EJSIT
European Journal of Science, Innovation and Technology
ISSN: 2786-4936
www.ejsit-journal.com
Volume 6 | Number 4 | 2026

Object Lifetime Models and Their Effect on Performance and Memory Fragmentation in Specialized C++ Allocators

Maksim Martynov*
Lead Programmer, Playrix
Novi Sad, Republic of Serbia

ABSTRACT

Modern C++ applications place heterogeneous demands on dynamic memory management. Allocator throughput and memory consumption depend on the workload request profile and on the lifetime model of allocated objects. This work presents an empirical study of four allocators: the system malloc, a monotonic arena, a fixed-size pool allocator, and a segregated free list. A matrix of 39 controlled scenarios is built by factorial variation of the lifetime model (long-lived, FIFO, LIFO, and random order), and the workload parameters: fixed small size, variable size, and churn. Each scenario records throughput and the ratio of reserved memory to peak live allocation volume. Additional experiments at 10× scale and at a reuse fraction of 0.9 expose the stability limits of each strategy. Specialized allocators outperform malloc in throughput by one to two orders of magnitude when the lifetime assumptions of their architecture hold. The pool allocator delivers 86.4 million operations per second on long-lived, fixed-size objects with an overhead coefficient of 1.000× and sustains 81.4 million operations under a 10× load increase. The segregated free list exposes a gap between throughput and memory efficiency: overhead reaches 236.5× under heavy churn, and the 10× scale test pushes the system into a fallback regime, with 2,002,194 capacity-exhaustion operations. A controlled ideal_fit_32 experiment confirms that making all requests fit a single size class does not reduce the overhead of an allocator that eagerly preallocates all size classes. The results justify treating the object lifetime model as a primary design axis for allocators.

Keywords: C++ memory allocator, object lifetime, memory fragmentation, pool allocator, segregated free list, monotonic arena, performance benchmarking

INTRODUCTION

Dynamic memory management remains one of the principal factors that shape application performance in C++. The cost of allocation and deallocation operations, data locality, fragmentation of the resident set, and consumption of physical pages all depend on the chosen allocator and the workload it serves. At the data-center scale, memory consumption has become a primary resource constraint: the TCMalloc allocator that serves the Google fleet was characterized by Zhou et al., 2024, in a large-scale profiling study showing that improvements within a single allocator translate into millions in cost savings as page-cache utilization rises. Maas et al., 2024, demonstrated that traditional C++ allocators are limited at the structural level by the absence of object-lifetime knowledge and that statistical predictions of this property yield gains in huge-page utilization. Related work by Maas et al., 2021, treats adaptive huge-page release in long-lived warehouse-scale processes and identifies the lifetime distribution as the main variable for policy selection.

In parallel, the community has begun to study fragmentation as an object of analysis in its own right. Lamprakos et al., 2023, showed that dynamic memory allocation is equivalent to a two-dimensional packing problem and proposed a fragmentation metric that correlates

* Email: m555ax@icloud.com

monotonically with peak resident size. Their later generalization, presented at MPLR, reduces the characterization of an allocator-workload interaction to a single quantitative scale. Navasca et al., 2023, trained neural networks on static code to predict dynamic heap properties such as lifetime and hotness. Gao, Xu, and Chen, 2024, used eBPF and LightGBM to predict fragmentation in high-load environments. All three directions converge on a single axis: knowledge about object lifetime is the limiting factor for any improvement in allocator policy.

Modern specialized allocators form a fragmented landscape. Pool allocators and arenas arise in the context of polymorphic memory resources of the C++17 standard and in production systems. Giannessi, Biondi, and Biasci, 2024, presented RT-Mimalloc, a variant with hard real-time guarantees for embedded systems. Yang et al., 2023, described NUMAlloc for NUMA architectures. Dang et al., 2024, developed PMAlloc for persistent memory. Security-oriented variants, StarMalloc by Reitz, Fromherz, and Protzenko, 2024, together with SeMalloc and S2malloc by Wang, Xu, and Asokan, 2024, form a separate design axis. Adaptive strategies such as Brug by Weng, Uta, and Rellermeyer, 2024, and architectural proposals such as NextGen-Malloc by Li et al., 2023, and Old is Gold by Li, John, and Yadwadkar, 2025, formalize the idea of accounting for application semantics. Profiling tools, among them MemPerf by Zhou et al., 2023, render allocator-induced latencies visible.

Ferreira, Matias, and dos Santos, 2024, performed an empirical investigation of memory allocation patterns in GUI applications and confirmed that distributions of object lifetimes and sizes differ across workload classes. A controlled comparison in which the variables of lifetime and workload are separated and exhaustively crossed on a single instrumental bench is absent from the literature. The present work fills this gap. The aim of the study is to construct an empirical map of the throughput-fragmentation tradeoff for four allocators of distinct design across a matrix of 39 scenarios, and to determine which properties of specialized strategies are produced by workload tuning and which are inherited from architecture. A secondary aim is to isolate the contribution of the object lifetime model from that of the size and churn-fraction parameters. The resulting map formalizes allocator selection as a function of an application's limiting properties.

