Reconsidering "Reconsidering Custom Memory Allocation"

Nicolas van Kempen
University of Massachusetts Amherst
Amherst, MA, USA
nvankempen@cs.umass.edu

Emery D. Berger*
University of Massachusetts Amherst
Amherst, MA, USA
Amazon Web Services
Seattle, WA, USA
emery@cs.umass.edu

Abstract

Programmers using native languages such as C, C++, or Rust can implement custom memory allocation strategies to improve execution time. In their paper titled "Reconsidering Custom Memory Allocation" almost 25 years ago, Berger et al. [2] showed that while per-class allocators provide no significant speedups over a state-of-the-art general-purpose allocator, region-based allocators can improve execution time by allocating and freeing objects in bulk. This paper revisits that work on a modern hardware platform with modern general-purpose allocators to evaluate whether their conclusions still hold. It also augments the benchmark suite with two large real-world applications (Clang and Blender), and introduces a methodology to explore the effect of memory fragmentation on locality in general-purpose allocators. Our results support and extend the original conclusions, demonstrating the locality advantages of region-based custom memory allocators.

CCS Concepts: • **Software and its engineering → Allocation / deallocation strategies**; **Software performance**; • **General and reference → Performance**.

Keywords: Memory, Performance

ACM Reference Format:
Nicolas van Kempen and Emery D. Berger. 2026. Reconsidering "Reconsidering Custom Memory Allocation". In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3814942.3816132

1 Introduction

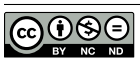
Low-level control over memory management is a fundamental aspect of native programming languages such as C, C++, or Rust. While garbage-collected systems provide increased safety guarantees and other software engineering benefits, manual memory management can provide significant performance improvements [14]. Notably, these performance enhancements can be obtained using a *custom memory allocator* by leveraging knowledge about the application's memory access patterns and object lifetimes.

Berger et al. [2] tackled the question of custom allocator performance over two decades ago in a paper entitled "Reconsidering Custom Memory Allocation"[1]. Their study classifies custom allocation techniques, and measures their performance across a suite of eight benchmarks. Those prior experiments identify *region allocators* as the only custom allocation technique providing substantial improvements to execution time (up to 44%) compared to a state-of-the-art general-purpose allocator. Region allocators (also named arena, pool, or bump-pointer allocators) operate by allocating objects contiguously without individually freeing them, instead returning all memory at once when the region is destroyed. This scheme allows for reduced allocator pressure thanks to fewer individual allocation and free operations: using knowledge of object lifetimes within the application, objects with identical or similar lifetimes can be grouped together, allowing efficient bulk interactions with the underlying allocator. Today, region allocators are still widespread. Most notably, they have been incorporated as first-class members of the C++ language since C++17 [9–13, 15]. Region allocator implementations are also available in many third-party libraries, including APR, Boost, and Folly for C/C++, as well as Bumpalo and typed-arena for Rust.

Over the two decades since Berger et al.'s study, hardware and state-of-the-art general-purpose allocators have changed substantially. Modern general-purpose allocators such as jemalloc [5] and mimalloc [19] have significantly narrowed the performance gap with custom strategies. At the same time, the growing gap between CPU speed and memory latency means cache misses are proportionally more expensive today [21, 26], making locality of reference an increasingly critical factor. Beyond the hardware and software landscape, we identify a methodological limitation in prior evaluations: benchmarks are run on a fresh, unfragmented heap, which

*Work done at the University of Massachusetts Amherst.

[1] Awarded Most Influential OOPSLA Paper in 2012.



ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816132

places the general-purpose allocator in ideal starting conditions. During execution, programs typically build up some level of natural heap fragmentation from earlier operations within the same process. Starting from a blank heap inadvertently advantages the general-purpose allocator, particularly for short-lived benchmarks.

To address this methodological gap, we introduce *adversarial allocation*, a technique that preconditions the heap with artificial fragmentation before the benchmark runs. Adversarial allocation first profiles the program to record its allocation size distribution and peak live object count, then uses this information to allocate and randomly free a large number of objects, leaving the heap in a fragmented state that mimics conditions in long-running or shared environments. On this preconditioned heap, general-purpose allocators must fill gaps scattered across memory, whereas region allocators still guarantee contiguous placement of objects. This exposes a locality benefit of region allocators that clean-heap evaluations systematically obscure: consecutive allocations within a region remain spatially co-located regardless of prior heap state, reducing cache misses when those objects are later traversed.

This paper makes the following contributions:

- Revisits custom allocators in the face of modern hardware and better high-performance general-purpose allocators.
- Proposes *adversarial allocation*, a refined methodology for accurately quantifying the locality upside of region allocators, without inadvertently giving an advantage to naïve general-purpose allocation.
- Evaluates the impact of region allocators in real programs, tackling benchmarks from Berger et al.'s evaluation and from two large-scale modern applications.

Our results largely support and extend Berger et al.'s original conclusions: per-class allocators provide no substantial benefit over a modern general-purpose allocator, while region allocators retain a meaningful advantage, particularly under heap fragmentation. However, our experiments also show that high-performance general-purpose allocators such as jemalloc and mimalloc have considerably closed the performance gap with region-based custom allocation, reducing region allocator speedups from up to 44% in the original study to at most 15% in our evaluation. Adversarial allocation further reveals a resilience advantage of region allocators that clean-heap evaluations miss entirely: naïve allocation slows down by up to 2× under heap fragmentation, while region allocators are unaffected.

2 Background

2.1 General-Purpose Memory Allocators

Dynamic memory allocation is necessary in low-level languages such as C and C++. General-purpose allocators must satisfy a wide set of requirements, without any knowledge of the application's specific allocation patterns: low overhead per operation, high throughput even in multi-threaded workloads, low memory waste, and good memory locality. A key technique used by general-purpose allocators is *size-class segregation*: the heap is partitioned into pools of fixed-size slots, and each allocation request is rounded up to the nearest size class and served from the corresponding pool.

