Lost Bytes At The Crossroads Between User- And Kernel-Level Memory Management

Pasha Fistanto
fistanto@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Sören Tempel
tempel@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Christian Dietrich
dietrich@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Abstract

With soaring DRAM prices, optimizing software for efficient memory use is becoming increasingly important. Essential to this end is the memory management component of the software's language runtime. This component requests page-sized memory from the kernel and subdivides it into smaller blocks to satisfy allocations performed by the software. Since DRAM was comparatively cheap for the preceding decade, contemporary prior work has focused on optimizing this component for performance. A key optimization is conducted on the free path where memory freed by the software is not directly returned to the kernel to satisfy future allocations and avoid costly context switches.

In this work-in-progress paper, we illustrate the lost potential of user-level allocators for system-wide memory use. We show that—since the kernel manages memory at page-granularity—the user-level allocation scheme must be designed with page alignment in mind. We provide a classification for physical memory that is lost due to the mismatch of user-space and kernel-space allocation granularities. To empirically illustrate the potential of such an allocation scheme, we conduct experiments with the `mallocng` allocator where we identify potential savings of up to 75 percent.

Keywords: Operating System, Memory Management

ACM Reference Format:
Pasha Fistanto, Sören Tempel, and Christian Dietrich. 2026. Lost Bytes At The Crossroads Between User- And Kernel-Level Memory Management. In *14th Workshop on Programming Languages and Operating Systems (PLOS '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3831586.3838144

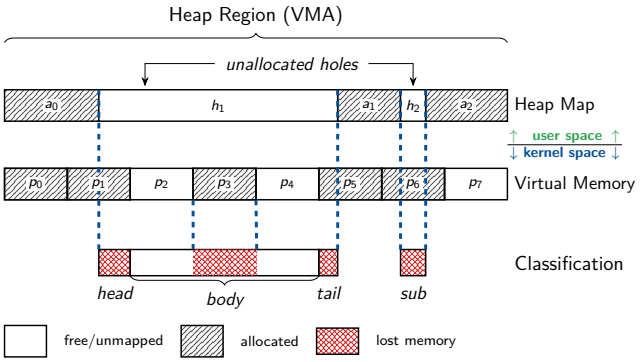


Figure 1. Userspace Heap Allocators manage "holes" of unallocated memory blocks that were already requested from the OS but not yet handed to the application. These holes not necessarily align with the page-grid of the virtual memory, resulting in memory lost for the operating system.

1 Introduction

Since late 2025, prices for DRAM have grown significantly due to an increased demand for AI data centers [10]. Consequentially, there has been a renewed interest in optimizing software to make efficient use of scarce memory resources.

The central component influencing memory consumption is the software's language runtime, which is responsible for memory management (e.g., through garbage collection [2]). For the preceding decade, conventional wisdom was that memory is cheap. Therefore, recent works on memory allocation techniques have predominantly focused on optimizing memory management for performance [12, 20, 16, 27].

Conceptually, memory management operates at the boundary between user- and kernel-space [8] (see Fig. 1). The application requests memory from the language runtime (e.g., via `malloc()`), which obtains it at page granularity from the kernel, divides it into smaller or larger chunks, and returns right-sized *blocks*. When the application returns memory via `free()`, the runtime does not commonly return it directly to the kernel. Either the memory is not returnable (e.g., due to some in-use blocks on the same page) or the allocator does not return memory deliberately as an optimization to satisfy future allocations. In both cases, from the OS point of view, this unallocated memory is *"lost"* as it cannot be made available to another process.

PLOS '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2877-8/2026/09
https://doi.org/10.1145/3831586.3838144

For system-wide memory efficiency, it is vital that such unused memory be returned—as soon as possible—to the kernel. To strike a balance between memory efficiency and performance, some runtimes support the explicit release of heap memory on pressure [21]. For instance, the GNU C library provides `malloc_trim()` to aggressively return memory to the OS. We believe that the responsibility to decide when memory is to be trimmed should not be forced upon applications since fundamentally, this task falls under the domain of the operating system. And application developers seem to agree: only 120 of the ≈ 42 000 Debian source packages call `malloc_trim()`.

But even when attempts are made to return memory, the mismatch between heap blocks and OS pages makes this challenging. For example, TCMalloc [27] reduces management overheads by rounding allocations to static size classes. Size class 22, for instance, covers all objects between 1153 and 1408 B. Using such "uneven" sizes reduces internal fragmentation and can be considered a standard technique [27, 4, 6]. But as a result, every third 1408-byte block crosses a 4 KiB page boundary, making it harder for the OS to reuse unallocated areas.

