International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Formalization of Experimental Methods for Evaluating Memory Allocators, Taking into Account the Semantics of the Object Lifecycle

Maksim Martynov*

Lead Programmer, Playrix, Novi Sad, Republic of Serbia
Email: m555ax@icloud.com

Abstract

Memory allocator evaluation still lacks a stable methodological basis, even though allocator behavior depends on workload heterogeneity, concurrency patterns, and the temporal semantics of allocated objects. This article addresses that gap through a formal analytical framework that treats object lifetime as an explicit experimental variable instead of a residual statistic. The study systematizes recent allocator research, identifies points at which benchmark practice loses methodological precision, and proposes a phase-based evaluation model that links allocation events with lifecycle classes, workload transitions, and metric interpretation. The presented materials include 11 recently peer-reviewed sources on allocator characterization, lifetime profiling, semantics-aware allocation, tiering, and benchmark methodology. Comparative analysis, conceptual synthesis, typologization, and analytical generalization shape the methodological basis. The analytical section derives a lifecycle-centered evaluation schema, distinguishes workload classes that require different observables, and formulates reporting rules for reproducible allocator assessment in server workloads, managed runtimes, heterogeneous-memory systems, and application-specific environments.

Keywords: memory allocator; benchmarking methodology; object lifetime; lifecycle semantics; workload characterization.

Received: *5/2/2026*
Accepted: *7/2/2026*
Published: *7/12/2026*

---

* *Corresponding author*.

1. Introduction

Memory allocator research has reached a stage where raw speed comparison no longer carries enough explanatory weight for a strong academic evaluation. The obstacle comes from the structure of the workloads that allocators serve. Allocation size, thread locality, object reuse, delayed reclamation, transfer across threads, phase shifts in application behavior, and long-tail object survival all shape allocator behavior. Many experimental setups still fold these variables into one flat benchmark result. Under that condition, one allocator can appear preferable in one paper and unconvincing in another, even if neither study contains an error.

This article aims to formalize experimental methods for evaluating memory allocators by making object lifecycle semantics explicit in experimental design, metric selection, and interpretation. Three tasks define the analytical path. The first task reconstructs the ways in which recent studies already organize allocator evaluation around lifetime structure, phase behavior, and workload realism, even when they do not name those categories directly. The second task identifies the distortions that arise when lifetime semantics remain hidden behind aggregate metrics. The third task develops a phase-based formalization in which allocation events are interpreted through lifecycle classes and through transitions between operational phases of the workload.

The novelty lies in moving the center of evaluation away from allocator internals taken in isolation and toward the relation between allocator policy and object existence over time. The working hypothesis states that allocator experiments become more reproducible, more comparable across runtime environments, and more useful for design decisions when researchers model object lifetime as a structured semantic variable with phase-sensitive observables instead of reducing it to average allocation statistics or terminal memory footprints.

2. Methods and materials

The source base was assembled through a structured narrative search rather than a full systematic review in ACM Digital Library, IEEE Xplore, publisher-hosted conference proceedings, Google Research publication pages, arXiv, and journal platforms that index recent work on allocator evaluation, object lifetime profiling, benchmark methodology, and workload characterization. The searches were conducted in April–May 2026. The search combined terms such as “memory allocator evaluation,” “object lifetime profiling,” “allocator benchmark methodology,” “heterogeneous memory tiering,” “allocation characterization,” and “semantics-informed allocator,” along with additional combinations including “memory allocator workload characterization,” “persistent memory allocation,” “producer-consumer allocator,” and “memory management benchmark methodology.”

The inclusion criteria were publication date from 2024 to 2026, direct relevance to allocator evaluation or allocation profiling, attention to object lifetime or workload structure, and the presence of a reusable methodological idea, evaluation logic, workload characterization principle, or reporting implication. Sources were retained when they provided enough technical detail to support analytical comparison across allocator behavior, lifecycle semantics, metric selection, or benchmark design. Duplicate records, papers centered on adjacent memory topics without an allocator-evaluation method, and studies that did not provide a reusable experimental

logic were removed during screening. The exclusion criteria removed papers focused only on low-level optimization without evaluation implications, studies centered on storage or operating-system mechanisms without allocation semantics, short abstracts without reproducible methodological content, and sources whose findings could not be connected to phase behavior, lifecycle classification, metric interpretation, or reporting practice.

