk allocations. With all 26 cores allocating in parallel, 4 KiB allocations slow down by a factor of *ten* (94 ns→984 ns), while 2 MiB alloca-

tions are even 27 times slower! This poor scalability affects many multicore and memory-heavy workloads [7]. The root causes are the scattered allocator state and the usage of global locks, both of which are also problematic [16, 17] for a crash-tolerant NVRAM adaptation.

About this paper

We propose LLFREE, a persistent, lock- and log–free page-frame allocator that: (1) focuses on *memory-management unit (MMU)*-specific memory sizes, (2) scales well on multi-core systems by reducing memory sharing, (3) is memory efficient due to its small amount of metadata, (4) has automatic huge-frame defragmentation, and (5) is always in a consistent state and, thus, well suited for persistent memory. In this paper, we claim the following contributions:

- We explore the weaknesses of the Linux buddy allocator and simpler list-based frame allocators.
- We derive design principles for hardware-centric lock- and log-free physical memory management.
- With LLFREE, we provide a page-frame allocator that is suited for both volatile and nonvolatile memory.
- We replace the Linux buddy allocator with LLFREE and conduct a comprehensive evaluation to compare the two allocators in terms of performance, scalability, spatial overhead, fragmentation behavior, and crash consistency.

2 Problem Analysis: Linux Frame Allocator

The page-frame allocator must provide physical-memory *frames*, which have MMU-specific granularities, are naturally aligned, and are used to set up virtual address spaces. For this paper, we will stick to the AMD64 MMU and its frame sizes (4 KiB, 2 MiB, 1 GiB), which we call *natural (allocation) sizes*. However, the general design can be adapted to other page sizes. For 4 KiB, we will use the term *base frame*, for 2 MiB *huge frame*, and for 1 GiB *giant frame*, respectively. While some kernels (e.g., Windows and Darwin) use simple free lists [42, 50], Linux (and FreeBSD) use the buddy system [18, 28].

Linux Buddy Allocator A buddy allocator avoids external fragmentation by allowing only allocation sizes of the form $2^o \times P$, where P is the smallest size and o is the allocation *order*. For each order, the buddy system keeps a bucket of naturally-aligned free blocks. If an allocation hits an empty bucket o, a block from the bucket $o+1$ is requested and halved into two *buddies*, whose start addresses differ only in a single bit. One buddy is returned, the other is put into the order-o bucket. The free operation tries to recursively merge the block with already freed buddy blocks before putting the block into a bucket. To speed up merging, buckets are usually implemented as doubly-linked lists and a one-bit flag is used to track which blocks are available for merging.

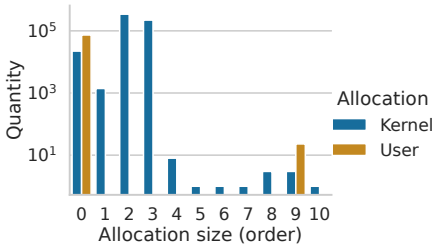

Figure 2: Requested allocation sizes during system startup and a 120 s memcached+memtier benchmark.

Linux employs one buddy allocator per memory zone (e.g., for each NUMA node), supports the orders $o \in \{0, \ldots, 10\}$, and uses the base-frame size as P. Therefore, the supported sizes are between 4 KiB and 4 MiB on AMD64. Unlike a general-purpose buddy allocator, Linux does not store the list pointers within the free memory, as this would require all memory to be mapped, which is not supported by all architectures. Instead, it uses the `struct page` for its metadata.[1] Also, each zone allocator is protected by a spin lock, which serializes all split and merge operations. In order to reduce contention on these locks, Linux further employs per-CPU caches for the most frequently-used orders.

Problem 1: Mixing of Concerns Linux's frame allocator is not only used for hardware-sized page frames but *also* for allocating contiguous physical-memory ranges of various orders. Although this was necessary for allocating *direct-memory access (DMA)* buffers before the widespread adoption of I/O-MMUs, there is no technical requirement for this anymore. However, the Linux developers still use those non-native sizes to allocate larger kernel objects (e.g., stacks). To get a feeling for this, we recorded the requested sizes during boot and a subsequent `memcached` benchmark (see Fig. 2): We see that userspace memory gets only requested for the natural orders 0 and 9, whereas the kernel mostly requests non-natural orders. Allocating contiguous blocks of frames for kernel objects might still be beneficial to save TLB entries (Linux identity-maps all physical memory into the kernel space with giant frames). However, by mixing frame- and kernel-object allocation, the allocator has to provide all orders via its interface, which leads to a number of secondary problems.

Problem 2: Merge Cost Because there are nine buddy orders between the base and huge frames, the transition between both is costly: In the worst case, starting from 512 4 KiB frames, it takes 511 buddy-merge operations to form a single 2 MiB frame. For each merge and under lock, we have to manipulate list pointers in five cache lines; four are in `struct pages` that are usually not yet in the cache.

Problem 3: Scalability Furthermore, as we have seen in Fig. 1, these already costly operations scale poorly if re-

[1] With `struct page`, Linux has a per-frame information store that is used and repurposed by different subsystems. On AMD64, it is 64 bytes large.

quested by multiple threads. The reason for this is the contention at the mentioned per-zone lock, which Linux tries to mitigate by maintaining per-CPU caches for some orders. Each per-CPU cache keeps a list of free blocks, which are drained on memory pressure or if the cache exceeds a watermark. While for a long time, only order-0 allocations were cached, Linux 5.13 extended the caches to order 1-3 and order 9 (2 MiB). However, allocation-heavy workloads easily overwhelm these per-CPU caches. Also, they aggravate fragmentation and complexity.

Problem 4: Huge-Frame Fragmentation Although the buddy system prioritizes the smallest-fitting bucket, it has a problem with huge-page fragmentation: For example, if a single 4 KiB piece of an otherwise free 2 MiB frame is in use, the other 511 base frames exist as 9 blocks of different orders (4 KiB to 1 MiB). Since buckets are unsorted sets and the allocator has no concept of "almost-full huge pages", those blocks have equal chances of being allocated as any other block from any other huge frame. As a result, the buddy system does not specifically aim to minimize the fragmentation of huge frames. We will discuss this further in Sec. 5.6.

The per-CPU caches aggravate fragmentation as they delay merge operations. Even if the last missing 4 KiB frame of a 2 MiB frame has already been freed, it may reside in a per-CPU cache that would have to be flushed to complete the huge frame. Furthermore, as the per-CPU cache hides memory from the buddy allocator, we cannot employ an intra-bucket heuristic that increases the likelihood of a huge-frame merge (e.g., appending to the end of the bucket list). On the contrary, a recently freed block that *could* complete a 2 MiB frame is more likely to be allocated again.

In order to reduce huge-page fragmentation, Linux has supported "high-atomic page blocks" since 2015 to isolate larger blocks and prevent them from being fragmented by small allocations. Furthermore, Linux also employs active defragmentation (memory compaction), where a background task iterates through a memory zone and moves pages to the beginning, clearing larger chunks at the end. However, both induce additional complexity.

Problem 5: Persistent Allocations Given its current structure, the Linux frame allocator is unsuitable for persistent allocations. For persistent NVRAM zones, the allocator must be able to recover its state in case of a power loss to ensure the persistence of the required metadata. This is challenging [17] for complex algorithms (such as lock-protected recursive frame merging), distributed state (such as doubly linked lists), or redundancy (such as per-CPU caches). While in theory, each of these problems could be solved with extra logging protocols [36, 41, 45], the performance, memory, and complexity impact of doing so would be excessive. Also, this logging overhead would have to be paid for every regular operation despite crashes being usually extremely rare. Thus, we do not consider this as a realistic option. To our knowledge, there are currently no persistent (page-frame) allocators that achieve allocation times close to their volatile counterparts [3, 9, 33, 36, 41, 45, 52].

Problem Summary: Complexity In the end, the Linux physical-memory allocator suffers from complexity. Mixing the concerns of frame-sized and other allocation quantities (Problem 1) motivates the buddy structure, which, however, leads to high merge costs (Problem 2) and lock-based, doubly-linked traversals that hamper multi-core scalability (Problem 3). This is mitigated by an increasing number of per-CPU caches, which (combined with the buddy system) unfortunately worsens huge-frame fragmentation (Problem 4), requiring additional mechanisms such as high-atomic page blocks and memory compaction that further increase the complexity. All this results in a design that is unsuitable for persistent allocations (Problem 5) due to redundant and distributed storage of data and state.

