USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

Vinay Banakar, *University of Wisconsin–Madison and Google;* Suli Yang, *Google;* Kan Wu, *xAI;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison;* Kimberly Keeton, *Google*

https://www.usenix.org/conference/osdi26/presentation/banakar

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering

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

Hardware and OS mechanisms for memory tiering are widely deployed, yet datacenters still overprovision DRAM. The root cause is *hotness fragmentation*: allocators place objects by size rather than access pattern, so hot and cold objects become interleaved within the same pages. A single hot object marks its page as active, trapping surrounding cold data in expensive DRAM. Our analysis of Google datacenter workloads shows that up to 97% of the bytes in active pages are cold and unreclaimable. We propose *address-space engineering*: dynamically reorganizing virtual memory so that hot objects cluster into uniformly hot pages and cold objects into uniformly cold pages. We present *OBASE*, a compiler-runtime system for unmanaged languages that serves as an object-aware *frontend* for page-aware OS *backends*. *OBASE* tracks accesses via lightweight pointer instrumentation and migrates objects at runtime using a lock-free protocol that is safe under concurrency. By reorganizing the address space, *OBASE* enables unmodified backends (kswapd, TMO, TPP, Memtis) to tier memory effectively. Across ten concurrent data structures, six backends, and production traces from Meta and Twitter, *OBASE* improves page utilization by 2–4× and reduces memory footprint by up to 70%, with only 2–5% overhead.

1 Introduction

Memory capacity, especially DRAM, has become the dominant cost driver in modern data centers, often accounting for 50% of server capital expenses [1, 49, 59]. Memory tiering promises a solution: by placing cold, less accessed data in cheaper, slower tiers—such as compressed memory, SSDs, or CXL-attached disaggregated memory [3, 4, 6, 11, 42, 43]—one could significantly reduce the expensive DRAM capacity required, thereby lowering costs [29, 44, 49, 56, 60, 62].

In principle, this approach shows great promise, primarily because in hyperscale workloads, only a small volume of memory is hot; the vast majority remains cold and un-accessed for extended periods. As shown in Figure 1, across six different Google workloads [13], only 1.7%–21.3% of bytes are accessed during the trace duration. Indeed, four of the six workloads access less than 3% of bytes, suggesting that we could reclaim 80% to 98% of DRAM for secondary tiers with minimal impact on application performance.

*Work done at Google

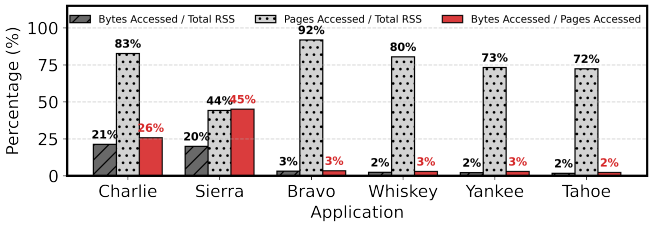


Figure 1. The granularity gap in memory access of Google workloads. Dark Grey: The percentage of total memory *bytes* actually accessed. Light Grey: The percentage of total memory *pages* accessed. Red: The page utilization (calculated as Bytes Accessed / Pages Accessed, see §2.2 for more details).

In practice, however, realizable gains often fall short of this potential. Google reported offloading only 20% of infrequently accessed data to a compressed memory tier [43], while Meta achieved similarly modest savings of 20–32% [62].

The crux of this gap lies in a mismatch between how applications organize data and how the OS manages memory: applications access data at *object* granularity, which varies in size, while the OS manages memory at fixed-size *page* granularity (4KB, 2MB, or 1GB) [20]. Modern allocators place objects on pages with no regard for how these objects will be accessed in the future, often placing frequently accessed objects and rarely touched objects intermingled on the same page. As a result, a few hot bytes (e.g., a single object) would make a whole page active from the OS's perspective, trapping the whole page in expensive DRAM. One can clearly see this phenomenon in Figure 1, where even though only 3.2% of *bytes* are accessed in a workload Bravo, 91.8% of the *pages* are accessed, making them unfit candidates for reclamation. Similar trends can be observed across all six workloads. We term this phenomenon **hotness fragmentation** and identify it as the root cause of inefficient tiering at hyperscalers. We quantify this fragmentation using the *page utilization* metric.

To combat hotness fragmentation, we introduce **address space engineering**, which *dynamically* reorganizes objects within the address space so that objects with similar access intensities are grouped together. By segregating hot and cold objects, address space engineering promises to close the gap between theoretical and realizable memory savings. It creates a memory layout with uniformly cold pages (or hot) that are significantly more amenable to tiering.

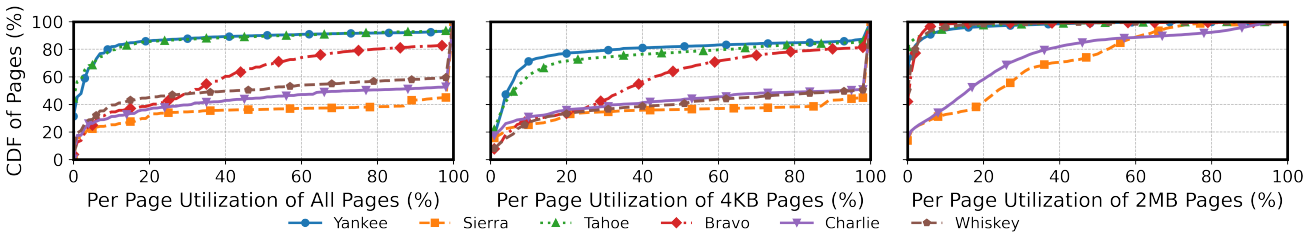


Figure 2. Low Per-Page Utilization in Google Data Center Workloads. CDFs of page utilization for six applications, shown for all pages combined (left) and separately for 4KB pages (center) and 2MB pages (right).

Crucially, by focusing on the address space *layout*, we decouple memory tiering into two orthogonal concerns, allowing independent innovation within each: the *layout* (frontend) problem, which organizes objects to yield high-quality page candidates for reclamation; and the *reclamation* (backend) problem, which focuses on migration policies and mechanisms across different tiers. This separation allows us to reuse existing page-based tiering infrastructures—both swap-based [43, 62] and byte-addressable [44, 49, 56]—and leverage future improvements in reclamation mechanisms. It also ensures that our layout optimization techniques remain applicable to future, currently non-existent memory tiers (e.g., CXL 3.0 fabrics or NVM). Further, it significantly lowers the adoption barrier for hyperscalers: rather than requiring changes to their existing tiering backends, address space engineering provides a superior memory layout that makes existing backends work more effectively.

In this paper, we present ***O*bject-*B*ased *A*ddress-*S*pace *E*ngineering (*OBASE*)**, a compiler-runtime system that engineers the address space for *unmanaged* languages like C/C++. *OBASE* acts as an intelligent *frontend* for memory organization: it manages pointer-based data structures, transparently profiles object access using lightweight instrumentation to determine access intensity, and employs a novel lock-free protocol to migrate objects safely, clustering hot and cold objects together. As a result, *OBASE* prepares the memory layout for any OS *backend*, enabling them to work more effectively. *OBASE* preserves a familiar programming model: developers annotate which pointer fields may relocate, and the compiler routes their accesses through guides. This does require giving up two assumptions (stable object addresses and pointer arithmetic over managed objects) and applies to pointer-based structures rather than arbitrary code; we make this boundary explicit in §7.

We evaluate *OBASE* with ten concurrent pointer-based data structures and six state-of-the-art reclamation and tiering backends [17, 44, 49, 60, 62]. Using key-value store workloads (YCSB) and production traces (Meta, Twitter), we demonstrate that *OBASE* improves page utilization (by 2–4×) and achieves higher memory savings (up to 70%) at the same performance overhead, or conversely, achieves equivalent savings with negligible performance impact. For tiered-memory configurations, *OBASE* achieves the same throughput with half the DRAM capacity.

2 Case for Dynamic Object Reorganization

In this section, we show that hotness fragmentation is both severe and unavoidable under static object placement. We introduce a metric to quantify fragmentation, analyze data center traces from Google [13], and demonstrate that object hotness changes continuously over time. Finally, we explain why unmanaged languages make dynamic reorganization especially challenging, motivating the design of *OBASE*.

2.1 Page Utilization: A Metric

The OS marks a page as "active" if it receives at least one access during a time window T. We define *per-page utilization* as the fraction of a touched page's capacity that is actually accessed. Let $P(T)$ be the set of touched pages during T, and let $U(p,T)$ be the number of unique bytes accessed within page p during T. We then define the aggregate *page utilization* as:

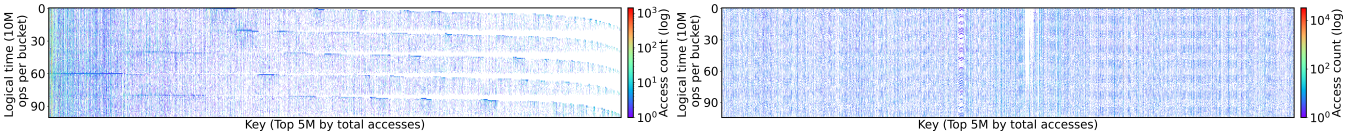
$$\text{Page Utilization}(T) = \frac{\sum_{p \in P(T)} U(p,T)}{\sum_{p \in P(T)} \text{Size}(p)}$$

Low utilization means a page appears hot to the OS even when most of its bytes are cold. Previous work [21] demonstrated low page utilization for open-source databases (e.g., Redis, MongoDB) where 75–90% of accessed pages utilized less than 15% of their capacity for YCSB workloads.