MATERIALS AND METHODS

Allocators and architectural properties

Four allocators are compared. The first is the Windows malloc system, with the ucrtbase implementation. The second is the monotonic arena monotonic_arena, which serves memory linearly out of a pool of fixed capacity, with deallocation available only for the entire pool at once. The third is pool_allocator, which maintains a free list of fixed-size blocks at a constant cost per allocation and return. The fourth is segregated_list, which maintains a set of free lists for several size classes, each preallocated at initialization.

Architectural properties define the admissible subset of workloads. The monotonic arena is applicable only under the long-lived model, in which individual deallocations are absent until the end of the scenario. The pool allocator is applicable only at a fixed block size. Workloads with variable sizes are inaccessible to it. The segregated list and malloc accept any combination of sizes and lifetime models.

Lifetime models and workloads

Four lifetime models are introduced. FIFO frees objects in the order in which they were allocated. LIFO frees in reverse order. The random model applies a Fisher–Yates permutation with a deterministic seed. Long-lived performs no individual deallocations; all objects survive until the end of the measurement. Three workload parameters are paired with each model. Fixed_small uses objects of a single fixed size of 32 bytes. Variable-size draws are uniformly

distributed over the interval 8–512 bytes. Churn produces a mixed sequence of allocations and deallocations with a tunable reuse fraction.

The models form a 4×3 matrix, with 12 base points per universal allocator and sparse coverage for the specialized ones. The base set of admissible cells totals 31 scenarios across all four allocators, and 8 stress configurations extend the experiment to 39 measurement points. Standard cells use 1,000 live objects under fixed_small and variable_size and 500 live objects under churn, with allocation counts of 255,687, 255,917, and 503,652 across the three workloads. Long-lived cells skip individual deallocations and reach 101,000 live objects under fixed_small and variable_size at 506,152 allocations and 200,500 live objects under churn at 1,003,652 allocations. Lamprakos et al., 2023, showed that the character of an allocator–workload interaction is shaped by the joint distribution of lifetimes and sizes. Ferreira, Matias, and dos Santos, 2024, documented the heterogeneity of these distributions across application classes, which motivates a factorial design for the experiment.

Additional stress scenarios

Two diagnostic scenarios extend the base matrix. The fixed_small_large scenario increases the workload by a factor of 10, to 10,000 objects and 2.55 million operations, to test scaling and detect transitions across capacity thresholds. The heavy_churn scenario sets the reuse fraction to 0.9, with 200 live objects and 502,147 operations, modeling an intense lifecycle with rapid address turnover. A further control segment ideal_fit_32 curates request sizes to a single size class of segregated_list, isolating the contribution of irrelevant size classes to total overhead.

Metrics and definitions

Throughput is defined as the mean number of completed operations, allocations, and deallocations per second, measured across 5 repetitions after 3 warmup runs. The overhead coefficient is defined as the ratio of bytes reserved by the allocator to the peak live memory volume observed by instrumentation. This indicator reflects the memory held by the allocator in reserve relative to the minimum volume required to hold all live objects at once. The third metric is fallback_count, the number of operations the allocator could not serve from its internal structures, and that required a fall-through to the system path. The formal relation between throughput, overhead, and resident set is treated by Lamprakos et al., 2023, who showed that a simplified use of throughput as a sole metric masks strategies of unequal quality.

Scenarios in which the peak live volume exceeds the total capacity of the preallocated size classes of an allocator yield overhead values below unity. Such values do not indicate memory efficiency. They indicate that the allocator failed to serve requests from its own structure and was forced to fall back. The corresponding cases are flagged in the results by a non-zero fallback_count and are interpreted as a capacity-exhaustion indicator.

Execution environment and instrumentation

Experiments were executed on an Intel Core Ultra 9 275HX processor with 24 logical and physical cores. The software environment consisted of Windows 10.0.26200, the MSVC compiler version 19.38.33145.0, and an x64 Release build with the flags debug=0, opt=1, asserts=0. The measurement platform was an in-house benchmark system version 1.55.0, build identifier git 35a2309. Random permutations and size distributions are deterministic under seed 42. All scenarios are single-threaded. Multi-threaded effects, treated in NUMAlloc by Yang et al., 2023, and in adaptive allocators by Weng, Uta, and Rellermeyer, 2024, lie outside the scope of the present study and form the subject of a separate publication.

Memory-related metrics are collected through allocator-side instrumentation. The reported overhead coefficient reflects allocator-reserved memory relative to peak live payload

and should not be interpreted as full OS-level RSS. Each measurement consists of 3 warmup iterations and 5 measured iterations of the main loop, with the minimum, median, mean, 95th percentile, and maximum time recorded per iteration. The tables report mean values across 5 iterations.

RESULTS AND DISCUSSION

Baseline malloc throughput as the reference point

The system malloc delivers a throughput of 0.87–1.07 million operations per second across all 15 base scenarios. The spread across lifetime models remains within 21% of the median. The source of this invariance is the architecture of malloc, which implements a general strategy for free-memory management by searching for a suitable block and falling back to the kernel when the heap requires extension. The allocator data structure does not specialize for any hypothesis about deallocation order or size distribution. The result establishes a baseline against which specialized allocators are evaluated. A similar stability of system-allocator characteristics was reported by Zhou et al., 2024, in their characterization of TCMalloc on the Google fleet.

