Tidying Up the Address Space

Vinay Banakar[1, 3], Suli Yang[3], Kan Wu[2*], Andrea C. Arpaci-Dusseau[1],
Remzi H. Arpaci-Dusseau[1], Kimberly Keeton[3]

[1]University of Wisconsin-Madison [2]xAI [3]Google

Abstract

Memory tiering in datacenters does not achieve its full potential due to hotness fragmentation—the intermingling of hot and cold objects within memory pages. This fragmentation prevents page-based reclamation systems from distinguishing truly hot pages from pages containing mostly cold objects, fundamentally limiting memory efficiency despite highly skewed accesses. We introduce address-space engineering: dynamically reorganizing application virtual address spaces to create uniformly hot and cold regions that any page-level tiering backend can manage effectively. HADES demonstrates this frontend/backend approach through a compiler-runtime system that tracks and migrates objects based on access patterns, requiring minimal developer intervention. Evaluations across ten data structures achieve up to 70% memory reduction with 3% performance overhead, showing that address space engineering enables existing reclamation systems to reclaim memory aggressively without performance degradation.

Keywords: Operating Systems, Memory Management, Memory Tiering, Virtual Address Space, Object Management

ACM Reference Format:
Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Kimberly Keeton. 2025. Tidying Up the Address Space. In *3rd Workshop on Disruptive Memory Systems (DIMES '25), October 13–16, 2025, Seoul, Republic of Korea.*ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3764862.3768179

1 Introduction

Memory is the most constrained and costly resource in modern datacenters, with utilization reaching 60-90% at hyperscalers like Google [47] and Meta [38] while driving 50% of server costs [34]. To control these mounting expenses, operators adopt *memory tiering* to move cold data from expensive DRAM to slower, cheaper storage [17, 31, 34, 38]. However, the effectiveness of tiering is fundamentally limited by a semantic gap between applications and the operating system. Applications operate on fine-grained data objects, while the OS manages memory in coarse-grained pages. This mismatch causes frequently accessed (hot) and infrequently accessed (cold) objects to become intermingled on the same pages—a problem we term *hotness fragmentation*. This fragmentation cripples page-level reclamation, as even a single hot object on a page prevents the entire page from being safely moved to a slower tier without risking performance-degrading page faults.

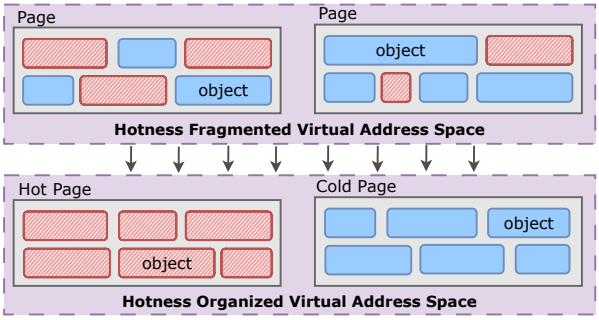


Figure 1. Address-Space Engineering. Hotness fragmentation (top) intermixes hot (red) and cold (blue) objects, making pages difficult to reclaim. Address-space engineering (bottom) reorganizes objects into uniformly hot and cold regions, enabling efficient page-level management.

Existing systems do not address the root cause of this problem. Page-level reclamation systems like the kernel's kswapd, Google's zswap [30], or Meta's TMO [49] are powerful but ignorant; they can only identify pages as hot or cold, not the objects within them, and thus cannot distinguish a truly hot page from a mostly-cold one. Allocation-time hinting approaches [27] are too static, making a one-time placement decision that cannot adapt as an object's access patterns change over its lifetime. While more radical object-level tiering systems [22, 44] offer fine-grained control, they impose high adoption costs by requiring significant application modifications and direct hardware access. Therefore, what is needed is a system that provides object-level awareness to guide page-level decisions, without abandoning the abstractions that make existing systems practical.

We propose **address-space engineering**: instead of making the OS object-aware, we engineer the application's virtual address space to be OS-tiering-friendly, adapting to workload access patterns. While garbage-collected languages have long employed object reorganization for locality [13, 25, 26, 50], unmanaged languages like C++ present unique challenges due

*Work done at Google

DIMES '25, October 13–16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2226-4/25/10
https://doi.org/10.1145/3764862.3768179

to the assumption that allocation-time decisions are final [9]. The key insight is to challenge this assumption for pointer-based data structures while preserving the language's semantic guarantees. As shown in Figure 1, this approach "tidies up" the fragmented address space by dynamically identifying and migrating objects based on their access patterns, transforming a fragmented layout, where hot and cold data are mixed, into a cleanly organized one, where objects are grouped by their access temperature. This continuous, dynamic engineering of the virtual address space is the key to bridging the semantic gap between application objects and OS pages.