All these design decisions were most probably well-founded when they were made. We argue that the time has come to revisit the design and structure of the most fundamental memory manager in our systems, the page-frame allocator.

3 The LLFree Page-Frame Allocator

We originally designed LLFree as a page-frame allocator for natural frame sizes (4 KiB, 2 MiB) only, with the goal of high scalability and suitability for persistent allocations. For the Linux integration, we later had to extend it to support also non-natural allocation orders, which to our surprise, worked out without having to compromise on any of our design goals. In the following, we describe the original design, while the integration particularities are left to Sec. 4.

3.1 Design Principles

For our allocator, we specified three major design principles:

Respect Hardware The hardware characteristics define both the structural elements for and the features of the software implementation. We leverage MMU-defined frame sizes, cache-line granularity in data structures, and available atomics in algorithms.

Avoid Sharing On systems with multiple CPUs, both true and false sharing are major bottlenecks for scalability, becoming even more significant on NUMA systems. Locks are a frequent cause of sharing. We reduce access to shared data structures and do not use locks.

Careful Redundancy Redundant information (such as software caches and replicas) must be kept in sync. This is especially difficult to accomplish when targeting crash consistency on persistent memory since a potential crash can disrupt the synchronization of these redundant data structures. Therefore, we strictly limit redundancy for the state that is required for crash recovery.

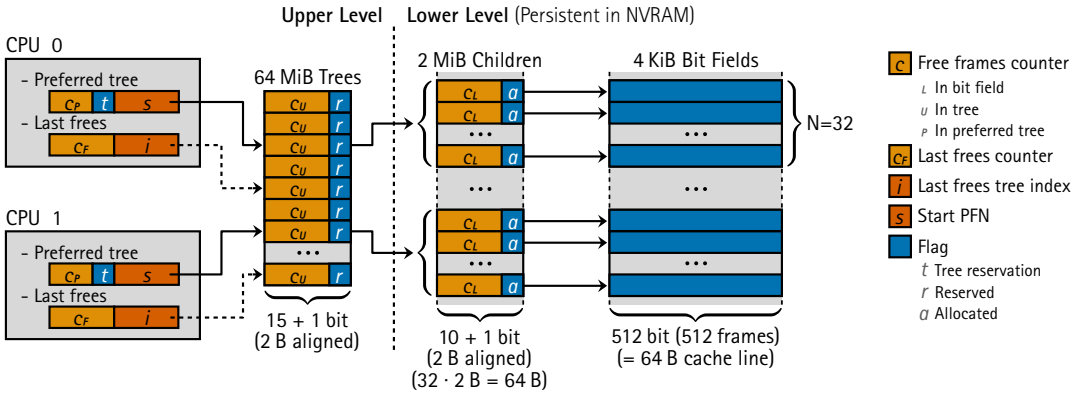

Figure 3: Architecture of the LLFREE allocator: The tree entries, child entries, and bit fields are stored consecutively in large arrays. From a PFN, we can directly extract the indices of the corresponding tree entry, child entry, bit field, and the frame's allocated bit within.

We do not claim fundamental novelty for these principles: The first two are well established in the domain of scalable OS kernel development; their benefits have been reported many times [6, 15, 22, 47]. This does not particularly hold for the third principle, which is more targeted at reaching crash consistency than scalability. On the contrary, employing rather than limiting redundancy is a common technique in kernel design to avoid sharing and *improve* scalability (as we have seen in the previous section), so there is a trade-off between the two. This deep entangling of persistence and scalability, as well as the importance of avoiding intermediate states for crash consistency on NVRAM, has already been reported in the domain of data structures and algorithms [8, 16, 26, 29, 46]. The contribution of this paper is the rigorous combination and application of these principles and dealing with their trade-offs in the design of a page-frame allocator that is *both* scalable on DRAM *and* optionally crash-consistent on NVRAM.

3.2 LLFREE: Design Overview

Fig. 3 depicts the architecture of LLFREE, which has been designed for the natural allocation orders 0 (4 KiB base frames) and 9 (2 MiB huge frames). Conceptually, LLFREE is divided into two levels: A *lower level* that performs the actual allocations and an *upper level* that implements allocation strategies to avoid sharing and fragmentation. Roughly speaking, the lower level takes responsibility for crash consistency – only its state needs to be kept persistent on a crash-consistent NVRAM zone – while the upper level provides for scalability.

3.2.1 Lower Level - Allocation Mechanisms

The lower level manages de/allocation of base and huge frames. For this, it employs a table entry and a bit field of 512 bits (Fig. 3: 2 MiB Children, 4 KiB Bit Fields) for every huge frame to mark the free (0) and taken (1) base frames from this huge frame. The number of free base frames is also maintained in the counter c_L. Base frames are allocated by first atomically decrementing c_L (this prevents races for the last free frame) and then searching the bit field for a 0 bit, an operation supported by special processor instructions. Huge frames are allocated by just changing the counter c_L from 512 to 0 and setting the *allocated flat* flag a, all within a single 16-bit *compare-and-swap (CAS)* operation. The bit field remains untouched, and all-zero in this case, the necessary bookkeeping (e.g., for crash recovery) is done in the a flag.

Thus, in most cases, the lower-level allocator needs to touch only two cache lines for de/allocating a base frame (table entry, bit field) and just one cache line for a huge frame (table entry).[2] Still, even if the current child/tree does not contain enough frames for an allocation, our sequential search fits well with the processor's cache-line prefetching.

3.2.2 Upper Level - Allocation Strategies

The upper level provides for scalability by using allocation strategies that constructively minimize (false) sharing and huge-frame fragmentation. Technically, it manages the physical memory as an array of chunks we call *trees* (Fig. 3): Each tree root refers to a lower-level table with N children that, in turn, refer to N bit fields, each managing 512 frames, similar to a page-table tree. We chose $N = 32$ so that child arrays (in the lower level) occupy a single cache line; hence a tree manages 16 384 base frames, respectively 64 MiB.

Each tree root contains the count c_U of free base frames (for allocation strategies), as well as a reserved flag r (for per-CPU pinning). Our early benchmarks showed that allocations suffer from false sharing when entries in the lower-level child array are updated concurrently. Hence, to avoid sharing, each CPU can pin ($r = 1$) a preferred tree for its allocations.

If there are no free frames left in the preferred tree, a new tree has to be reserved. The reservation algorithm follows a search heuristic to avoid fragmentation of huge frames by using c_U to classify trees into one of three classes:

[2] One additional cache-line access will happen in the upper level.

allocated	Almost all frames are taken ($c_U < 12.5\,\%$).
free	Almost all frames are free ($c_U > 87.5\,\%$).
partial	Everything in between.

The heuristic prioritizes *partial* trees over *free* and *allocated* ones. Thereby, (almost) *free* trees have a higher chance of becoming entirely free over time. However, we still select *free* trees before *allocated* ones for a new CPU-preferred tree, as the latter bear a high probability that allocations might (soon) fail again, especially for huge frames. This is a trade-off between performance and fragmentation.

The thresholds determining whether a tree is *free*, *partial*, or *allocated* are configurable. Our benchmarks showed that the thresholds of 12.5 percent (2048 free base frames for $N = 32$) for *free* and 87.5 percent for *allocated* ones are sufficient to avoid fragmentation (Sec. 5.6). The actual search for an appropriate tree is done in the following order:

1. First, the *neighborhood* of the current CPU tree is searched for *partial* or *free* trees, with *neighborhood* being defined as the 31 other tree roots that reside on the same cache line.
2. If this does not succeed, the whole trees array is sequentially searched (first-fit) for a *partial* tree.
3. If no *partial* tree is found, the search is repeated for a *free* tree or an *allocated* one with enough free pages.
4. As last resort, the allocator drains (unreserves) the trees of other CPUs and steals them.

Note, however, that even the slow path of reserving a new tree does not require locks and can be done fully parallel, as the reservation itself requires only the atomic update of the reserved flag of an entry. Given a memory zone of 256 GiB, the algorithm accesses 128 cache lines in the worst case.