In this work-in-progress paper, we argue that the memory-management component of language runtimes should be optimized for memory efficiency from day one and that doing so requires considering the available kernel interface. For this purpose, we quantify the lost memory that we could free today and identify potentials for a tighter allocator–OS co-design. For this, we conduct experiments with the `mallocng` [6] memory allocator that illustrate potential memory savings of up to 75 percent. In future work, we want to build upon these insights to design a memory allocator that is optimized for efficiently returning memory to the kernel.

Outlook. In Sec. 2, we discuss the interaction between the application, language runtime, and kernel. We then expand on the runtime and kernel, classifying lost bytes occurring at this boundary (Sec. 3). According to our classification, we perform experiments to empirically assess the amount of bytes lost in the `mallocng` (Sec. 4). Finally, we discuss the future (Sec. 5) and related work (Sec. 6) before concluding this work-in-progress paper (Sec. 7).

2 Background

User-space allocators are part of the language runtime and sit between the application and the *operating system (OS)*. On a memory request (e.g., via `malloc()`), the allocator provides a block whose size matches or exceeds the request in size. Although it could forward every request to the OS, the allocator maintains a cache of free memory blocks, which we will call *holes*. It hands out memory from this cache according to its *allocation strategy* and requests more memory from the OS when the cache runs dry. Whether and when memory is returned to the OS is highly allocator-specific but can only be done at the granularity of whole pages.

On POSIX systems, the allocator can use system calls at two different granularities to manage the *virtual address space (VAS)*. At the coarser level, it creates and shrinks anonymous *virtual-memory areas (VMAs)* using the `mmap()`/`munmap()` system calls. VMAs [23] are OS-internal objects used to structure and manage a process's VAS. VMAs are aligned and sized according to the page size of the CPU's memory management unit and are usually populated lazily with physical-memory frames. As `mmap()`/`munmap()` change the VAS structure, they require extensive locking and synchronization and are therefore considered a bottleneck of current virtual-memory subsystems [15, 13].

On the fine-grained level, POSIX processes can invoke `madvise()` with the `WILLNEED`/`DONTNEED` flags to request or remove page-aligned subranges from an existing VMA. The `madvise()` system call then leaves the VMA untouched but only changes the physical-memory mapping. When applied to anonymous mappings, `madvise(DONTNEED)` has to remove the pages immediately, triggering a costly TLB shootdown [15]. Hence, Linux introduced the non-standard `MADV_FREE` flag with 4.16 that has a relaxed semantic and allows for lazy and batched page removal.

Despite these possibilities, performance-oriented allocators often hoard memory even when parts of a VMA could be madvise'd. For example, jemalloc [4] waits for a decay timeout (default 10 seconds) before a background thread calls `madvise()`. Similarly, without calling `malloc_trim()`, the GNU C library's allocator does not return holes that are surrounded by allocated blocks. Currently, eagerly returning individual pages to the OS is considered too expensive for the hot allocation path.

3 Classification of Lost Bytes

To better understand the mismatch between OS and allocator granularity, we want to examine the unallocated heap blocks (*holes*) and how they intersect with the page-size grid of virtual memory (see Fig. 1). This intersection between both granularities allows us to classify the bytes in each hole: each hole is either a sub-page hole that does not cross a page boundary or a large hole, which can be subdivided into a *head*, *body*, and *tail*.

When P is the constant page size, v is a virtual address, H is the set of unallocated holes, and $h^i = [v_s^i, v_e^i)$ is a single hole, we use the following page-aligned floor/ceiling helpers to define our classification:

$$\lceil v \rceil_P = \lceil v/P \rceil \cdot P \qquad\qquad \lfloor v \rfloor_P = \lfloor v/P \rfloor \cdot P$$

If the hole is within a single physical page, $\lfloor v_s^i \rfloor_P = \lfloor v_e^i \rfloor_P$ and $v_s^i \neq \lfloor v_s^i \rfloor_P$, the entire hole is classified as a *sub-page hole*:

$$\mathrm{sub}(h_i) = [v_s^i, v_e^i)$$

If the hole crosses at least one page boundary, it is a *large hole* that we decompose into three segments:

$$\begin{aligned} \text{head}(h_i) &= [v_s^i, \quad \lceil v_s^i \rceil_P) \\ \text{body}(h_i) &= [\lceil v_s^i \rceil_P, \ \lfloor v_e^i \rfloor_P) \\ \text{tail}(h_i) &= [\lfloor v_e^i \rfloor_P, \ v_e^i \quad) \end{aligned}$$

Please note that $\text{head}(h_i)$ and $\text{tail}(h_i)$ are empty if the start address, respectively the end address, is page aligned. If the hole spans only across two pages, $\text{body}(h_i)$ is empty.