Dynamic reorganization enables a powerful architectural principle: decoupling object-level placement from page-level reclamation. We propose a model with a frontend system responsible for making object placement decisions and a backend responsible for managing the underlying pages. The frontend provides the backend with an address space containing uniformly hot or cold regions, making the backend's job easier: it can confidently reclaim entire cold regions or promote hot regions to huge pages. This architecture achieves the intelligence of an object-level system while retaining the compatibility and simplicity of existing page-level systems.

In this paper, we present ***Hierarchically Aware Data structurES (HADES)***, a compiler-runtime system that realizes this frontend/backend vision for pointer-based data structures in non-managed languages. HADES transparently tracks object-level access intensity and uses a safe, lock-free protocol to migrate objects between hot and cold heaps. We demonstrate that by adding a HADES frontend to standard reclamation approaches like kswapd and proactive reclamation, we can reduce memory usage by up to 70% on YCSB workloads with minimal performance degradation (3-5%). This work makes the following contributions: (1) a decoupled frontend/backend model for memory tiering based on dynamic object reorganization; (2) the design and implementation of HADES, an object-level frontend; and (3) an evaluation demonstrating that this model allows existing, unmodified backends to reclaim memory far more effectively.

2 Semantic Gap in Memory Management

Real-world workloads exhibit highly skewed access patterns, with large portions of datasets remaining untouched [10, 35] or accessed only once [52]. Modern allocators [19, 54] optimize for allocation speed and spatial fragmentation reduction, making placement decisions at allocation time without considering future access patterns. Meanwhile, Linux's memory reclamation system operates at page granularity, using access bits in page table entries to track activity. This fundamental mismatch scatters frequently accessed objects across memory pages, intermixing them with rarely accessed data and creating a semantic gap between application-level object access and OS-level page management.

From a memory tiering perspective, the core inefficiency arises from the fragmentation of large regions of cold data by a few hot objects, which makes entire pages unreclaimable. This intermingling of hot and cold objects within the same pages is a problem we term *hotness fragmentation*. To quantify the extent of hotness fragmentation, we introduce the metric *Page Utilization*. Page Utilization is defined as the ratio of unique accessed bytes to the total size of accessed pages over a specific time period T:

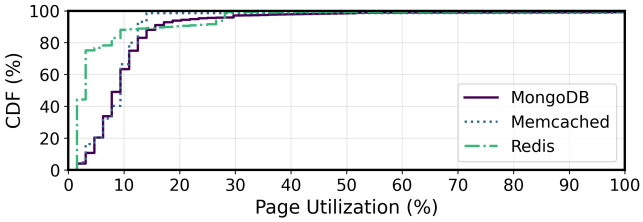


Figure 2. Page Utilization. Redis, Memcached, and MongoDB Page Utilization for 360s epochs running a YCSB-C (read-only) workload with a Zipfian distribution. A small fraction of bytes per page are accessed, and more pages are touched than necessary.

$$PageUtilization(T) = \frac{TotalUniqueBytes(T)}{UniquePages(T) \times PageSize}$$

A low Page Utilization value serves as a strong indicator of this problem: it signifies that only a small fraction of the bytes on an accessed page are actually being used. This forces the entire page to remain resident in the faster memory tier, trapping the large volume of co-located, un-accessed (and likely cold) data on the same page.

Our investigations reveal consistently low Page Utilization across diverse systems and workloads. Using PinTool [1] to track memory access patterns in popular key-value stores running YCSB workloads [15], we found that 75% of accessed pages in Redis utilize just 3% or less of their capacity, while 90% of pages in MongoDB and Memcached use less than 15% (Figure 2). This scattered placement creates an illusion of high memory activity when only a small fraction of bytes receives regular access, directly constraining the effectiveness of any page-based reclamation system.

The issue of non-uniform access within larger memory granularities is well-established. Cache prefetching studies [7, 28, 45], using techniques like spatial footprint bitmasks [7, 28] and block access counts per region [45], have consistently revealed that only a fraction of data within multi-kilobyte granularities is often hot. These studies confirm that fine-grained access locality exists, but their block-level metrics are primarily designed to identify specific cache blocks for prefetching.

To assess the impact of this fine-grained skew on OS-level memory tiering, we use *Page Utilization*. This metric quantifies the byte-level efficiency within the set of all unique OS pages touched, not just within a prefetch-specific region. By calculating the ratio of unique bytes accessed to the total size of these pages, Page Utilization directly measures the memory wasted due to "hotness fragmentation"—where sparsely located hot objects force entire pages to remain in the fast tier.

Unlike metrics aimed at prefetch accuracy, low Page Utilization serves as a direct proxy for the reclaimability challenges faced by page-based memory management, highlighting the potential benefits of improving the access uniformity of pages.