CPU-Local Tree Roots: Despite using per-CPU reservations, the updates on the tree array can still suffer from false sharing when multiple CPUs concurrently update the free counters of entries sharing a cache line. This can be solved in two ways: either by aligning the 2 B tree entries to the cache line size (64 B on x86) or by splitting the counter into a global and a CPU-local part. We followed the second approach to keep the memory and cache overhead low. On tree reservation, a CPU moves the free counter c_U of the reserved tree to its CPU-local data c_P and sets c_U to zero. Allocations and frees from this CPU now change only the local counter c_P. Allocations from other CPUs no longer happen, as the tree is reserved, but foreign frees from previous allocations may still happen. In this case, only the global counter c_U in the trees array is incremented, which avoids invalidating the cache line of the respective local entry. The counters are synchronized if a local allocation runs out of memory ($c_P = 0$) or, in rare cases, by remotely draining a reserved tree from another CPU, which also clears the tree's reservation.

Besides the local free counter c_P, the CPU-local entries also contain an *in-tree reservation* flag t, and the *start PFN*, combined into a single 64 bit data type. The *start PFN* contains the last allocated base frame number; it acts as a last-fit pointer to speed up allocations in the child array (divide by 512) and also identifies the reserved tree (divide by $512 \cdot N$).

The t flag is atomically set during the reservation of a new per-CPU tree to prevent races with remote draining or parallel reservation attempts. The latter could only happen if the zone is too small for one tree per core. If this is the case, the per-CPU data is shared between multiple cores and the t-flag coordinates the allocation of a new tree.[3]

Reserve-on-Free: While per-CPU trees prevent contention and false sharing for concurrent allocations, frees must always go to the tree that was the origin of the respective frame. If the source CPU of the frame has meanwhile switched to a new tree or the free is invoked from another CPU, this tree is not the per-CPU tree. Especially on memory-intensive workloads, frees thereby may still suffer from false sharing.

To mitigate this, LLFREE provides a *reserve-on-free* heuristic that assumes that allocation and free workloads exhibit locality: A CPU reserves a tree as its preferred tree after F consecutive free operations targeted it, expecting that subsequent frees will also affect this tree (c_F and i in Fig. 3). The threshold $F = 4$ performed best in our benchmarks.

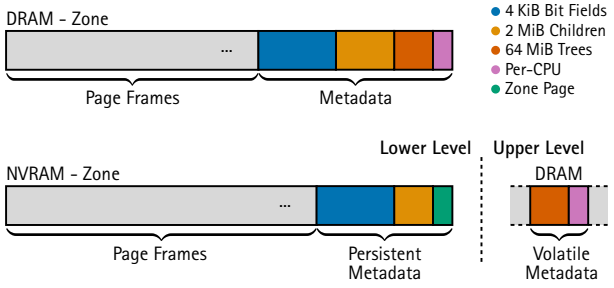

Figure 4: Layout of LLFREE-managed memory zones in persistent and non-persistent memory.

3.2.3 Crash Consistency

LLFREE optionally provides crash-consistency on NVRAM zones. For a crash-consistent NVRAM zone, only the allocator's lower level (cf. Fig. 3) has to be stored within the persistent memory – plus one extra zone page storing a magic identifier, the size of the memory region, and a startup/shutdown marker to detect if the system needs to be recovered after a power loss. The upper level always resides in DRAM to reduce access latencies and avoid unnecessary writes to NVRAM. It is restored into DRAM from the lower-level information. Fig. 4 depicts the zone layout for persistent and non-persistent memory zones.

[3] On Linux-x86, this holds only for the 16 MiB DMA zone, which Linux still offers for compatibility with legacy 16-bit ISA devices.

The key point of LLFree's design is that all transactions authoritatively happen within a single atomic change of a single cache line. For base frames, this is the atomic change in the respective bit field, while huge-frame de/allocation is implemented by the atomic change of a and c_L in the child entry. This ensures *memory consistency* as each authoritative change is visible to all cache-coherent cores, and all derived information (i.e., counter values) are altered by atomic commutative operations, which are not affected by reordering.

The single–cache-line property also makes it easy to implement *persistence consistency* on all systems that provide a *persist granularity* [38] of at least one cache line to be written atomically to the NVRAM at the end of the operation, which is considered to be the minimum standard for NVRAM hardware [10, 38]. In case of a recovery, each entry in the child array is checked for the a flag: If it is set, the child entry is the authoritative information – the entry is a huge page with $c_L = 0$ (the bit field is all-zero). Otherwise, the entry describes a set of base pages – the bit field is the authoritative information, which is used to restore the value of c_L. The upper-level state can then be restored from the child array.

While a crash during a de/allocation would never result in an unrecoverable allocator state, it could lead to a lost frame. This would happen if in an allocation the bit has already been set, but the frame has not yet reached the caller, or if the deallocation has been invoked but not yet cleared the bit. The theoretical worst-case bound for this effect is the maximum number of parallel operations, that is, the number of CPUs. This could be mitigated by two-phase de/allocation protocols, which, however, would also change the interface of the page allocator. Crashes are rare events – and a crash during the critical phase of a page-frame allocation even more so (the endangered code sequences take only a few clock cycles). Hence, the probability of an actual frame loss during the lifetime of a system is extremely low, while even in the case of such an incident, the costs would be acceptable.

4 Implementation

We implemented LLFree as a Rust module, integrated it with (and extended it for) the Linux kernel, and replaced the original Linux buddy allocator with LLFree. Its allocate and free algorithms, discussed in the previous section, can be found in the appendix (Fig. 14).

4.1 Approach

As the Linux community has started to adopt the Rust language, we took the chance to explore how well it is suited for a performance-critical low-level kernel module. Compared to C, Rust has much more restricted (i.e., safer) memory management and avoids undefined behavior. We are convinced that these properties, which prevent entire classes of memory bugs, simplified the development of the allocator.

The modular LLFree implementation contains a test environment that initializes the allocator on a virtual memory mapping in user space for unit testing and benchmarking. This made it possible to profile and find performance bottlenecks early on. Besides standard unit tests, we developed specific race-condition tests for the possible orders of atomic operations, which proved quite helpful in finding several design and logic errors.

Following this approach, we were able to quickly implement and compare strategies for the upper and lower level of the allocator. The final implementation consists of 2 199 lines of safe Rust code (with 25 unsafe lines for initialization and address translations) and 1 318 lines of unit tests. The Linux buddy allocator is written in C and mainly contained in `page_alloc.c` (excluding reclaiming and memory compaction), which alone has 6 060 lines of code – without any tests. However, the buddy allocator is tightly coupled with other memory subsystem components, making it difficult to estimate its actual contribution to this source base.

4.2 Replacing the Linux Buddy Allocator

We modified Linux to boot with our LLFree allocator. The integration required some modifications to LLFree (support for non-natural orders), but especially to Linux itself due to the tightly coupled implementation of the buddy allocator. Nonetheless, the system seems stable for everyday workloads.

4.2.1 LLFree Changes

LLFree was specifically designed for the natural orders defined by the hardware. Linux, however, requires supporting all orders up to 10, which we implemented as follows: Orders 1 to 6 (2 to 64 frames) can be allocated similarly to order 0. The allocation now searches for a large enough, aligned set of zeros in the bit fields and allocates it, toggling the bits with a single 64-bit CAS operation.

Orders 7 and 8 (128 and 256 frames) are allocated with an optimistic lock-free algorithm using 2–4 atomic operations. If one fails due to a race, the others are safely reverted, and the search continues. However, these orders are rarely allocated (cf. Fig. 2), and contention could only occur if a tree is stolen during an allocation or we explicitly share a tree among multiple CPUs. Hence, an actual conflict is a rare event.[4]

Order 10 was implemented by allocating two aligned child entries at once, similar to an order 9 allocation. As these child entries are only 16 bit large (with alignment), this is done with a single 32-bit CAS operation. If a higher-order allocation fails because the tree is fragmented, another one is reserved. In this case, the allocator searches for a *free* tree that is not fragmented before falling back on *partial* ones.

[4] This also breaks our assumption for persistence consistency from Sec. 3.2.3 for orders 7 and 8. However, we can safely ignore this for now, as non-natural orders are only employed by the kernel, which currently does not use persistence.