Berger et al.'s experiments focus on two general-purpose allocators: the default Windows XP allocator ("Win32"), and the Doug Lea allocator ("dlmalloc") [17]. At the time, dlmalloc was a state-of-the-art high-performance allocator. Today, both of those allocators are severely outdated. This paper discusses three general-purpose allocators widely used today: glibc malloc, jemalloc [5], and mimalloc [19]. glibc malloc evolved from dlmalloc and is the default Linux allocator. jemalloc and mimalloc are modern high-performance allocators.

2.2 Custom Allocators

Using a custom allocator is a common technique to obtain performance gains and/or software engineering benefits. Most often, the motivation for implementing a custom allocator is performance. In some applications, custom allocators also allow for easier memory management; for example, a server handling requests may use a custom allocator to facilitate memory reclamation of objects after request handling is complete.

Custom allocators are an old idea, but they are still used today. In fact, there has been significant work to integrate custom allocator support as an integral part of the C++ standard library with C++17 [9–13, 15]. This integration is in the form of the `std::pmr` library, which provides a standard interface for custom allocators named `memory_resource`. A few subclasses of `memory_resource` are provided in the standard library, most notably a region allocator implementation named `monotonic_buffer_resource`.

We follow the taxonomy established and used by Berger et al. [2]. Namely, we split allocators into three categories: per-class, region, and *hybrid*. Table 1 provides an overview of the benchmarks used in this paper and in the original study, and a few characteristics of their custom allocator. The following sections provide more explanation of each type, along with examples taken from those benchmarks.

2.2.1 Per-Class Allocators. Per-class allocators optimize for a single object size/type, providing the usual `malloc`/`free` API. On deallocation, they keep a freelist of objects ready to be reused on the next allocation instead of returning memory to the system allocator. To minimize memory overhead, linked list pointers can be embedded in freed objects. Section 2.1 describes how general-purpose allocators already separate their allocations into size classes: therefore, intuitively, per-class allocators may only offer limited savings from size computations. Section 4.3 quantifies these savings.

Table 1. Classification of the custom allocators used in Berger et al.'s original study and in this paper (§2.2). Allocators are marked either per-class, region, or hybrid. This table also lists specific characteristics and interface details of those custom allocators: whether or not they use chunks, and what free interface is available. Three benchmarks from the original study are excluded: we could not find complete sources for C-Breeze, or a working and useful input for lcc. Apache's benchmarking is noisy and not compute-intensive enough to draw meaningful allocator performance conclusions.

Benchmark	Type	Original	This Study	Chunks	Individual Free	Bulk Free	Reason for Exclusion
boxed-sim	per-class	✓	✓		✓		
mudlle	region	✓	✓	✓		✓	
175.vpr	region	✓	✓	✓		✓	
176.gcc	region	✓	✓	✓		✓	
197.parser	hybrid	✓	✓	✓	✓		
Clang	region		✓	✓		✓	
geometry_nodes	region		✓	✓		✓	
sculpt	hybrid		✓	✓	✓	✓	
C-Breeze	per-class	✓			✓		missing sources
Apache	region	✓		✓		✓	not compute-bound
lcc	region	✓		✓		✓	missing benchmark input

Algorithm 1: Region Allocation (§2.2.2). If the current chunk doesn't have sufficient space, allocate a new one and set it as the current chunk. Then, bump the current chunk's allocation pointer. Allocators often have additional characteristics, such as alignment, special handling of larger allocations, or monotonically increasing chunk sizes.

```
function AllocateObject(region, size)
    if Available(region.current_chunk) < size then
        region.current_chunk ← CreateNewChunk()
        Append(region.chunks, region.current_chunk)
    end if
    object ← CurrentBumpPtr(region.current_chunk)
    IncrementBumpPtr(region.current_chunk, size)
    return object
end function
```

Algorithm 2: Region Reset (§2.2.2). In region allocators, memory is retained until the entire region is ready to be disposed of. This is one source of savings region allocators provide: significantly reduced total memory operations.

```
function Reset(region)
    for each chunk in region.chunks do
        free(chunk)
    end for each
end function
```

2.2.2 Region Allocators. Region allocators [6] aim to group together objects that share a similar lifetime and/or will be frequently accessed together. They follow the idea of a "bump-pointer"; incrementing a pointer for each object allocation, freeing everything in one single operation, but only when finished using all allocated objects. While some implementations use a fixed-size maximum, most others extend this functionality to a monotonically growing set of chunks, allowing support for an arbitrary number of objects. Algorithms 1 and 2 provide the outline for a simple region allocator implementation. On allocation, the pointer is simply bumped if the current chunk has enough available bytes, otherwise a new chunk must first be allocated. On reset and destruction, all allocated chunks are simply passed directly to the free function. This interface and implementation match C++17's `std::pmr::monotonic_buffer_resource`, both in libc++ and libstdc++.

Custom allocators in mudlle, 175.vpr, 176.gcc, and Clang, as well as Blender's LinearAllocator (geometry_nodes), all follow this mechanism. 176.gcc adds one special feature: the ability to partially free the region. By providing a pointer, all memory from that pointer to the region's tail will be deallocated.

Region allocators may suffer from increased memory footprint due to deallocations being deferred until the entire region is disposed of. This increased memory requirement is largely dependent on program and allocator implementation rather than measurement platform or machine architecture, and has been studied by Berger et al. [2]; this paper does not further discuss it.

2.2.3 Hybrid Allocators. Allocators not strictly falling into either one of the categories above typically employ a combination of techniques. In our benchmarks, this is the case for 197.parser and Blender's mempool (sculpt).

197.parser's allocator employs a fixed-size chunk of memory provisioned at startup. It then operates with a standard malloc/free API. Freeing an object marks it as free, and if it

is the last allocated object, the allocator resets its internal bump-pointer to the new last live object. In other words, an object will be re-used only if all subsequently allocated objects are first marked free. This method is effective for a stack-like use of memory.