Penalties of Poor Page Utilization. Memory reclamation in modern operating systems requires completely cold pages, as even a single active object prevents the entire page from being swapped out. Figure 3 illustrates this problem with Redis running a YCSB-C workload, where despite requiring 1.2 GiB of resident memory, Redis actively touches only ~0.5 MiB of cache lines. Most pages contain at least one hot object but remain mostly unused, creating vast regions of theoretically reclaimable memory that current systems cannot efficiently recover. This leads to our **observation #1: improving Page Utilization increases reclaimable memory by reducing the number of pages needed to serve skewed workloads**.

Poor Page Utilization significantly impacts CPU performance by forcing hot objects to scatter across virtual address space. Processors must perform frequent TLB lookups and page table walks even for cache-friendly workloads, with TLB misses requiring 150-600 cycles compared to 4-cycle hits. At Google, 11% of fleet CPU cycles are consumed by dTLB load misses [54]. While transparent huge pages reduce TLB pressure, applying them indiscriminately increases memory footprint by up to 69% [29]. Object grouping creates dense regions ideal for targeted huge page promotion, preserving TLB efficiency without memory bloat. This leads to our **observation #2: improving Page Utilization enables targeted huge page promotion, preserving CPU cycles without sacrificing reclaimable memory**.

Poor address space layout forces datacenter infrastructure waste through memory overprovisioning and CPU underutilization. Operators must spread jobs with small working sets but large allocation footprints across multiple machines to avoid CPU stranding [47], even when actual memory needs are much smaller than allocated. With DRAM accounting for 50% of server costs [34] and producing 12x more emissions per bit than SSDs [39], reorganizing the address space by object access patterns creates a foundation for both memory efficiency and CPU optimization. This leads to our **observation #3: better address space layout enables both efficient huge page usage and memory reclamation, allowing stable skewed workloads to run on fewer machines for more cost-effective and environmentally sustainable datacenters**.

3 Workload-Optimized Address Space

The semantic gap between object-level allocation and page-level management demands a new approach to organizing the virtual address space. Instead of treating memory layout as a static side-effect of allocation, a system can dynamically engineer it to match workload behavior. This section outlines the core principles for designing such a system.

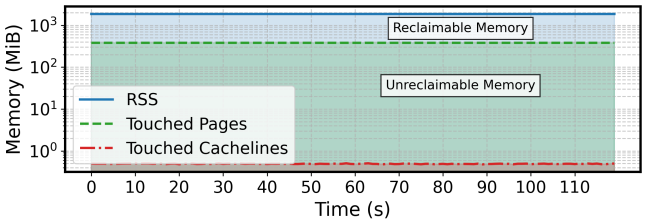


Figure 3. Unreclaimable Memory. YCSB-C with Zipfian distribution running on Redis shows memory used (RSS), pages needed (Touched Pages), and cachelines needed (Touched Cachelines). Only 0.5 MiB of actual data is required whereas 1.2 GiB remains resident. The gap represents theoretically reclaimable memory that current systems cannot efficiently recover.

3.1 Enabling Object Mobility

The first challenge in reorganizing the address space is that applications written in unmanaged languages like C++ assume object addresses are fixed after allocation. To enable dynamic reorganization, a system must be able to move objects without violating this stability assumption from the application's perspective. The most practical way to achieve this is to **focus on pointer-based data structures,** where data is accessed through pointers rather than by arithmetic offsets. By intercepting pointer dereferences, a system can track and update pointers when an object moves, making the relocation transparent to the application. This approach requires no code modifications from the developer, as it works directly with objects allocated via `new` or `malloc`. This approach must be safe in a highly concurrent environment, requiring a lock-free mechanism that guarantees correctness without acquiring coarse-grained locks. This ensures forward progress at all times and compatibility with the full spectrum of concurrency models, from simple locking to sophisticated lock-free algorithms.

Applying this principle is fundamentally harder in C++ than in managed environments like Java or .NET. Managed runtimes have built-in moving garbage collectors that provide the machinery to safely relocate objects [13, 26]. They can pause application threads at well-defined "safe-points" to scan the heap and atomically update all references to a moved object. C++ offers no such infrastructure. Therefore, the challenge is not the idea of moving objects, but introducing a mechanism to do so safely and concurrently in a non-cooperative environment. A system must be able to relocate an object while other threads may be actively trying to access it, without a "stop-the-world" pause and without breaking language semantics.

3.2 Grouping Objects by Access Intensity

Once objects can be moved safely, the system needs a policy to guide their placement. Relying on static, allocation-time placement [5, 14, 16] is insufficient, as individual objects follow their own unique access trajectories—some remain hot throughout their lifetime, others cool down quickly after initialization, and many transition between hot and cold states

as application phases change. An effective approach is to **dynamically group objects based on their observed access intensity,** a technique explored in managed runtimes for improving CPU cache locality [13, 26]. The system monitors which objects are accessed over time, allowing it to differentiate between hot (active) and cold (inactive) data at runtime. This allows the system to adapt to shifting hot sets and application phase changes without prior knowledge of the workload.