In order to calculate the physical memory that is lost, we query the operating system for the physical frames that are mapped in the heap regions. This can be done efficiently with the `mincore()` system call or via the `pagemap` pseudo-file in the `proc` file system. As a result, we get the set of mapped addresses M, which we can intersect with our classification to get a statistic about the lost memory:

$$M \cap \left(\bigcup_{h_i \in H} \text{sub}(h_i) \cup \text{head}(h_i) \cup \text{body}(h_i) \cup \text{tail}(h_i) \right)$$

In Fig. 1, the hole h_2 is a sub-page hole that is surrounded by live allocations. As page p_6 is backed by physical memory, the bytes of h_2 are entirely lost for the OS. In comparison, h_1 is a large hole that stretches across four boundaries and five pages. The *head* and *tail* parts of h_1 are located on allocated pages and thus cannot be returned. The *body* covers three entire pages, which in principle could all be returned to the OS. In the example, however, a single page p_3 is still backed by physical memory. Please note that the allocator can usually not return bytes in the *head*, *tail*, or the *sub* class to the kernel, but only the *body* class is directly reusable.

The base of our classification is the set of holes H that we extract from the allocator. However, many allocators do not eagerly merge consecutive holes that are adjacent to each other. For example, `mallocng` [6] considers two holes in an area as two separate *slots* that are not considered for slot-spanning operations. Hence, we perform our classification on the original hole list H^{alloc} and on the minimal hole list H^{min}, where H^{min} is defined such that $\nexists v_e^i = v_s^j$ with $h^i, h^j \in H^{\text{min}}$.

3.1 Allocator Integration

We integrated our classification into `mallocng`, which is the standard allocator of musl libc, a Linux-based C library that is especially popular for embedded systems.[1] `mallocng` is a slab allocator that separates allocations into fixed-size classes. Due to its focus on embedded Linux, it has a configuration option to make use of `madvise(FREE)` to release page-sized holes to the kernel (see Sec. 2). Since the allocator is tailored to memory efficiency, we use it as the object of our empirical evaluation in Sec. 4 and enable the `madvise` option.

[1] https://musl.libc.org/

To employ our classification with `mallocng`, we compile the allocator as a separate dynamic library that we `LD_PRELOAD` with our benchmark programs. At load time, we spawn a separate sampling thread that periodically inspects the process state. It extracts the original hole list H^{alloc} from `mallocng`'s internal data structures. Further, the sampling thread invokes `mincore()` to determine the actually allocated physical memory frames.

From H^{alloc}, we derive the minimal hole list H^{min} by merging adjacent holes. Adjacent holes result from `mallocng`'s design as a slab-like allocator, which treats larger regions as up to 32 adjacent but independent *slots* of homogeneous size. For efficiently calculating H^{min}, we insert the elements of H^{alloc} in a sorted red-black tree, where all core operations are $O(\log N)$. When inserting a hole, we find the closest adjacent holes before and after and merge it in case.

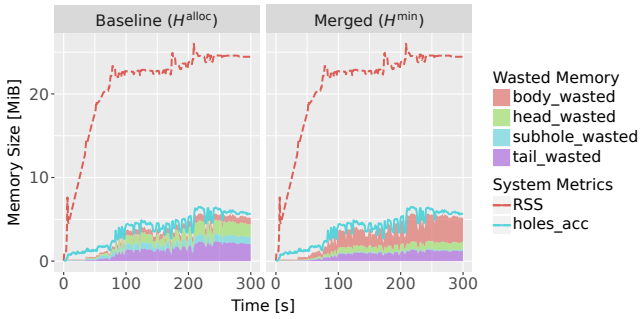
4 Evaluation

With our `mallocng` allocator integration (Sec. 3.1), we now empirically apply our classification of lost bytes (Sec. 3) to real-world benchmarks. Our evaluation focuses on the categorization of virtual and physical bytes across varying workloads where objects of diverse lifetimes are continuously created and destroyed. We compare the existing `mallocng` strategy (H^{alloc}) and what happens when we virtually merge adjacent holes (H^{min}, see Sec. 3.1) to form larger returnable regions.