Blender's mempool mixes region-based and freelist-based custom allocation. Each pool has a fixed object size, and a fixed number of objects per chunk, with new chunks allocated as needed from the system allocator. In addition to standard region allocation, BLI_mempool also has a freelist mechanism: on deallocation, objects are kept in a linked list for immediate reuse on the next allocation (last in, first out). Another particularity of this allocator is support for iteration over all allocated objects. During iteration, objects continue to be allocated in and freed from the pool.

3 Adversarial Allocation

Applications often gradually build up heap fragmentation throughout their execution, as allocations and deallocations of different sizes interleave. Therefore, running short-lived benchmarks from a clean-state heap does not give the full picture of the benefits of custom allocators, as general-purpose allocators generally behave optimally from a blank starting point, allocating contiguous objects. Figure 1 illustrates the effect of allocating from a blank heap versus an already fragmented heap: in the blank heap case, memory allocators typically return objects in contiguous memory. In contrast, when the heap is fragmented, memory allocators must fill distant gaps to keep memory footprint under control.

In order to fairly measure the full performance benefits of custom allocators, we introduce *adversarial allocation*, a method preconditioning the heap to simulate heap fragmentation that may occur in long-running applications or in tail-latency cases. Adversarial allocation consists of exercising the memory allocator before the program runs, simulating prior executions. This technique requires two phases: first, the program runs with lightweight instrumentation to record the size class distribution of all allocations made. This distribution is then used for all subsequent runs to repeatedly call malloc and free before the program's execution actually starts.

3.1 Phase 1: Detecting Allocation Sizes

General-purpose allocators typically segregate allocations by size classes. Therefore, for representative artificial heap fragmentation, we need to gather data on the program's allocation sizes. We achieve this with a small library interposing on the standard memory allocation functions. The output of this tool consists of the count of allocations for each size under 4096 bytes by default; allocations of 4096 or more bytes are ignored as they meet or exceed the standard page size. While this paper's benchmarks are not particularly impacted by physical memory footprint and address translation, this cutoff setting can easily be adjusted to account for large allocations in applications sensitive to TLB misses. Extending this study to huge pages is an interesting future direction. Figure 2 depicts an example allocation size distribution generated by this tool for 197.parser. The tool also increments and decrements a counter on malloc and free respectively, recording the maximum number of live allocations at any point in the program. Together with the distribution of allocation sizes, the adversarial allocation phase will use this maximum live allocation number to obtain an estimated peak allocation footprint.

Algorithm 3: Preconditioning the heap before program execution by allocating and randomly freeing objects (§3.2). The size distribution and the peak number of live allocations are retrieved from the previous detection phase. The multiplier and occupancy settings control the preconditioning footprint and density, respectively.

```
Require: distribution, peak_allocations
  function PreCondition(multiplier, occupancy)
      objects ← CreateEmptyArray()
      n ← multiplier × peak_allocations
      for i ∈ [1; n] do
          size ← SampleRandomSize(distribution)
          Append(objects, Malloc(size))
      end for
      Shuffle(objects)
      m ← n × (1 − occupancy)
      for i ∈ [1; m] do
          Free(objects[i])
      end for
  end function
```

3.2 Phase 2: Preconditioning the Heap

Algorithm 3 formalizes the preconditioning procedure happening during static initialization, before the program executes. The allocation size distribution and the peak number of live allocations are retrieved from the previous detection phase. With a given multiplier M, the tool allocates $M \times$ `peak_allocations` objects, sampling each object's size proportionally from the recorded distribution. The multiplier simulates the possible heap state of a long-running application that has, over its lifetime, allocated and mostly freed a quantity of memory far exceeding its current working set. After allocation, the objects are randomly and uniformly shuffled, and freed, leaving only a given occupancy fraction of them permanently live on the heap. By randomizing the order of allocated objects before freeing, the algorithm ensures that live and free blocks are interleaved throughout the heap, producing a fragmented free list that resembles what accumulates in practice [25].

Performance counters show that adversarial allocation increases cache misses at every level, indicating that heap

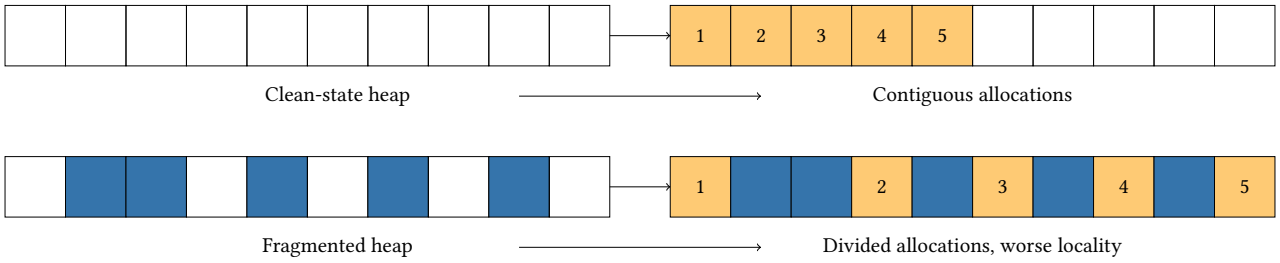


Figure 1. Comparing the performance of short-lived benchmarks from a clean-state heap masks potential locality improvements brought by the region allocator (§3). When starting from a blank, clean-state heap, the program’s allocator can typically deliver contiguous allocations. A region allocator on the other hand guarantees this contiguity, even in the presence of heap fragmentation.