This grouping creates an address space layout that directly supports memory efficiency. Hot objects are consolidated into dense regions, making them ideal candidates for huge pages to improve TLB performance. Cold objects are clustered together into separate regions, creating uniformly cold pages that can be easily identified and reclaimed. Crucially, the mechanism for tracking this activity must have very low overhead. Unlike heavyweight profilers or dynamic instrumentation tools, a production-ready solution requires a lightweight mechanism to monitor object accesses without impacting application performance. By grouping objects based on their actual usage, the system organizes the address space to reflect the workload's true temporal access patterns.

3.3 Decoupling Layout from Reclamation

The final principle is to **decouple object-level placement from page-level reclamation** to best leverage existing OS mechanisms. This separation establishes a frontend system that organizes the virtual address space and a back-end policy that acts upon it. The front-end's responsibility is to group objects by access intensity, creating regions of uniformly hot or cold pages. It provides an address space layout that is affable for any reclamation policy to act upon, whether that back-end is the kernel's kswapd or a user-space agent like TMO.

This separation makes existing back-ends more effective even with simple page placement policies. When a back-end is presented with a page from a cold region, it no longer has to guess whether the page contains hidden hot objects; the front-end's organization provides a strong guarantee of its temperature. This allows reclamation policies to act more decisively and reclaim more memory without risking performance degradation from unexpected page faults. This decoupled design enables independent innovation: front-ends can improve their tracking and placement algorithms, while back-ends can evolve to support new hardware like CXL memory [8, 18, 36, 46], all without requiring changes to the other layer.

4 HADES

HADES is a compiler-runtime co-design that implements the front-end role described in Section 3. As shown in Figure 4, it works seamlessly with existing page-level reclamation back-ends by providing them with an address space that is already organized by access activity. HADES monitors object access patterns and dynamically reorganizes the virtual address space, grouping recently accessed objects together while

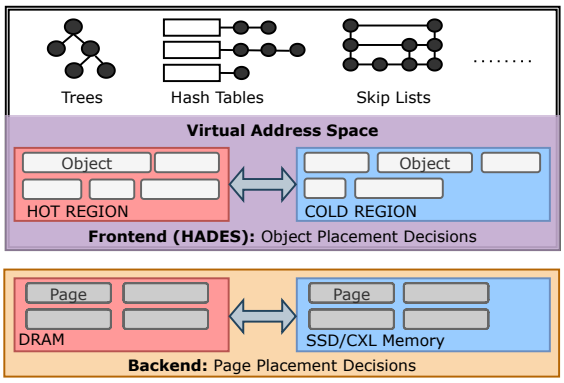


Figure 4. HADES Overview. As a front-end, HADES makes object placement decisions to organize the virtual address space into hot and cold regions. This enables any back-end to make more effective page placement decisions between DRAM and tiered storage.

segregating rarely accessed ones. While HADES requires no changes to application logic, developers provide a one-time annotation to indicate which pointer-based data structures should be managed, enabling fine-grained optimization without altering program semantics.

Tracking and Grouping Objects. To implement the principle of grouping objects by access intensity (§3.2), HADES first needs a low-overhead way to track access. Modern 64-bit processors leave high-order address bits unused, which HADES repurposes to store metadata directly within pointers [4]. It uses these "tagged pointers," or guides, to atomically set an access bit upon dereference, minimizing overhead by skipping the update to the access bit if it is already set. A runtime component, the Object Collector, periodically scans a sparse bitmap referencing all managed objects to read these access bits. Based on this information, it groups objects into three distinct heaps: a NEW heap for initial allocations, a HOT heap for frequently accessed objects (placed on huge pages), and a COLD heap for inactive objects targeted for reclamation (Figure 5). A custom memory allocator ensures these heap regions are contiguous, enabling efficient madvise operations to form hugepages or page out the entire regions.

Safe Concurrent Migration. To enable object mobility (§3.1), HADES must relocate objects without locks or application stalls. The key insight is to track an object's active use in real time, rather than its lifetime. HADES embeds a small Active Thread Count (ATC) in each guide's unused bits, which counts the number of threads currently executing within a function that accesses the object. To manage these counts efficiently, compiler-inserted scope guards automatically increment the ATC at the start of a function and decrement it upon exit. This entire tracking mechanism is selectively activated using an epoch-based protocol only when migration is occurring, eliminating overhead during normal execution [11, 20]. During a migration window, the system can safely move any object with an ATC of zero using an optimistic approach.

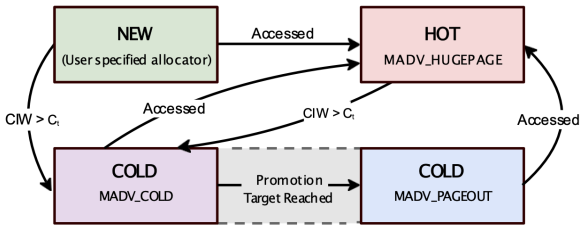


Figure 5. Object Migration State Diagram. An object starts in NEW heap and migrates to HOT or COLD based on access patterns and if its Consecutive Inactive Windows (CIW) is greater than a threshold (C_t), with heap properties adjusted through madvise flags.