Throughput of specialized allocators

Figure 1 presents the measured throughput of all four allocators across the 12 base combinations of lifetime model and workload, on a logarithmic vertical axis.

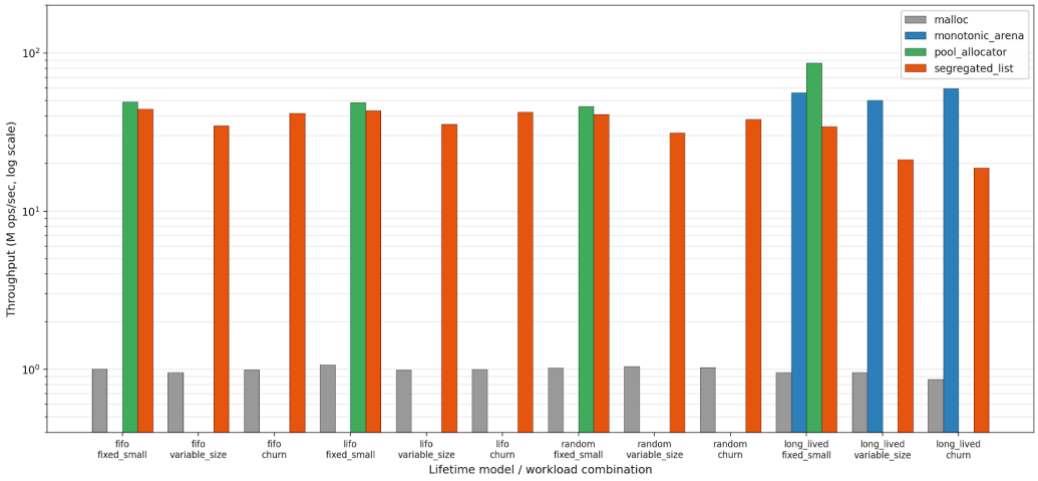

Figure 1. Throughput of all allocators across 12 combinations of lifetime model and workload

The vertical axis is logarithmic. The system malloc holds 0.87–1.07 million operations per second across all scenarios. Specialized allocators raise this level by one to two orders of magnitude when the assumptions of their architecture are met.

The pool allocator raises throughput to 46.0–49.0 million operations per second on the FIFO, LIFO, and random scenarios at fixed_small. The maximum is reached at 86.4 million operations per second on long-lived/fixed_small. This rise comes from eliminating the cost of returning blocks to the free list: under long-lived, all 101,000 objects are served sequentially from preinitialized blocks, and the hot path reduces to fixed-size block acquisition with no search and no variable-size bookkeeping. The value of 86.4 million places the pool allocator in the same order of magnitude as the strongest known designs for long-lived objects and aligns with the quantitative expectations for C++17 polymorphic memory resources.

The monotonic arena applies only to long-lived scenarios and reaches 50.3–59.5 million operations per second, depending on object size. This level is below the pool allocator peak of 86.4 million operations per second. The difference may come from implementation-specific overheads such as alignment handling, bounds checks, block-growth checks, and the fact that the pool allocator operates on a fixed block size. The convergence of the two strategies to the 50–86 million range under long-lived confirms that elimination of deallocation work is the principal source of the speedup.

The segregated free list spans 6.8–45.1 million operations per second, a wide range that reflects nonlinear sensitivity to workload parameters. On FIFO, LIFO, and random at fixed_small, the allocator delivers 41.0–44.3 million operations per second. On variable_size, the band shifts down to 31.2–35.5 million. On churn throughput rises to 38.0–42.4 million through the short reuse cycle of freed blocks. On long-lived, the allocator runs slower, at 18.8–34.4 million operations per second, owing to size-class capacity exhaustion and the resulting need for fallback. A description of this nonlinearity is given by Lamprakos et al., 2023, who formalized the dependence of allocator-workload behavior on boundary conditions through a packing metric. The full numerical breakdown of throughput across the matrix is collected in Table 1.

Table 1. Allocator throughput across the scenario matrix, in millions of operations per second

Allocator / workload	fifo	lifo	random	long_lived	scale 10×	heavy_churn
malloc / fixed_small	1.01	1.07	1.02	0.96	0.89	—
malloc / variable_size	0.96	0.99	1.05	0.96	0.97	—
malloc / churn	0.99	1.00	1.03	0.87	—	0.93
monotonic_arena / fixed_small	—	—	—	56.01	—	—
monotonic_arena / variable_size	—	—	—	50.32	—	—
monotonic_arena / churn	—	—	—	59.46	—	—
pool_allocator / fixed_small	49.01	48.46	46.03	86.36	81.43	—
segregated_list / fixed_small	44.27	43.19	41.01	34.36	7.57	45.10
segregated_list / variable_size	34.67	35.54	31.19	21.19	6.84	—
segregated_list / churn	41.71	42.41	38.04	18.80	—	—

Note: A dash denotes architectural incompatibility.

Memory overhead as an independent axis

Figure 2 lays out the overhead coefficient as a heatmap over the same matrix of scenarios. The color scale is logarithmic, since the segregated free list spans four orders of magnitude across the matrix.