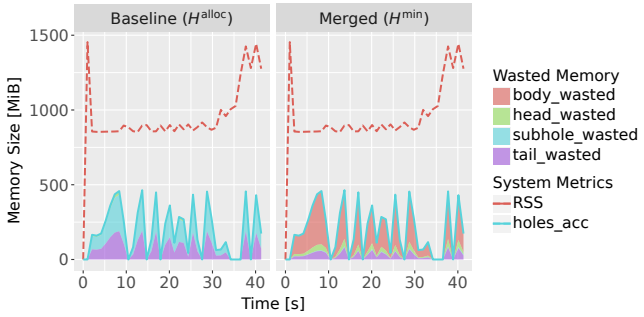
Benchmark Scenario. For an in-depth investigation, we evaluate `mallocng` using Redis (an in-memory key-value store) and LuaJIT (a JIT compiler for the Lua programming language) to cover different use cases. Redis and LuaJIT have been configured to make use of our instrumented `mallocng` allocator. This enables us to assess lost bytes for conceptually different applications: a popular key-value store and a garbage-collected programming language. In Sec. 4.4, we will present more results for the mimalloc-benchmark suite [14].

For Redis, we generate the database workload using the `memtier_benchmark` tool, provided by Redis developers. Specifically, `memtier_benchmark` is configured to use 800 concurrent connections distributed over 16 threads. The total benchmark runtime is set to 300 s. Within this timeframe, the benchmark spawns multiple Redis clients that perform SET and GET queries, with a SET:GET ratio of 1:5. The objects generated by the clients have a *time-to-live (TTL)* of 30 s to 60 s; thus, the objects are guaranteed to be freed during the benchmark's runtime. The size of generated objects ranges from 2048 B to 8192 B.

LuaJIT runs a simplified adaptation of Hans Boehm's GCBench (Garbage Collection Benchmark) [19], which allocates small and short-lived memory objects during its runtime. Specifically, this workload creates multiple short-lived and one long-lived binary trees. We configured the benchmark to generate complete binary trees with a depth of 21.



(a) Redis benchmark.



(b) LuaJIT benchmark.

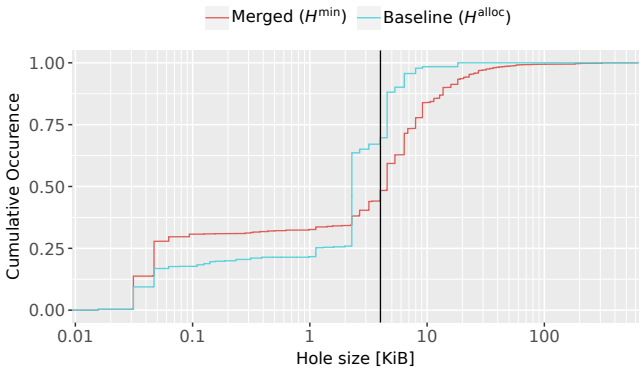
Figure 2. Classification results over individual benchmark runs. The graph shows RSS, the extend of the allocator's hole list in virtual memory (holes_acc), and the wasted physical memory in the different classes.

Metric	Redis		LuaJIT	
	H^{alloc}	H^{min}	H^{alloc}	H^{min}
Head Size (MiB)	0.97 ± 0.025	0.55 ± 0.003	13.32 ± 1.330	22.8 ± 1.999
Body Size (MiB)	0.83 ± 0.024	2.38 ± 0.090	0 ± 0.000	181.04 ± 19.250
Tail Size (MiB)	1.41 ± 0.038	0.79 ± 0.011	99.38 ± 9.936	33.9 ± 2.707
Subhole (MiB)	0.59 ± 0.024	0.09 ± 0.003	127.81 ± 12.761	2.82 ± 0.267
RSS (MiB)	20.77 ± 0.136		925.82 ± 13.764	

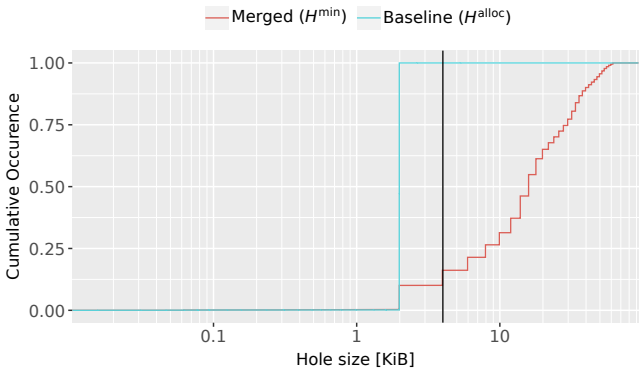
Table 1. Average Hole Size (mean ± std)