Adaptive Workload Response. Fixed threshold migration policies are brittle and cannot adapt to changing workloads. HADES therefore employs a dynamic feedback loop to adjust its reclamation aggressiveness. It monitors the "promotion rate" [30] -— the rate at which applications access data from the COLD heap, which serves as a proxy for page-fault pressure. If this rate exceeds a configurable performance target (e.g., 1%), it suggests the system is being too aggressive. In response, we adapt concepts from TCP congestion control [6] to introduce a multiplicative increase/additive decrease (MIAD) strategy, which makes it harder for objects to be demoted. Initially, cold pages are marked with `MADV_COLD` for reactive reclamation; the system only transitions to proactive `MADV_PAGEOUT` once the promotion rate is safely below the target.

Backend Integration. HADES validates the principle of decoupling layout from reclamation (§3.3) by enhancing existing page-level mechanisms without modifying them. The Object Collector produces uniformly cold regions that kernel mechanisms like kswapd, TPP [38], or HeMem [43] can migrate with high confidence. As shown in Sec. 5, this architectural separation allows backends to operate more effectively, validating our proposed paradigm where a frontend provides object-level intelligence and the backend manages memory using its established policies.

5 Evaluation

We evaluate HADES as a frontend that reorganizes address spaces to work with existing memory management backends. Our experiments demonstrate that object-level temporal tracking eliminates hotness fragmentation while enabling unmodified page-level reclamation systems to achieve both memory savings and performance preservation.

Setup. We evaluate HADES' effectiveness using ten popular pointer-based data structures with diverse concurrency mechanisms (Table 1), borrowed from ASCYLIB [21]. For consistent testing, we develop CrestDB, a lightweight concurrent key-value store that can use any of these structures as its backend. Experiments were conducted on an Intel Xeon Gold 5218 CPU (16 cores), 32GB DRAM, and a 512GB Intel P4800x SSD for swap on Ubuntu 22.04. Each test ran six server threads and six client threads to stress concurrent access patterns.

Structure	Concurrency Control	Used In
HashTable Harris [23]	Lock-free algorithm	NGINX
HashTable Pugh [42]	Fine-grained r/w lock	Redis, Memcached
HashTable Java CHM [3]	Segmented bucket locks	Linux kernel, HAProxy
SkipList Coarse	Global lock	LevelDB/RocksDB
SkipList Fraser [20]	Lock-free algorithm	Redis Sorted Sets
SkipList Herlihy [24]	Optimistic fine-grained	Cassandra, CockroachDB
B+Tree Coarse	Global lock	SAP HANA
B+Tree OCC	OCC w/ epoch reclaim	VoltDB index
MassTree [37]	OCC + RCU	LMDB
Adaptive Radix Tree [32]	Fine-grained r/w lock	DuckDB, PostgreSQL

Table 1. Concurrent data structures evaluated with HADES

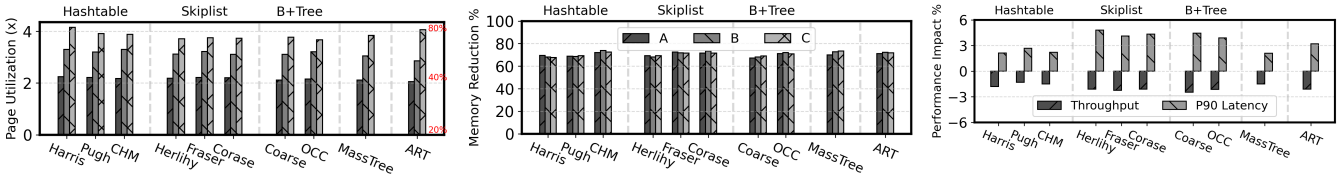
5.1 Frontend Effectiveness on YCSB Workloads

We tested our key-value interface against three YCSB workloads using zipfian distributions that scatter hot keys throughout the entire key space, creating realistic access skew unlike YCSB's default hot key concentration patterns [10, 12, 51]. Each experiment loads 10M KV pairs with 30B keys and 1024B values to demonstrate address space optimization effectiveness across diverse data structure implementations.

HADES increases page utilization from 18-20% to substantially higher levels through object grouping across all data structure types. Once the system completes its initial object classification phase, it achieves 2x improvement for workload A, 3x for workload B, and 4x for workload C (Figure 6(a)). Read-only workload C reaches 80% page utilization as HADES migrates hot objects to dedicated regions without interference from new allocations, while update-heavy workloads show lower improvements because new value allocations initially appear in the NEW heap. The consistent improvement across all ten data structures—from simple hash tables to complex B+Trees with different concurrency mechanisms—confirms that object-level tracking eliminates hotness fragmentation regardless of the underlying index implementation or synchronization approach.