2.2 Fragmentation in Data Center Workloads

To understand hotness fragmentation across a broader range of applications, we analyze memory access traces from six data center workloads at Google [13], collected using DynamoRIO [12]. We process each unique access, annotate pages as 4KB or 2MB using `/proc/self/smaps`, and count unique 64B cache lines touched per page. Based on instruction counts, core counts, and typical warehouse-scale CPU characteristics [41], we estimate the traces capture up to ~30s of steady-state execution.

Figure 2 shows the cumulative distribution of per-page utilization for all workloads, ordered as: (a) aggregated across both 4KiB and 2MiB page sizes, (b) 4KB pages only, and (c) 2MB pages only.



(a) Meta KV Trace. White horizontal bands indicate coordinated quiet periods where access drops across many keys. These bands reveal phased workload behavior.

(b) Twitter Trace. A sparse, scattered pattern indicates sporadic hotness. A few keys on the far left remain consistently hot, but the majority exhibit bursty access with long idle gaps.

Figure 3. Temporal Evolution of Key Hotness. Heatmaps showing access frequency (log scale) for the top 5M keys over logical time buckets (10M operations each). If hotness were static, we would observe continuous vertical bands on the left side of each plot. Instead, we see shifting phases (Meta) and intermittent bursts (Twitter), demonstrating that the hot working set evolves continuously.

Aggregated view. Aggregated across both page sizes, all six workloads show populations of poorly-utilized pages. Yankee and Tahoe are the most fragmented: median utilization is around 3%, meaning half of their touched pages waste over 95% of capacity on cold data. Bravo follows with a median near 35%. The remaining workloads have higher overall utilization but still carry long left tails: roughly 25–30% of pages in Sierra, Charlie, and Whiskey use less than 20% of their capacity. Page-based reclamation cannot distinguish these poorly-utilized pages from well-utilized ones without object-level information.

4KB pages. Fragmentation is visible even at the smallest page size: for Tahoe and Yankee ~80% of pages fall below 20% utilization, for Bravo 60% fall below 40%, and Charlie, Whiskey, and Sierra have 35–40% of pages below 40%. Hotness fragmentation is thus inherent to access patterns, not merely an artifact of huge pages.

2MB pages. Fragmentation worsens dramatically for huge pages. Tahoe, Bravo, and Yankee are extreme: 85–90% of their huge pages utilize under 10% of their 2MB capacity, and even Sierra has 40% below 20%. Each such page, consuming 512× a base page, holds over 1.8MB of cold data alongside a few accessed cache lines.

Finding 1: *Active pages are mostly cold: 70–90% of bytes in pages the OS considers hot receive no accesses.*

2.3 Object Hotness Changes Over Time

One potential solution for hotness fragmentation is to segregate hot and cold objects at allocation time, using static hints or profiling [40]. This strategy assumes that hotness is both identifiable at allocation time and relatively stable afterward.

First, identifying hotness at allocation time is challenging because the same code path allocates objects with vastly different lifecycles. For example, in Redis and Memcached, a single `SET` handler allocates memory for all incoming records, yet one record might be a session token accessed every millisecond, while another is a user profile never touched again. Static analysis cannot distinguish these cases at allocation time. Second, this approach assumes that object hotness is relatively stable: objects identified as hot when allocated will remain hot throughout their lifetimes.

We show that these assumptions do not hold. We analyze object-level traces from Meta [22] and Twitter [65], which record operations in logical time (10M operations per bucket). Figure 3 visualizes access intensity for the top 5 million keys (ranked by total accesses) over 100 such buckets. If hotness were static, the most popular keys (left) would exhibit continuous vertical bands. Instead, both traces show significant churn: bursts of activity followed by long idle gaps.

Meta: Phased hotness. The Meta workload shows coordinated phases where many keys become inactive at once, visible as white horizontal bands. Even keys that are repeatedly accessed overall alternate between active bursts and extended dormancy. A page packed with currently-hot objects eventually becomes a page dominated by cold objects as access patterns shift.

Twitter: Sporadic hotness. The Twitter workload shows a sparse pattern. A small slice of keys remains consistently hot, but the majority exhibit brief access bursts separated by long idle gaps, even among the top 5 million keys.

To quantify churn, we measure reuse-distance variability (75th to 25th percentile of number of operations between accesses to the same key). For mid-sized objects (64B–4KB), representing 94% of Meta keys and 98.2% of Twitter keys, 75% of keys have a reuse spread exceeding 5×, and 65% exceed 30×. Thus, access gaps for the majority of keys fluctuate by more than an order of magnitude.

Finding 2: *Hotness is transient; object hotness is neither knowable at allocation time nor stable over time. As a result, one-time placement cannot prevent hotness fragmentation, and dynamic object migration based on changing hotness is required.*

2.4 Object Mobility in Unmanaged Languages

Moving objects dynamically presents varying degrees of difficulty depending on the language runtime. Managed runtimes, such as the JVM and Go, can relocate objects during garbage collection by updating references atomically, making mobility relatively straightforward. In unmanaged languages like C++, however, programs assume that an object's address is stable [23], rendering dynamic object migration significantly

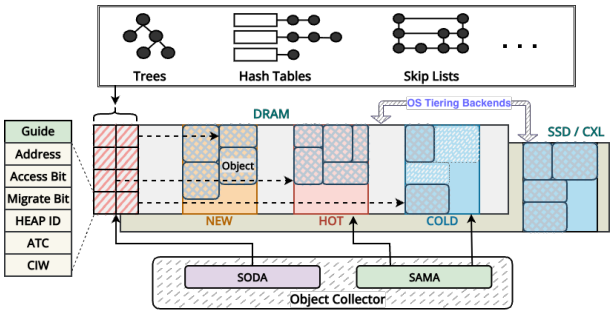


Figure 4. *OBASE* Overview. *OBASE* acts as a frontend that organizes the virtual address space into hot and cold regions. The Object Collector monitors access via Guides and SODA, migrates objects using SAMA, and presents a re-organized address space to the OS backend.

more challenging. Thus, the challenge is not merely to reorganize the layout dynamically, but to do so without breaking program correctness under aliasing and concurrency.

n this paper, we demonstrate how to enable object mobility in *unmanaged* languages specifically because of this challenge. We show that address-space engineering is achievable in C++, where stable addresses are the default assumption, by routing accesses through an indirection rather than preserving the address itself; techniques proven in this setting generalize naturally to managed runtimes.

To make the problem tractable in an unmanaged context, we focus on **pointer-based data structures**, where elements are accessed indirectly through pointers. These structures are ubiquitous in memory-intensive applications (Table 2). Rather than attempting to intercept arbitrary raw-pointer dereferences, which is intractable in C++, we route accesses to relocatable objects through an explicit indirection object (a *guide*, §3.2) that the application uses in place of a raw pointer. When an object moves, the system redirects the guide to its new location transparently. This mechanism must operate safely in highly concurrent environments, without resorting to coarse-grained locks or stop-the-world pauses that would negate the performance benefits of an improved memory layout.

In § 3, we show how *OBASE* provides this transparency and concurrency within the C++ execution model, enabling dynamic reorganization.

3 *Object-Based Address-Space Engineering*

Our goal is to engineer an application's address space so that page-based backends can efficiently reclaim or tier memory. *OBASE* achieves this by reshaping object placement: cluster hot objects into dense regions to increase page utilization, and segregate cold objects into separate regions to expose large pools of reclaimable memory.

OBASE operates in environments where the operating system manages memory in page-sized units and may migrate pages across tiers [20, 42, 43, 58]. The design does not assume a particular tiering policy or memory hierarchy; it only assumes that backends observe per-page activity. By presenting backends with regions that are uniformly hot or cold, *OBASE* allows existing mechanisms—from traditional page reclaim (kswapd, zswap) to tiering engines (such as TMO, TPP, Memtis, and HeMem)—to make better decisions without becoming object-aware [43, 44, 49, 56, 62].

Achieving object-level placement with page-level backends poses four challenges: **(C1) object mobility**—C++ ties object identity to its address, so relocation must be transparent and safe under aliasing; **(C2) low-overhead tracking**—hotness classification must run in the common path without degrading performance; **(C3) dynamic adaptation**—hotness shifts continuously (§2.3), so the layout must evolve; and **(C4) safe concurrent migration**—objects must move without global pauses even while threads hold pointers to them.

3.1 System Overview

OBASE's design rests on separating the *layout* problem (organizing objects so pages contain uniformly hot or cold data) from the *tiering* problem (deciding which pages to evict and where to put them). We call systems addressing the layout problem *frontends* and those addressing reclamation *backends*: existing tiering systems like Kswapd, TPP, Memtis, and TMO are backends that assume the layout is given. *OBASE* is a frontend that takes responsibility for layout so any backend can operate on uniformly hot or cold pages.

The following subsections describe how *OBASE* enables object mobility (§3.2), tracks accesses efficiently (§3.3), organizes the address space (§3.4), and migrates objects safely under concurrency (§3.5).

Figure 4 illustrates the architecture. *OBASE* runs in a continuous control loop. All dereferences of managed objects pass through a lightweight *guide* abstraction, which records whether an object was accessed recently. A background *Object Collector (OC)*, periodically processes this metadata to classify objects by access activity and decide whether they should reside in the NEW, HOT, or COLD heaps. Based on this classification, the OC reorganizes the address space by migrating objects between heaps using a safe, lock-free protocol based on *Active Thread Counts (ATC)*. Finally, *OBASE* exposes these organized regions to the OS through a *Spatially-Aware Memory Allocator (SAMA)*, which lays out each heap as a contiguous virtual range, enabling coarse-grained OS hints. This control loop ensures that the virtual address space continuously adapts to the application's shifting working set while presenting page-based backends with clear hot and cold targets.

