Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 2

Open Access | PP: 77-83

DOI: https://doi.org/10.70315/uloap.ulete.2026.0302013

Architectural Patterns of Specialized Memory Allocators in C++ and Trade-Offs between Latency and Fragmentation

Maksim Martynov

Lead Programmer, Playrix, Novi Sad, Republic of Serbia.

Abstract

This article examines specialized memory allocation patterns in C++ through an architectural interpretation of allocator behavior under different object lifetime regimes. The topic remains relevant because engineering teams often optimize allocation speed at the hot path and detect the cost of that choice later, once memory retention, locality decay, and fragmentation begin to shape runtime behavior. The study aims to clarify how bump and arena allocators, pools, stack allocators, segregated lists, thread-local paths, and deferred reclamation schemes reshape the design space of high-performance software. The material base consists of ten recent scholarly publications selected from major computer science databases. The analytical procedure combines source analysis, comparative interpretation, conceptual synthesis, and typologization. The article develops a mechanism-centered account of allocator behavior, explains why latency and fragmentation belong to the same design problem, and formulates decision logic for choosing allocator patterns under bounded, mixed, and concurrent workloads. The practical value lies in offering a structured basis for memory architecture decisions in performance-critical C++ systems.

Keywords: *C++, Memory Allocators, Arena Allocation, Pool Allocation, Fragmentation.*

INTRODUCTION

Memory allocation in C++ shapes far more than local execution cost. Once a system starts to create objects with different sizes, uneven access patterns, and incompatible lifetime horizons, allocator choice influences cache residency, synchronization pressure, reuse timing, and long-run memory retention. Teams often treat allocation as a narrow implementation concern and postpone architectural judgment until performance anomalies appear under load. By that point, the allocator has already influenced data placement, reclamation timing, and the degree to which memory remains reusable after bursts of activity.

A concrete reference point for this analysis comes from a production-grade memory subsystem organized as a dedicated core/memory module. That module encodes allocator patterns as explicit architectural constraints instead of exposing them as interchangeable low-level utilities. Each allocation path carries assumptions about object lifetime, ownership boundaries, and reuse discipline. This framing anchors the discussion in an implementation where allocator behavior belongs to system structure and does not sit outside it as a narrow optimization layer.

The aim of this article is to develop an architectural interpretation of specialized allocator patterns in C++ and to explain how the trade-off between latency and fragmentation changes once allocator structure is read together with object lifetime semantics. The first task is to reconstruct the operating logic of major specialized patterns, including bump or arena, pool, stack, and segregated-list designs, from the standpoint of fast-path cost and reuse discipline. The second task is to explain how lifetime regularity, mixed churn, and concurrency reshape fragmentation exposure and locality quality. The third task is to formulate a decision model that relates workload structure, reclamation discipline, and placement policy to allocator suitability in real systems.

The article adds a different angle to the subject by treating allocator architecture as a coupled arrangement of reservation, placement, reuse, and reclamation. That framing reaches further than benchmark ranking or library comparison. The working hypothesis states that low allocation latency persists only while the allocator embeds assumptions that match the temporal structure of live objects. Once those assumptions drift away from the actual workload, the apparent gain at the fast path is repaid through stranded memory, weaker

Citation: Maksim Martynov, "Architectural Patterns of Specialized Memory Allocators in C++ and Trade-Offs between Latency and Fragmentation", Universal Library of Engineering Technology, 2026; 3(2): 77-83. DOI: https://doi.org/10.70315/uloap.ulete.2026.0302013.

locality, or delayed reclamation. Within this setting, lifetime becomes a controllable design parameter rather than an emergent side effect. The core/memory design exposes lifetime boundaries through allocator selection itself, which lets engineers constrain object survival domains and prevent uncontrolled mixing of allocation regimes.