4.2.2 Linux Changes

During boot, the data structures for the LLFREE allocator are allocated by the early-boot `memblock` allocator and initialized. Like the buddy allocator, we create one LLFREE instance per memory zone and store a pointer to its data directly in the `struct zone`. Most further changes in the code base are to conditionally disable and replace the buddy allocator, its per-CPU caches, zone locks, and high-atomic page blocks when our implementation is activated (via a Kconfig option). In total, the LLFREE module, a thin wrapper around the LLFREE allocator, added 942 lines. Outside this module, we changed 415 lines with 296 alone in `page_alloc.c`.

Most functionalities, including page reclamation, were easily adapted to the new allocator. However, some higher-level services that directly use the buddy allocator's internal data structures, e.g., its free lists, turned out impossible to adapt without completely rewriting them. Hence, we disabled them for both allocators in the benchmarks. First, this includes the memory fragmentation heuristic that decides whether active memory compaction should be executed. The heuristic uses the internal counters of the buddy-allocator's free lists. However, the need for active memory compaction is an exceptional state, only triggered when the allocator is highly fragmented with almost no huge pages left. It does not happen in our benchmarks. Instead, we compare fragmentation and compaction costs in Sec. 5.6. Second, we deactivated the out-of-memory (OOM) handler, as its checks directly rely on the internal free lists of the buddy allocator. Considering the complexity of the OOM procedure, we consider its redesign is outside the scope of this paper. Our benchmarks do not trigger OOM events for both allocators.

To make the higher order allocation speeds in Linux competitive, we had to implement two workarounds: The first reduces the number of write accesses to the `struct page` entries. As an aid for kernel-internal debugging (e.g., detecting double frees), Linux reinitializes the flags of *all* `struct pages` backing a higher order allocation, which is a costly and mostly unnecessary overhead. Linux also differentiates between standard allocations, which can be freed in parts, and *compound* allocations, which can only be freed at once. For *compound* allocations, all but the first `struct page` are marked as *tail* page. This scattered, redundant encoding of *compound* frames is costly, especially for huge frames. Modifying all 512 entries (i.e., 512 cache lines) for every de/allocation is detrimental to the overall performance, making it hard to distinguish and compare the allocation speeds of both allocators. Thus, we benchmarked standard allocations and disabled the flag reinitialization. The latter did not have any observable consequences.

The second bottleneck that affected especially LLFREE is the updating of the vmstat free-frames counter, which provides an estimate of the available free memory. This counter is updated for de/allocations that escape the per-CPU caches. To reduce congestion, each CPU has a local counter absorbing updates up to a certain threshold. However, its current value of 125 is too small for huge frames. We increased the threshold by 1024 (the size of the order 9 per-CPU caches). Because the LLFREE allocator does not use these caches, the global counter remains as accurate as with the buddy allocator using the original value, as the latter may hide as many frames in its per-CPU caches.

5 Evaluation

In our evaluation, we show that LLFREE scales well for different allocation patterns and sizes. We also look at the fragmentation behavior, quantify the memory overhead, and investigate crash recovery for the NVRAM case.

5.1 Evaluation Setup and Benchmarks

Our test system is a DELL PowerEdge R750 with two Gen 3 Intel(R) Xeon(R) Gold 5320 CPUs (2× 26 physical cores @ 2.20 GHz). Each of the two NUMA nodes has four 32 GiB DRAM DIMMs (total: 256 GiB DRAM) and four 128 GiB Optane Gen 2 DIMMs (total: 1 TiB NVRAM). We assume the eADR persistence guarantees offered by the Gen 3 Xeon Gold architecture [24] (memory consistency ↦ persistence consistency) and omit explicit persisting cache flushes (`clwb`) in the implementation, as this also provides for a more direct comparison of DRAM and NVRAM allocation speeds.[5] For stable results, we disable proactive memory zeroing (a recently introduced optional hardening feature) and hyper-threading, which yields similar general performance characteristics, with the exception that memory sharing is costlier between physical cores than between logical ones. We execute our benchmarks on the modified Linux 6.0 kernel with and without LLFREE. As Linux instantiates allocators per memory zone (e.g., NUMA-1-DRAM) that do not impact each other, we perform isolated tests with a single NUMA-node allocator.

As allocator performance is highly work-load specific, we created three synthetic benchmarks that cover a wide range of allocation patterns: (1) For the *bulk* benchmark, all cores allocate half of the available memory at once and directly free it up again; this process is repeated. (2) The *random* benchmark allocates all memory(similar to the bulk benchmark) and frees the frames in random order; we only measure free operations. (3) For the *repeat* benchmark, each core allocates and frees a single frame as fast as possible. *Repeat* has been deliberately constructed as a best-case scenario for the Linux buddy allocator, as it maximizes the expected benefit of the local per-CPU caches. Nevertheless, this is the least realistic scenario: Common is lazy allocation (due to demand paging) with bulk/random free (at program termination).

[5] Note that LLFREE could also work with weaker persistency (Sec. 3.2.3).

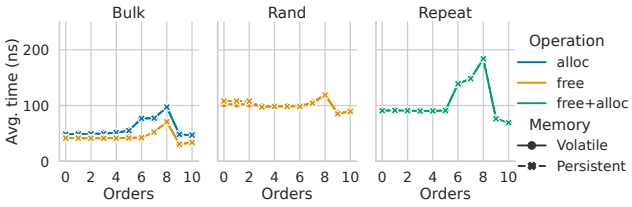

Figure 5: Per-order allocation time of standalone LLFREE on DRAM and NVRAM (8 cores, 128 GiB)

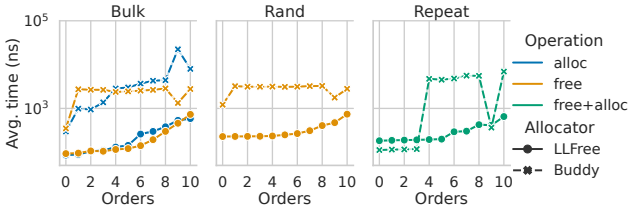

Figure 6: Per-order allocation time of Linux-integrated LLFREE (8 cores, 128 GiB DRAM, logarithmic scale)

As individual operations execute fast, per-operation time measurements would distort the results. Therefore, we measure the time for all operations and divide it by their number. Since LLFREE benefits from free locality, we expect the random benchmark to be especially challenging.

5.2 Allocation Sizes

First, we look at the allocation speeds of the different request sizes (4 KiB – 4 MiB). For this, eight CPU-pinned threads manage 128 GiB of DRAM or DAX-mapped Optane memory. The benchmarks are executed both in our userspace benchmark environment to measure the *standalone* performance of LLFREE and with a kernel module in the modified Linux.

Fig. 5 shows the average time per operation for the userspace benchmarks. For bulk and repeat, *one* operation costs less than 100 ns, while order 8 (1 MiB), which is the furthest from the next lower natural order, is the most expensive one. Due to random's cache-miss and invalidation behavior, a free operation can take up to 120 ns. Even though Optane is known to have around twice the random-access latency of DRAM [49], the resulting allocator performance is very similar for DRAM and NVRAM, as most updates remain in the L3 caches.

After these userspace results, we replaced the Linux buddy allocator with LLFREE for a quantitative in-situ comparison. This integration induces higher management overheads (e.g., updating `struct page`), which causes additional cache misses compared to the previous userspace benchmark. As Linux's allocator is not crash-consistent, we now only look at DRAM performance. Again, eight cores on the first NUMA node execute the respective benchmark in parallel.

Fig. 6 shows that LLFREE is about one order of magnitude faster than the original Linux allocator for the bulk and random benchmarks. For the repeat benchmark, where a single frame is reallocated repeatedly, the per-CPU caches (order 0-3, 9) make the buddy allocator faster than LLFREE (e.g., for order 0: 112 ns vs.183 ns). This stems not only from the caching itself but also from the fact that some statistics (i.e., `vmstat` counters, NUMA hit/miss rates) are not updated if the cache services a frame request. Nevertheless, for sizes that are not covered by per-CPU caches, LLFREE is about 100 times faster in the repeat benchmark.

However, the per-CPU caches are not beneficial for all workloads: In the bulk benchmark, we see that for order-9 allocations, the buddy allocator is about ten times slower than orders 8 and 10. An in-depth analysis revealed the problem: As the order-9 caches only have a capacity of two, the cache-refill operation is invoked for every other frame. This refill batches the allocation of multiple frames - two in this case - into a single critical section, reducing the number of acquire and release operations to the buddy lock. However, this critical section also contains a check for all `struct pages` of the allocated frames, which is especially expensive for higher-order allocations. Therefore, the lock is held longer, increasing lock contention compared to the other orders without caches that perform these checks after releasing the lock.