Screening was performed in two stages. At the first stage, titles, abstracts, keywords, and venue descriptions were reviewed for topical relevance. At the second stage, full texts were examined to determine whether each source contributed to at least one of five analytical categories: allocator characterization in production and large applications, direct profiling of object lifetime, benchmark methodology for memory-managed runtimes, semantics-aware allocator design, or environment-specific evaluation in persistent and heterogeneous memory systems. The final corpus contains 11 sources published from 2024 to 2026. These works cover five connected groups of questions: allocator characterization in production and large applications, direct profiling of object lifetime, benchmark methodology for memory-managed runtimes, semantics-aware allocator design, and environment-specific evaluation in persistent or heterogeneous memory systems.

The study uses comparative analysis to identify recurring experimental structures, source analysis to isolate methodological assumptions, conceptual synthesis to unify them into one analytical vocabulary, typologization to distinguish classes of workloads and observables, and analytical generalization to derive a formal evaluation model aligned with the three research tasks stated in the introduction. To improve reproducibility of the analytical procedure, each source was coded according to four extraction fields: workload type, lifecycle signal, allocator-related mechanism, and reported or implied observable. Workload type recorded whether the study addressed server-scale production behavior, managed-runtime benchmarking, producer-consumer communication, single-threaded execution, browser workloads, mobile environments, persistent memory, heterogeneous memory, or semantics-informed allocation. Lifecycle signal recorded references to object lifetime, retention, ownership transfer, phase transition, delayed reclamation, placement, reuse, or profiling fidelity. Allocator-related mechanism described the allocator policy, placement logic, cache behavior, metadata organization, page-release mechanism, or semantic classification principle under analysis. The observable field recorded which metrics or reporting units were treated as meaningful: allocation latency, throughput, RSS, peak memory, fragmentation, page retention, reclaimable memory, transfer cost, tier migration, NUMA locality, profiling overhead, or class-specific overhead.

3. Results

Recent allocator studies do not support the idea that one benchmark suite and one fixed metric set can provide a stable basis for comparison. A more complex picture emerges once the literature is read through the lens of workload structure. Production-scale allocator characterization shows that object sizes, lifetime distributions, and platform sensitivity vary across real deployments even inside one allocator family. In the warehouse-scale study of TCMalloc, engineers had to account for workload diversity at the level of cache hierarchy behavior and allocator redesign choices. Their conclusions depended on size and lifetime distributions, not on one summary indicator alone [1]. A related pattern appears in large-application profiling. The object-centric characterization of

Chromium exposed marked variation in lifetime events, type distributions, reuse, and memory activity across user-facing workloads, while measurement overhead itself became part of the methodological problem [2]. These studies matter for the present article because they shift the analytical focus. Workload semantics vary across phases and object classes. An allocator experiment that leaves that structure implicit will mix allocator policy with workload accident.

A second group of studies brings lifetime into the foreground and exposes where common evaluation practice remains underspecified. Lifetime-aware allocation already functions as an operational design principle in recent research. The LLAMA work links predicted object lifetime with placement decisions that reduce fragmentation under huge pages, which means lifetime shapes allocator quality in a direct way and not as a descriptive afterthought [3]. Research on finalization-based object lifetime profiling in managed environments reaches a related conclusion from the measurement side. That work shows that researchers can derive actionable lifetime traces, yet it also shows that the profiling method itself changes the boundaries of what counts as observable lifetime and how closely that observation matches optimization targets [4]. Read together, these contributions point to one methodological consequence. Allocator experiments lose explanatory power when lifetime appears only as a post hoc statistic.

The same point becomes sharper in work that injects semantic information into allocator design. SeMalloc assigns heap objects through a thread-, context-, and flow-sensitive semantic model, so the allocation decision depends on richer descriptors than size class alone [5]. The security orientation of that paper does not reduce its value for allocator methodology. Its broader contribution lies in the change of unit analysis. Once researchers classify allocation events by semantic type, benchmark interpretation must distinguish at least four dimensions: allocation demand, semantic class, lifetime trajectory, and cross-thread movement. Averages that pool those dimensions obscure the mechanism under study.

Benchmark-methodology papers help explain why allocator results often resist comparison across studies. The recent reassessment of Java performance analysis argues that benchmark suites age, drift away from current workloads, and require explicit methodological renewal through richer characterization and integrated analysis frameworks [6]. Research on memory management in mobile devices reaches a parallel conclusion under different operating constraints. Mobile environments complicate principled evaluation because heterogeneity, external services, and the weak supply of common open benchmarks inject noise that server-style methods do not absorb well [7]. These papers do not study allocators in isolation. Their relevance lies elsewhere. They show that the benchmark environment shapes the experiment as much as the allocator itself.