METHODS AND MATERIALS

The material base was formed through a targeted search in Scopus, Web of Science, ACM Digital Library, and IEEE Xplore for publications issued between 2021 and 2025 on memory allocators, huge-page fragmentation, NUMA-aware allocation, profile-guided heap placement, concurrent reclamation, and allocator verification. Search strings combined terms such as memory allocator, fragmentation, object lifetime, NUMA, heap allocation, concurrent reclamation, C++ systems, and locality-aware allocation. The screening process excluded textbooks, blog posts, vendor documentation, tutorials, and papers centered on managed runtimes without transfer value for unmanaged C and C++ memory systems. After title, abstract, and venue screening, ten publications remained. The final corpus covers five connected problem groups: lifetime-aware allocation and huge-page efficiency, allocator behavior at warehouse scale, NUMA-sensitive heap management, profile-guided locality control, and the coupling between allocation and reclamation in concurrent systems (Maas et al., 2021, 2024, Yang et al., 2023, Zhou et al., 2024, Oh et al., 2023, Moreno & Rocha, 2023, Singh et al., 2024, Reitz et al., 2024, Dang et al., 2024, Li et al., 2025). The selected sources are interpreted against the logic of the core/memory module, where allocator patterns are already separated along lifetime and ownership boundaries. This makes it possible to map theoretical models from the literature onto concrete allocation paths instead of leaving them at the level of abstract allocator description.

The study uses comparative analysis to identify recurring allocator patterns across different system settings, conceptual synthesis to connect fast-path behavior with fragmentation dynamics, typologization to group allocator designs by lifetime assumptions and reuse discipline, and analytical generalization to derive an implementation-oriented decision model for C++ memory architecture. Allocator behavior is interpreted within a simplified formal model of workload and object lifetime. Workload is treated as a parameterized sequence of allocation and deallocation events, while lifetime is defined as the interval between allocation and release within this sequence. This representation allows different allocator patterns to be compared under controlled variations of allocation dynamics without relying on a fixed execution trace. A phase-based view of execution is used to distinguish between growth, steady reuse, and churn, which makes it possible to relate allocator assumptions to the evolution of fragmentation and reuse over time.

RESULTS

Specialized allocators differ less by surface API than by the temporal assumptions hidden inside their internal paths. A bump or arena allocator presumes that deallocation can move away from individual objects and converge at a phase boundary. Under that condition, allocation collapses into pointer advancement, metadata traffic shrinks, and fragmentation remains constrained by construction. The same pattern loses coherence once medium-lived and long-lived objects stay in the same region. Bulk reclamation then loses selectivity, and memory retention starts to accumulate under the appearance of cheap allocation. In the core/memory module, this limitation is handled by isolating arena usage to strictly bounded execution scopes. The allocator is not expected to tolerate lifetime drift. The surrounding system enforces phase alignment so that reclamation remains structurally valid. Recent lifetime-aware work follows that same logic. Researchers report allocator gains once runtime systems expose lifetime regularity to the allocator instead of hiding it behind a uniform malloc-like contract (Maas et al., 2024, Maas et al., 2021).

Pool allocation addresses a different workload shape. It works best when object size repeats and reuse frequency stays high. Under those conditions, the allocator avoids expensive search paths and limits external fragmentation because blocks circulate inside a fixed-size class. The architectural benefit still carries a cost. Once object populations spread across classes, or once a pool tuned for short-lived objects begins to retain occupants that stay for much longer, internal slack rises and usable capacity contracts without a clear signal at the API boundary. The core/memory implementation addresses this by binding pools to specific ownership domains instead of exposing them as general-purpose allocators. Objects allocated from a pool share both size constraints and an expected lifetime horizon, which reduces the probability that long-lived objects occupy slots intended for short reuse cycles. PMAlloc reaches a related conclusion from the persistent-memory side. The authors show that size-segregated organization remains effective for small allocations, while performance and space behavior depend on the arrangement of metadata placement and reclamation paths (Dang et al., 2024).

Segregated-list allocators occupy a broader middle ground. They preserve fast allocation for recurring size classes, yet they do so through a wider metadata surface and more involved refill logic. In practice, they fit workloads with size dispersion that still clusters into reusable bands. Their weakness appears once temporal dispersion overtakes size regularity. Free lists continue to classify memory by size, while the waste originates from objects that share pages and regions despite dying at very different times. Within the module design, this effect is mitigated by restricting cross-

domain allocation into shared size classes. The allocator layer enforces separation between lifetime regimes before they reach the same free-list structure. Recent lifetime-aware allocator work breaks with that size-first view and reorganizes placement around predicted lifetime classes, since huge-page fragmentation cannot be reduced through size segregation alone (Maas et al., 2024).