If we compare Fig. 5 and Fig. 6, we see that there is still potential for improvement in Linux's allocation path. As the other overheads scale linearly with the number of covered 4 KiB frames (for updating `struct page`), we are currently unable to fully harvest LLFREE's performance for 2 MiB frames. For example, while 4 KiB random frees are equally fast within the kernel, freeing a huge frame takes 4.48 times longer.

5.3 Multicore Scalability

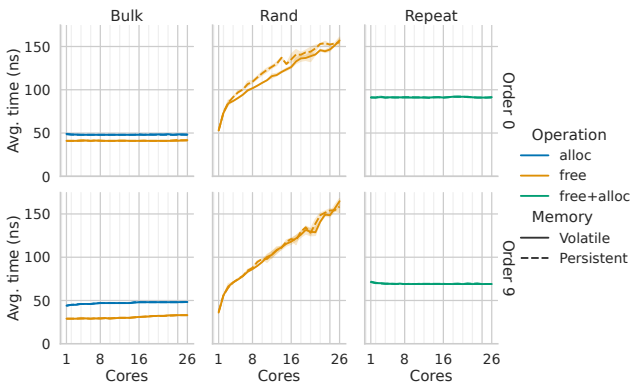

Figure 7: Average time per core count on orders 0 and 9 and 128 GiB memory of LLFREE in volatile and persistent memory

To evaluate multicore scalability, we focus on the two natural frame sizes and scale the number of requesting cores from 1 to 26. Again, Fig. 7 shows the raw LLFREE performance (in

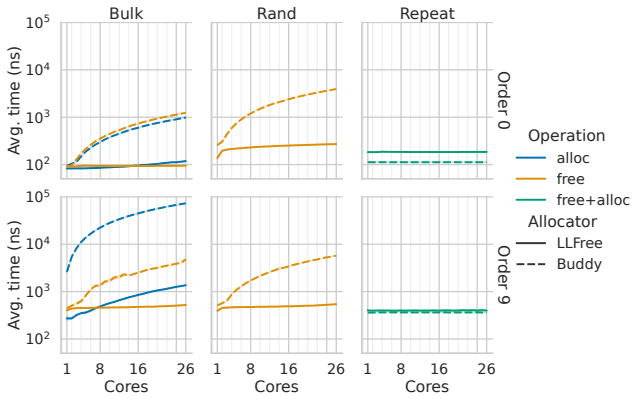

Figure 8: Average time per core count on orders 0 and 9 and 128 GiB memory in the Linux kernel on a logarithmic scale

userspace) on DRAM and NVRAM, while Fig. 8 (log-scale!) shows the in-kernel performance on DRAM.

For the bulk and repeat userspace benchmarks (Fig. 7), we see that LLFREE's operation times remain almost constant, independent of the number of cores, and the memory type has only an insignificant influence on the performance. Here, LLFREE's allocation and free reservation system avoids most sharing. Only for random, where cache invalidations and update conflicts on child counters are more frequent, we see a significant impact of more workers. However, even with 26 workers that request frames in parallel, a DRAM allocation/free of either order takes less than 170 ns.

In Fig. 8, we see that the Linux performance is heavily influenced by its per-CPU caches for the natural sizes. For the repeat benchmark, which is the best case for per-CPU caches, we see that LLFREE is up to 65.18 percent (26 cores, 4 KiB) slower than Linux. However, for bulk and random, which exceed the capacity of the per-CPU caches, the Linux allocator shows severe performance drops for more cores as memory is now requested directly from the buddy system. While the single-core performance for 4 KiB is still almost equal, Linux takes 7.3/13.5 (bulk allocations/random frees) *times* longer with 26 cores than LLFREE. For 26 cores requesting 2 MiB frames, this changes to 52.5/9.6 times.

5.4 List-Based Allocators

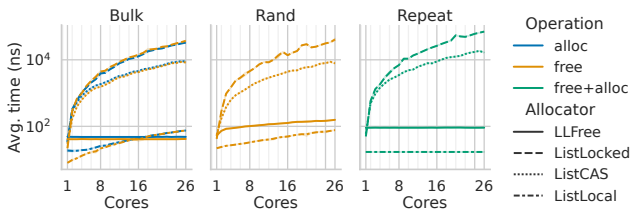

Figure 9: Average speed of the list and LLFREE allocators per core count for order 0 on 128 GiB DRAM

Besides the buddy system, simple free lists are a common design for page-frame allocators in commodity (Windows [50], Darwin [42]) and research (Twizzler [5]) operating systems. Like Linux, they mitigate lock contention on the global structures by additional core-local lists. To compare these allocator concepts with LLFREE, we built three prototypical list-based allocator implementations and evaluated their scalability. (1) The *ListLocked* allocator, consisting of a lock-based shared singly-linked list (Windows, Darwin), (2) the *ListCAS* allocator, which uses a LIFO lock-free list instead [44] (supposed to be preferable over locking), and (3) the *ListLocal* allocator, a theoretical allocator that maintains per-core lists only, does not need any protection and never drains (ideal case). All list allocators store their next pointers in a 64 B aligned array, similar to Linux's `struct page` array.

In Fig. 9, we see the results of this comparison on a logarithmic scale. As expected, the locked variant has the worst performance due to the high degree of lock contention. However, while replacing the lock with atomic operations improves the situation by 81 percent (26 cores, random), we see that this still does not solve the fundamental scaling issue; contention basically just moves from the lock to the cache line that contains the head pointer of the list.

To our surprise, LLFREE even outperforms (from 16 cores onwards) the ideal *ListLocal* variant, which is not even suited as a global frame allocator, but takes 36 percent more time on 26 cores for *bulk* allocations. This is caused by the state dispersion of linked lists: Virtually every allocation and free touches at least one new cache line – compared to LLFREE's cache-friendly structure, where in the best case, 512 allocations reuse the same three cache lines. In the *random* benchmarks, LLFREE also suffers from cache misses due to non-local frees (the worst case for LLFREE).

5.5 Allocator State Dispersion

To compare allocators' temporal and spatial costs, we propose the *state-dispersion* metric – a quantitative measure denoting the byte count utilized for metadata storage. Intuitively, state dispersion is the number of bytes accessed for a full enumeration of the internal state. However, it is critical to understand that state dispersion does not directly translate to memory overhead, given that allocators often repurpose the free memory or overload other shared data structures (i.e., `struct page`) for their metadata, whereby the plain memory overhead becomes a less meaningful metric. However, an allocator exhibiting high state dispersion will likely induce more cache misses during its operation, impacting run-time efficiency. This, of course, does not only depend on the size of the state but also its spatial distribution.

In Tab. 1, we break down the state dispersion of LLFREE and the Linux allocator into the different components. To put these numbers into perspective, we also show how the disper-

Allocator	Per Zone	Per CPU	Per 1 GiB	1 Zone 128 GiB 52CPUs	Full Scan: Accessed Cache Lines
LLFree				**4.1 MiB**	**67 754**
4 KiB Bit Fields			32.0 KiB	4.0 MiB	65 536
2 MiB Counters			1.0 KiB	128.0 KiB	2 048
64 MiB Trees		128 B	32 B	10.5 KiB	168
Global	128 B			128 B	2
Linux Buddy Allocator				**516.0 MiB**	**33 555 169**
Free Lists	988 B		4.0 MiB	512.0 MiB	33 554 448
Allocated Flag			32.0 KiB	4.0 MiB	(within above)
Pageblock Bits			256 B	32.0 KiB	512
Per-CPU Caches	8 B	256 B		13.0 KiB	209

Table 1: Allocator-State Dispersion and Cache Overhead

sion scales to our benchmark machine (52 cores, 128 GiB, 1 memory zone) and how many distinct 64B cache lines would be accessed for a complete enumeration in this setting.

Due to LLFree's usage of bit fields and counter arrays, its state disperses only to 4.14 MiB (0.0032 % of DRAM) on the benchmark machine. The primary contributor to this are the 4 KiB bit fields (4 MiB). Thus, even a full-state scan can comfortably fit within the machine's 35 MiB L3 cache, for which only 67 754 cache lines need to be loaded. Also, as LLFree does not repurpose memory, it does not require physical memory to be mapped by the kernel, and its state dispersion is equal to its memory overhead.