3.2 Object Mobility in Unmanaged Languages

OBASE decouples an object's logical identity from its location using a lightweight *guide* abstraction. A guide carries the current location of the object as well as additional metadata needed by *OBASE*. Developers interact with objects by using guides rather than raw pointers. When a guide is dereferenced, *OBASE* resolves it to the object's current address and

records that the object was accessed (described in §3.3). The indirection layer is the mechanism that makes later relocation transparent; code that previously operated on pointers continues to operate on guides.

Developers choose which pointers can participate in relocation by marking them with annotations (e.g., the pointers to keys and values in hash-table buckets or the record pointers in B+ trees). In practice this annotation is small: for the ten data structures in our evaluation (Table 2), each managed type carries one to three relocatable pointer fields (typically the structure's authoritative child) while container algorithms and client code are untouched. Guides are enforced through three compiler passes (detailed in §4.7). A type-level pass identifies annotated pointers and rewrites their dereferences to invoke the guide. An instrumentation pass injects hooks at access sites so *OBASE* can observe uses without modifying application logic. A validation pass ensures that managed objects are not used in unsupported ways (e.g., pointer arithmetic over nodes or assumptions about physical contiguity). These passes allow developers to adopt *OBASE* incrementally, starting with the portions of a codebase where object residency matters most.

The division of labor is deliberately narrow: the *developer* marks a small set of pointer fields and guarantees that no unannotated raw pointer aliases a managed object; the *compiler* performs everything else: rewriting declarations and dereferences, and inserting the tracking hooks. An annotated guide carries unique ownership of its object, analogous to `std::unique_ptr`: *OBASE* updates that single guide when relocating, and does not track hidden aliases, so structures that share a node through multiple pointers (e.g., graphs, doubly-linked lists) are out of scope. The validation pass rejects pointer arithmetic and physical contiguity assumptions over managed objects rather than silently miscompiling them. A raw pointer obtained from a guide dereference (e.g., held in a register) remains valid for the duration of the enclosing public operation, because that operation keeps the object's active-thread count above zero and thereby vetoes any concurrent migration (§3.5); callers must not cache such pointers beyond the operation.

Two further programming-model restrictions keep these invariants intact under concurrent migration. First, the `Guide<T>` type appears only inside the data structure's implementation, where Pass 2 rewrites the developer-marked pointer fields (§4.7); it does not appear in calling code. Second, guides do not cross the data structure's public boundary as parameters or return values; they are held as internal fields, and callers interact through primitive-typed APIs (e.g., `HashMap::get(int idx)` in Figure 7). Together these ensure that every guide dereference is reachable from a public-method entry whose TAG participates in the migration protocol.

3.3 Low-Overhead Access Tracking

To classify objects by temperature, *OBASE* must observe which objects are accessed over time, but tracking must be cheap enough to run continuously. Existing mechanisms fall short at both ends of the spectrum. Hardware page table access bits operate at page granularity and cannot distinguish a few hot cache lines from megabytes of cold data on the same page [8, 43]. Software profilers such as DynamoRIO and LLVM's memory profiling provide fine-grained information but impose prohibitive overheads for always-on deployment [12, 14]. Hardware sampling via PEBS offers lower overhead but provides statistical coverage rather than precise per-object tracking [7].

Instead, *OBASE* embeds access tracking directly into guide pointer dereferences, yielding object-level information without significant overhead. A guide overloads the dereference operator so that each access records that the underlying object was used. Modern 64-bit architectures reserve high-order address bits in canonical user-space pointers, so *OBASE* stores a small amount of per-object metadata in these unused bits [16]. Inline metadata avoids external side tables and ensures that recording access is part of normal pointer use.

Metadata is updated on every dereference using a single atomic read–modify–write (RMW): an `accessed` bit is set if it has not already been set, thereby skipping subsequent updates to avoid unnecessary cache-coherence traffic for frequently touched objects. This design keeps the common path small, and the resulting per-access cost is comparable to a cache hit. The metadata also holds state used by the runtime to classify objects or coordinate relocation, without requiring separate allocations or indirection.

To let the runtime observe all managed objects without walking application-specific pointer graphs, *OBASE* maintains a Sparse Object Data Activity (SODA) bitmap (§4.3) over the process heap. SODA records which virtual regions contain managed objects and enables the Object Collector (OC) to discover objects by scanning these regions rather than interpreting container internals.

3.4 Dynamically Engineering the Layout

The access signals from §3.3 feed into a layout policy that continuously reorganizes the virtual address space. OBASE groups objects by observed temperature so that page-based backends see large, uniform regions rather than interleaved hot and cold data. Three logical temperature heaps capture this temperature: NEW holds freshly allocated objects whose access pattern is not yet clear, HOT holds the current working set, and COLD holds objects inactive for multiple scan periods. Each heap occupies a dedicated virtual address range, so an object's address directly encodes its temperature.

Objects move between heaps as access intensity changes (Figure 5): inactive HOT objects are demoted to COLD, and COLD objects that become active again are promoted back.

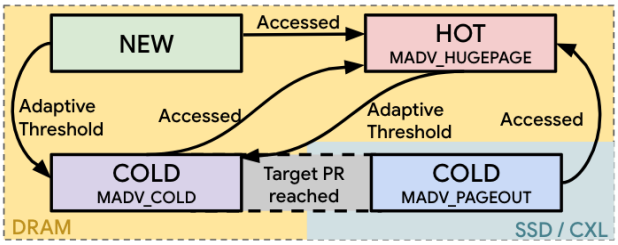


Figure 5. Object Migration State Diagram. Objects transition between heaps based on access intensity. The Object Collector promotes accessed objects to HOT and demotes inactive objects to COLD, allowing SAMA to apply different `madvise` policies to each region if required.

This continuous reclassification tracks the workload's evolving working set rather than its allocation history.

To realize these logical heaps in the virtual address space, *OBASE* employs a Spatially-Aware Memory Allocator (SAMA) that reserves a contiguous virtual address range for each heap and sub-allocates objects within that range. Contiguity is a deliberate design choice that allows coarse-grained OS hints to be applied to whole heaps rather than individual pages, exposing large pools of cold memory to page-based tiering systems. The Object Collector (OC) periodically scans SODA to classify each managed object. Objects touched since the last scan are candidates for promotion; objects that remain untouched accumulate evidence of coldness.

Reacting to a single inactive window would make classification sensitive to transient bursts. *OBASE* therefore tracks a per-object Consecutive Inactive Window(CIW) counter: each scan window without an access increments CIW; any access resets it to zero. Objects whose CIW exceeds a cold threshold become eligible for demotion to COLD; objects in COLD that are accessed rejoin HOT. This hysteresis ensures that only sustained inactivity triggers migration to COLD.

The cold threshold C_t governs how long an object must remain inactive before migration. Too aggressive and COLD objects are frequently re-accessed; too conservative and reclaimable memory lingers in HOT. *OBASE* adapts C_t using a promotion-rate target. We define the promotion rate (PR) as the fraction of the working set drawn from COLD heap in each scan window:

$$PR = \frac{\text{unique COLD pages accessed}}{\text{working set size}} \times \frac{60}{\text{scan interval (s)}}$$

Crucially, *OBASE* measures COLD-heap accesses regardless of where those pages physically reside—it cannot determine which tier a page occupies and does not attempt to. If the observed rate exceeds a target, C_t increases by one window; if below, C_t decreases. This additive adjustment converges to a workload-specific regime. The goal is not to minimize COLD-heap accesses, but to ensure that pages classified as COLD are *safe targets* for any backend policy. §4 details initialization and tuning.

By design, *OBASE* is decoupled from any specific backend. The base system reorganizes the address space but issues no reclamation hints. This separation is intentional: the value of address-space-engineering lies in making pages uniformly hot or cold, which improves the precision of any page-based mechanism. Backends such as kswapd, TMO, TPP, and Memtis observe per-page activity through their existing interfaces (PTE-A bits, PEBS samples, or PSI signals) and naturally make better decisions when COLD pages contain only cold objects.

Optionally, *OBASE* can issue proactive hints to accelerate reclamation. In this hinted mode, once the promotion rate stabilizes below the target, *OBASE* marks COLD pages with `MADV_COLD` or `MADV_PAGEOUT` to signal that they are safe to reclaim. Similarly, SAMA may request huge pages for HOT (`MADV_HUGEPAGE`) to selectively improve TLB coverage over dense hot data. These hints are strictly advisory and complement rather than replace backend policies.

3.5 Safe Concurrent Migration

Reorganizing the address space requires relocating objects while application threads may hold pointers to them. In managed languages, garbage collectors solve this problem using load barriers or stop-the-world pauses. C++ offers neither: there is no runtime to intercept pointer loads, and production systems cannot tolerate pauses. OBASE must therefore migrate objects without blocking threads, while guaranteeing that every dereference sees a valid location. As described in §3.2, all dereferences pass through guides, so relocation reduces to atomically updating a single pointer-sized word. Callers never follow stale pointers and do not need explicit synchronization.

Lightweight Scoping. An object can be migrated only when no thread is actively using it. Classic approaches—stop-the-world pauses or per-access load barriers—are unsuitable for C++ systems code [30, 50, 63]. Instead,*OBASE* introduces a per-object *Active Thread Count* (ATC) that tracks how many public data structure operations have observed a guide during a migration window. The ATC captures the notion of active use: if a thread begins an operation that may dereference an object, ATC is incremented once for that scope; when the operation completes, the ATC is decremented. Migration is permitted only when the ATC reaches zero, indicating that no thread is currently executing an operation that could read or modify the object.