Several sources become especially informative once they are read against the same technical question, namely why low-latency paths so often produce memory waste later in execution. The warehouse-scale study of TCMalloc places per-CPU caches and placement effects at the center of both throughput and locality. The adaptive huge-page subrelease study shows that high huge-page coverage improves translation behavior, while delayed return of broken huge pages inflates memory footprint. The NUMAlloc paper adds a third layer and demonstrates that topology-blind allocation raises the cost of remote access on NUMA systems. These works point toward one shared conclusion. Engineers improve latency by shortening the hot path, reducing contention, and preserving locality during placement. The relationship between allocator specialization, latency reduction, and delayed memory costs is presented below (Figure 1).

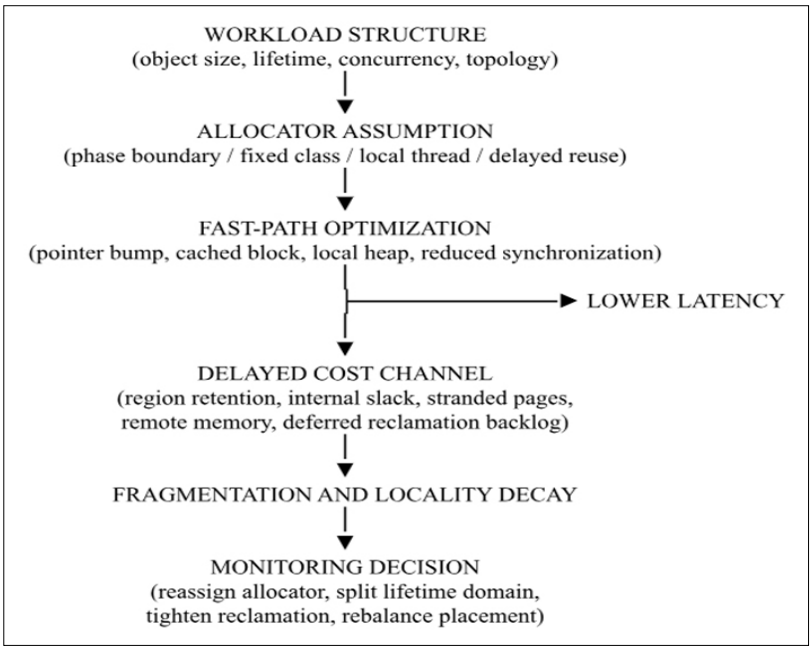

Figure 1. Scheme of the latency–fragmentation mechanism in specialized C++ allocators (compiled by the author based on Maas et al., 2021; Yang et al., 2023; Zhou et al., 2024; Reitz et al., 2024)

Fragmentation then grows when those same mechanisms strand memory across pages, nodes, or caches that the allocator does not rebalance with enough speed or precision (Maas et al., 2021, Yang et al., 2023, Zhou et al., 2024). Table 1 condenses that relation. Allocator patterns can be arranged along two connected axes: predictability of object lifetime and dispersion of reclamation. High lifetime predictability supports arena and stack-like strategies because these designs minimize path length and metadata churn. Once predictability breaks down and object lifetimes interleave, the system shifts toward pools, segregated lists, distributed heaps, or reclamation-assisted designs.

Table 1. Architectural alignment between lifetime regularity, allocation path length, and fragmentation exposure (developed by the author based on Maas et al., 2024; Yang et al., 2023; Zhou et al., 2024)

Lifetime regularity	Dominant allocator pattern	Fast-path latency tendency	Main fragmentation risk
Phase-bounded and predictable	Bump, arena, stack	Minimal	Region retention under mismatch
Repetitive size classes with stable reuse	Pool	Very low	Internal slack across classes
Mixed sizes and mixed lifetimes	Segregated lists	Low to moderate	Page breakage and stranded free space
Thread-local bursts under concurrency	Distributed heaps, per-thread caches	Low on hot path	Memory stranded across threads or CPUs
Deferred reclamation after unlinking	Reclamation-assisted allocation	Moderate	Temporary memory inflation before safe reuse

The figure replaces the idea of a universally fast allocator with a conditional map of architectural fit. Latency and fragmentation do not sit on separate scales. Engineers link them through the same design decision, since a short allocation path always carries an implied reuse pattern that unfolds later in time (Maas et al., 2024, Yang et al., 2023, Zhou et al., 2024).