In contrast, the Linux allocator stores most of its state in `struct page`. There, it requires one flag and repurposes the 16 bytes of the LRU list pointers (double-linked list) for its per-order free lists and for the per-CPU page caches. Due to the scattered nature of linked lists, the Linux allocator (potentially) spreads its state over 516 MiB (0.39 % of DRAM). Even worse, as each `struct page` resides on its own cache line, a complete state scan would have loaded 33 555 169 cache lines. Hence, in comparison to LLFree, Linux's allocator state not only disperses over 125 times more memory, but even requires 495 times more cache lines to be loaded for the full scan.

Furthermore, as LLFree does not rely on the `struct page`, this also raises the question if they could be shrunk or eliminated. These records currently occupy 1.56 percent of DRAM. Unfortunately, removing the allocator's dependency on `struct page` does not directly result in smaller per-frame records, as other kernel subsystems reuse the LRU list pointers for various purposes when the frame is allocated (in the Linux source code, `struct page` is basically a mess of `unions`). Therefore, shrinking or even eliminating `struct page` is a deeply cross-cutting and challenging task.

Nevertheless, an allocator that, like LLFree, does not require a per-frame record significantly eases this challenge, since we then would only need per-frame records for *allocated* frames. For example, with Linux's current move to `struct folio` [11], which describes a bundle of physically contiguous frames, it could become possible to allocate this record dynamically. In this sense, we see LLFree as an important first step into untangling `struct page` dependencies. However, its complete elimination remains a topic of further research.

5.6 Fragmentation and Compaction Cost

Next, we look at the huge-frame fragmentation behavior of both allocators. For this, we first define a metric for this fragmentation and for the memory-compaction cost that would be required to remove this fragmentation. To measure fragmentation, we count the number of huge frames that can be allocated if we drain all caches but perform no memory compaction. We then compare this to the number of *possible* huge frames that could be allocated with compaction to get an idea of the fragmentation level in the system.

To gauge the compaction cost, we consider the minimal number of 4 KiB copy operations required to free up the *possible* maximum of huge frames. To calculate this metric, we (1) count the number of free 4 KiB frames in each possible huge frame, (2) sort the resulting array, and (3) match the "fullest" with the "emptiest" huge frames while counting the number of required copy operations. Please note that Linux skips step 1 and 2 and moves memory to the beginning of the zone, which results in suboptimal memory compaction. With LLFree, however, sorting the children array (see Fig. 3) would bring us close to the optimal variant.

To compare both allocators, we conduct the following synthetic benchmark: First, we generate an initial memory configuration that is a worst-case scenario for a maximally fragmented physical memory. For this, we allocate 90 percent of a 125 GiB region before freeing up half of all 4 KiB frames randomly again. Starting with this fragmented state, we perform 100 iterations, each freeing 10 percent of the allocated memory in the form of randomly-selected 4 KiB frames and reallocating the same amount again as individual 4 KiB frames. After each iteration, we drain the CPU-local caches (buddy), respectively the tree reservations (LLFree), and measure the huge-frame fragmentation and the compaction cost. Note that we still leave the Linux memory compactor turned off (as stated in Sec. 4.2.2). Like in our other benchmarks, we do not trigger unfulfillable allocation requests, which would require synchronous compaction. We measured the hypothetical compaction costs for each iteration for both allocators using the described offline calculation.

Fig. 10 shows the change of both metrics over time. The Linux allocator can recover only a single huge frame in this benchmark, although the whole memory was cycled ten times over (100 iterations). Also, the compaction cost decreased only by 3.3 percent, indicating that huge frames are only getting slowly defragmented over time. Additionally, our scenario benefits Linux as we drain the per-CPU caches and use the optimal compaction-cost metric. In contrast, LLFree can recover 46.6 percent of the initially polluted huge frames over

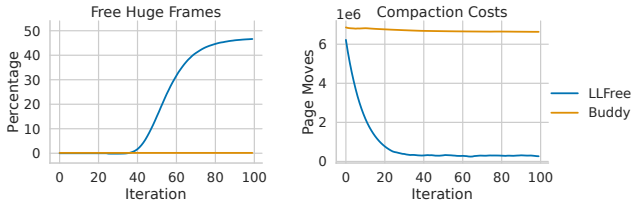

Figure 10: Free huge frames (left) and compaction cost (right) over iterations that randomly reallocate 10% of the allocated memory.

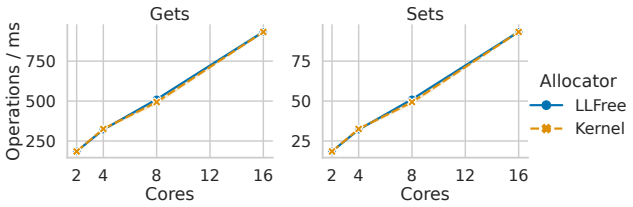

Figure 11: Number of Gets / Sets per millisecond for the *memtier* benchmark.

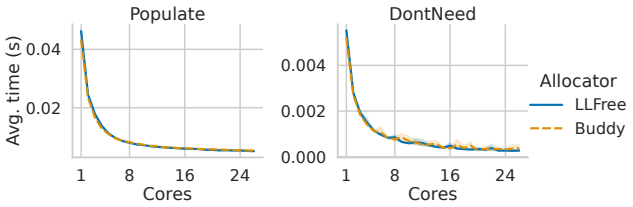

Figure 12: Average time to populate/free a 128 GiB memory map in the *write* benchmark.

the benchmark. Although it looks like defragmentation only starts to kick in around iteration 50, a look at the compaction cost indicates that entropy decreases right from the beginning; there are just no completely free huge-frame frames yet. After performing reallocations summing up to the total amount of memory (10 iterations), we are at 39.1 percent compaction cost and after 50 iterations, we would only require 4.9 percent of the initially-required copy operations.

Overall, we see that LLFREE shows a passive defragmentation behavior steered by our subtree-allocation policy. As a buddy system does not track the "fullness" of split-up buckets, it cannot imitate this on the cheap.

5.7 Crash Recovery

As validating crash consistency by real system crashes is too time-consuming to obtain robust results, we simulate LLFREE's recovery (Sec. 3.2.3) for regular shutdowns and crashes using a userspace benchmark on a DAX-mapped NVRAM region: For this, (1) we initialize the allocator on a 128 GiB region, (2) allocate half of it randomly, and (3) allocate and free memory repeatedly as in Sec. 5.6. For regular shutdowns, the process terminates after (2), while we simulate crashes by randomly killing the benchmark with `SIGKILL` during (3). Afterward, another process recovers the allocator's state from the persistent memory.

In total, we injected 1 000 crashes and LLFREE could recover its state in all cases; in about half of the experiments, we actually lost frames (at most one per core), which is expected, as the cores spent all their time in alloc/free. For the recovery procedure, LLFREE iterates through the bit fields to correct the child counters, which takes on average 2 460 µs. In comparison, a regular NVRAM re-initialization, where LLFREE only needs to iterate over the child tables, takes only 477 µs. Recovery was done single-threaded but could be parallelized and partially performed in the background if recovery times should become an issue.

5.8 Application-Level Benchmarks

As a real-world application benchmark, we used `memtier`[6], which evaluates the performance of the *memcached* key-value store. It measures the throughput of *Get* and *Set* requests. Unfortunately, we see no significant difference as shown in Fig. 11. As the page allocator is primarily used by the page-fault handler, which lazily allocates memory, the overhead of the other involved memory management components might overshadow any performance gains.

To investigate this hypothesis further, we created the *write* benchmark, which maps a large memory region and populates it in parallel. The population is done by writing a non-zero value into the first byte of the page, triggering a page fault and subsequent allocation request. For unmapping, the `madvice/DONTNEED` syscall is used. The benchmark is executed for 1–26 cores, with the memory region split evenly between the cores. Again, the results in Fig. 12 show no significant difference between the buddy and LLFREE allocators, just like the *memtier* benchmark.

Utilizing the *perf* profiler, we measured where most of the runtime is spent. The flame graph in Fig. 13 shows that the page allocator (yellow) only accounts for 5.3 percent of the runtime. Primarily dominant are the `struct mm` rw-lock (orange, 17.3 %) and the updates of the LRU, cgroups, reverse mappings, and `struct page` flags (green, 28.7 %). Because most `struct page` update macros are inlined, the actual time is probably higher. These memory-management bottlenecks are consistent with other research [7, 12, 14, 32, 35].