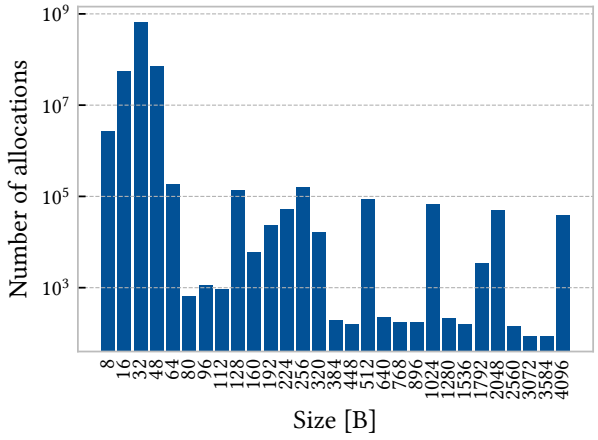
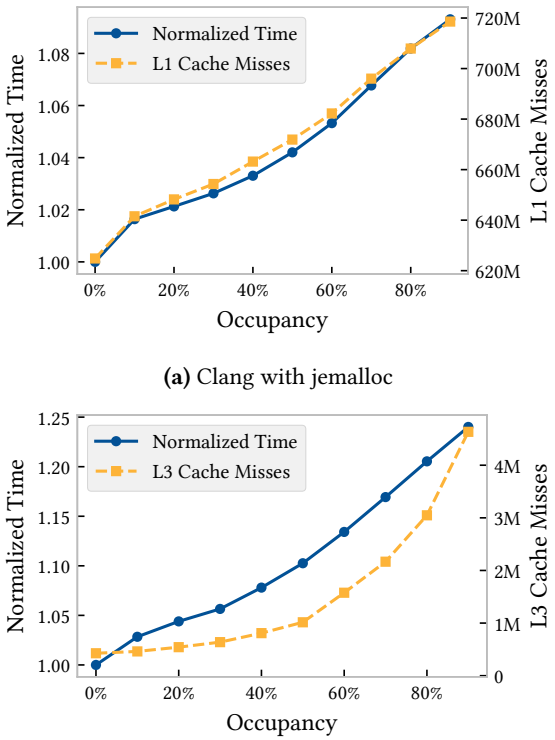


Figure 2. Distribution of allocations for 197.parser, grouped in size class bins, with a log-scale (§3.1). The detection phase also tracks the maximum count of live allocations: 491k for this benchmark.

fragmentation is the primary cause of the observed slowdown. Figure 3 shows examples of this effect, by gradually increasing adversarial allocation occupancy with a fixed multiplier. As occupancy increases, fewer contiguous object slots remain available to the allocator during execution. In turn, cache misses increase at every level, degrading performance. Adversarial allocation is a useful tool to quantify a program’s resilience to memory fragmentation, even when it is not always representative of realistic long-term fragmentation occurring in a given program. General-purpose allocators may implement heuristics to defeat this particular implementation, but an adversarial pattern can always be constructed for any allocator [25].

4 Evaluation

Berger et al.’s experimental setup used a “600 MHz Pentium III system with 320MB of RAM, a unified 256K L2 cache, and 16K L1 data and instruction caches”. For reference, Pentium



(a) Clang with jemalloc

(b) 197.parser with jemalloc

Figure 3. Adversarial allocation occupancy controls synthetic heap fragmentation (§3.2). These two examples, covering L1 misses in Clang and L3 misses in 197.parser under a fixed adversarial allocation multiplier of 10, highlight the correlation between cache misses (poor locality) and increased execution time.

III processors were discontinued in 2004. In contrast, this paper’s evaluation was conducted on a server equipped with two Intel Xeon Gold 6430 processors (initial release: 2023), running Linux version 6.8.0-84-generic, with 128GB of memory and a total of 120MB L3 cache. The tested underlying general-purpose allocators are glibc malloc 2.39, jemalloc

Table 2. Applications used to evaluate custom allocators, along with their version and input (§4).

Benchmark	Version	Input	Description
boxed-sim	sha256:fc65ed0	-n 128 -s 13	simulates *n* balls bouncing inside a box
mudlle	2020-04-30	time.mud	MUME's extension language interpreter
175.vpr	spec2000v1.3	train input	FPGA circuit placement and routing
176.gcc	spec2000v1.3	200.i	C compiler
197.parser	spec2000v1.3	ref input	natural language processing
Clang	21.1.8	sqlite3.c	C/C++ compiler
geometry_nodes	5.0.1	foreach_zone_bfield	geometry node modification
sculpt	5.0.1	1000 × 1000 mesh	diagonal brush stroke on mesh

Table 3. Configurations of adversarial allocations used in our evaluation (§4). Each configuration is further tested with the region allocator enabled or falling back to the underlying allocator, and with three backing general-purpose allocators: glibc malloc, jemalloc, and mimalloc.

Name	Multiplier	Occupancy
Adv_0	0	–
Adv_1	1	0.33
Adv_3	3	0.66
Adv_{10}	10	0.8

5.3.0, and mimalloc 2.2.5. We compile both benchmarks and allocators with Clang 21 and all optimizations enabled, except for glibc malloc which comes from the system's library. Reported results are averaged over repeated runs; variance was low across all measurements. Table 3 details all adversarial allocation settings discussed in our evaluation. This section aims to address the following research questions:

- **RQ1**: Do per-class custom allocators provide meaningful performance benefits?
- **RQ2**: Have high-performance general-purpose allocators closed the gap with region allocators?
- **RQ3**: Do region allocators offer more resilience to heap fragmentation?

The first two research questions mirror and aim to refresh Berger et al.'s evaluation. The third question extends our experiments to adversarial allocation and resilience to heap fragmentation.

4.1 Benchmarks

Table 2 describes the benchmarks used, as well as their version and input. We include all benchmarks from Berger et al.'s study we could successfully obtain and run, namely boxed-sim, mudlle, 175.vpr, 176.gcc, and 197.parser [23]. To obtain modern real-world examples of C and C++ programs using custom allocation, we searched Google and GitHub, and queried the ChatGPT and Claude large language models for projects using custom allocator implementations. Candidate applications were then filtered, selecting those amenable to meaningful benchmarking of their custom allocator. This process allows us to extend our evaluation with three additional benchmarks representative of modern application workloads: one from LLVM/Clang [4] and two from Blender [3].