Concurrency changes the structure of the trade-off once again. Thread-local caches and distributed heaps reduce contention because most allocations no longer touch a shared global structure. That design improves scalability, while the memory system pays through a different channel. Idle or lightly loaded threads can hold reusable memory that other threads cannot reach without extra transfer cost. NextGen-Malloc pushes this logic further and argues for allocator management outside the main processing cores. NUMAlloc binds allocation to NUMA locality in order to reduce remote traffic. These studies converge on one architectural lesson. Teams sustain low-latency concurrent allocation through decentralization, and that same decentralization raises the chance that free space remains partitioned across ownership domains before the system can reuse it at a global level (Li et al., 2023, Yang et al., 2023). A comparative scoring of allocator patterns by latency efficiency and fragmentation exposure is presented below (Figure 2).

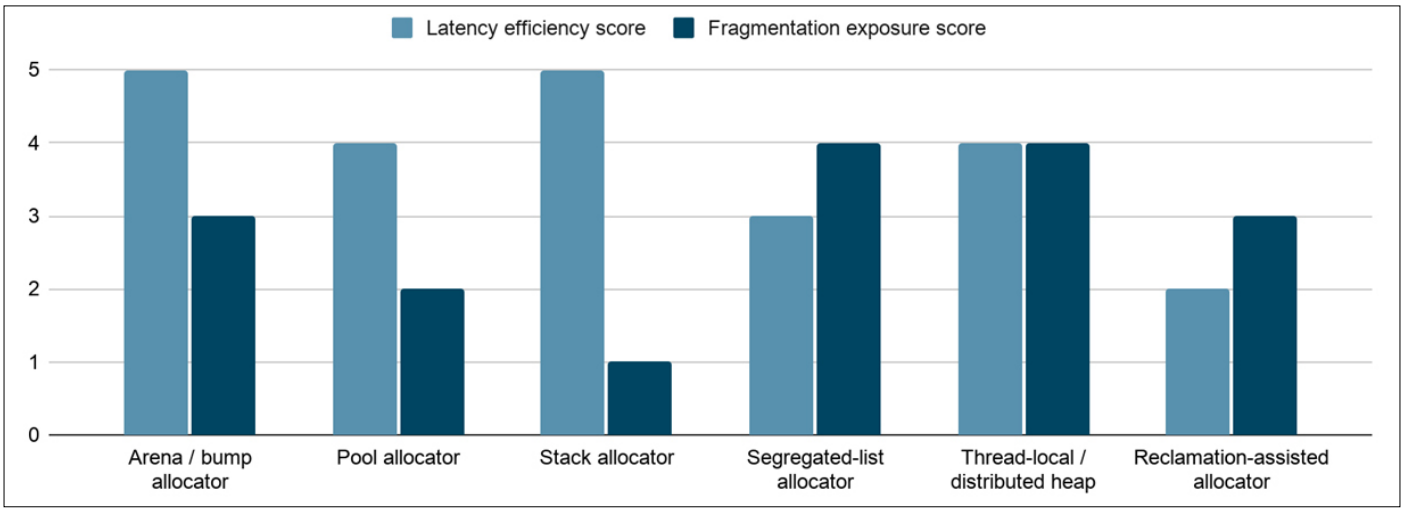

Figure 2. Comparative scoring of allocator patterns by latency efficiency and fragmentation exposure (compiled by the author based on Maas et al., 2021; Maas et al., 2024; Yang et al., 2023; Zhou et al., 2024; Moreno and Rocha, 2023; Singh et al., 2024; Dang et al., 2024)

The core/memory module reflects this trade-off by treating thread-local caches as controlled leakage zones. Memory remains local for latency reasons, while periodic consolidation paths prevent unbounded accumulation across threads. Locality-aware allocation extends the analysis beyond allocator internals and into data layout. MaPHeA shows that engineers can guide heap placement through memory-hierarchy behavior rather than block availability alone. Once allocation participates in placement policy, the allocator starts shaping layout itself. That shift matters for C++ because allocator choice then affects cache residency, TLB behavior, and traversal cost even when the abstract data structure stays the same. Performance gains that engineers often attribute to a "better allocator" may instead arise from a more coherent physical arrangement of objects that the program accesses together (Oh et al., 2023, Zhou et al., 2024).