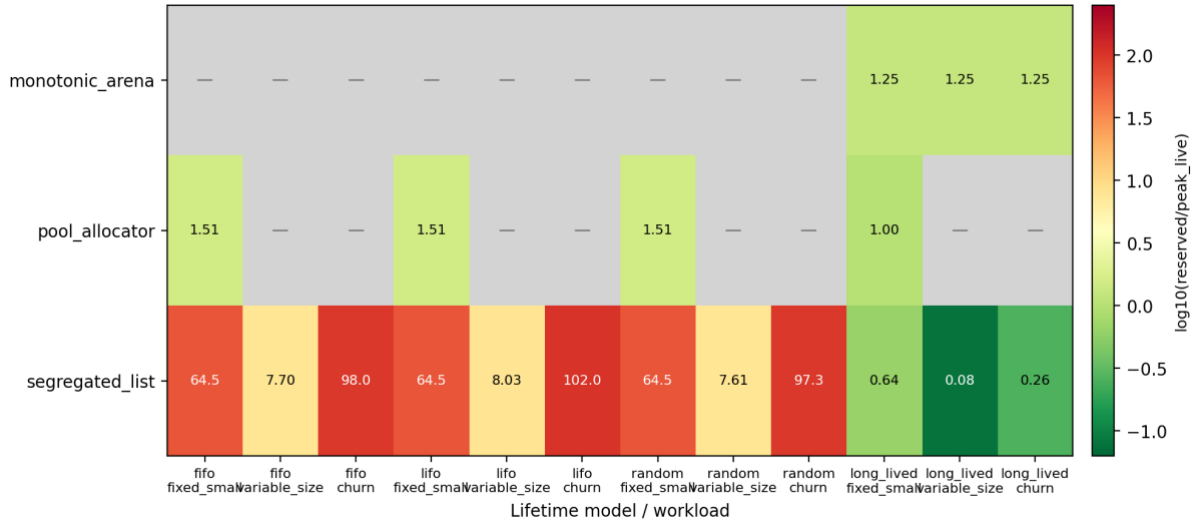

Figure 2. Heatmap of the overhead coefficient, the ratio of reserved memory to peak live allocation volume

Grey cells correspond to scenarios that the allocator architecture does not admit. Empty but admissible cells carry a numeric value. The color scale is logarithmic and reflects four orders of magnitude in segregated free list variability.

The monotonic arena holds overhead at 1.25× across all three long-lived workloads. The value is set by the arena growth policy: when the current block overflows, a new fixed-size chunk is allocated, and a small portion of the last chunk remains unused. The pool allocator returns an overhead of 1.51× under scenarios with surplus preallocation and 1.00× on long-lived/fixed_small, where the capacity is matched to the peak live count. The 1.00× ratio means that the entire reserve was occupied by useful payload at measurement time.

The segregated free list yields overhead values ranging from 0.08× to 236.5×, four orders of magnitude on a single architecture. High values arise when the reserve exceeds the peak live volume. On a churn workload with 200 live objects, 8729 bytes of live memory coexist with 2,064,384 bytes of reserved size classes, giving an overhead of 236.5×. On fixed_small with 1000 objects and 32,000 bytes of live memory, the ratio is 64.5×. The source of the divergence is an architectural decision to preallocate every size class at initialization, regardless of which classes the actual workload demands.

The long-lived scenarios deliver overhead values below unity, 0.08, 0.26, and 0.64. These values do not signify memory efficiency. They mean that the total volume of 101,000 live objects exceeds the joint capacity of the preallocated size classes. The allocator cannot serve further requests from its own structure and shifts into a fallback regime. The fallback_count in the matching cases reads 445,340, 494,760, and 971,780 operations, a count comparable to total allocations. The correct reading is that the allocator has run out of capacity, and the overhead metric is no longer comparable to other strategies. A similar analytic break is discussed by Lamprakos et al., 2023, where a simplified treatment of resident volume produces erroneous conclusions when the fallback path is ignored. Table 2 records the overhead coefficient for each specialized allocator and scenario.

Table 2. Overhead coefficient, the ratio of reserved memory to peak live allocation volume, for specialized allocators across the base matrix

Allocator / workload	fifo	lifo	random	long_lived
monotonic_arena / fixed_small	—	—	—	1.25
monotonic_arena / variable_size	—	—	—	1.25
monotonic_arena / churn	—	—	—	1.25
pool_allocator / fixed_small	1.51	1.51	1.51	1.00
segregated_list / fixed_small	64.51	64.51	64.51	0.64*
segregated_list / variable_size	7.70	8.03	7.61	0.08*
segregated_list / churn	98.04	101.97	97.34	0.26*

Note: The metric is undefined for malloc because the instrumentation captures the volume of live allocations visible to the client and has no access to the system allocator's internal heap state.
** A value below unity does not signify memory efficiency. It signals exhaustion of size-class capacity and a transition to the fallback path. The corresponding scenarios contain 445,340–971,780 fallback operations.*

Behavior under scale and under churn

Figure 3 reports the response of the three measured allocators to a tenfold increase in the object count. The left panel shows throughput at 1,000 and 10,000 objects, and the right panel shows the fallback operation count. The malloc bar is absent on the right panel because no fallback metric is defined for the system allocator. The bar for the pool allocator is at zero because no fallback occurred.