The ATC must be updated efficiently. Rather than incrementing the ATC on every pointer dereference—which would impose unacceptable overhead—*OBASE* scopes tracking to public API boundaries. When a thread enters a data-structure operation (e.g., get, insert), it registers with a Thread-local Active scope guard (TAG); when the operation completes, the guard decrements the ATC for all objects touched. This design reflects how C++ programmers reason about pointer validity: pointers are valid for the duration of the operation that

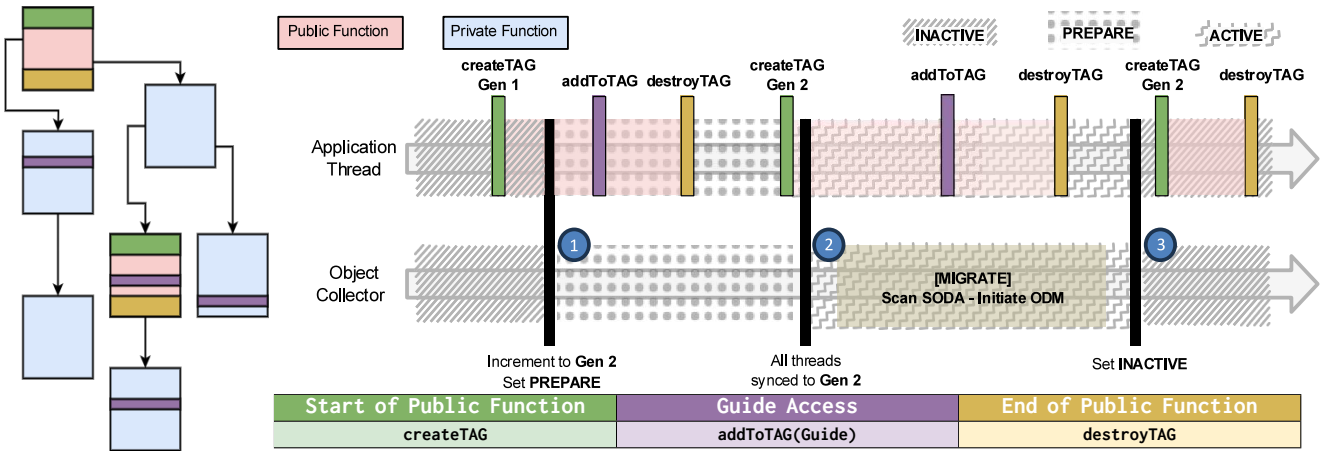


Figure 6. Application Thread Execution Flow And Interaction With OC The left side shows the instrumented call graph, code is inserted in three scenarios as shown. Public functions create and destroy Thread-local Active scope Guards (TAGs), maintaining nesting levels and registering the thread in the Thread Activity Index (TAI). Guides increment active reference counts only if added to the TAG. Reference counts are decremented only when the outermost public function exits. Active Thread Count (ATC) is enabled only during `PREPARE` and `ACTIVE` states.

obtained them, not indefinitely. Compiler instrumentation (detailed in §4.2) inserts the necessary hooks automatically.

Active Windows via Epochs. Always-on ATC tracking would impose overhead even when no migration is in progress. *OBASE* therefore activates ATC only during periodic migration epochs. Outside these windows, guide dereferences record only access activity (§3.3) with no ATC overhead. When the OC initiates migration, it coordinates an epoch transition that ensures all active threads enable ATC tracking before any object is moved. This epoch-based activation confines the synchronization cost to brief, infrequent windows.

Optimistic Migration. Given ATC=0, *OBASE* moves objects using an optimistic protocol inspired by optimistic concurrency control. The Object Collector copies the object to its target heap, then attempts to atomically swing the guide to the new address. If any thread accesses the object during the copy, the commit fails and the move is abandoned—the object remains in place, and the thread sees valid data. This optimistic approach has two key properties: (1) threads never block on migration, and (2) concurrent access safely vetoes relocation rather than observing inconsistent state. §4.5 details the CAS-based protocol.

Safety and Non-Blocking Progress. Safety follows from a single invariant: the guide is updated atomically, and migration aborts on any conflicting access. No thread can ever follow a stale pointer. Progress is non-blocking: application threads never wait on the collector, and the collector performs bounded work per object before committing or abandoning. Frequently accessed objects naturally resist migration (their ATC rarely reaches zero), while cold objects eventually move. The result is an address space that reshapes continuously without stop-the-world pauses, while preserving familiar pointer semantics.

4 Implementation

This section details *OBASE*'s concrete realization.* We cover the guide metadata encoding (§4.1), scope-guard data structures (§4.2), SODA bitmap layout (§4.3), controller parameters (§4.4), the epoch-based migration protocol (§4.5), kernel reclamation optimizations (§4.6), and the compiler passes that automate guide management (§4.7).

4.1 Guide Metadata Encoding and Heap Allocation

Each guide uses the 48 bits required for canonical x86-64 user-space addresses and repurposes the upper 16 bits for metadata: 7 bits encode ATC (supporting up to 128 concurrent threads per object), 5 bits track CIW (up to 32 windows, or 60+ minutes at the default 120s interval), 2 bits identify the current heap (NEW, HOT, COLD), and 2 bits store the access and migration-lock flags. As 128-bit addressing becomes more prevalent [5], implementations can expand these fields without changing the guide abstraction. We implement the three heaps using a Spatially-Aware Memory Allocator (SAMA) built on jemalloc's extent management. SAMA reserves large `mmap` regions for each heap and allocates objects within them, returning unused extents to the OS as objects migrate or are freed.

4.2 Efficient Scope Guard Tracking

Figure 6 illustrates the instrumented call graph. The compiler inserts three hooks: `createTAG` (green) at public API entry, `addToTAG(guide)` (purple) at each guide dereference, and `destroyTAG` (yellow) at public API exit. Public functions may

OBASE is open-sourced at https://github.com/WiscADSL/obase

call private helpers that also dereference guides; the TAG tracks all such accesses but only decrements the ATC when the *outermost* public function returns. This nesting-aware design ensures that the ATC remains positive throughout the entire operation, even when internal helpers are inlined or called multiple times.

We implement TAGs using a `BaseDeltaPtrSet` that exploits pointer locality: it encodes pointers as a base address plus 32-bit deltas, grouping up to 16 nearby pointers within two cache lines. Insertion is O(1) when the pointer falls within an existing group; otherwise a new group is created in O(log G) time, where G is the number of groups. Since pointers within a single operation cluster tightly (e.g., keys and values in the same bucket, nodes along a tree path), most insertions hit existing groups. Across the ten data structures in our evaluation, the median number of unique guides per operation ranges from 3 (hash tables) to 12 (B+Tree traversals), keeping per-operation TAG overhead under 100 ns.

4.3 SODA Bitmap Structure and Object Discovery

The Sparse Object Data Activity (SODA) bitmap uses a two-level structure to cover the heap address space without allocating storage for empty regions. SODA divides the address space into coarse-grained blocks; each block contains 64-bit words where individual bits indicate guide presence at fixed-size slots. Blocks are allocated lazily and reclaimed when empty. Because SODA tracks guide pointers rather than object addresses, it remains valid across relocations; the guide's address does not change when the object moves. Memory overhead is one bit per potential guide slot plus block-level bookkeeping. The OC scans SODA at a configurable interval (120 s by default), chosen to align with cold-memory detection thresholds in warehouse-scale tiering systems [29, 43].

4.4 Cold Threshold Controller

OBASE initializes C_t to three scan windows; at the default 120 s interval, this is approximately six minutes of inactivity before demotion, consistent with the five-minute rule for tiered storage [19, 32, 33]. The target promotion rate of 1% is based on budgets used in production compressed-memory and CXL tiering deployments [29, 43]. C_t is bounded between 1 and 32 windows. After each scan window, an additive-increase/additive-decrease (AIAD) loop (Algorithm 1) compares the observed promotion rate PR_{actual} (§3.4) against the target and adjusts C_t accordingly, so the threshold settles into a workload-specific regime without large swings.

4.5 Epoch Protocol and Optimistic Migration

As shown in Figure 6 the OC coordinates migration through three states (`INACTIVE`, `PREPARE`, and `ACTIVE`) using a global epoch counter and a Thread Activity Index (TAI) - a per-thread slot that lets the OC verify, without blocking threads, that every active thread has observed the new epoch before migration begins.

Algorithm 1 Cold-Threshold Controller (per scan window)

$PR_{actual} \leftarrow \dfrac{\text{unique COLD pages accessed}}{\text{working-set size}} \times \dfrac{60}{\text{scanInterval}}$
if $PR_{actual} > PR_{target}$ **then** ▷ demotions revisited too often
  $C_t \leftarrow \min(C_t + 1, 32)$ ▷ demote more conservatively
else if $PR_{actual} < PR_{target}$ **then** ▷ slack in the budget
  $C_t \leftarrow \max(C_t - 1, 1)$ ▷ reclaim more aggressively
end if

Time	Actor	Action	ATC	Lock	Outcome
(a) Concurrent dereference aborts migration:					
t_0	OC	read guide	0	0	eligible
t_1	OC	CAS: set lock	0	1	copying begins
t_2	Thread	dereference	1	0	lock cleared
t_3	OC	CAS: commit	*mismatch*		**aborted**
(b) No interference, migration succeeds:					
t_0	OC	read guide	0	0	eligible
t_1	OC	CAS: set lock	0	1	copying begins
t_2	OC	CAS: commit	0	0	**success**

Table 1. Race between migration and concurrent access. (a) A dereference at t_2 modifies the guide, causing the OC's commit CAS to fail. (b) When no thread intervenes, both CAS operations succeed and the object moves.