Mixed-lifetime workloads expose another limit of allocator specialization. Once unlinking an object and reusing its storage no longer happen at the same moment, reclamation policy enters allocator architecture as an active design variable. Hybrid optimistic-access reclamation and neutralization-based reclamation address the same structural tension in concurrent unmanaged systems. Teams need memory to return fast enough to prevent inflation, while stale references still require protection against premature reuse. In these designs, latency and fragmentation acquire a third companion, namely reclamation lag. The front edge of the allocator may look fast, while the system accumulates retired but unreclaimed memory behind it. Under those conditions, observed fragmentation partly reflects the safety protocol rather than a weakness in the allocation path alone (Moreno & Rocha, 2023, Singh et al., 2024). In the module architecture, reclamation is not treated as an afterthought layered on top of allocation. It is modeled as part of the allocation contract itself, where some allocators guarantee delayed reuse and others enforce immediate recycling according to the safety constraints of the data structure.

Verification-oriented work sharpens that point from another angle. StarMalloc shows that allocator design now often combines safety features, metadata separation, quarantine logic, and concurrency support inside the allocator core. Each added safeguard changes hot-path shape, metadata volume, and reclaim timing. That detail matters for architectural analysis because specialized allocators no longer differ only by free-list policy or region discipline. Engineers now

build compound mechanisms in which latency, safety, fragmentation, and auditability press against one another within the same implementation (Reitz et al., 2024).

A final comparison clarifies where specialization succeeds and where it begins to fail. ExGen-Malloc revisits single-threaded allocator design and shows that metadata and control logic introduced for multithreaded generality impose measurable cost once concurrency is absent. When that result is read together with warehouse-scale and NUMA-oriented studies, a clear pattern appears. Specialization helps when it removes machinery that the workload will never use. The same move becomes costly once specialization strips away mechanisms that the workload will later require. Arena allocation without bounded phases, per-thread caches without stable parallel reuse, or size segregation without temporal regularity all fit that pattern. The architectural question therefore concerns the durability of assumptions. Teams need to know which assumptions about lifetime, concurrency, and reuse can remain trustworthy long enough for the chosen allocator to stay coherent over time (Li et al., 2025, Zhou et al., 2024, Yang et al., 2023).

DISCUSSION

The reviewed material supports a practical decision logic in which allocator selection starts from temporal structure, moves next to concurrency, and only then turns to raw allocation speed. In applied C++ development, the first design question concerns the shape of object lifetimes. Teams need to determine whether objects are phase-bounded, class-repetitive, weakly predictable, or densely interleaved. That distinction guides the allocator toward one of several roles. One design minimizes path length. Another limits fragmentation inside size classes. A third preserves placement locality across nodes. A fourth delays reuse until safety conditions are satisfied. In the core/memory design, this selection logic is implemented as an explicit mapping between workload categories and allocator entry points. Engineers do not choose allocators at call sites arbitrarily. They route allocations through domain-specific interfaces that already encode lifetime expectations. Table 2 organizes that mapping and compares allocator patterns through architectural fit instead of benchmark rank.

Table 2. Allocator pattern selection by workload structure (compiled by the author based on Maas et al., 2021, 2024; Oh et al., 2023; Yang et al., 2023; Zhou et al., 2024)

Workload condition	Preferred allocator pattern	Expected advantage	Likely architectural cost	Suitable deployment logic
Strict phase boundaries and batch reset	Arena or bump allocator	Minimal allocation overhead	Coarse reclamation and retention under lifetime drift	Use for request-scoped, frame-scoped, or transaction-scoped object graphs
Fixed-size recurrent objects	Pool allocator	Stable low latency and bounded external fragmentation	Internal slack when class mix drifts	Use for nodes, messages, handles, and stable object classes
Nested call depth with disciplined unwind	Stack allocator	Cheap allocation and release symmetry	Very low tolerance for ownership escape	Use where lexical lifetime mirrors runtime lifetime
Mixed sizes with moderate reuse regularity	Segregated-list allocator	Good balance between reuse speed and flexibility	Metadata growth and page breakage	Use as a general specialized heap with tuned class boundaries
Topology-sensitive multithreading	NUMA-aware or distributed heap allocator	Reduced contention and improved locality	Stranded free space across threads or nodes	Use when thread and node affinity remain stable
Concurrent unlink and delayed safe reuse	Reclamation-assisted allocator	Safe reuse in non-blocking structures	Memory inflation during grace periods	Use when correctness constraints dominate immediate reuse