Clang [4] is a very widely used C and C++ compiler, based on the LLVM toolchain [16]. Among a few other less frequently used custom allocator strategies, it implements a region allocator in `llvm::BumpPtrAllocator`. This implementation follows the general description in Section 2.2.2: it provides Allocate and Reset methods that respectively allocate an object of the given size, and free all allocated objects. We benchmark Clang on the SQLite3 v3.49.1 amalgamation, a very large (165k lines of code) C file, with all optimizations enabled (`-O3`).

Blender [3] is a state-of-the-art 3D computer graphics tool, widely used for 3D modeling and animations, including full-length animated films [18]. Its source contains three different implementations of region allocators, each used in different subsystems. This evaluation focuses on two:

1. LinearAllocator: This allocator follows the general mechanism outlined in Section 2.2.2.
2. BLI_mempool: This hybrid allocator mixes freelist-based and region-based custom allocation, and supports iteration. Notably, it requires stability during iteration in the event of object allocations or frees.

We benchmark those two allocators with built-in Blender performance benchmarks, with some changes to input size and/or output format: geometry_nodes for LinearAllocator and sculpt for BLI_mempool.

4.2 Disabling Custom Allocation

To effectively compare custom allocation strategies to relying solely on the general-purpose allocator, we maintain two separate versions of each benchmark: a first version as-is, with the custom allocator enabled, and a second where all individual object requests are forwarded directly into the system's malloc and free functions (naïve allocation). We use the same methodology as Berger et al., and manually replace

Table 4. Methods to disable custom allocation (§4.2).

Benchmark	Disable Method
boxed-sim	remove freelist
mudlle	reduce chunk size to fit single objects
175.vpr	reduce chunk size to fit single objects
176.gcc	reduce chunk size to fit single objects
197.parser	use malloc/free directly
Clang	reduce large allocation threshold to 0
geometry_nodes	reduce large allocation threshold to 0
sculpt	track live objects with a hash set

each custom allocator so that it falls back to the default underlying system allocator in the most efficient way possible. Disabling custom allocation can be as simple as redirecting back to the underlying malloc/free functions (197.parser). For most region allocators, ensuring each chunk contains exactly one allocation is a simple but effective way to guarantee objects are allocated via malloc, while maintaining object ownership tracking and bulk free capabilities. Per-class allocators (boxed-sim) are also straightforward to replace by removing each type-specific freelist. Blender's mempool is the most complex allocator to disable, due to its wide interface features: it must support individual and bulk frees, as well as iteration over all allocated objects. In this case, the most efficient way to emulate naïve allocation is keeping a hash set of all live objects. Table 4 summarizes the methods employed to disable custom allocation in each benchmark.

4.3 RQ1: Do per-class custom allocators provide meaningful performance benefits?

This section focuses on the boxed-sim benchmark. It uses a per-class allocator: on deallocation, objects are placed on class-local freelists and are recycled for the next allocation (last in, first out). Figure 4 shows normalized execution time results for this benchmark. For boxed-sim, naïve allocation with mimalloc outperforms glibc malloc with the custom allocator even on the highest adversarial allocation settings. Custom per-class allocation on top of mimalloc only shaves an additional 2.3%.

Berger et al. discuss one additional benchmark using per-class allocators: C-Breeze. However, it has not received any updates in two decades, and we were unable to successfully compile and run the program on our experimental platform. Further, our search for modern C and C++ applications meaningfully using per-class allocation yielded no compelling candidates.

Nevertheless, we extend our experiments with an upper-bound microbenchmark: it exclusively measures allocation throughput using intrusive pointers for zero memory overhead in building the linked freelist. With 16-byte allocations, this upper bound shows the per-class allocator outperforms naïve allocation with glibc malloc by only 24%, jemalloc by 11%, and mimalloc by 8%. In a realistic application where memory operations account for only a fraction of total cycles, these gains drop by an order of magnitude.

Per-class allocators offer no additional protection against heap fragmentation. The freelist ordering depends on deallocation history: even if objects are initially allocated contiguously, they become arbitrarily interleaved as allocations and frees occur.

RQ1 Summary: The boxed-sim per-class allocator provides only a 2.3% improvement to execution time with mimalloc, while requiring significant implementation effort. The per-class allocation microbenchmark confirms that potential savings are minimal. These results suggest that, in most cases, the engineering complexity of per-class custom allocators outweighs their performance benefits, which modern general-purpose allocators already largely eliminate. This aligns with Berger et al.'s conclusions.

4.4 RQ2: Have high-performance general-purpose allocators closed the gap with region allocators?

Figure 5 summarizes and compares the execution time using custom allocation (with the best possible underlying general-purpose allocator) versus naïve allocation. Since no adversarial allocation is involved, execution time reductions can for the most part be attributed to the reduction in memory operations that chunked allocators provide. Berger et al. [2] reported that "region-based allocators often outperform general-purpose allocation"; our results suggest high-performance general-purpose allocators today have significantly closed that gap. Among the three tested general-purpose allocators, mimalloc provides the best overall performance on our selection of benchmarks. Naïve allocation with mimalloc provides equal or nearly equal performance to custom allocation for 175.vpr and Clang. For 197.parser, geometry_nodes, and mudlle, the slowdown is 7.1%, 14%, and 15%, respectively.

RQ2 Summary: High-performance general-purpose allocators have significantly closed the performance gap with region allocators, down from the 44% upper bound reported by Berger et al. Regions can still offer execution time improvements, up to 15% in mudlle.

4.5 RQ3: Do region allocators offer more resilience to heap fragmentation?

This section leverages adversarial allocation, described in Section 3, to better surface any potential locality-related benefits region allocators can bring. Region allocators are able to fully bypass any natural fragmentation that gradually builds up during program execution, making them ideal candidates in tail-latency and locality-sensitive workloads.