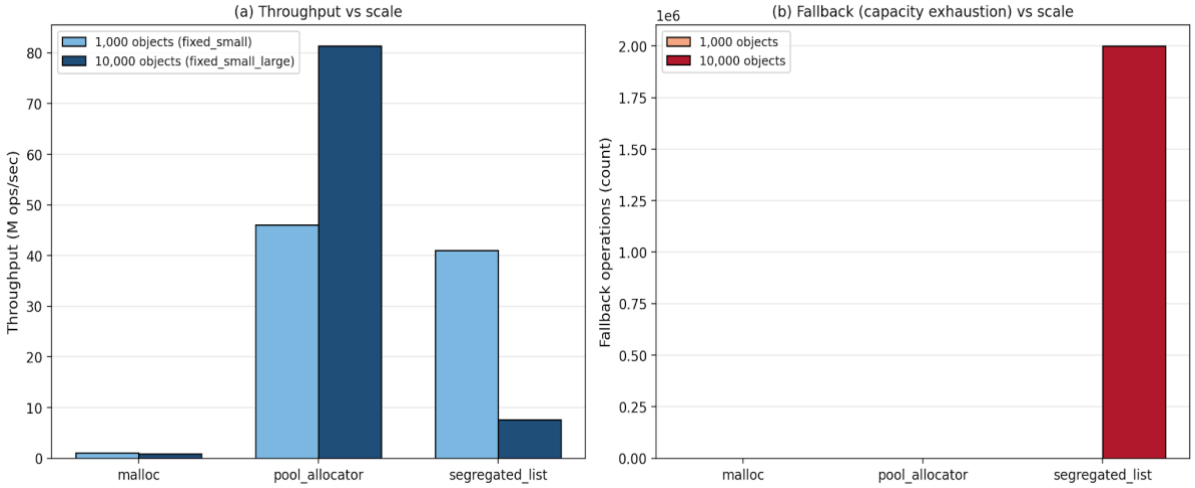

Figure 3. Stability test under scaling

Left, throughput at 1,000 and 10,000 objects. Right, fallback operation count. The pool allocator raises throughput from 46.0 to 81.4 million operations per second while keeping fallbacks at zero. The segregated free list drops from 41.0 to 7.6 million operations per second as 2,002,194 fallback operations appear. The system malloc holds 1.02 and 0.89 million operations at the two points

The fixed_small_large scenario raises the object count from 1,000 to 10,000 while preserving the random model and the fixed block size. The pool allocator achieves a 77%

increase in throughput, from 46.0 to 81.4 million operations per second. This effect is explained by amortization of free-list initialization cost: with a larger operation count, the fixed amount of setup work is divided across more requests, reducing the average per-operation cost. The overhead coefficient holds at 1.05, and no fallback occurs.

The segregated free list shows the opposite trajectory. Throughput drops from 41.0 to 7.6 million operations per second, a 5.4-fold reduction. The fallback counter reads 2,002,194 operations against 2.55 million total allocations. The fallback counter reaches 2,002,194, which is comparable to the total operation count and indicates that fallback dominates the measured execution. Although the aggregate reserved capacity is larger than the peak live payload, most of that capacity is distributed across size classes that are irrelevant to the fixed-size workload. The effective capacity of the 32-byte class is exhausted, so the allocator transitions to fallback despite having unused memory in other classes.

The system malloc retains 0.89 million operations per second under a 10× load increase, at the level of its baseline. This property, throughput stability under a change in scale, distinguishes the universal allocator from the specialized ones. The pool allocator and segregated_list trace opposite trajectories: the first improves as load rises, while the second collapses. Adaptive approaches discussed in Brug by Weng, Uta, and Rellermeyer, 2024, formalize the idea of a dynamic strategy switch when threshold conditions are crossed. The heavy_churn scenario forms a further extreme. A reuse fraction of 0.9 at 200 live objects yields 0.93 million operations under malloc with no losses, and 45.1 million operations under segregated_list at an overhead of 236.5×. The high throughput of segregated_list at this point hides the breakdown of the link between live memory and reserved memory. Table 3 collects the throughput, overhead, and fallback counts for the stress scenarios.

Table 3. Allocator behavior under stress scenarios

Scenario	Allocator	Throughput, M op/s	Overhead	Fallback ops
random / fixed_small_large	malloc	0.89	—	—
random / fixed_small_large	pool_allocator	81.43	1.05	0
random / fixed_small_large	segregated_list	7.57	6.45	2,002,194
random / variable_size_large	malloc	0.97	—	—
random / variable_size_large	segregated_list	6.84	0.78*	834,524
random / heavy_churn	malloc	0.93	—	—
random / heavy_churn	segregated_list	45.10	236.50	0
random / ideal_fit_32	segregated_list	41.43	64.51	0

Note: The fixed_small_large scenario corresponds to 10,000 objects and 2.55 million operations. The heavy_churn scenario models a reuse fraction of 0.9 with 200 live objects.
** A value below unity reflects capacity exhaustion. The accompanying fallback counter confirms the reading.*

The ideal_fit_32 experiment as isolation of the architectural contribution

Figure 4 compares the overhead of the segregated free list on the standard fixed_small workload with that of the curated ideal_fit_32 workload, in which every request is sized to fit in a single preallocated class.