Coarse workload labels create another methodological weakness. Terms such as “server,” “single-threaded,” or “memory-intensive” often carry more weight in allocator papers than they can support. Recent work shows why that vocabulary falls short. ExGen-Malloc revisits allocator design for single-threaded applications and ties allocator impact to cache behavior and overall application performance, which means concurrency class alone cannot organize a reliable taxonomy of benchmarks [8]. BatchIt studies producer-consumer workloads and demonstrates that allocator behavior shifts when allocation and deallocation occur in communicating threads rather than inside one local control path. In that case, transfer structure becomes an experimental variable in its

own right [9]. Work on object tiering in heterogeneous memory systems arrives at the same destination through another route. Allocation outcomes depend on object records, monitoring triggers, tier assignment logic, and workload phase shifts, not on nominal memory pressure in isolation [10]. Persistent-memory allocation studies reinforce the need for environment-specific tags, because metadata placement, slab conversion, NUMA sensitivity, and deallocation structure all change what the experiment is measuring [11]. Across these sources, one defect recurs. The benchmark name often stands in for the workload description, while the real experimental object is the interaction between allocator policy and lifecycle structure.

A stronger formalization therefore needs a different organizing unit. The proposed model starts from lifecycle semantics. Each object belongs to a lifecycle class described through three coordinates: persistence horizon, ownership path, and reclamation mode. Persistence horizon separates ephemeral, medium-lived, persistent, and phase-residual objects. Ownership path identifies whether one thread creates and reclaims the object, whether the object moves across threads, whether shared pools hold it, or whether framework-level containers retain it. Reclamation mode distinguishes immediate free, batched free, deferred free, runtime-managed reclamation, and reclamation blocked or delayed by indirection layers such as pooling or tiering. Experimental design then groups allocator behavior by phases rather than by program runs taken as undivided wholes. A phase marks a bounded interval in which the distribution of lifecycle classes stays stable enough for the selected metrics to carry clear meaning. Phase boundaries arise from shifts in request mix, thread topology, working-set temperature, object-transfer intensity, or retention pressure. Figure 1 summarizes the proposed relation between workload phases, lifecycle classes, allocator actions, and observable outcomes.

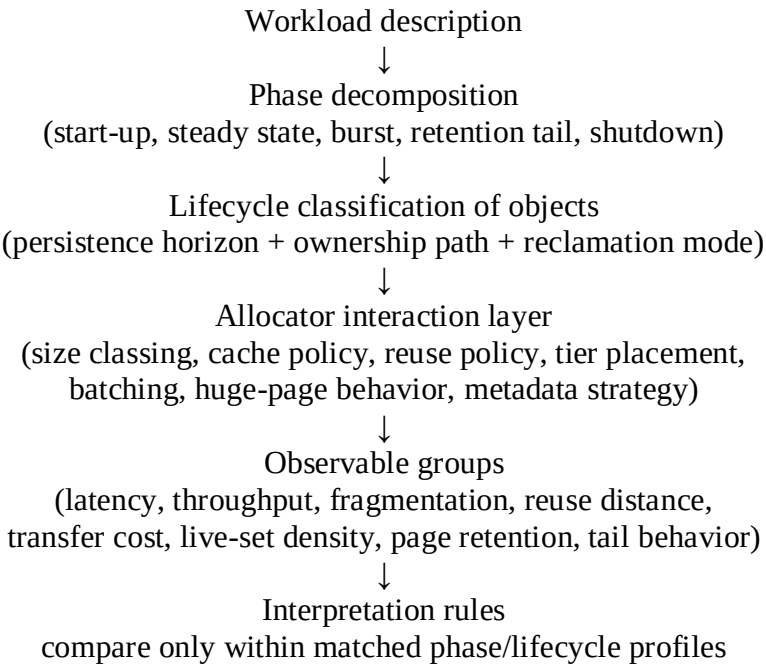

Figure 1: Proposed phase-based formalization of allocator evaluation with explicit object lifecycle semantics (adapted by the author from [1–4])

This figure changes the logic of comparison. Researchers need to compare allocators inside matched phase and lifecycle profiles first, then across phase transitions, and only after that at the whole-program level. Recent literature supports that sequence from several directions. The warehouse-scale characterization of TCMalloc documents wide lifetime diversity across production workloads [1]. LLAMA ties predicted survival time to fragmentation behavior under huge pages [3]. Finalization-based profiling shows that lifetime tracing itself requires methodological care [4]. Chromium characterization demonstrates that long-lived containers can keep pages' resident even when much of their content no longer serves active work [2]. Taken together, these studies support a clear inference for the first research task. Current allocator evaluation already depends on lifecycle structure, even where papers leave that dependence unnamed.