Benchmark Setup. To quantify lost bytes in `mallocng`, we invoke the benchmarks and categorize the lost bytes, the total size of all holes in virtual memory, and the allocator's *resident set size (RSS)* (physical memory). For each sample, we accumulate the lost bytes in each category. The sample rate is set to 1 Hz for the Redis and LuaJIT benchmark. By looking at H^{alloc}, we cover the behavior of `mallocng` in its current state. With the categorization over H^{min}, we determine the upper bound opportunity of what `mallocng` *could* achieve with a paging-aware memory-return policy.

All experiments have been performed on a Linux system with an Intel Ultra 9 processor and 94 GiB of DRAM.



(a) Redis benchmark



(b) LuaJIT benchmark

Figure 3. The CDF of the hole sizes. The vertical line marks a page size 4 KiB.

4.1 Baseline Classification of `mallocng` (H^{alloc})

First, we will look at the *existing* `mallocng` implementation and its H^{alloc} hole list. To which degree are the different categories unnecessarily backed by physical memory? For this, Fig. 2a and Fig. 2b (left facets) depict the results over the runtime of one benchmark run of Redis and LuaJIT.

For both benchmarks, `mallocng` manages on average around 20 percent (Redis: 20 %, LuaJit: 25 %) of its RSS as holes. Most of those holes are backed by physical memory (Redis: 82 %, LuaJit: 100 %). Only for Redis, `mallocng` returns physical memory to the OS without unmapping the VMAs. If we integrate over the gap between virtual and physical memory, `mallocng` returns 195.58 MiB · *seconds* in its holes, while 1 070.45 MiB · *seconds* are wasted.

We can explain this by looking at the classification: Redis can return memory to the OS via `madvise()` as 20 percent of its holes are *body* pages. But still, `mallocng` falls short of its potential as 60 percent of *body* pages remain backed by memory as the used `madvise(FREE)` does not eagerly return memory. In contrast, for LuaJIT, `mallocng` has no *body* pages, but 53 percent of the holes are subpage holes.

Our results illustrate the limitations of `mallocng`'s approach of returning memory to the OS. As `mallocng` makes no effort to form page-sized holes, it oftentimes cannot return unused memory to the kernel. Consequentially, it neglects small holes, which constitute the entirety of the LuaJIT workload, and is thus not memory efficient for such workloads.

4.2 Hole Merging of Baseline `mallocng`

In order to return as much memory as possible to the kernel, the *body* size would have to be maximized. Currently, even for the Redis benchmark, the largest class with 38 percent is the *tail* class (see Fig. 2a, left side). Hence, we want to examine to which extent `mallocng` keeps adjacent holes separate and falls short of its potential.

As it is, `mallocng` already performs some adjacent hole merging for small holes. This becomes apparent when considering the hole-size distribution for LuaJIT in Fig. 3b. There, the majority of holes in H^{alloc} have a size of 2032 B (size class 23). However, in the benchmark code itself [19] and the JIT implementation, no allocations of this size are performed. Instead, `mallocng` merges multiple smaller holes into larger holes of the aforementioned size. Unfortunately, holes of 2032 B do not align with page boundaries (4096 B in our setup), which leads to unreturnable memory in *head* and *tail*. Therefore, the underlying memory cannot be returned to the kernel and is thus lost at the crossroads.

In summary, `mallocng` performs some merging of adjacent holes, but it is not optimized for creating page-sized holes. Consequentially, `mallocng` loses memory through the unreturnable *head*, *tail*, and *sub* classes.

4.3 Potential Memory Savings (H^{min})

In the following, we demonstrate the potential memory savings that an allocator could achieve if it would eagerly merge all adjacent holes. For this, we repeat our classification after *virtually* merging the original hole set H^{alloc} to get H^{min}.

In Fig. 3a and Fig. 3b, we see how collapsing adjacent holes substantially shifts the hole-size distribution towards larger holes. Without merging, LuaJIT's distribution consists only of holes smaller than a page, while for Redis, 30.8 percent of holes are page size or larger (see Fig. 3a and Fig. 3b). With H^{min}, the distribution shifts to 84 percent for LuaJIT and 52 percent for Redis.

This increase influences not only the frequency of large holes but also their average size. Averaged over 5 runs, the *body* size increases for Redis by a factor of 2.85x (see Tab. 1). For LuaJIT, the *body* size goes from zero to 181 MiB. As bodies are always page-aligned, these areas could be returned.