The frontend identifies and reclaims cold memory effectively across all workloads and data structure types. Figure 6(b) shows HADES reduces memory usage by up to 70% through object-level cold identification and heap organization. When promotion rates reach target levels indicating accurate cold classification, HADES transitions from marking pages with `MADV_COLD` to issuing `MADV_PAGEOUT` for proactive reclamation. This object-level reorganization enables precise working set identification that creates uniformly cold pages suitable for safe reclamation without performance risks.

HADES introduces 2.5% average throughput reduction and 5% latency increase from tracking instrumentation overhead. Performance impact varies by data structure complexity, with hash tables showing lower overhead than skiplists and B+Trees due to differences in traversal patterns and key comparison requirements (Figure 6(c)). Access bit operations consume 4-5 ns (comparable to L1 cache hits), while the primary overhead occurs during scope guard operations requiring O(logN) complexity for tracking first-time object observations. The modest overhead across diverse concurrency mechanisms demonstrates that object-level tracking remains practical regardless of the underlying synchronization approach.



(a) Page Utilization Improvement. After HADES migrates objects, utilization increases for read-heavy workloads; by up to 84% and 40-60% for workloads A and B.

(b) Memory Usage. HADES identifies cold objects and reclaims their pages, reducing memory usage by up to 70% for all three workloads.

(c) Performance Impact Run-A. HADES incurs throughput and latency degradation due to ATC instrumentation, varying by data structure complexity.

Figure 6. CrestDB with YCSB. HADES effects on page utilization, memory usage, and performance across YCSB workloads A (50% writes), B (5% writes), and C (read-only) under a zipfian distribution.

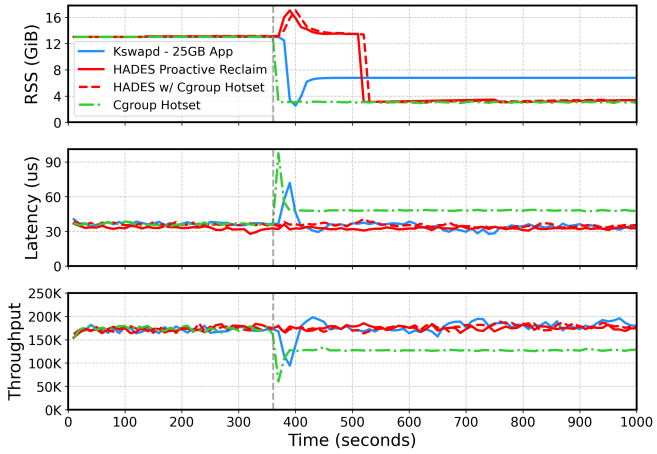


Figure 7. CrestDB vs. Baselines on YCSB-C. HADES resolves the trade-off between performance and memory savings. Standard backends must either sacrifice performance to save memory (Cgroup hotset) or sacrifice memory savings to preserve performance (kswapd). By providing an organized address space, HADES enables both reactive (HADES w/ Cgroup hotspot) and proactive (HADES Proactive) reclamation to achieve maximum memory savings with no performance degradation.

5.2 Backend Integration Validation

We demonstrate that HADES enables existing backends to achieve both memory savings and performance preservation by running YCSB-C with a 12GB footprint while actively accessing only ~4GB. Traditional systems force operators to choose between aggressive reclamation that degrades performance and conservative approaches that waste memory. HADES eliminates this trade-off by providing backends with uniformly cold pages that reclaim without affecting hot data.

Figure 7 reveals the fundamental limitation of page-level backends operating on poorly organized address spaces. The memory-saving-first approach (green line) sets cgroup limits to 4GB, achieving target memory usage but causing 50% latency increase and 30% throughput degradation from aggressive reclamation of pages containing hot objects. The performance-first approach (blue line) creates pressure through background applications (allocates 25 GB), preserving performance but achieving poor memory savings with usage remaining around 6GB rather than the 4GB target. Both backends face the same constraint: page-level granularity cannot distinguish truly hot pages from those containing scattered hot objects.

HADES transforms the same backends from facing painful trade-offs to achieving both objectives simultaneously. The reorganized address space enables `kswapd` to reduce memory usage to 4GB without performance degradation (dashed red line) because the COLD heap contains uniformly cold objects safe for reclamation. Proactive reclamation through `madvise` achieves identical results (solid red line), demonstrating that multiple backend approaches benefit from frontend organization. The backend integration validates that object-level intelligence creates conditions where page-level mechanisms operate effectively without modification.

6 Limitations

HADES optimizes the placement of managed objects in the virtual address space to achieve better page utilization, which enables proactive memory reclamation and improved TLB efficiency. Despite these advantages, HADES faces several limitations that affect its applicability across different scenarios.