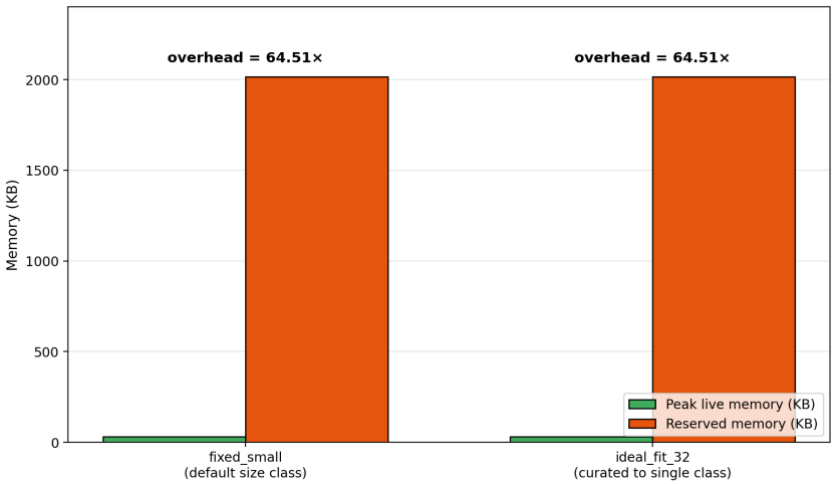

Figure 4. Comparison of segregated-free-list overhead on the standard fixed_small workload and on the curated ideal_fit_32, where every request lands in a single size class

Reserved memory holds at 2,064,384 bytes in both scenarios. Peak live load is the same. The shared overhead value of 64.51× points to a source of waste rooted in architecture and independent of configuration.

The ideal_fit_32 scenario is designed to test the hypothesis that segregated-free-list overhead arises from irrelevant size classes. All requests receive a single size that matches one of the preallocated classes. If overhead were determined by the ratio of classes to workload, then routing requests to a single class should reduce memory consumption. The measurement returns an overhead of 64.512×, identical to the overhead in the fixed_small scenario with the same size distribution and model seed.

The match rules out the hypothesis of a configurational source. The source of overhead is the eager preallocation of all size classes at allocator initialization. Size classes that the workload does not exercise continue to occupy memory. Curating the workload to a single class does not free the memory occupied by the others. The result has a direct implication for design: in architectures with a fixed set of size classes and eager preallocation, the workload-configuration variable cannot compensate for an architectural choice. A related observation about other strategies appears in the design-space survey by Maas et al., 2024.

The pool allocator as throughput champion within its niche

The pool allocator delivers the largest measured advantage over malloc. Pool throughput on long-lived/fixed_small reaches 86.4 million operations per second, compared to a malloc baseline of 0.96 million, for a 90× advantage. On FIFO, LIFO, and random with the same fixed_small, the advantage holds within 45–48×. The architectural account rests on constant-cost push and pop operations at the head of the free-list and on the absence of search. The pool allocator remains vulnerable along one dimension: the block size must be known in advance. Requests at any other size are not served, and the architecture provides no dynamic reconfiguration. Production use is described by Maas et al., 2024, in connection with huge-page utilization: under an estimate of the lifetime distribution, routing short-lived objects to one pool and long-lived ones to another reduces page-cache fragmentation.

Comparison with production allocators places the result in context. Giannessi, Biondi, and Biasci, 2024, in RT-Mimalloc, show that pool strategies can be embedded in more universal systems, at the cost of additional measures for predictability under tight constraints. Yang et al., 2023, treated NUMA effects, which are absent from the single-threaded measurements of the present study. Reitz, Fromherz, and Protzenko, 2024, in StarMalloc, and

Wang, Xu, and Asokan, 2024, in SeMalloc and S2malloc, demonstrate that safety and verification objectives form a separate design axis. These directions share a common feature with the present study: architectural decisions about which properties to treat as primary determine the space of achievable tradeoffs.

Object lifetime as the primary design axis

The empirical map shows that observed allocator behavior is shaped by the lifetime model. Under long-lived, the monotonic arena and the pool allocator dominate. On FIFO, LIFO, and random with fixed_small, the pool allocator leads. On variable_size without long-lived, the universal malloc and segregated_list partition the space, with segregated_list trading memory for throughput and malloc trading throughput for flexibility. Table 4 formalizes allocator selection as a function of two axes.

Table 4. Decomposition of allocator selection along two axes, the lifetime model and the character of sizes

Model / size	Fixed small size	Variable size	Heavy churn / unstable load
long_lived: bulk release	pool_allocator: 86.4 M op/s, overhead 1.00. monotonic_arena: 56.0 M op/s, 1.25.	monotonic_arena: 50.3 M op/s, 1.25.	monotonic_arena: 59.5 M op/s, 1.25.
fifo / lifo / random: individual deallocations	pool_allocator: 46–49 M op/s, 1.51. segregated_list when capacity exceeds peak load: 41–44 M op/s.	segregated_list: 31–35 M op/s, overhead 7.6–8.0. malloc: 0.9–1.0 M op/s with stable consumption.	malloc as the safe choice: 0.9–1.0 M op/s. segregated_list yields high throughput with overhead up to 102.
Scale 10× and beyond	pool_allocator rises to 81.4 M op/s. segregated_list drops to 7.6 M op/s.	segregated_list shifts to fallback mode. malloc holds 0.97 M op/s.	Adaptive strategies in the family of Brug call for separate evaluation.