The table reveals a consistent rule. Faster allocation paths appear once the allocator is allowed to ignore part of the future. Arena allocators ignore fine-grained deallocation. Pools ignore cross-class flexibility. Distributed heaps ignore global symmetry of reuse. Reclamation-assisted systems ignore immediate recycling. Each omission removes work from the hot path, while each one shifts cost into another layer of the system. The implementation decision therefore depends on identifying where delayed cost remains acceptable and where it threatens the dominant workload.

Production monitoring creates a second practical issue. Many allocator decisions fail late. Code remains stable, short tests may even show strong latency, and only prolonged execution reveals page breakage, stranded spans, remote access drift, or retired-memory accumulation. A useful observability layer for allocator integration in C++ therefore needs to stay compact and still capture the structural symptoms of mismatch. Table 3 outlines a monitoring model that connects allocator choice to the signals worth tracking during deployment.

Table 3. Monitoring metrics for allocator fit in production-oriented C++ systems (compiled by the author based on Maas et al., 2021, 2024; Moreno & Rocha, 2023; Singh et al., 2024; Reitz et al., 2024; Dang et al., 2024)

Metric	What it captures	When it becomes decisive	Interpretation rule
Allocation latency distribution	Hot-path stability under load	Early integration and regression testing	Rising tail latency points to refill pressure, contention, or metadata expansion
Live to reserved memory ratio	Effective retention and hidden slack	Long-running services	Divergence indicates fragmentation or delayed reclamation
Reuse delay	Time between free eligibility and actual reuse	Concurrent or deferred reclamation designs	Persistent growth signals safety overhead dominating allocator efficiency
Span or page utilization	Packing quality inside large backing units	Huge-page and segregated-list systems	Low utilization reveals lifetime mismatch inside shared regions
Remote allocation or access share	NUMA placement quality	Multisocket deployments	Growth implies topology-blind placement or thread migration
Cache or TLB miss sensitivity after allocator swap	Layout impact of placement policy	Data-intensive code paths	Improvement or regression indicates allocator-driven layout change
Metadata footprint	Structural overhead of allocator design	Specialized heaps with rich control logic	High footprint can erase nominal fast-path gains
Quarantine or deferred free backlog	Safety-related memory delay	Hardened or reclamation-assisted allocators	Backlog growth shows safety policy converting into memory inflation

These metrics support a staged implementation model. The first stage classifies lifetime regularity and concurrency shape from the application architecture. The second stage selects the narrowest allocator pattern that still matches that structure. The third stage instruments the signals tied to the chosen risk surface. The fourth stage revisits the decision once lifetime assumptions drift, which often happens after feature growth. This sequence helps teams avoid a common engineering mistake. Many projects adopt an aggressively specialized design at the start and then force later workload complexity into an allocator architecture built for a simpler temporal regime.

The same module uses these metrics as guardrails against architectural drift. When live-to-reserved ratios diverge or reuse delay grows, the system signals that lifetime assumptions encoded in allocator choice no longer match actual object behavior. At that point, allocator reassignment becomes a design task rather than a low-level optimization.

A durable C++ memory subsystem usually combines a small number of patterns and assigns each one a clear boundary of use. Arena allocation serves phase-bounded object graphs. Pools serve repetitive stable classes. A segregated or topology-aware heap backs irregular allocations. Reclamation protocols protect non-blocking structures where immediate reuse would be unsafe. The practical gain comes from separating incompatible lifetime regimes before they collapse into the same heap path and begin to interfere with one another through hidden retention, weak locality, or delayed reuse.

CONCLUSION

The analysis establishes that specialized C++ allocators work best when they are treated as temporal architectures of reuse. Their performance profile depends on how closely internal discipline matches the lifetime structure of allocated objects. Fast-path efficiency remains stable under bounded phases, stable size classes, or explicit ownership domains. That stability weakens once those regularities disappear.