- **Lack of pointer stability:** HADES invalidates cached pointers by moving objects across memory, requiring users to query keys and values each time rather than caching pointers. This mirrors constraints in Abseil containers [2] and STL structures like `std::vector` and `std::deque`, which similarly don't guarantee pointer stability.
- **Unique Object Ownership:** When a pointer is annotated for HADES management, users implicitly assert that it has exclusive access control over the object. This unique ownership model, akin to `std::unique_ptr` semantics, is essential for safe object migration, as HADES only updates the address via the annotated pointer. This model aligns with the internal object management within the kinds of concurrent data structures evaluated (see Table 1), which are fundamental to various systems like databases, in-memory caches, and web servers. These structures naturally create

unique ownership paths for their elements or nodes. Objects shared and accessed through multiple aliases cannot be safely managed by HADES.

- **Incompatibility with pointer arithmetic:** HADES places objects across multiple heaps without maintaining contiguous placement, eliminating implicit ordering guarantees and making it unsuitable for data structures like arrays and matrices that require contiguous memory.
- **Language support restrictions:** HADES only works with languages supporting operator overloading (C++, Rust) to intercept access during pointer dereferencing. Languages without this capability (Go, Java) cannot implement this approach, though they might enable direct object management through garbage collection.
- **Requirement for explicit annotations:** Users must designate which objects HADES manages, which is straightforward in key-value stores but perhaps challenging in more complex scenarios. Determining automatic management candidacy falls outside this work's scope.

7 Related Work

Object-Level Management. Recent works like AIFM [44] and MIRA [22] operate at object granularity but focus exclusively on far-memory over RDMA, requiring direct hardware access that limits production adoption. Alaska [48] uses handle-based indirection to reduce RSS through heap compaction, addressing fragmentation reactively without object hotness classification. HADES takes a fundamentally different approach by proactively reshaping the virtual address space through temporal access tracking, creating tiering-friendly object clusters that bridge the gap between application-level object access and page-level OS memory management. This organization of objects across specialized heaps based on access frequency enables HADES to work effectively with existing OS reclamation mechanisms while adapting to changing workload characteristics.

Runtime vs. Allocation-Time Placement. Allocation-time hinting approaches [14, 16, 27, 40, 54] fail to capture objects transitioning between hot and cold states or distinguish between objects from the same allocation site with different access patterns. Systems like PGHO [27] can apply hints automatically based on profile data for allocators like TCMalloc [54], but still make placement decisions only at allocation time. These instrumentation-based profile collection solutions are too slow for production workloads and rely on representative workloads that are often unavailable. HADES instead tracks access patterns at runtime, enabling migration based on actual usage rather than static predictions.

Page-Level Optimizations. In contrast to HADES's object-level address space reorganization, several systems optimize memory tiering and efficiency at the page level. For instance, Johnny Cache [33] manipulates physical page allocation to minimize address conflicts in the hardware sets of direct-mapped DRAM caches. Similarly, Memstrata [53] manages host physical page mappings in virtualized environments to isolate tenants and optimize performance within Intel Flat Memory Mode, where hardware tiers cachelines between local DRAM and CXL memory. These systems improve how pages interact with the underlying cacheline-granular hardware, but they treat the page contents as opaque. HADES's approach is orthogonal, as it ensures the cache lines within each virtual page are more uniformly hot or cold, making the hardware's tiering decisions more effective.

Approaches also exist for page-level decisions about tiering and page size. HawkEye [41] improves huge page management through fine-grained page-level access tracking for better TLB utilization. Memtis [31] dynamically determines page tier placement and page size in heterogeneous memory systems. Based on the access skew within huge pages detected via hardware sampling, Memtis decides whether to split a huge page into smaller base pages, migrating only the hot subpages to the fast tier, thus balancing tiering benefits against address translation costs. While Memtis optimizes page sizes and placement, it still operates at the page level. If a base page still contains a mix of hot and cold objects, Memtis cannot separate them. HADES, by organizing objects, ensures that pages are more likely to be homogeneously hot or cold, potentially improving the effectiveness of systems like Memtis, Johnny Cache, and Memstrata.

8 Conclusion

We demonstrate that the key to efficient memory tiering in non-managed languages lies not in altering the OS, but in engineering the application's virtual address space to be OS-friendly. We introduced a decoupled frontend/backend model where an object-level frontend dynamically reorganizes the address space to create uniformly hot and cold regions, enabling any standard page-level backend to reclaim memory far more effectively. HADES realizes this vision by introducing novel, lock-free techniques to safely migrate objects in a concurrent environment. By resolving hotness fragmentation at its source, HADES allows standard backends to reduce memory usage by up to 70% without the performance trade-offs that have made aggressive memory tiering impractical.