Note: The recommendations rest on the empirical data of the matrix.

The idea of lifetime as a primary design property recurs in several independent studies. Maas et al., 2024, formalize it through ML prediction from the call stack. Navasca et al., 2023, train neural networks on static code. Gao, Xu, and Chen, 2024, use eBPF and LightGBM in high-load settings. All these approaches introduce an additional layer of instrumentation or learning to extract information that the standard allocator interface does not expose. The present work addresses a different facet of the same problem: when knowledge of object lifetime is provided by design, through the choice of a pool layout or an arena allocator, that knowledge produces concrete numerical gains, measurable and verifiable across a matrix of scenarios. Combining the two paths, extracting lifetime through ML and exploiting it via specialized allocators, is a research direction discussed by Li et al., 2023, and Li, John, and Yadwadkar, 2025.

A practical decomposition of allocator selection

The selection decision can be structured along two axes. The first axis is the character of object lifetime: whether the application is dominated by long-lived objects released as a

group, by objects with a local LIFO pattern, or by objects with a complex lifecycle. The second axis is the character of sizes: a fixed set of small sizes, a variable size with a known upper bound, or an unbounded distribution. Table 4 provides a decomposition based on the empirical data. Profiling support, discussed in MemPerf by Zhou et al., 2023, gives the tools required to locate a real application on these axes.

Further practical recommendations concern the limits of stability. The pool allocator is robust under load growth and delivers a positive throughput trajectory, up to a 10-fold increase. The segregated free list requires calibration of size-class capacities against the largest observed working set and loses quality beyond that set. The system malloc serves as a safe default: its throughput is unremarkable, yet stable across all tested scenarios. Specialized contexts, persistent memory, treated in PMAlloc by Dang et al., 2024, cache-aware data placement in CacheAware by Alanazi, Alanazi, and Albalawi, 2026, and GUI applications studied by Ferreira, Matias, and dos Santos, 2024, introduce further axes that the matrix of the present study does not cover.

LIMITATIONS

Measurements come from a single hardware-software combination, an Intel Core Ultra 9 275HX under Windows 10.0.26200, with the ucrtbase implementation of the system malloc and the MSVC compiler at version 19.38.33145.0. Allocator throughput and memory footprint are influenced by platform features such as virtual-page granularity, kernel heap policy, and CRT runtime layout. The set of compared allocators excludes industrial implementations such as jemalloc, tcmalloc, and mimalloc, with the system malloc serving as the single universal baseline. The directional findings, pool dominance under fixed-size long-lived workloads, segregated overhead driven by eager preallocation, malloc stability across the matrix, derive from architectural reasoning that ports across platforms with comparable allocator families.

All experiments are single-threaded, and the matrix omits persistent-memory workloads, secure-allocation regimes, and hard real-time constraints. PMAlloc by Dang et al., 2024, StarMalloc by Reitz, Fromherz, and Protzenko, 2024, and RT-Mimalloc by Giannessi, Biondi, and Biasci, 2024, define design spaces orthogonal to the throughput-fragmentation tradeoff, while NUMAlloc by Yang et al., 2023, and Brug by Weng, Uta, and Rellermeyer, 2024, address contention and adaptive multithreaded behavior. Cross-implementation and multithreaded extensions of the matrix form the subject of follow-on work.

The 5 measured iterations after 3 warmup runs under a fixed seed fall below the customary 20–30 range of microbenchmark studies. The benchmark records median, minimum, maximum, and 95th-percentile values per scenario, and the relative spread between maximum and minimum is under 10 % of the mean in 27 of the 39 scenarios and under 20 % in 33 of them. The higher-variance points concentrate in long-lived workloads where individual runs hold more state. The overhead coefficient itself loses interpretive sharpness when peak live volume drops to a few kilobytes. The heavy_churn point at 8,729 live bytes against a 2,064,384-byte reserve yields a ratio of 236.5 driven by the small denominator, and side-by-side reading of absolute reserved volume helps when peak live volume sits below 50 KB.

CONCLUSION

The empirical map drawn over the 39 measurement points, 31 cells of the factorial matrix and 8 stress configurations, places object lifetime at the center of the C++ allocator design space. Architectural choices determine which combinations of lifetime model and request size an allocator can serve and which it discards. Within the admissible subset, the throughput of specialized strategies rises by one to two orders of magnitude over the level of system malloc. The price for these gains is paid in memory or in flexibility, and the form of payment is fixed by architecture. The pool allocator trades universality for throughput and

scaling stability. The segregated free list trades throughput for the eager preallocation of size classes. The monotonic arena trades the option of individual deallocations for a compact allocation path.

The ideal_fit_32 control isolates the architectural and configurational contributions to memory waste. The match in overhead values between curated and standard workloads establishes that the peak overhead of the segregated free list arises from design. Tuning does not account for them. This finding sets a direction for future work. Architectures in which size-class preallocation runs on demand or in response to observed workload feedback could close the gap between observed throughput and actual memory efficiency. A related direction is a multithreaded matrix implementation that incorporates NUMA effects and adaptive strategies. A third direction is the union of specialized allocators with lifetime-prediction approaches, in which routing requests across pools occurs at runtime under the output of an ML classifier. The present data offer a quantitative reference point for evaluating such combinations.