Epoch Transitions. In the `INACTIVE` state, the ATC updates are disabled; dereferences only set access bits. When the OC initiates migration, it increments the epoch counter and enters `PREPARE` state 1. Threads entering public methods record the current epoch in the TAI (a small array indexed by thread-ID hash); exits clear their slot. The OC repeatedly scans the TAI; once every non-empty slot reflects the new epoch 2, all active threads have enabled ATC tracking, and the OC enters the `ACTIVE` state . After migration completes, the OC returns to `INACTIVE` state 3. Crucially, threads never block; they simply record epoch participation, while the OC performs convergence checks.

Optimistic Data Migration. Within `ACTIVE` states, the OC migrates objects using an Optimistic Data Migration (ODM) protocol (Algorithm 2), similar in spirit to Optimistic Concurrency Control (OCC), where a job is performed first and verified later, essentially acting as a weak transaction. For a candidate with ATC=0 and migration-lock clear, the OC: (1) atomically sets the migration-lock bit on the guide slot; (2) allocates space in the target heap via SAMA and copies the object; (3) constructs a new guide (new address, heap ID, reset CIW, cleared lock) and publishes it with a single commit CAS, the same atomic both installs the new address and releases the lock. If the commit CAS fails, a concurrent dereference has changed the guide and the migration is abandoned (the OC explicitly clears the lock it set; see Algorithm 2).

Race Resolution. Table 1 illustrates a race between migration and access. Every dereference performs an atomic

Algorithm 2 Optimistic Data Migration (ODM)

```
1:  procedure MigrateObject(ptr, targetHeap)                    ▷
    ptr: address of guide slot
2:      guide ← AtomicLoad(ptr)                                 ▷ Load object guide
3:      if RefCount(guide) > 0 then
4:          return false                                        ▷ Object in use
5:      end if
6:      srcAddr ← ExtractAddr(guide)
7:      srcHeap ← ExtractHeapType(guide)                        ▷ Get current heap type
8:      SetMigrationLock(ptr)                                   ▷ CAS; mark as migrating
9:      size ← GetSize(srcAddr)
10:     dstAddr ← samalloc(size,targetHeap)
11:     if dstAddr = null then
12:         ClearMigrationLock(ptr)
13:         return false                                        ▷ Allocation failed
14:     end if
15:     Copy(dstAddr,srcAddr,size)                              ▷ Copy data
16:     newGuide ← CreateGuide(guide,dstAddr,targetHeap)        ▷
    Inherits lock from guide metadata
17:     ClearMigrationLock(newGuide)                            ▷
    Local clear of inherited lock; commit via CAS below
18:     success ← AtomicCAS(ptr,guide,newGuide)
19:     if success then
20:         Free(srcAddr,srcHeap)                               ▷ Release old memory
21:         return true
22:     else
23:         Free(dstAddr,targetHeap)                            ▷ Rollback
24:         ClearMigrationLock(ptr)                             ▷ CAS; release lock we set
25:         return false                                        ▷ Migration failed, guide changed
26:     end if
27: end procedure
```

CAS that sets the access bit and *clears* the migration-lock bit. If a thread accesses object x while the OC is copying it (t_2), the guide changes, the commit CAS fails (t_3), and x remains at its original address. The thread always sees valid data; concurrent access supersedes migration.

4.6 Kernel Page Reclamation Optimization

When objects migrate to the COLD heap and the Object Collector issues `MADV_PAGEOUT`, Linux's page reclamation path becomes the bottleneck for efficient memory tiering. The default path in `shrink_folio_list()` processes each page individually: it clears the page table entry (PTE), issues a TLB flush or shootdown that triggers an inter-processor interrupt (IPI) to every core, and submits the page to the block I/O layer. This fine-grained approach creates substantial overhead when reclaiming large regions, as the cumulative cost of per-page TLB invalidations and IPIs degrades performance even for threads accessing unrelated hot data.

We modify `shrink_folio_list()` to batch these operations: it aggregates pages (up to a full PMD spanning 512 base pages), clearing each PTE and marking it for pageout, but defers TLB invalidation until a complete batch is prepared, then issues one flush over the entire range and submits all pages together for I/O. By amortizing TLB shootdowns and I/O submissions, this reduces IPIs by more than 99% when demoting 10 GiB of memory versus the unmodified kernel. Beyond local tiering, batched invalidation also addresses a scalability bottleneck in RDMA-based far-memory systems [35, 48, 57], where per-page shootdowns cause IPI storms whose latencies grow super-linearly with thread count; aggregating across hundreds of pages lets the reclamation path scale for both CXL and RDMA backends.

```
1  void HashMap::get(int idx) {
2      createTAG();                    // public entry
3      Guide<char>& v = buckets[idx]; // HashMap member
4      addToTAG(&v);                   // before deref
5      if (*v > 37) { destroyTAG(); return; }
6      addToTAG(&v);  *v = 42;
7      addToTAG(&v);
8      std::cout << "Value: " << *v << std::endl;
9      destroyTAG();                   // public exit
10 }
```

Figure 7. Compiler transformation. The developer marks one pointer field of `HashMap`; the pass rewrites its type from `char*` to `Guide<char>`, brackets the public method with `createTAG`/`destroyTAG`, and inserts `addToTAG` before each guide dereference. The `buckets` array is unmarked and stays in memory, so indexing itneeds no guard; only the guide dereference (`*v`) does. Callers invoke `HashMap::get` with a primitive key (no `Guide` crosses the public-method boundary), so caller code neither holds nor dereferences a `Guide`. ATC increments inside `addToTAG` only when the TAG state is not `INACTIVE`.

4.7 Compiler Passes for Guide Management

OBASE uses three complementary compiler passes, implemented on Clang and LLVM, to convert raw pointers to guides and insert the TAG/ATC instrumentation, confining developer effort to marking which pointer fields are managed. **Visibility extraction.** A Clang frontend pass ensures TAGs are created only at public boundaries by using a `RecursiveASTVisitor` to find public methods that serve as data-structure entry points (e.g., `get`, `set`, `delete`) and records their visibility for later stages. **Pointer-to-guide conversion.** A second Clang pass uses the rewriter to convert the developer-listed pointer declarations and usages into guides, giving precise control over which objects *OBASE* manages. **IR instrumentation.** The third pass runs on the LLVM IR. A fixed-point analysis first identifies the set of functions that directly use guides (via conversions, destructor calls, operator overloads, or assignments), propagates this set through the call graph to find functions that touch guides indirectly, and combines it with the visibility data from Pass 1 to classify each function. It then inserts `createTAG`/`destroyTAG` at the entry and exit of public functions that touch guides, and `addToTAG` before each guide dereference in all transformed functions (public and private). This restricts TAG creation to public boundaries while still tracking every access throughout the call stack (Figure 7).

Structure	Concurrency Control	Used In
HashTable Harris [36]	Lock-free algorithm	NGINX
HashTable Pugh [54]	Fine-grained r/w lock	Redis, Memcached
HashTable Java CHM [15]	Segmented bucket locks	Linux kernel, HAProxy
SkipList Coarse	Global lock	LevelDB/RocksDB
SkipList Fraser [31]	Lock-free algorithm	Redis Sorted Sets
SkipList Herlihy [37]	Optimistic fine-grained	Cassandra, CockroachDB
B+Tree Coarse	Global lock	SAP HANA
B+Tree OCC	OCC w/ epoch reclaim	VoltDB index
MassTree [47]	OCC + RCU	MICA, Silo
Adaptive Radix Tree [45]	Fine-grained r/w lock	DuckDB, PostgreSQL

Table 2. Concurrent data structures evaluated with *OBASE*. These structures span lock-free, fine-grained, and coarse-grained concurrency mechanisms, demonstrating *OBASE*'s compatibility with diverse synchronization approaches.

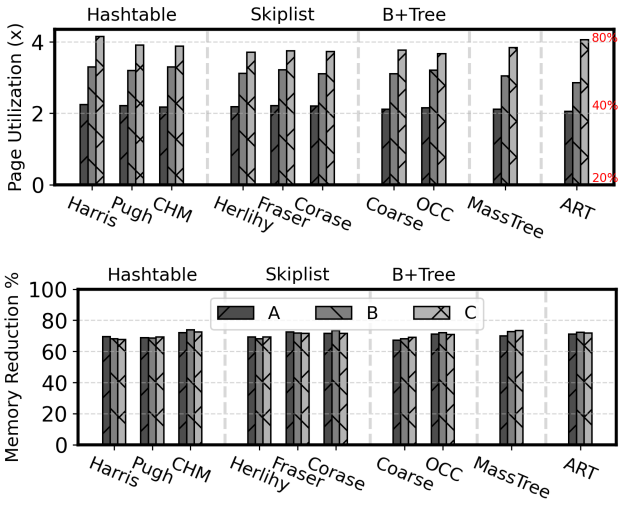


Figure 8. *OBASE* effectiveness (YCSB, 10M keys). Top: Page utilization improvement relative to the baseline allocator; *OBASE* increases utilization by 2–4× across workloads and data structures. Bottom: RSS reduction after *OBASE* pages out the COLD heap (via Kswapd). Memory footprint shrinks by 65–72%.