The reviewed studies also show that fragmentation does not stand apart as an isolated memory metric. It grows from the same architectural choices that reduce latency, including bulk reclamation, size segregation, per-thread caching, topology-sensitive placement, and delayed reuse. Allocator evaluation therefore becomes more informative once lifetime semantics, placement policy, and reclamation lag are read together.

The core/memory module illustrates how these principles can be embedded directly into system structure, where allocator choice acts as a boundary that constrains object lifetime and prevents uncontrolled interaction between incompatible allocation regimes.

The hypothesis is confirmed. Low latency remains sustainable only while allocator assumptions stay aligned with the real workload over time. Once alignment breaks, the system repays the early speed gain through retention, locality loss, stranded memory, or delayed reuse. The practical implication follows from that result. Engineers should design memory architecture in C++ as a controlled composition of allocator patterns, with each pattern attached to a recognizable lifetime regime and monitored through structure-sensitive metrics.

REFERENCES

1. Dang, Z., He, S., Zhang, X., Hong, P., Li, Z., Chen, X., Song, H., Sun, X.-H., & Chen, G. (2024). PMAlloc: A holistic approach to improving persistent memory allocation.

ACM Transactions on Computer Systems, 42(3–4). https://doi.org/10.1145/3643886

2. Li, R., John, L. K., & Yadwadkar, N. J. (2025). Old is gold: Optimizing single-threaded applications with ExGen-Malloc. *IEEE Computer Architecture Letters, 24*(2), 225–228. https://doi.org/10.1109/LCA.2025.3587582
3. Li, R., Wu, Q., Kavi, K., Mehta, G., Yadwadkar, N. J., & John, L. K. (2023). NextGen-Malloc: Giving memory allocator its own room in the house. In *Proceedings of the ACM*. Association for Computing Machinery. https://doi.org/10.1145/3593856.3595911
4. Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., & Raffel, C. (2024). Combining machine learning and lifetime-based resource management for memory allocation and beyond. *Communications of the ACM, 67*(4). https://doi.org/10.1145/3611018
5. Maas, M., Kennelly, C., Nguyen, K., Gove, D., McKinley, K. S., & Turner, P. (2021). Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the ACM*. Association for Computing Machinery. https://doi.org/10.1145/3459898.3463905
6. Moreno, P., & Rocha, R. (2023). Releasing memory with optimistic access: A hybrid approach to memory reclamation and allocation in lock-free programs. In *Proceedings of the ACM* . Association for Computing Machinery. https://doi.org/10.1145/3558481.3591089
7. Oh, D.-J., Moon, Y., Ham, D. K., Ham, T. J., Park, Y., Lee, J. W., Ahn, J. H., & Lee, E. (2023). MaPHeA: A framework for lightweight memory hierarchy-aware profile-guided heap allocation. *ACM Transactions on Architecture and Code Optimization, 22*(1). https://doi.org/10.1145/3527853
8. Reitz, A., Fromherz, A., & Protzenko, J. (2024). StarMalloc: Verifying a modern, hardened memory allocator. *Proceedings of the ACM on Programming Languages, 8*(OOPSLA2). https://doi.org/10.1145/3689773
9. Singh, A., Brown, T. A., & Mashtizadeh, A. J. (2024). Simple, fast and widely applicable concurrent memory reclamation via neutralization. *IEEE Transactions on Parallel and Distributed Systems, 35*(2), 203–220. https://doi.org/10.1109/TPDS.2023.3335671
10. Yang, H., Zhao, X., Zhou, J., Wang, W., Kundu, S., Wu, B., Guan, H., & Liu, T. (2023). NUMAlloc: A faster NUMA memory allocator. In *Proceedings of the ACM* . Association for Computing Machinery. https://doi.org/10.1145/3591195.3595276
11. Zhou, Z., Gogte, V., Vaish, N., Kennelly, C., Xia, P., Kanev, S., Moseley, T., Delimitrou, C., & Ranganathan, P. (2024). Characterizing a memory allocator at warehouse scale. In *Proceedings of the ACM*. Association for Computing Machinery. https://doi.org/10.1145/3620666.3651350