Extending the result beyond C++ requires care. Environments with automatic memory management have a different cost structure, and the architectural classes of allocators in such environments form along different axes. The principle of lifetime primacy as a design factor applies to them as well. Future work that compares allocators under real production loads can use the matrix of the present study as a calibration grid for the observed effects. An open question is the manifestation of the lifetime effect in tasks with persistent memory, secure allocation, and real-time allocation, where additional design constraints reshape the tradeoff.

ACKNOWLEDGMENTS

The author reports no financial conflicts of interest in connection with the work presented.

REFERENCES

Alanazi HA, Alanazi AG, Albalawi NS. CacheAware: data locality-aware scheduling for distributed memory systems. Computers, 2026, 15(3), 181. DOI: 10.3390/computers15030181

Dang Z, He S, Zhang X, Hong P, Li Z, Chen X, Song H, Sun XH, Chen G. PMAlloc: a holistic approach to improving persistent memory allocation performance. ACM Trans Comput Syst, 2024, 42(3–4), 1–52. DOI: 10.1145/3643886

Ferreira A, Matias R, dos Santos C. Empirical study on memory allocation patterns in GUI-based applications. In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC 2024), IEEE, 2024, 772–778. DOI: 10.1109/SMC54092.2024.10831777

Gao Y, Xu D, Chen L. Prediction and optimization of memory fragmentation based on eBPF and LightGBM in high-load environments. In: Proceedings of the 2024 7th International Conference on Artificial Intelligence and Pattern Recognition (ICAIPR 2024), ACM, 2024, 798–805. DOI: 10.1145/3703935.3704087

Giannessi R, Biondi A, Biasci A. RT-Mimalloc: a real-time memory allocator for embedded systems. In: Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2024), IEEE, 2024, 173–185. DOI: 10.1109/RTAS61025.2024.00022

Lamprakos CP, Xydis S, Catthoor F, Soudris D. The unexpected efficiency of bin packing algorithms for dynamic storage allocation in the wild: an intellectual abstract. In: Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM 2023), ACM, 2023, 58–70. DOI: 10.1145/3591195.3595279

Li R, Wu Q, Kavi K, Mehta G, Yadwadkar NJ, John LK. NextGen-Malloc: giving memory allocator its own room in the house. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS 2023), ACM, 2023. DOI: 10.1145/3593856.3595911

Li R, John L, Yadwadkar N. Old is gold: optimizing single-threaded applications with ExGen-Malloc. IEEE Comput Archit Lett, 2025, 24(2), 225–228. DOI: 10.1109/LCA.2025.3587582

Maas M, Kennelly C, Nguyen K, Gove D, McKinley KS, Turner P. 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), ACM, 2021, 28–38. DOI: 10.1145/3459898.3463905

Maas M, Andersen DG, Isard M, Javanmard MM, McKinley KS, Raffel C. Combining machine learning and lifetime-based resource management for memory allocation and beyond. Commun ACM, 2024, 67(4), 87–96. DOI: 10.1145/3611018

Navasca C, Maas M, Maniatis P, Lim H, Xu GH. Predicting dynamic properties of heap allocations using neural networks trained on static code: an intellectual abstract. In: Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM 2023), ACM, 2023. DOI: 10.1145/3591195.3595275

Reitz A, Fromherz A, Protzenko J. StarMalloc: verifying a modern, hardened memory allocator. Proc ACM Program Lang OOPSLA, 2024, 8, 1757–1786. DOI: 10.1145/3689773

Wang R, Xu M, Asokan N. S2malloc: statistically secure allocator for use-after-free protection and more. In: Proceedings of DIMVA 2024, Lecture Notes in Computer Science, vol. 14828, Springer, 2024, 23–43. DOI: 10.1007/978-3-031-64171-8_2

Wang R, Xu M, Asokan N. SeMalloc: semantics-informed memory allocator. In: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS 2024), ACM, 2024. DOI: 10.1145/3658644.3670363

Weng W, Uta A, Rellermeyer JS. Brug: an adaptive memory (re-)allocator. In: Proceedings of CCGrid 2024, IEEE, 2024, 67–76. DOI: 10.1109/CCGrid59990.2024.00017

Yang H, Zhao X, Zhou J, Wang W, Kundu S, Wu B, Guan H, Liu T. NUMAlloc: a faster NUMA memory allocator. In: Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM 2023), ACM, 2023. DOI: 10.1145/3591195.3595276

Zhou J, Silvestro S, Tang SJ, Yang H, Liu H, Zeng G, Wu B, Liu C, Liu T. MemPerf: profiling allocator-induced performance slowdowns. Proc ACM Program Lang OOPSLA, 2023, 7, 1418–1441. DOI: 10.1145/3622848

Zhou Z, Gogte V, Vaish N, Kennelly C, Xia P, Kanev S, Moseley T, Delimitrou C, Ranganathan P. Characterizing a memory allocator at warehouse scale. In: Proceedings of ASPLOS 2024, ACM, San Diego, USA, 2024, 192–206. DOI: 10.1145/3620666.3651350