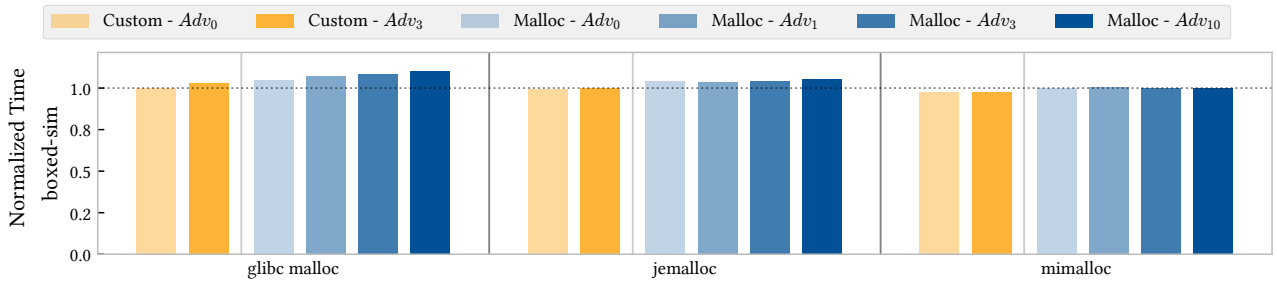


Figure 4. Per-class custom allocation in boxed-sim only provides marginal execution time improvements (§4.3). Execution time is normalized to using the custom allocator under glibc malloc with no adversarial allocation. Table 3 describes the adversarial allocation settings used.

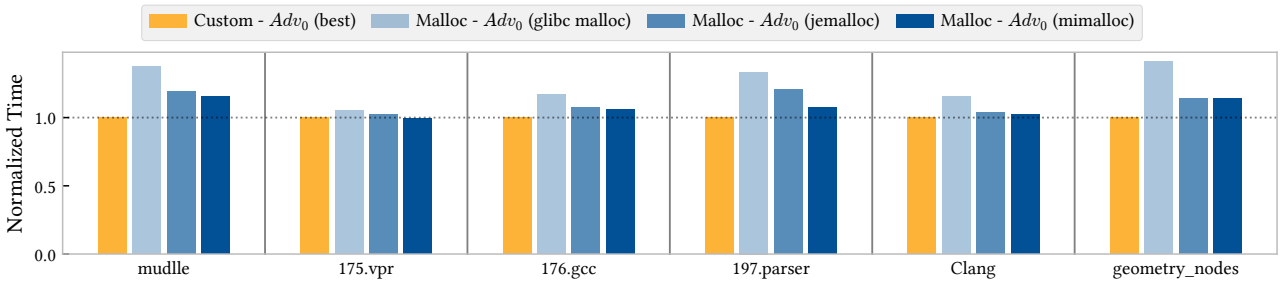


Figure 5. Region allocators significantly outperform naïve general-purpose allocation in four benchmarks (§4.4). For each benchmark, execution time is normalized to using custom allocation with the best underlying general-purpose allocator. While Section 2.2 classifies 197.parser as using a hybrid allocator, its use of chunked allocation exhibits roughly the same reduction in memory operations as other region allocator benchmarks.

Figure 6 shows the runtime for all region-allocator-based benchmarks in this paper. Nearly every region allocator — including the hybrid allocators in 197.parser and sculpt, whose chunked allocation provides the same locality properties — shows perfect resilience to adversarial allocation, even with higher settings. The only exception is Clang: this is because not all allocations in Clang go through the region allocator. Instead, regions are integrated only in high-impact functions and procedures throughout the program. Therefore, the remaining allocations are still susceptible to performance degradation from heap fragmentation. Nevertheless, under Adv_3 settings, Clang with its region allocator still outperforms naïve allocation by 3.4% for jemalloc, 3.2% for mimalloc, and 9.4% for glibc malloc.

In contrast, naïve allocation is heavily impacted by adversarial allocation, even with modern high-performance allocators. Based on the Adv_3 and mimalloc settings, adversarial allocation slows down mudlle, 197.parser, and 176.gcc by 18%, 25%, and 27%, respectively. This slowdown is mitigated using custom, region-based allocation. The default Linux allocator is considerably more affected by fragmentation than jemalloc and mimalloc, with both mudlle and 197.parser taking over 2× longer under Adv_3 settings.

Interestingly, 175.vpr and geometry_nodes appear relatively unaffected by adversarial allocation. 175.vpr is very processor-intensive with a low memory footprint, and hence has low sensitivity to fragmentation issues. 70% of allocations in the geometry_nodes benchmark are bigger than 64 bytes, the cache line size on our experimental platform. It also has a small working set of objects: only 18k. Because of their allocation properties, locality in these programs is not a major concern; region allocation can still provide performance benefits from reduced memory operations (Section 4.4).

RQ3 Summary: Region allocators provide strong resilience to heap fragmentation, while naïve allocation degrades execution time by up to 2× under adversarial allocation (mudlle and 197.parser, glibc malloc). Even with modern high-performance allocators, slowdowns up to 27% are observed. Clang is the only exception, as not all of its allocations are region-managed, yielding only partial resistance (3.2–9.4% depending on the underlying allocator). Programs with low memory sensitivity, such as 175.vpr and geometry_nodes, see little impact from fragmentation regardless of allocator.