5 Evaluation

We evaluate *OBASE* along four dimensions:

E1: Effectiveness. Does *OBASE* reduce hotness fragmentation and memory footprint across different data structures and workloads? (§5.2)

E2: Backend synergy. How much does *OBASE* improve the effectiveness of existing page-based reclamation and tiering backends? (§5.3)

E3: Overhead and scalability. What runtime overheads do tracking and migration introduce, and how do they scale with thread count? (§5.4)

E4: Dynamic behavior. How does *OBASE* adapt to changing hotsets in long-running, real-world workloads? (§5.5)

5.1 Experimental Setup

All experiments run on an Intel Xeon Gold 5218 (16 cores, SMT disabled) server configured in `performance` governor mode with Ubuntu 22.04 and Linux kernel 6.12. The memory subsystem comprises two tiers: a fast tier of 2×16 GB DDR4 DRAM modules at 2400 MHz, and a slow tier of 4×128 GB Intel Optane DC Persistent Memory 100 modules at 2666 MHz. All six memory devices occupy distinct channels to avoid interface. We configure Optane PMEM in `Memory Mode` and expose it as a separate NUMA node, creating a two-tier hierarchy representative of emerging CXL-attached memory deployments [46, 49]. The Optane tier provides approximately 2.5× higher access latency than local DRAM [64], consistent with first-generation CXL memory characteristics [58]. For reclamation experiments requiring swap, we use a 512 GB Intel P4800X Optane SSD.

CrestDB testbed. We implemented CrestDB, a concurrent in-memory key-value store, to evaluate *OBASE* across diverse data structures and concurrency mechanisms. CrestDB integrates ten high-performance data structures spanning the concurrency-control spectrum (Table 2), from lock-free algorithms to global locks. Many structures are borrowed from ASCYLIB [34], which provides production grade implementations. This diversity demonstrates that *OBASE*'s compiler instrumentation and migration protocol are compatible with a wide range of synchronization approaches without structure-specific modifications. All data structures maintain guide pointers to key and value objects; CrestDB deep-copies inserted data, ensuring *OBASE* manages the authoritative copy rather than application-held aliases. Unless otherwise noted, CrestDB runs with six server threads and six client threads.

Workloads. We use the YCSB benchmark suite with Zipfian-distributed keys to model skewed access patterns typical of production workloads. To evaluate real-world adaptivity, we replay production traces from Meta (CacheLib [22], DBbench [24]) and Twitter (Cluster 7, Cluster 23) [65].

Controller parameters. Unless otherwise noted, the Object Collector scans every 120 s, targets a 1% promotion rate, and adjusts C_t by ±1 window per scan within $1 \le C_t \le 32$ (§3.4); the conservative 1% target follows production compressed-memory deployments [43, 62]. The optimal rate is hardware-dependent (faster tiers tolerate higher cold-access fractions), and hardware-aware tuning is future work. §5.5 examines C_t dynamics over time.

5.2 Effective Address-Space-Engineering (E1)

We first evaluate whether *OBASE* achieves its core objective: reducing hotness fragmentation and converting unreclaimable cold data into reclaimable pages. We run CrestDB with all ten data structures under YCSB workloads A, B, and C with Zipfian keys. We load 10M keys with 30-byte keys and 1024-byte values, creating a 13 GiB dataset. Unless stated otherwise, all results in this section are measured after *OBASE* has converged (promotion rate below the 1% target).

Page utilization. Figure 8(a) reports page utilization before and after *OBASE* reorganizes objects into NEW, HOT,

and COLD heaps. Initially, the data structures exhibit 18–20% utilization when measured over 120 s windows: most pages contain only a handful of accessed cache lines, reflecting the hotness fragmentation observed in production traces (§2).

After three scan intervals, the Object Collector classifies objects based on guide access bits and migrates them into HOT or COLD heaps. Compared to the baseline, *OBASE* improves page utilization by 2× for workload A (50% writes), approximately 3× for workload B (5% writes), and up to 4× for the read-only workload C. Across data structures, absolute utilization after convergence ranges from roughly 40% (workload A) to 80% (workload C).

The variation across workloads reflects how the NEW and HOT heaps interact. Workload C has no updates: once objects are classified as hot, they remain in the HOT heap and no new objects are allocated there. Nearly all accessed bytes end up densely packed in a small number of HOT pages, and utilization approaches 80%. In workload B, occasional updates allocate fresh values in NEW, so the working set splits between NEW and HOT and overall utilization stabilizes around 60–70%. Workload A performs frequent updates, continuously injecting new objects into NEW. Utilization still roughly doubles, but cannot reach read-only levels because a larger fraction of hot objects reside in NEW during their initial epochs.

The consistency of the improvement across ten structurally different data structures shows that *OBASE*'s benefits derive from object-temperature clustering rather than data-structure-specific layout optimizations.

Memory footprint. Higher page utilization translates directly into reclaimable cold memory. Once the promotion rate falls below the 1% target—typically after 3–4 scan intervals (6–8 minutes) for YCSB—*OBASE* proactively issues `madvise(MADV_PAGEOUT)` on the COLD heap. Figure 8(b) shows the resulting RSS reduction relative to a baseline without reclamation.

Across all data structures and workloads, *OBASE* reduces RSS by 65–72%. For workload B with 10M keys, the baseline uses 12.4 GiB; after *OBASE* converges and pages out COLD, RSS drops to 3.5–4.0 GiB. Because COLD pages contain almost exclusively inactive objects, proactive paging does not cause swap-in storms or noticeable throughput degradation (we quantify overheads in §5.4).

Takeaway #1: *OBASE improves page utilization across all data structures as it tracks object hotness without the semantic knowledge of each structure. This enables uniform hotness fragmentation reduction across diverse concurrency mechanisms.*

5.3 Backend Synergy (E2)

The memory savings demonstrated in §5.2 are valuable only if OS tiering backends can exploit them without degrading performance. We now show that *OBASE* enables existing reclamation and tiering systems to achieve aggressive memory savings while preserving throughput.

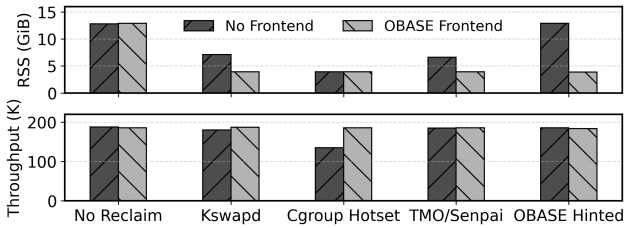


Figure 9. *OBASE* with reclamation backends (YCSB-C, MassTree). Top: RSS after convergence. Bottom: throughput. Without *OBASE*, backends face a trade-off between memory savings and performance. With *OBASE* (hatched bars), all backends achieve near-optimal memory savings with minimal throughput loss.

5.3.1 Paging-based Reclamation Backends. We run CrestDB with MassTree under YCSB-C in a memory-constrained configuration: the workload has a 13 GiB footprint but an active working set of approximately 4 GiB. We compare four reclamation strategies, each with and without *OBASE* as a frontend:

- **Kswapd:** Linux's background reclaimer, triggered by memory pressure from a co-located process.
- **Cgroup:** Memory limit set to working-set size (4 GiB), forcing aggressive reclamation.
- **TMO:** Meta's PSI-based proactive reclaimer [62].
- ***OBASE* Hinted:** Proactive `MADV_PAGEOUT` on the COLD heap after convergence.

Throughout the evaluation, *OBASE* denotes the frontend alone, address-space reorganization with proactive paging disabled, so combinations like *OBASE*+TMO isolate the benefit of reorganization from any hinting. *OBASE* Hinted additionally issues `MADV_PAGEOUT` on the COLD heap.

Figure 9 reveals a fundamental trade-off that *OBASE* resolves. **Without *OBASE***, backends must choose between memory efficiency and performance. Kswapd reduces RSS from 13 GiB to 7 GiB (1.8×) with no throughput loss, but leaves 3 GiB of cold data trapped in mixed-temperature pages due to its page level view through PTE scans. Cgroup reaches 4 GiB (3.2×), but throughput collapses by 38% as the kernel inevitably evicts hot objects. TMO achieves a 6.5 GiB RSS (2×) with no throughput loss, but cannot reclaim further because PSI signals page-level pressure regardless of object-level coldness.

With *OBASE*, all backends reach 4 GiB RSS—matching the most aggressive policy—with no throughput degradation. Kswapd now reclaims COLD pages preferentially. Cgroup no longer thrashes because evicted pages contain genuinely cold objects. TMO's PSI probes no longer encounter mixed-temperature pages that resist reclamation. *OBASE* Hinted achieves the same result proactively, without relying on any backend policy.

Takeaway #2: *OBASE achieves aggressive reclamation while preserving application performance.*

5.3.2 Tiering Backends. To demonstrate the memory tiering benefits of reduced hotness fragmentation, we evaluate

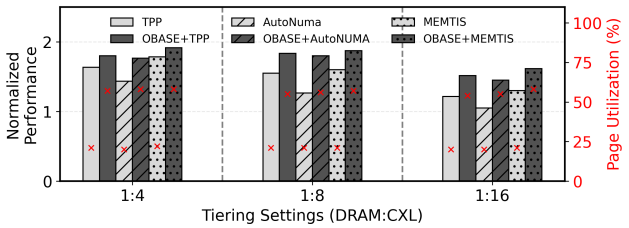


Figure 10. *OBASE* with tiering backends (YCSB-B, MassTree, 50M keys). Throughput normalized to CXL-only baseline (higher is better). Without *OBASE*, performance degrades as DRAM shrinks because the hot set cannot fit. With *OBASE*, the hot set compacts, enabling stable performance even at 1:16.

OBASE with three page based migration systems: TPP [49], AutoNUMA [60], and Memtis [44].

Setup. We load CrestDB (MassTree) with 50M keys (30-byte keys, 1024-byte values), a 67 GiB dataset. We vary the DRAM:CXL ratio across three configurations: 1:4 (14.8 GiB DRAM), 1:8 (7.4 GiB DRAM), and 1:16 (3.9 GiB DRAM), with the remaining capacity on Optane PMEM. As in [44], performance is normalized to a baseline where all data resides on the slow tier; values above 1.0× indicate speedup from effective DRAM use.