Treating adjacent holes as a unit implicitly increases the number of pages that can be returned to the kernel. Therefore, the *body* class grows significantly, and for Redis it is 63 percent and 75 percent for LuaJIT.

	RSS	$\sum H^{\text{min}}$	Lost Phys. Mem [% of $\sum H^{\text{min}}$]			
Benchmark	[MiB]	[MiB]	Sub	Head	Body	Tail
mimallloc-bench: Real-World Applications						
barnes	40.21	0.002	100	0	0	0
cfrac	0.43	0.027	21.5	9.4	0.4	1.7
espresso	0.40	0.191	9.2	7.8	31.5	4.5
gs	7.45	0.240	12.3	5.4	4.9	4.1
larsonN-sized	33.36	1.255	81.3	3.2	0	8.8
z3	122.24	0.600	37.8	5.8	15.7	4.7
mimallloc-bench: Allocator Benchmarks						
cache-scratchN	0.02	0.004	100	0	0	0
glibc-simple	0.05	0.006	72.8	3.1	0	1.5
glibc-thread	0.81	0.216	56.2	10.3	5.1	13.4
mstressN	77.87	5.091	15.9	6.6	45.6	8.1
rptestN	13.81	1.759	10.5	12.9	33.9	24.4
rbstress1	15.93	0.638	27.4	10.5	13.3	10.7
rbstressN	14.14	0.963	17.7	6.7	29.1	7.3
sh6benchN	109.67	0.241	47.1	10.5	10.3	17.3
sh8benchN	119.12	9.978	65.2	5.8	5.9	11.5
xmalloc-testN	29.56	8.070	9.5	9.3	61.8	15.5
Custom Real-World Application Benchmarks						
LuaJIT	932.90	235.941	1.2	9.5	75	14.3
Redis	20.50	4.002	2.1	14.2	46.2	19.6
CLang SQLite3	102.43	22.959	74.3	4	11.2	6.6

Table 2. Lost Byte Statistic for MiMalloc Benchmark Suite (H^{min}). Average over the benchmark, sampled at 10 Hz.

4.4 MiMalloc Benchmark Suite

After this in-depth discussion of LuaJit and Redis, we want to apply our classification to a larger set of benchmarks. For this, we selected the MiMalloc benchmark suite [14], which consists of real-world applications and hand-crafted allocator benchmarks. We excluded three benchmarks (Redis, Lua, and Lean) for redundancy and for using multiple processes. Instead, we added a compilation of the amalgamated SQLite3 database (clang-21, -O2) as a larger compilation job. The results of our measurement are shown in Table 2.

Since the MiMalloc suite is performance-oriented, it contains many short running test cases, and we cannot expect a large hole/RSS ratio (see Tab. 2). Still, the distribution of the different categories within the existing holes is characteristic of the lost-memory problem. While the table shows the results for H^{min}, merging continues to reduce sub-page holes significantly. With the transition to H^{min}, the geometric mean over the subpage column decreases by −38 percent, and the *body* column increases by +229 percent.

The last four columns of Table 2 show the lost bytes in percentage of $\sum H^{\text{min}}$. Please note that these percentages do not have to add up to 100 percent as physical memory could have been returned. We see that the subpage and *body* class are the most important reason for lost bytes.

While the amount of subpage holes can only be reduced by better allocation strategies, memory in *body* regions could be directly returned. As `mallocng` does not prioritize this,

we lose 31 percent for espresso, 62 percent for xmalloc-testN, and 75 percent for LuaJIT of the accumulated hole size. Especially for longer running processes (Redis) and processes with varying memory demand (LuaJIT/xmalloc-testN), the easily returnable *body* pages are a significant factor.

We conclude that losing physical-memory bytes from *body* and subpage holes is a common problem that should be addressed by future allocators.

5 Future Work

As DRAM prices keep rising, designing heap allocators that return more memory to the OS is a promising research direction. Based on our insights, we see at least four important learnings from our work for future work.

Block Alignment Allocators intentionally use unaligned size classes to reduce fragmentation, at the cost of returnability. In Sec. 4, we have seen that `mallocng` loses memory through the resulting unreturnable *head* and *tail* areas. We argue that allocators should size and place their blocks to better match page-alignment requirements.

Page-level Tracking Slab-based allocators use bit fields to track unallocated slots in an area. From the discrepancy between H^{alloc} and H^{min}, we could quantify that `mallocng` loses significant amounts of *body* bytes by only considering large slots for `madvise()` operations. We argue that they should also track enough information to decide which pages in the area are not occupied by allocated slots.