6 Discussion

Our results show (besides crash consistency) that LLFREE provides excellent scalability in user- and kernel-level benchmarks, achieved by its consequent lock-free and cache-friendly design. Nevertheless, these benefits are not yet visible in end-to-end benchmarks, even though there are applications

[6] https://github.com/RedisLabs/memtier_benchmark

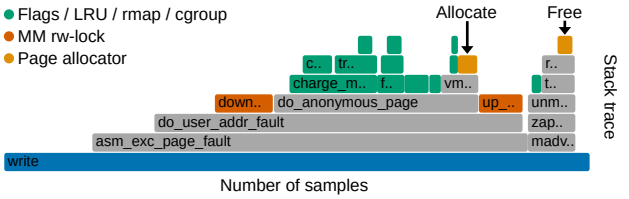

Figure 13: Flame graph for the write benchmark with the Buddy allocator on 8 cores and 128 GiB DRAM.

for which OS-level memory de/allocation performance already *is* a big issue [12, 14, 32, 35].

We argue that, given the deep entangling and grown complexity, the scalability problem could only be solved in a bottom-up manner and provide LLFREE and its design concepts as a first step in this direction. We believe if nonvolatile memories will play a role *some* day, their kernel-level management has to be designed together with volatile memory – and see LLFREE as an important step in this direction. Finally, our results also show that the conjunction of striving for scalability *and* persistence [8, 16, 46] works out particularly well in kernel design if considered from the very beginning.

7 Related Work

Many general-purpose allocators [3, 9, 33, 36, 41, 45, 52] for nonvolatile memory have been proposed. In contrast to LLFREE, all of them use logging to ensure crash consistency, which increases NVM wear [49], and locks for multithreaded operation; some reduce lock contention by using multiple allocator instances [36, 41], per-CPU/thread free lists [3, 9] or range locking [52]. From these, *PAllocator* [36] has similarities to LLFREE as it comes with antifragmentation measures and, similar to our lower level, stores only parts of its state in NVRAM and recovers its volatile state on boot. Nevertheless, all of these persistent allocators are general-purpose userspace allocators and, thus, have different design goals compared to a kernel page-frame allocator like LLFREE.

On the OS side, *Twizzler* [5] is explicitly built around nonvolatile memory but nevertheless does not contain a persistent page-frame allocator. Instead, the system rebuilds the allocator state from the persistent objects on each reboot in DRAM. To our knowledge, LLFREE is the first persistent page-frame allocator to be used within the operating system.

While the immunity to external fragmentation was one of the original motivations for paging [2], its extension to different frame sizes brought back the problem. To ease active huge-frame reclamation, placement strategies [19–21] categorize allocations (i.e., movable, reclaimable) and spatially cluster them onto separate huge frames. For this, Linux has multiple free lists per buddy order, each of which serves a different category. LLFREE currently does not support such categorization. However, it could easily be extended for this by specialized trees (see Sec. 3.2.2). For Linux, strategies with better clustering characteristics have been suggested [37].

Other measures to reduce huge-frame fragmentation include proactive compaction [30] and anticipated continuous memory reservation [27, 34]. Even hardware solutions have been proposed, such as building huge frames of noncontinuous memory [43], or an additional level in address translation [51] similar to nested paging [4]. In contrast, LLFREE is a pure software solution that passively defragments huge frames while being fast and crash-consistent at the same time.

8 Conclusion

The page-frame allocator, which manages the physical memory, is at the core of all memory management in modern operating systems. However, as we have shown in the example of Linux, its classical lock-based design with many lists and distributed metadata has not kept up with the progress in hardware towards massive-parallel systems with large amounts of heterogeneous volatile and nonvolatile memories. This results in internal complexity, poor scalability, high memory fragmentation, and general unfitness for achieving crash-consistent allocations on nonvolatile memories.

We presented LLFREE, a new log- and lock-free page-frame allocator that, by its lockless and cache-centric design, achieves excellent scalability for parallel allocations (53 times faster than the Linux buddy allocator for parallel 2 MiB DRAM allocations on 26 cores), while constructively keeping huge-page fragmentation low. All de/allocations manifest in memory by a one–cache-line transaction, whereby LLFREE can provide crash-consistency for persistent NVRAM without logging and at near-DRAM speed on eADR systems. Our integration of LLFREE into Linux was successful, but it also revealed many further bottlenecks of its memory-management subsystem and the deep entangling of the buddy allocator with it. These topics demand further investigation and redesign. We consider LLFREE a crucial first step towards a complete structural rethinking of the OS memory management.

Acknowledgments

We thank our anonymous reviewers for their helpful and constructive comments. Special thanks go to Godmar Back, whose demanding and encouraging shepherding has helped us tremendously improve this paper's content and quality.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.

References

[1] AKRAM, S. Exploiting Intel Optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2021), ISMM 2021, Association for Computing Machinery, p. 80–93.

[2] BENSOUSSAN, A., CLINGEN, C. T., AND DALEY, R. C. The Multics virtual memory. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles (SOSP '69)* (New York, NY, USA, 1969), ACM Press, pp. 30–42.

[3] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H. Makalu: fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), pp. 677–694.

[4] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, Association for Computing Machinery, p. 26–35.

[5] BITTMAN, D., ALVARO, P., MEHRA, P., LONG, D. D. E., AND MILLER, E. L. Twizzler: a data-centric OS for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)* (July 2020), USENIX Association, pp. 65–80.

[6] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *8th Symposium on Operating System Design and Implementation (OSDI '08)* (Berkeley, CA, USA, 2008), USENIX Association, pp. 43–57.

[7] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *9th Symposium on Operating System Design and Implementation (OSDI '10)* (Berkeley, CA, USA, 2010).

[8] CHEN, Z., HUA, Y., DING, B., AND ZUO, P. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 799–812.

[9] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News 39*, 1 (Mar. 2011), 105–118.

[10] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)* (New York, NY, USA, 2009), ACM, pp. 133–146.

[11] CORBET, J. Clarifying memory management with page folios, 05 2023.

[12] CROTTY, A., LEIS, V., AND PAVLO, A. Are you sure you want to use mmap in your database management system? In *CIDR 2022, Conference on Innovative Data Systems Research* (2022).

[13] DAVID, T., DRAGOJEVIĆ, A., GUERRAOUI, R., AND ZABLOTCHI, I. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 373–386.

[14] DURNER, D., LEIS, V., AND NEUMANN, T. Experimental study of memory allocation for high-performance query processing. In *International Conference on Very Large Databases (VLDB)* (2019), pp. 1–9.

[15] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (New York, NY, USA, Dec. 1995), ACM Press, pp. 251–266.

[16] FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PETRANK, E. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, Association for Computing Machinery, pp. 28–40.

[17] FU, X., KIM, W.-H., SHREEPATHI, A. P., ISMAIL, M., WADKAR, S., LEE, D., AND MIN, C. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 100–115.

[18] GORMAN, M. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[19] GORMAN, M. The performance and behaviour of the anti-fragmentation related patches. Linux Kernel Mailing List, Mar. 2007. https://lkml.org/lkml/2007/3/1/92.

[20] GORMAN, M., AND HEALY, P. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management - ISMM '08* (Tucson, AZ, USA, 2008), ACM Press, p. 41.

[21] GORMAN, M., AND WHITCROFT, A. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, Jul 2006), vol. Volume 1, p. 370–384.

[22] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., SCHÖNBERG, S., AND WOLTER, J. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (New York, NY, USA, Oct. 1997), ACM Press.

[23] HAYOT-SASSON, V., BROWN, S. T., AND GLATARD, T. Performance benefits of intel optane dc persistent memory for the parallel processing of large neuroimaging data. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* (2019), 509–518.

[24] INTEL. eADR: new opportunities for persistent memory applications, 2021. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, visited 2021-10-04.

[25] INTEL. 2022 Q2 – 10-Q earnings report, 2022. https://www.intc.com/filings-reports/all-sec-filings/content/0000050863-22-000030/0000050863-22-000030.pdf.

[26] IZRAELEVITZ, J., MENDES, H., AND SCOTT, M. L. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing* (2016), Springer, pp. 313–327.