The hot-set mismatch. Without *OBASE*, the working set spans 16.3 GiB of pages at 21% utilization— slightly larger than the 1:4 DRAM budget of 14.8 GiB. No configuration can keep all accessed data in DRAM. With *OBASE*, the same logical working set compacts to 6.33 GiB at 57% utilization, fitting comfortably at 1:4 and 1:8 ratios.

TPP. TPP uses hysteresis-based promotion and proactive demotion to manage DRAM headroom. Figure 10 shows TPP achieves 1.65× speedup at 1:4, degrading to 1.25× at 1:16 as the DRAM budget shrinks below the fragmented hot set. With *OBASE*, TPP reaches 1.85× at 1:4 and retains 1.45× at 1:16—a 16% improvement at the most constrained ratio.

AutoNUMA. AutoNUMA promotes any accessed remote page immediately, without hysteresis, causing thrashing when the hot set exceeds DRAM capacity. Without *OBASE*, AutoNUMA underperforms TPP by 15–20%, achieving only 1.05× at 1:16—barely better than CXL-only. With *OBASE*, AutoNUMA matches TPP-alone performance: at 1:8, *OBASE*+AutoNUMA (1.6×) exceeds TPP alone (1.55×). When pages are uniformly hot or cold, even naive promotion decisions become correct.

Memtis. Memtis uses hardware sampling (PEBS) to identify hot pages, achieving the best baseline results: 1.8× at 1:4 and 1.55× at 1:16. With *OBASE*, Memtis improves to 1.95× and 1.7×, respectively. The gains are smaller (3–10% vs. 12–29% for TPP/AutoNUMA) because Memtis already captures much of the page-level signal—but even the most sophisticated page-level policy benefits from object-level reorganization.

Because *OBASE* reduces the effective hot set by 2.5×, ratio **1:X** performs comparably to baseline at **1:(X/2)**. For example, *OBASE* + TPP at 1:16 reaches 1.45×, within ~6% of TPP alone at 1:8 (1.55×). For operators, this means the same performance with half the DRAM, or double the effective capacity of existing tiered deployments.

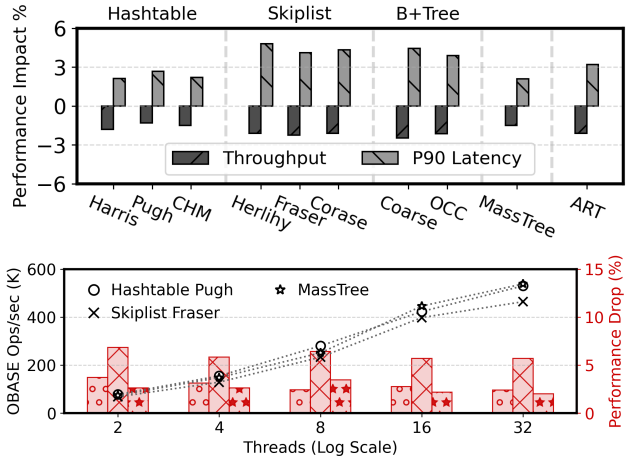


Figure 11. *OBASE* overhead and scalability (YCSB, 10M keys). Top: Throughput and p90 latency overhead across data structures. Overhead ranges from 1.5–5% depending on structure. Bottom: Scalability from 2 to 32 threads. Bars show absolute throughput; markers show overhead relative to baseline. Overhead remains bounded at 1–8% with no upward trend.

Takeaway #3: *OBASE compacts the hot set so it fits in smaller DRAM budgets, making page-based tiering backends effective.*

5.4 Overhead and Scalability (E3)

Given the memory savings and backend improvements demonstrated above, we now quantify *OBASE*'s runtime costs.

5.4.1 Steady-State Overhead. Figure 11(a) reports throughput and p90 latency overhead when no reclamation or tiering backend is active, normalized to an uninstrumented baseline.

On average, *OBASE* reduces throughput by 2.5% and increases p90 latency by 5%. Hash tables see the smallest impact (1.5–3% throughput drop), while skiplists, B+Trees, and ART experience 3–5% overhead. This variation correlates with the number of guides touched per operation: hash table lookups dereference few nodes, whereas tree traversals visit more nodes and incur proportionally more tracking overhead.

The overhead has two main sources: (1) a tagged-pointer read-modify-write on each guide dereference (4–5 ns, comparable to an L1 cache hit), and (2) TAG/ATC bookkeeping during `ACTIVE` migration epochs. The Object Collector runs in a dedicated thread and consumes less than 1% of CPU time.

5.4.2 Thread Scalability. A natural concern is whether *OBASE*'s atomic operations become contention bottlenecks at higher thread counts. We measure scalability by varying CrestDB server threads from 2 to 32 on three representative data structures spanning different synchronization mechanisms: Hashtable Pugh (fine-grained locking), Skiplist Fraser (lock-free), and MassTree (OCC with epoch reclamation).

Figure 11(b) shows that throughput scales and overhead remains bounded at 1–8% regardless of thread count. Critically, overhead does not increase with concurrency: all three data structures exhibit similar overheads at 2 and 32 threads.

This scalability follows from *OBASE*'s design: guide metadata updates target per-object state (no cross-thread contention), the test-and-set optimization skips redundant writes for hot objects, TAGs are thread-local, and ATC increments occur only during brief `ACTIVE` epochs. The 2–5% overhead is measured against a DRAM-only baseline with no memory pressure—an idealized scenario that production systems rarely enjoy. In the tiered-memory environments where *OBASE* is designed to operate, the comparison reverses: as §5.3 showed, backends *without OBASE* suffer 10–38% throughput loss from poor page-selection decisions, while backends *with OBASE* match DRAM-only performance.

Takeaway #4: *OBASE imposes 1.5–5% overhead across data structures and stays within 8% from 2 to 32 threads, a modest cost relative to the backend improvements in §5.2–5.3.*

5.5 Real World Traces

Synthetic workloads exercise controlled forms of skew, but production systems exhibit substantially more complex behavior: shifting hotsets, mixed read/write/delete ratios, and locality patterns that evolve over hours. We therefore evaluate *OBASE* on four real-world traces to assess whether its fragmentation-reduction benefits generalize beyond YCSB and whether the feedback controller adapts stably to long-term changes in access patterns.

We evaluate four traces that span a range of access patterns:

- **Meta CacheLib** [22]: Read-heavy (83% GET) with gradually shifting popular keys.
- **DBench Mixgraph** [24]: Models Meta's ZippyDB with key-range locality across 30 prefixes. Read-heavy (85% GET, 14% PUT, 1% SEEK).
- **Twitter Cluster 7** [65]: High skew (α=1.07) with small, concentrated working set of reads and writes.
- **Twitter Cluster 23** [65]: Write-heavy (31% SET, 30% INCR) with low skew (α=0.274) and deletes.

These traces cover the spectrum from highly skewed (Cluster 7) to nearly uniform (Cluster 23), and from read-dominated (CacheLib) to write-heavy (Cluster 23). We replay each trace on CrestDB with ART and measure memory reduction and page utilization improvement.

Page utilization (E1). Page utilization improves by 1.8–3.4× across traces. Cluster 23 shows the highest gain (3.4×) because its low skew disperses accesses across many keys, yielding very low baseline utilization. Cluster 7's high skew naturally concentrates accesses, so the baseline is already reasonable and the relative improvement is smaller (1.8×).

Memory reduction (E1, E2 & E4). Figure 12 shows that *OBASE* Hinted reduces RSS by 36–58% compared to no reclamation. Cluster 7 achieves the largest reduction (58%) because its high skew concentrates the working set into fewer hot objects; DBench shows the smallest (36%) because key-range locality spreads accesses more uniformly. Adding *OBASE* to TMO provides 15–30% additional savings relative to TMO-alone, demonstrating that *OBASE* and TMO are complementary: TMO identifies reclaimable pages, while *OBASE* ensures those pages contain uniformly cold data.

Figure 12. Production trace results. Memory reduction (left axis): *OBASE* Hinted vs. no-reclaim, and *OBASE*+TMO vs. TMO-alone. Page utilization improvement (right axis, hatched) from address-space reorganization.

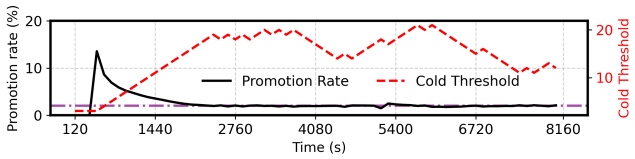


Figure 13. Cold threshold adaptation (Meta CacheLib). Promotion rate (black) and cold threshold C_t (red) over time. The controller automatically adjusts C_t to maintain the 1% target (purple).

Adaptive cold threshold (E4). Figure 13 demonstrates *OBASE*'s ability to adapt to workload dynamics over 2.3 hours of the Meta CacheLib trace. At startup, the initial C_t=3 causes premature demotions, spiking the promotion rate to 14%. The controller increments C_t each scan interval; within 25 minutes, C_t reaches 18 and the promotion rate drops below the 1% target. A high promotion rate during convergence *does not indicate degraded performance*: COLD-heap pages remain in DRAM until a backend explicitly pages them out, and the promotion rate reflects workload behavior rather than backend decisions (§3.4).

Around t=5400 s, a workload shift causes a brief spike, as previously cold keys become active; the controller raises C_t and the system re-converges as the new hotset stabilizes. Throughout the trace, C_t varies between 10–20 windows while maintaining the promotion rate near the target, demonstrating that *OBASE* tracks workload evolution.