Cheap OS Hints As `madvise()` is a system call, it will never become cheap enough to be part of the hot path of a performance-oriented allocator. This is also the reason why using `madvise()` in `mallocng` is optional [5]. We argue that a cheaper OS interface is needed to inform the OS about pages that are currently not allocated by the user-space allocator.

Sources of Memory Bloat Keeping memory and results that are not strictly necessary around is a common pattern to avoid OS, allocator, or recalculation overheads; caching speeds up programs. However, LRU result caches, lists of half-initialized objects, and custom allocators are all potential sources of memory bloat. We argue that we need a holistic view on the impact of those patterns in memory-intensive applications; looking at the allocator is not enough.

Paging-Aware Memory-Return Policy In this work we introduced H^{min} to find the upper-bound of returnability. However, we disregard the performance of the H^{min} calculation, as we only use it for evaluation purposes. To create a paging-aware memory-return policy based on H^{min}, the runtime overhead of the bookkeeping method must also be considered and optimized in future work.

6 Related Work

Memory allocation has been an active research area for many decades [26]. As such, a large variety of different allocation schemes with varying optimization goals have been proposed in the existing literature. To this end, an established body of prior work focuses on boosting allocation performance [12, 20, 16, 27]. We, however, concern ourselves with bytes lost due to the design choices of the user-level allocator. This is closely related to prior work on memory fragmentation [24, 11, 1, 22, 9], on which we thus focus on the following.

Early work on fragmentation investigated the overall fragmentation rate of existing allocation schemes [11], under varying workloads [1], showing that even best-fit heuristics can lead to severe memory fragmentation [24]. Building upon this insight, compaction techniques have been proposed in prior work to periodically defragment the heap [3, 22, 25]. These techniques are especially popular in conjunction with garbage-collected languages [2] but have recently also been applied to unmanaged languages [22]. Conceptually, compaction techniques are complementary to the more foundational free-space merging that we evaluated in Section 4.

With the advent of huge pages, there has been a renewed interest in optimizing memory allocation schemes to reduce huge-page fragmentation in physical memory [8]. For example, to mitigate huge-page fragmentation, Maas et al. propose placing memory objects in the heap based on their lifetime, which they infer through machine learning [17]. Follow-up work by the same authors also suggested an adaptive policy to "break up huge pages" and return them to the OS to enable system-wide optimizations [18]. Especially the latter work is reciprocal to our own as it illustrates the tight integration between the user- and kernel-level allocator required for efficient memory utilization. Conceptually, our insights regarding the importance of page alignment for memory-efficient user-level allocators also apply to huge pages and have not been sufficiently described in the existing literature.

7 Conclusion

Efficient memory management requires a close integration between the user- and kernel-level memory allocator. In this work-in-progress paper, we illustrate that bytes are lost at this boundary if the user-level allocator is not designed with the kernel-level memory view in mind. Specifically, the source of this inefficiency is a mismatch between the page-based and block-based view on unallocated memory that results in many page-crossing blocks that cannot be returned to the kernel. With our classification scheme, we contribute a theoretical framework to talk about the amount of memory that is lost at the boundary. And with our `mallocng` experiments, we demonstrate potential memory savings of up to 75 percent. Building upon these insights, we want to design a memory allocator that is optimized for efficiently returning memory to the kernel.

Data Availability

We published an artifact of our evaluation to reproduce the results in this paper [7].

Acknowledgments

We thank our reviewers for their valuable feedback. This work was partly supported by the German Research Foundation (DFG) under grant no. 468988364 and 501887536.

References

[1] Aniruddha Bohra and Eran Gabber. 2001. Are mallocs free of fragmentation? In *2001 USENIX Annual Technical Conference* (ATC '01). USENIX Association, Boston, MA, (June 2001).

[2] Jacques Cohen and Alexandru Nicolau. 1983. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5, 4, (Oct. 1983), 532–553. doi:10.1145/69575.357226.

[3] Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. 2008. A compacting Real-Time memory management system. In *2008 USENIX Annual Technical Conference* (ATC '08). USENIX Association, Boston, MA, (June 2008).

[4] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. Tech. rep. FreeBSD Project, (Apr. 2006). https://www.freebsd.org/publishing/whitepapers/malloc.pdf.