[27] KIM, S.-H., KWON, S., KIM, J.-S., AND JEONG, J. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the 2015 International Symposium on Memory Management* (New York, NY, USA, 2015), ISMM '15, Association for Computing Machinery, p. 1–14.

[28] KNOWLTON, K. C. A fast storage allocator. *Communications of the ACM 8*, 10 (1965), 623–624.

[29] KORGAONKAR, K., IZRAELEVITZ, J., ZHAO, J., AND SWANSON, S. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 435–444.

[30] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and efficient huge page management with ingens. In *12th Symposium on Operating Systems Design and Implementation (OSDI '16)* (USA, 2016), USENIX Association, p. 705–721.

[31] LEE, S., KWON, M., PARK, G., AND JUNG, M. Lightpc: Hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 289–305.

[32] LEIS, V., ALHOMSSI, A., ZIEGLER, T., LOECK, Y., AND DIETRICH, C. Virtual-memory assisted buffer management. In *Proceedings of the ACM SIGMOD-/PODS International Conference on Management of Data (SIGMOD'23)* (New York, NY, USA, June 2023), ACM.

[33] MEMARIPOUR, A. S., IZRAELEVITZ, J., AND SWANSON, S. Pronto: Easy and fast persistence for volatile data structures. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), pp. 789–806.

[34] NAVARRO, J., IYER, S., AND COX, A. Practical, transparent operating system support for superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)* (Boston, MA, Dec. 2002), USENIX Association.

[35] NEUMANN, T., AND FREITAG, M. J. Umbra: A disk-based system with in-memory performance. In *Conference on Innovative Data Systems Research (CIDR)* (2020).

[36] OUKID, I., BOOSS, D., LESPINASSE, A., LEHNER, W., WILLHALM, T., AND GOMES, G. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow. 10*, 11 (aug 2017), 1166–1177.

[37] PANWAR, A., PRASAD, A., AND GOPINATH, K. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 679–692.

[38] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)* (2014), IEEE Press, p. 265–276.

[39] PMDK TEAM, I. C. Intel Persistent Memory Develpment Kit (PMDK). https://pmem.io/pmdk, visited 2021-10-05.

[40] RAO, D. S., KUMAR, S., KESHAVAMURTHY, A. S., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth ACM European Conference on Computer Systems (EuroSys '14)* (2014), pp. 15:1–15:15.

[41] SCHWALB, D., BERNING, T., FAUST, M., DRESELER, M., AND PLATTNER, H. nvm malloc: Memory allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015* (2015), R. Bordawekar, T. Lahiri, B. Gedik, and C. A. Lang, Eds., pp. 61–72.

[42] SINGH, A. *Mac OS X Internals: A Systems Approach: A Systems Approach*. Addison Wesley, 2016.

[43] SWANSON, M., STOLLER, L., AND CARTER, J. Increasing TLB reach using superpages backed by shadow memory. *SIGARCH Comput. Archit. News 26*, 3 (apr 1998), 204–213.

[44] TREIBER, R. K. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.

[45] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. *ACM SIGARCH Computer Architecture News 39*, 1 (Mar. 2011), 91–104.

[46] WANG, T., LEVANDOSKI, J., AND LARSON, P.-A. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 461–472.

[47] WENTZLAFF, D., AND AGARWAL, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43* (Apr. 2009), 76–85.

[48] XU, J., AND SWANSON, S. 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, 2016), USENIX Association, pp. 323–338.

[49] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. 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, Feb. 2020), USENIX Association, pp. 169–182.

[50] YOSIFOVICH, P., RUSSINOVICH, M., SOLOMON, D. A., AND IONESCU, A. *Windows Internals, Part 1 (7th Edition)*. Microsoft Press, 2017.

[51] ZHANG, L., SPEIGHT, E., RAJAMONY, R., AND LIN, J. Enigma: Architectural and operating system support for reducing the impact of address translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)* (New York, NY, USA, 2010), ICS '10, Association for Computing Machinery, p. 159–168.

[52] ZHANG, L., AND SWANSON, S. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 897–912.

Appendix

(1) If reserved counter $c_P \geq 2^o$ then $c_P \leftarrow c_P - 2^o$ and continue with (2).
  (a) Otherwise, sync with global c_U and repeat (1) if the counter is now large enough.
  (b) Otherwise, reserve a new tree and repeat (1).
(2) Search the corresponding children array sequentially for an entry with $c_L \geq 2^o$. If this search fails, reserve a new subtree and repeat (1).
  (a) For base frames, decrement c_L, search the corresponding bit field for a zero bit, and set it.
  (b) For huge frames set $c_L = 0$ and $a = 1$.
(3) Return the allocated *page-frame number (PFN)* or `NULL`.

(a) The allocation of an order o frame.

(1) Check the corresponding child entry.
  (a) For base frames, check if $c_L \leq 512 - 2^o$ and $a = 0$, and continue with (2).
  (b) For huge frames, check for $a = 1$, set it to zero, and continue with (4).
(2) Toggle the corresponding bits in the layer-one bit field.
(3) Increment the child counter ($c_L \leftarrow c_L + 2^o$).
(4) Increment the reserved c_P or the global c_U if this free is in another tree.
  (a) When a global, *partial* tree entry is updated, reserve it if the past F allocations also affected it.

(b) The free operation of an order o frame.

Figure 14: The allocation and free algorithms. This description does *not* include all edge cases and error paths.

A Artifact Appendix

Abstract

The artifact contains the necessary tools and resources required to evaluate LLFree, a new lock- and log-free allocator design that scales well, has a small memory footprint, and is readily applicable to non-volatile memory. To simplify the evaluation, the artifact is packaged as a Docker image, which includes the different benchmarks from the paper's evaluation and any dependencies. These benchmarks are designed to stress the allocator in various scenarios. They allow other researchers to compare the performance of LLFree with the traditional Buddy allocator and reproduce our experimental results. Additionally, this image contains the raw data and scripts for the paper's figures, making our evaluation as transparent as possible.

Scope

These benchmarks show that the LLFree allocator out scales the buddy allocator on systems and workloads with high parallelism. However, executing them in a virtual machine (even with KVM) leads to less accurate results. Therefore, in the paper, we built and tested the modified Linux on raw hardware. Nonetheless, the results should show similar trends as in the paper's evaluation.

Contents

The Docker image includes all required dependencies and scripts for building and running the benchmarks, as well as generating relevant plots and data. It also features a Python script (run.py) that serves as the central command center to manage the building, benchmarking, and plotting processes. The allocator can be tested in both user space and within a custom-built Linux kernel that incorporates LLFree executed in a QEMU+KVM vm. For the latter, the image contains a QEMU+KVM virtual machine and scripts to boot it and run the kernel benchmarks. It further contains the raw data and plots from the benchmarks shown in the paper.

Hosting

The docker image and the repositories of the allocator, the modified Linux kernel, and the benchmarks are hosted on GitHub:

- **Docker Image**: This image contains an execution environment that makes it easy to run and evaluate the benchmarks.
- **llfree-bench**: The benchmark scripts and results.
  (tag = `atc23-artifact-eval`)
  - The artifact instructions can be found in `artifact-eval/README.md`.

- **llfree-rs**: The Rust implementation of the LLFree allocator.
  (tag = `atc23-artifact-eval`)
- **llfree-linux**: The modified Linux Kernel that can be configured to use LLFree instead of the Buddy allocator.
  (tag = `atc23-artifact-eval`)
- **linux-alloc-bench**: Kernel module for benchmarking the page allocator.
  (tag = `atc23-artifact-eval`)

Requirements

As our benchmarks are packaged in a Docker image and do not rely on specific hardware, the only prerequisites are:

- A Linux-based system for KVM. We have tested this on Linux 6.0, 6.1, and 6.2.
- At least 8 physical cores and 32GB RAM (more is better). Lower specifications should work, but the results may be less meaningful.
- Hyperthreading and TurboBoost should be disabled for more stable results. As the VM is not configured for this, the kernel benchmarks might be especially affected.
- A properly installed and running Docker daemon.

Next Steps

The artifact instructions can be found in the *llfree-bench* repository under artifact-eval/README.md. The *artifact-eval* directory also contains all the scripts mentioned in the document and also the Dockerfile for building the image by oneself if desired.