The second research task concerns the distortions that appear once lifetime semantics remain hidden. Four distortions stand out. Aggregate throughput hides lifecycle mismatch. An allocator tuned for ephemeral objects can post strong numbers during a short burst and weak numbers during a retention tail, even though each result reflects a different operational condition rather than a contradiction. RSS and peak memory hide page-retention mechanisms. Reuse, deferred reclamation, and pool lifetime can keep pages resident long after logical object death, so raw footprint loses interpretive value without lifecycle classes. Concurrency labels hide ownership transitions. Producer-consumer transfer, shared pools, and thread-local reclamation impose different stresses even when request sizes look similar. Benchmark portability hides environment semantics. A benchmark moved from a server suite into a mobile runtime or a heterogeneous-memory platform no longer preserves the same experimental meaning because instrumentation cost, page policy, and allocator hooks shift the measurement target itself [6,7,9,10]. These distortions arise before measurement output reaches the page. They enter at the point where researchers construct the experimental object.

The third research task calls for a formal reporting protocol that future allocator studies can adopt without forcing one allocator architecture, runtime model, or benchmark suite. The proposed scheme proceeds through five stages. Researchers first describe the workload through phase decomposition instead of using a benchmark label as a sufficient descriptor. They then map sampled allocations into lifecycle classes through profiling, code inspection, trace reconstruction, or application-level ownership knowledge. The third stage aligns each reported metric with the lifecycle class and phase for which that metric has interpretive validity. The fourth stage separates within-phase behavior from transition costs such as burst entry, cache refill, tier migration, or reclamation lag. The fifth stage reports whole-program outcomes only after profile-matched comparisons are complete. This scheme does not erase allocator diversity. It turns diversity into something that researchers can interpret, compare, and reuse across studies.

4. Discussion

The analytical reconstruction of recent literature leads to a practical conclusion that reaches beyond allocator engineering. Experimental rigor in this field depends on whether the study declares the lifecycle situation that a reported number belongs to. Once object existence over time enters the design of the experiment as an explicit variable, allocator evaluation starts to resemble protocol specification. Researchers define states, transitions, observables, and limits of comparison. Without that structure, reproducibility weakens even if the measurement

process itself looks careful.

The implementation model proposed here keeps the entry barrier low enough for real research practice. The proposed protocol can be illustrated through a controlled reporting scenario involving two allocator strategies under one workload divided into startup, burst, steady-state processing, and retention-tail phases. The example is not presented as an independent benchmark. Its purpose is to show how the proposed framework changes the interpretation of conventional allocator metrics when phase and lifecycle labels are added to the report.

In a traditional aggregate comparison, allocator A may appear preferable if the final table privileges mean allocation latency and burst throughput. Allocator B may appear preferable if the same workload is judged by retained RSS or peak memory after the active request stream has declined. Under a flat reporting regime, these results look like competing conclusions. Under the lifecycle-centered protocol, they describe different experimental objects. The burst phase is dominated by ephemeral objects and allocator-cache refill behavior. The retention tail is dominated by phase-residual objects, delayed reclamation, page release policy, and container-level retention. A single whole-program score therefore collapses two allocator properties that should be interpreted separately.

The lifecycle-centered report would first mark the phase boundary, then identify the dominant object classes inside each interval, and only after that bind metrics to their valid interpretive domain. Allocation latency and throughput would be reported for the burst phase together with the share of short-lived objects and the ownership path of allocations. RSS, reclaimable-page structure, and retained live-set density would be reported for the retention tail together with information about phase-residual objects and delayed free behavior. The comparison would not ask which allocator is “better” in the abstract. It would specify which allocator behavior is preferable under a defined lifecycle profile.