[5] Rich Felker. 2022. Disable madv_free usage in mallocng. Version e6e8213. Git Commit. (Sept. 28, 2022). Retrieved May 29, 2026 from https://git.musl-libc.org/cgit/musl/commit/src/malloc?id=e6e8213.

[6] Rich Felker. 2020. Mallocng-draft. Next-gen malloc for musl libc. Version 2ed5881. (June 15, 2020). Retrieved May 29, 2026 from https://github.com/richfelker/mallocng-draft.

[7] Pasha Fistanto. Lost bytes plos 2026 artifact. Zenodo, (Aug. 2026). doi:10.5281/zenodo.21772263.

[8] Mel Gorman and Andy Whitcroft. 2006. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Ottawa Linux Symposium* (OLS '06). Ottawa, Canada, (July 2006). https://www.kernel.org/doc/ols/2006/ols2006v1-pages-369-384.pdf.

[9] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '21). USENIX Association, (July 2021), 257–273. ISBN: 978-1-939133-22-9. https://www.usenix.org/conference/osdi21/presentation/hunter.

[10] Francisco Jeronimo, Tom Mainelli, Bryan Ma, Ryan Reith, and Jeff Janukowicz. 2025. Global memory shortage crisis: market analysis and the potential impact on the smartphone and pc markets in 2026. IDC Research, Inc. (Dec. 2025). Retrieved May 27, 2026 from https://www.idc.com/resource-center/blog/global-memory-shortage-crisis-market-analysis-and-the-potential-impact-on-the-smartphone-and-pc-markets-in-2026/.

[11] Mark S. Johnstone and Paul R. Wilson. 1998. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34, 3, (Oct. 1998), 26–36. doi:10.1145/301589.286864.

[12] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: accelerating memory allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '17). Association for Computing Machinery, Xi'an, China, 33–45. doi:10.1145/3037697.3037736.

[13] Tae Woo Kim, Youngjin Kwon, and Jeehoon Kang. 2025. Scalable address spaces using concurrent interval skiplist. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, 1131–1148.

[14] Daan Leijen. 2026. Mimalloc-bench. (June 3, 2026). Retrieved June 3, 2026 from https://github.com/daanx/mimalloc-bench.

[15] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-memory assisted buffer management. In *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data*. ACM, Seattle, WA, USA, (June 2023). doi:10.1145/3588687.

[16] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J. Yadwadkar, and Lizy K. John. 2023. Nextgen-malloc: giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (HotOS '23). Association for Computing Machinery, Providence, RI, USA, 135–142. doi:10.1145/3593856.3595911.

[17] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '20). Association for Computing Machinery, Lausanne, Switzerland, 541–556. doi:10.1145/3373376.3378525.

[18] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (ISMM 2021). Association for Computing Machinery, Virtual, Canada, 28–38. doi:10.1145/3459898.3463905.

[19] Mike Pall. 2025. Binary-trees lua program. (Mar. 2, 2025). Retrieved May 29, 2026 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-lua-2.html.

[20] Aidi Pi, Junxian Zhao, Shaoqi Wang, and Xiaobo Zhou. 2021. Memory at your service: fast memory allocation for latency-critical services. In *Proceedings of the 22nd International Middleware Conference* (Middleware '21). Association for Computing Machinery, Québec city, Canada, 185–197. doi:10.1145/3464298.3493394.

[21] Lennart Poettering et al. 2026. Memory pressure handling in systemd. Version 5ade3f. (Apr. 9, 2026). Retrieved May 27, 2026 from https://systemd.io/MEMORY_PRESSURE/.

[22] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: compacting memory management for C/C++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '19). Association for Computing Machinery, Phoenix, AZ, USA, 333–346. doi:10.1145/3314221.3314582.

[23] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (ASPLOS '87). IEEE Computer Society Press, Palo Alto, California, USA, 31–39. ISBN: 0818608056. doi:10.1145/36206.36181.

[24] M. John Robson. 1977. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20, 3, (Jan. 1977), 242–244. doi:10.1093/comjnl/20.3.242.

[25] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. Corm: compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data* (SIGMOD '21). Association for Computing Machinery, Virtual Event, China, 1811–1824. doi:10.1145/3448016.3452817.

[26] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: a survey and critical review. In *Memory Management*. Henry G. Baler, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 1–116. ISBN: 978-3-540-45511-0.

[27] Zhuangzhuang Zhou et al. 2024. Characterizing a memory allocator at warehouse scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (ASPLOS '24). Association for Computing Machinery, La Jolla, CA, USA, 192–206. doi:10.1145/3620666.3651350.