Acknowledgments

We thank David Culler, Lilian Tsai, Qian Ge, Teresa Johnson, colleagues in SystemsResearch@Google, the anonymous reviewers, and our shepherd, Antonio Barbalace, for valuable feedback on earlier drafts of this manuscript.

References

[1] 2024. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[2] 2025. Abseil Pointer Instability. https://abseil.io/docs/cpp/guides/container#fn:pointer-stability

[3] 2025. Overview of Package util.concurrent. https://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

[4] 2025. Tagged Pointers. https://en.wikipedia.org/wiki/Tagged_pointer.

[5] 2025. TCMalloc Leveraging hot/cold hints. https://google.github.io/tcmalloc/temeraire.html#leveraging-hotcold-hints.

[6] 2025. TCP Additive increase/multiplicative decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease

[7] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 399–411. https://doi.org/10.1109/HPCA.2019.00053

[8] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2023. WiscSort: External Sorting for Byte-Addressable Storage. *Proc. VLDB Endow.* 16, 9 (May 2023), 2103–2116. https://doi.org/10.14778/3598581.3598585

[9] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (jun 2004), 25–36. https://doi.org/10.1145/1012888.1005693

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[11] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[12] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 239–252. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

[13] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-guided proactive garbage collection for locality optimization. *SIGPLAN Not.* 41, 6 (June 2006), 332–340. https://doi.org/10.1145/1133255.1134021

[14] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/3368826.3377922

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[16] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[17] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[18] Pouya Esmaili-Dokht, Francesco Sgherzi, Valeria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, German Llort, Petar Radojkovic, Miquel Moreto, Judit Gimenez, Xavier Martorell, Eduard Ayguade, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. 2024. A Mess of Memory System Benchmarking, Simulation and Application Profiling . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 136–152. https://doi.org/10.1109/MICRO61859.2024.00020

[19] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[20] Keir Fraser. 2003. Practical lock-freedom. https://api.semanticscholar.org/CorpusID:11933396

[21] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic concurrency with OPTIK. *SIGPLAN Not.* 51, 8, Article 18 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851146

[22] Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Program-Behavior-Guided Far Memory System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 692–708. https://doi.org/10.1145/3600006.3613157

[23] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[24] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm *(SIROCCO'07)*. Springer-Verlag, Berlin, Heidelberg, 124–138.

[25] Mark D. Hill, James R. Larus, and Trishul M. Chilimbi. 2000. Making Pointer-Based Data Structures Cache Conscious . *Computer* 33, 12 (Dec. 2000), 67–74. https://doi.org/10.1109/2.889095

[26] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. *SIGPLAN Not.* 39, 10 (Oct. 2004), 69–80. https://doi.org/10.1145/1035292.1028983

[27] Teresa Johnson, Snehasish Kumar, , and David Li. 2021. RFC: IR metadata format for MemProf. https://groups.google.com/g/llvm-dev/c/aWHsdMxKAfE/m/WtEmRqyhAgAJ

[28] Sanjeev Kumar and Christopher Wilkerson. 1998. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA '98)*. IEEE Computer Society, USA, 357–368. https://doi.org/10.1145/279358.279404

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[30] H. Andrés Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers.. In *ASPLOS*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 317–330.

[31] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. https://doi.org/10.1145/3600006.3613167

[32] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[33] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 519–534. https://www.usenix.org/conference/osdi23/presentation/lepers

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[35] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. arXiv:arXiv:2203.00241

[36] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. *Systematic CXL Memory Characterization and Performance Analysis at Scale*. Association for Computing Machinery, New York, NY, USA, 1203–1217. https://doi.org/10.1145/3676641.3715987

[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[38] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. arXiv:arXiv:2206.02878

[39] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. 2024. FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 745–764. https://www.usenix.org/conference/osdi24/presentation/mcallister

[40] Svetozar Miucin and Alexandra Fedorova. 2018. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/3240302.3240417

[41] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[42] William Pugh. 1990. *Concurrent maintenance of skip lists*. Technical Report. USA.

[43] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[44] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[45] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 252–263. https://doi.org/10.1109/ISCA.2006.38

[46] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[47] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[48] Nick Wanninger, Tommy McMichen, Simone Campanoni, and Peter Dinda. 2024. Getting a Handle on Unmanaged Memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 448–463. https://doi.org/10.1145/3620666.3651326

[49] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[50] Jon L. White. 1980. Address/memory management for a gigantic LISP environment or, GC considered harmful. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming* (Stanford University, California, USA) *(LFP '80)*. Association for Computing Machinery, New York, NY, USA, 119–127. https://doi.org/10.1145/800087.802797

[51] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[52] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 130–149. https://doi.org/10.1145/3600006.3613147

[53] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

[54] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 2024. Characterizing a Memory Allocator at Warehouse Scale. In *Proceedings of the 29th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 192–206. https://doi.org/10.1145/3620666.3651350