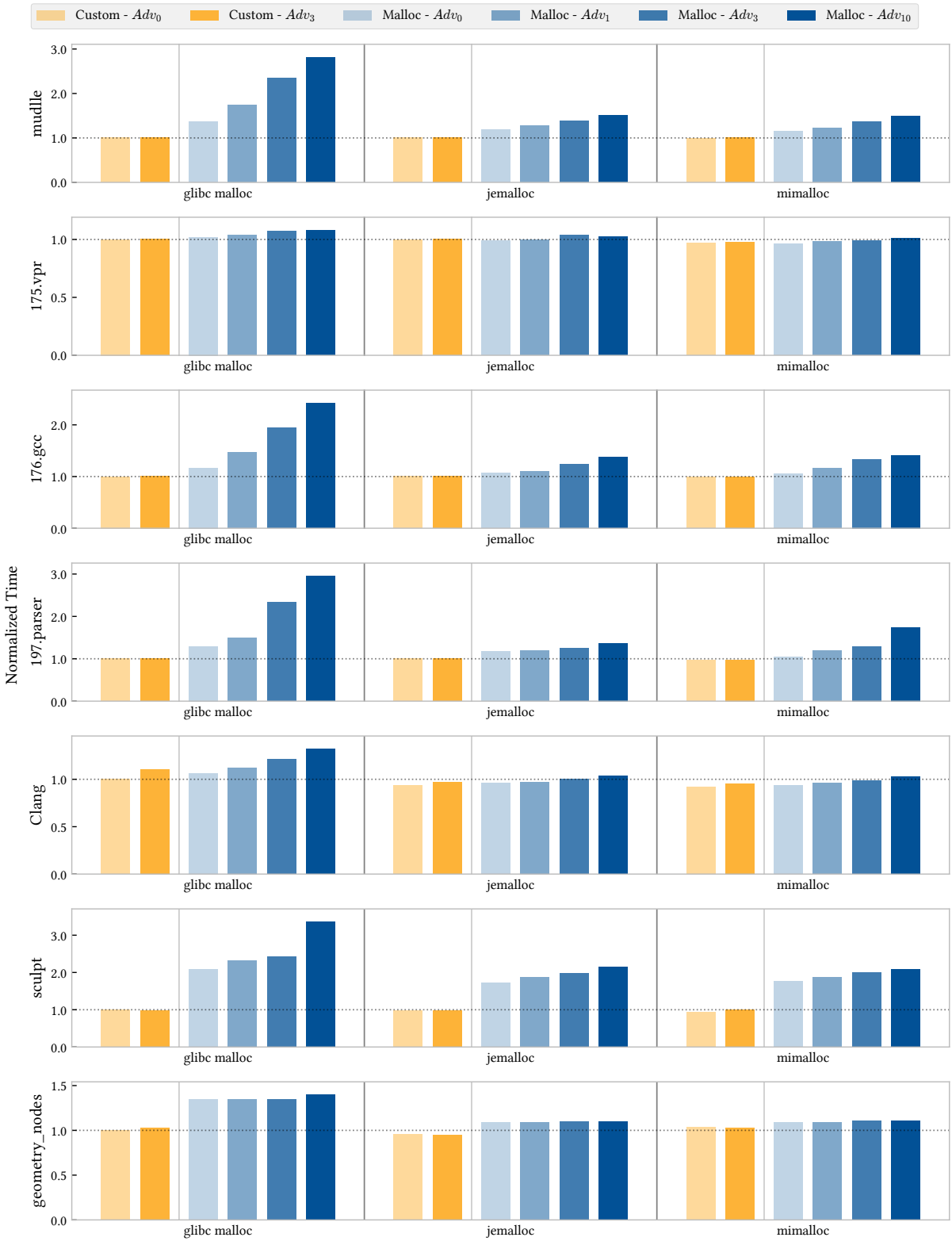


Figure 6. Region allocators offer significant resilience to adversarial allocation, and more generally heap fragmentation (§4.5). For each benchmark, execution time is normalized to using the custom allocator under glibc malloc with no adversarial allocation. Table 3 describes the adversarial allocation settings used.

5 Threats to Validity

This paper's evaluation relies on a specific set of benchmarks that use custom allocators: we extend a set of five programs taken from Berger et al.'s study [2] with three more benchmarks from two modern widely used applications. These programs are single-threaded; multi-threaded workloads may exhibit different performance characteristics and memory behavior. Custom allocators are virtually always used in single-thread or per-thread mode; therefore comparison in this environment provides a useful baseline. As discussed in Section 3.1, these benchmarks also each have a specific allocation size class distribution. Applications that often use larger allocation sizes such as geometry_nodes will suffer less from fragmentation-induced, cache-related performance degradation.

Experiments in this paper are conducted on a server architecture, detailed in Section 4. In particular, our platform has a large cache size (450× the last-level cache size of Berger et al.'s processor). We expect advantages from custom allocators to remain in machines with equal or smaller cache size, since this reduced capacity will tend to increase the rate of cache misses. Measurements on a different machine (i7-8559U processor with an 8MB L3 cache) support this claim and yield results equivalent to those presented in Section 4.

Section 3 describes adversarial allocation, a stress test for custom and general-purpose allocation that allows applying increasing degrees of fragmentation. This allows allocator comparison across a range of behavior, whereas Berger et al.'s previous evaluation inadvertently focused exclusively on the best-case scenario for the general-purpose allocator. As noted in Section 3.2, while this fragmentation may not necessarily occur, at least to that level, during normal execution, adversarial allocation remains useful to quantify allocator resilience to synthetic fragmentation and simulate tail latency scenarios.

Section 4.2 and Table 4 describe the methods used to simulate naïve allocation in each of the benchmarks discussed in this paper. Switching to naïve allocation requires additional object bookkeeping previously handled by the custom allocator; we follow Berger et al.'s methodology and carefully replace each allocator for maximal performance. A manual program rewrite moving entirely to explicit memory management may reduce the gap to region allocation, but will still suffer from diminished contiguity guarantees as discussed in Section 4.5.

6 Related Work

This paper revisits prior work from Berger et al. [2], which classified custom allocator techniques across eight benchmarks. Section 4's evaluation with modern hardware, allocators, and benchmarks supports the original paper's conclusions. Namely, only region allocators can provide any significant benefit over naïve allocation with a high-performance general-purpose allocator. We extend our evaluation using adversarial allocation as described in Section 3, surfacing the additional locality guarantees region allocators offer.