This illustrative application clarifies the operational value of the framework. The same raw metrics remain available, yet their meaning changes once the workload is decomposed into phases and object lifecycle classes. Aggregate throughput explains short-lived pressure only inside the phase where short-lived objects dominate. Peak memory explains retention only when paired with information about object survival and page release. The framework therefore does not add a decorative taxonomy to allocator benchmarking. It prevents misinterpretation by assigning each metric to the phase and lifecycle structure in which that metric has methodological validity. A full predictive allocator or a heavy tracing framework is not required at the starting point. A research team can begin with a three-layer evaluation pipeline. The first layer isolates execution phases. The second assigns lifecycle tags to object groups at the highest fidelity that the environment permits. The third binds each metric to a justified interpretive domain. Under that discipline, a fragmentation number does not appear alone. The authors report it together with the phase in which it was measured, the dominant lifecycle mix at that point, and the ownership pattern that shaped reuse or retention. The reporting format becomes more elaborate, yet the gain in interpretability repays that cost.

Table 1: Lifecycle-centered comparison of methodological regimes for allocator evaluation (compiled by the author on the basis of [1,3,4,6,7])

Methodological regime	Primary unit of comparison	Strength	Blind spot	Recommended use
Aggregate benchmark regime	Whole-program score	Fast screening and simple reporting	Conceals phase shifts and lifecycle heterogeneity	Preliminary comparison
Micro-workload regime	Isolated allocation pattern	Sharp mechanism isolation	Weak ecological validity	Sensitivity testing
Profile-guided regime	Observed object groups	Connects policy with real traces	Depends on profiler fidelity	Mature workloads with stable instrumentation
Semantics-aware regime	Typed lifecycle class	High interpretive precision across policies	Requires richer workload knowledge	Design-oriented causal studies
Phase-based lifecycle regime	Phase and lifecycle profile together	Strong basis for cross-study comparison	Higher preparation cost	Publication-grade formal evaluation

This comparison clarifies why allocator papers often remain informative without accumulating into one coherent methodological line. The divergence does not come only from different allocators or different benchmark suites. It comes from the level at which each paper defines comparison. Once that point is made explicit, disagreement between studies becomes easier to interpret. In many cases, the studies are not refuting one another. They answer different experimental questions because they define the unit of comparison at different levels of abstraction.

A second implication concerns the sequence by which future studies choose observables. Many evaluation setups begin with a familiar metric set because those metrics fit standard tables and plots. A stronger logic starts elsewhere. Researchers first need to define the lifecycle failure they want to expose. After that, they select observables that make that failure visible. When page retention by mostly dead objects creates the concern, reuse density and reclaimable-page structure carry more explanatory weight than raw allocation latency. When cross-thread handoff creates the concern, transfer distance and remote reclamation behavior deserve central attention. When burst fragility becomes the question, phase-entry and phase-exit metrics need a visible place in the evaluation design. The method becomes sharper once the failure mode is declared before the chart is drawn.

Table 2: Decision matrix for selecting observables under the proposed formalization (compiled by the author on the basis of [2,5,8–11])

Experimental concern	Lifecycle signal to identify first	Core observables	Reporting rule
Short-lived burst pressure	High share of ephemeral objects	Allocation latency, cache refill cost, burst-tail latency	Separate warm entry from steady state
Cross-thread transfer	Ownership migration between producer and consumer	Transfer cost, remote free cost, queue contention	Report transfer topology with thread counts
Retention and fragmentation	Phase-residual objects that hold pages	Live-set density, reclaimable pages, page retention, RSS	Pair footprint with lifecycle composition
Heterogeneous or persistent memory placement	Long-lived objects with tier sensitivity	Tier migration cost, locality, metadata traffic, NUMA effects	State hardware policy and placement logic
Semantics-driven specialization or isolation	Distinct semantic classes at allocation sites	Class-specific overhead, false-sharing risk, isolation effect	Compare within matched semantic classes

This decision matrix gives the proposed framework operational shape. The sequence is direct. Researchers identify the experimental concern, infer the lifecycle signal that makes that concern visible, bind the metric set to that signal, and report results only within that declared interpretive frame. Under that logic, allocator benchmarking no longer splits into two unsatisfactory options, one built on standard suites and the other built on ad hoc traces. It becomes a controlled mapping between workload semantics and allocator observables.