Region-based memory management has been studied both as a language construct [24] and as an explicit programmer-controlled interface [6–8], and has more recently been included in the C++ standard library as of C++17. The inclusion process produced extensive analysis of region allocator performance and locality benefits [9–13, 15]. Notably, the N4468 document [15] discusses locality implications of region allocation, using a microbenchmark iterating over an increasingly shuffled vector of subsystems. We refine this methodology with synthetic fragmentation for any application, allowing evaluation of real programs.

Recent work has sought to bring locality and fragmentation improvements to general-purpose allocators. In addition to jemalloc [5] and mimalloc [19] discussed in Section 4, Hoard [1] and tcmalloc [27] are other examples of modern high-performance and low-fragmentation general-purpose memory allocators. Mesh [22] compacts physical memory pages automatically, potentially consolidating objects onto the same cache lines. However, Mesh requires periodic "meshing" and its compaction is probabilistic, hence remaining susceptible to adversarial allocation. Llama [20] (Learned Lifetime-Aware Memory Allocator) uses machine learning to predict object lifetimes, grouping objects with similar lifetimes together to reduce fragmentation. While this reduces peak and steady-state memory usage, it adds prediction overhead to each allocation and cannot guarantee spatial contiguity, unlike region allocators which provide it by construction.

7 Conclusion

This paper revisits the question of custom memory allocator performance with modern hardware, general-purpose allocators, and applications. Berger et al.'s original conclusions hold: per-class allocators provide no substantial benefit over a state-of-the-art general-purpose allocator, while region allocators remain the only custom allocation strategy with meaningful performance advantages. Nonetheless, modern high-performance general-purpose allocators have narrowed the gap to custom region-based allocators, with region allocator speedups falling from up to 44% in the original study to at most 15% on a clean heap in our evaluation.

We identify a methodological limitation in prior evaluations: starting from a blank heap inadvertently places general-purpose allocators in ideal conditions, masking a key advantage of region allocators. Adversarial allocation addresses this limitation by preconditioning the heap with synthetic fragmentation derived from the program's own allocation behavior. Under these conditions, the contiguous placement guarantees of region allocators translate directly into resilience to fragmentation: naïve allocation degrades

by up to 2×, while region-allocated programs are unaffected. Region allocators are therefore most valuable not just for their raw throughput advantage, but for their predictability in long-running and tail-latency-sensitive environments.

Acknowledgments

We thank Joshua Berne, Frank Birbacher, and John Lakos from Bloomberg for many productive discussions on this topic and helpful feedback. We thank Bloomberg and Meta Platforms for financial support. We thank our reviewers and particularly our shepherd, Jeremy Singer, for valuable feedback and guidance.

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/378993.379232

[2] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/582419.582421

[3] Blender 1994. *Blender - The Free and Open Source 3D Creation Software*. Blender. Retrieved April 3, 2026 from https://blender.org/

[4] LLVM 2010. *Clang: A C Language Family Frontend for LLVM*. LLVM. Retrieved February 2, 2026 from https://clang.llvm.org

[5] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. *BSDCan* (2006).

[6] David Gay and Alex Aiken. 1998. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 313–323. doi:10.1145/277650.277748

[7] David Gay and Alex Aiken. 2001. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 70–80. doi:10.1145/378795.378815

[8] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. doi:10.1145/512529.512563

[9] Pablo Halpern. 2023. Making C++ Software Allocator Aware. C++ Standards Committee Paper P2127. https://wg21.link/P2127

[10] Pablo Halpern. 2024. Policies for Using Allocators in Library Classes. C++ Standards Committee Paper P3002. https://wg21.link/P3002

[11] Pablo Halpern and Dietmar Kühl. 2019. polymorphic_allocator<> as a Vocabulary Type. C++ Standards Committee Paper P0339. https://wg21.link/P0339

[12] Pablo Halpern and John Lakos. 2020. Unleashing the Power of Allocator-Aware Software Infrastructure. C++ Standards Committee Paper P2126. https://wg21.link/P2126

[13] Pablo Halpern and John Lakos. 2020. Value Proposition: Allocator-Aware (AA) Software. C++ Standards Committee Paper P2035. https://wg21.link/P2035

[14] Matthew Hertz and Emery D. Berger. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 313–326. doi:10.1145/1094811.1094836

[15] John Lakos, Jeffrey Mendelsohn, Alisdair Meredith, and Nathan Myers. 2015. On Quantifying Memory-Allocation Strategies. C++ Standards Committee Paper N4468. https://wg21.link/N4468

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[17] Doug Lea. 1996. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[18] Benjamin Lee. 2025. *Flow Wins Best Animated Feature Oscar*. The Guardian. Retrieved March 16, 2026 from https://theguardian.com/film/2025/mar/03/oscar-flow-best-animated-feature

[19] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. *Mimalloc: Free List Sharding in Action*. Technical Report MSR-TR-2019-18. Microsoft Research.

[20] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 541–556. doi:10.1145/3373376.3378525

[21] Sally A. McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers* (Ischia, Italy) *(CF '04)*. Association for Computing Machinery, New York, NY, USA, 162. doi:10.1145/977091.977115

[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* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 333–346. doi:10.1145/3314221.3314582

[23] SPEC. 1999. SPEC CPU 2000 v1.3. Retrieved April 3, 2026 from https://www.spec.org/cpu2000/

[24] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. doi:10.1006/INCO.1996.2613

[25] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *Memory Management, International Workshop IWMM 95, Kinross, UK, September 27-29, 1995, Proceedings (Lecture Notes in Computer Science)*, Henry G. Baker (Ed.). Springer, 1–116. doi:10.1007/3-540-60368-9_19

[26] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. doi:10.1145/216585.216588

[27] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 192–206. doi:10.1145/3620666.3651350

Received 2026-04-03; accepted 2026-05-04