The lifecycle-centered reporting protocol can be summarized as a checklist for future allocator studies:

1. State the allocator versions, runtime environment, operating system, hardware configuration, compiler or interpreter settings, and relevant allocator flags.
2. Describe the workload source, input size, execution mode, concurrency pattern, warm-up procedure, and number of repeated runs.
3. Decompose the workload into phases such as startup, burst, steady state, phase transition, and retention tail, and justify the boundary used for each phase.
4. Identify dominant lifecycle classes in each phase through profiling, tracing, source-level inspection, or documented application knowledge.
5. Report persistence horizon, ownership path, and reclamation mode for the object groups that shape allocator behavior.

6. Match each metric to the lifecycle signal it is meant to expose; do not report throughput, RSS, fragmentation, or peak memory as self-sufficient indicators.
7. Separate within-phase measurements from transition costs, including cache refill, remote free behavior, delayed reclamation, tier migration, and page-release lag.
8. Present aggregate whole-program metrics only after phase-specific and lifecycle-specific results have been reported.
9. State instrumentation limits, profiling overhead, unobservable lifetime boundaries, and any allocator hooks that may alter the measured behavior.
10. Provide enough configuration detail for another researcher to repeat the evaluation or to compare a new allocator under the same lifecycle profile.

The proposed formalization still faces a clear limitation. Lifecycle semantics are easier to define than to observe with perfect fidelity. In unmanaged languages, semantic reconstruction and profiling overhead remain demanding. In managed runtimes, the runtime mediates what the profiler can see and how that visibility maps onto object death or retention. In production systems, instrumentation can perturb retention or reuse processes that the study aims to measure. For that reason, the framework works best as a discipline for experimental design and reporting before it becomes a promise of full semantic transparency. That limit does not weaken the proposal. It marks the boundary within which the proposal remains academically honest and practically useful.

5. Conclusion

Allocator evaluation becomes methodologically stronger once researchers describe workloads through phases and classify allocated objects by lifecycle semantics before they compare performance numbers. Recent studies already imply that size distributions, reuse patterns, transfer structure, and long-tail retention shape allocator outcomes in ways that whole-program averages conceal.

The formalization proposed here addresses that problem by replacing flat benchmark comparison with phase-matched and lifecycle-matched interpretation. Metrics gain meaning from the operational conditions in which they remain valid. Allocator results then become more comparable across runtime environments, allocator families, and platform classes.

The hypothesis stated in the introduction receives support from the analytical reconstruction of recent literature. Explicit modeling of object lifecycle semantics provides a firmer basis for reproducible allocator experiments, clearer design conclusions, and more defensible comparison than evaluation schemes built around aggregate throughput and terminal memory footprint alone.

References

[1]. 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. *Association for Computing Machinery*. https://doi.org/10.1145/3620666.3651350

[2]. Upadhyay, S., & Venkat, A. (2026). Shiny objects: Object-centric characterization of Chromium. *ACM*

Transactions, 10(1). https://doi.org/10.1145/3788102

[3]. 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

[4]. Jordan Montaño, S., Polito, G., Ducasse, S., & Tesone, P. (2024). Evaluating finalization-based object lifetime profiling. *Association for Computing Machinery*. https://doi.org/10.1145/3652024.3665514

[5]. Wang, R., Xu, M., & Asokan, N. (2024). SeMalloc: Semantics-informed memory allocator. *Association for Computing Machinery*. https://doi.org/10.1145/3658644.3670363

[6]. Blackburn, S. M., Cai, Z., Chen, R., Yang, X., Zhang, J., & Zigman, J. (2025). Rethinking Java performance analysis. *Association for Computing Machinery*. https://doi.org/10.1145/3669940.3707217

[7]. Sareen, K., Blackburn, S. M., Hamouda, S. S., & Gidra, L. (2024). Memory management on mobile devices. *Association for Computing Machinery*. https://doi.org/10.1145/3652024.3665510

[8]. Li, R., & Yadwadkar, N. (2025). Old is gold: Optimizing single-threaded applications with Exgen-Malloc. *arXiv*. https://doi.org/10.48550/arXiv.2510.10219

[9]. Filardo, N. W., & Parkinson, M. J. (2024). BatchIt: Optimizing message-passing allocators for producer-consumer workloads: An intellectual abstract. *Association for Computing Machinery*. https://doi.org/10.1145/3652024.3665506

[10]. Kammerdiener, B., McMichael, J. Z., Jantz, M., Doshi, K., & Jones, T. (2025). Flexible and effective object tiering for heterogeneous memory systems. *ACM Transactions, 22*(1). https://doi.org/10.1145/3708540

[11]. 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, 42*(3–4). https://doi.org/10.1145/3643886