Takeaway #5: *OBASE delivers consistent memory savings across real world workloads with diverse characteristics and dynamically adapts to shifting access patterns.*

6 Related Work

Object-Level Management. AIFM [57] and MIRA [35] manage memory at object granularity for disaggregated far memory, bypassing OS page management for specialized user-space runtimes; AIFM builds on Shenango's green threads and kernel-bypass networking, with explicit `DerefScope` guards around far-memory accesses. *OBASE* is complementary: it

acts as a *frontend* that reorganizes the address space for unmodified OS *backends*, optimizing *what* to tier rather than the latency of remote access, so we do not compare directly. The `DerefScope` contrast is instructive: it guards each individual far-memory dereference so the runtime can fault the object in, whereas *OBASE*'s TAG/ATC scopes bracket a whole public operation as a quiescence check that permits migration only when no thread is active, closer to epoch-based reclamation than to per-access load barriers.

Other systems indirect through pointers for different ends. Midas [55] harvests idle memory for application-managed *soft state*, dropping recomputable data under pressure, while Atlas [26] accelerates far-memory applications with a hybrid paging/runtime data plane; both optimize access to *remote* memory, whereas *OBASE* reorganizes *local* resident objects by observed access intensity so page-based backends see uniformly hot or cold pages. Alaska [61] uses handle-based indirection for heap compaction but does not track object hotness or organize memory by access intensity. ObjecTier [53] proposes a non-invasive object-consolidation framework with the same high-level vision as *OBASE*, motivated by simulations and they defer implementation to future work.

Allocation-Time Placement. Static placement approaches [27, 28, 40, 51, 66] make decisions at allocation time based on hints or profiling. However, object hotness is neither knowable at allocation nor stable over time(§2). *OBASE* tracks access patterns at runtime, enabling runtime migration.

Page-Level Tiering. TPP [49], Memtis [44], HawkEye [52], and TMO [62] all operate at page granularity. Even efforts to shrink the tiering granularity (e.g., Memtis's dynamic 4KB classification [44]) are bounded by the page floor, since subpage objects of differing hotness still share a 4KB page (§2). *OBASE* works below this floor by reorganizing at object granularity, and is designed to improve these backends.

Garbage Collection and Object Relocation. GCs have long relocated objects to improve locality: generational collectors cluster recent allocations, profile-guided reordering [39] groups hot objects during copying, and the Bookmarking Collector [38] avoids paging during collection. *OBASE* differs in relocation safety: where ZGC [63] and Shenandoah [30] use load barriers to redirect accesses while threads hold stale pointers, *OBASE* adopts a *quiescence-based* approach inspired by epoch-based reclamation [25, 31] and RCU [50], relocating only when an object's active-thread count reaches zero and aborting on any concurrent access via CAS failure. It also differs in purpose: GCs relocate to reclaim memory or improve cache locality, whereas *OBASE* relocates to improve *page utilization* so that tiering mechanisms see uniform pages.

7 Limitations and Applicability

OBASE targets a specific class of code, pointer-based concurrent data structures in unmanaged languages, rather than aiming for universal applicability. Its requirements bound where it applies, which we make explicit so users can judge fit.

- **No pointer stability.** Relocation invalidates raw pointers cached across operations (e.g., addresses retained by Abseil maps or STL iterators). Callers must re-resolve through the guide, as pointer-unstable containers already require [10].
- **Single ownership.** A guide asserts exclusive ownership like `std::unique_ptr`, and migration updates only that guide. Multiple guides aliasing one object (shared graph or doubly-linked nodes) are unsupported. CrestDB enforces this by deep-copying on insert.
- **No pointer arithmetic.** Guides do not support `+`/`[]` across object boundaries, so contiguous arrays, matrices, and packed columnar data are not supported in *OBASE*; a validation pass rejects such uses.
- **Language support.** The design relies on dereference-operator overloading, which is not supported by Go or Java.
- **Annotation.** Developers mark a small set of pointer fields, so adoption is not fully transparent; the compiler handles the rest (§4.7).
- **Hardware composability.** The guide encoding reuses high-order address bits, which can contend with Intel LAM [2], ARM TBI [9], or HWASAN [18] if those features claim the same bits; wider addressing [5] relaxes this.

We target unmanaged languages, as most latency-sensitive datacenter code is C++ or Rust; the mechanisms generalize to managed runtimes.

8 Conclusion

OBASE shows that memory tiering improves when applications cooperate with the OS rather than bypass it. By reorganizing virtual address space so that page boundaries align with object temperature, *OBASE* makes existing backends more effective without modifying them. The approach has limitations: it requires developer annotation of relocatable pointers and applies only to pointer-based structures. However, the principle of address-space engineering extends beyond tiering. The same techniques could improve generational garbage collection (grouping by access intensity rather than age), reduce false sharing of pages across NUMA nodes (separating objects by access pattern), and strengthen isolation (grouping by trust level). An application's address-space layout is itself a channel to the OS [20]: by grouping objects of similar temperature, *OBASE* turns layout into an object-level signal that unmodified page-based tiering can act on, with no new interface between the two.

Acknowledgments

We thank David Culler, Abhinav Sharma, Lilian Tsai, Qian Ge, Teresa Johnson, Sujay Yadalam, colleagues in SystemsResearch@Google and the ADvanced Systems Laboratory (ADSL), the anonymous reviewers, and our shepherd for valuable feedback. This work was supported by gifts from Google.

References

[1] 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherentinterconnect-strategy/

[2] 2021. Enable Intel LAM in Linux. https://lwn.net/Articles/902094/

[3] 2022. Compute Express Link. https://www.computeexpresslink.org/.

[4] 2022. Intel Optane DC PMM. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html

[5] 2022. Road to 128-bit Linux. https://lwn.net/Articles/908026/

[6] 2022. Samsung Memory-semantic ssd. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022

[7] 2024. Advanced profiling topics. PEBS and LBR. https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR

[8] 2024. DAMON: Data Access Monitor. https://sjp38.github.io/post/damon/

[9] 2024. Top-Byte-Ignore (TBI) ARM. https://en.wikichip.org/wiki/arm/tbi

[10] 2025. Abseil Pointer Instability. https://abseil.io/docs/cpp/guides/container#fn:pointer-stability

[11] 2025. Azure delivers cloud VM with Intel Xeon 6 and CXL memory. https://techcommunity.microsoft.com/blog/sapapplications/azure-delivers-the-first-cloud-vm-with-intel-xeon-6-and-cxl-memory---now-in-priv/4470067

[12] 2025. DynamoRIO: Dynamic Binary Instrumentation Framework. https://dynamorio.org/

[13] 2025. Google Workload Traces Version 2. https://console.cloud.google.com/storage/browser/external-traces-v2.

[14] 2025. LLVM-dev RFC:Sanitizer-based Heap Profiler. https://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html.

[15] 2025. Overview of Package util.concurrent. https://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html

[16] 2025. Tagged Pointers. https://en.wikipedia.org/wiki/Tagged_pointer.

[17] 2025. VM Linux Kernel Doc. https://docs.kernel.org/admin-guide/sysctl/vm.html.

[18] 2026. Hardware-Assisted AddressSanitizer Design Documentation. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[19] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. 2019. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM* 62, 11 (Oct. 2019), 114–120. https://doi.org/10.1145/3318163

[20] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[21] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. 2025. Tidying Up the Address Space. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)*. 63–72. https://doi.org/10.1145/3764862.3768179

[22] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[23] 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

[24] 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)*. 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[25] 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 (SIGMOD '18)*. 275–290. https://doi.org/10.1145/3183713.3196898

[26] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 77–95. https://www.usenix.org/conference/osdi24/presentation/chen-lei

[27] 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 (CGO 2020)*. 293–304. https://doi.org/10.1145/3368826.3377922

[28] 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 (EuroSys '16)*. Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[29] 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 (ASPLOS 2023)*. 727–741. https://doi.org/10.1145/3582016.3582031

[30] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

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

[32] Jim Gray and Goetz Graefe. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.* 26, 4 (Dec. 1997), 63–68. https://doi.org/10.1145/271074.271094

[33] Jim Gray and Franco Putzolu. 1987. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. 395–398. https://doi.org/10.1145/38713.38755

[34] 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

[35] 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 (SOSP '23)*. 692–708. https://doi.org/10.1145/3600006.3613157

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

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

[38] Matthew Hertz, Yi Feng, and Emery D. Berger. 2005. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 143–153. https://doi.org/10.1145/1065010.1065028

[39] 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

[40] 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

[41] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[42] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 2019. Faster than Flash: An In-Depth Study of System Challenges for Emerging Ultra-Low Latency SSDs . In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 216–227. https://doi.org/10.1109/IISWC47752.2019.9042009

[43] 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.). 317–330.

[44] 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 (SOSP '23)*. 17–34. https://doi.org/10.1145/3600006.3613167

[45] 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

[46] 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 (ASPLOS 2023)*. 574–587. https://doi.org/10.1145/3575693.3578835

[47] 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 (EuroSys '12)*. 183–196. https://doi.org/10.1145/2168836.2168855

[48] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[49] 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

[50] Paul Mckenney and JOHN SLINGWINE. 1998. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems* (01 1998).

[51] Svetozar Miucin and Alexandra Fedorova. 2018. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. 243–253. https://doi.org/10.1145/3240302.3240417

[52] 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 (ASPLOS '19)*. 347–360. https://doi.org/10.1145/3297858.3304064

[53] Vinita Pawar, Ankit Bhardwaj, and Ryan Stutsman. 2025. ObjecTier: Non-Invasively Boosting Memory Tiering Performance. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25)*. 180–186. https://doi.org/10.1145/3680256.3721319

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

[55] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. 2024. Harvesting Idle Memory for Application-managed Soft State with Midas. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1247–1265. https://www.usenix.org/conference/nsdi24/presentation/qiao

[56] 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 (SOSP '21)*. 392–407. https://doi.org/10.1145/3477132.3483550

[57] 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)*. 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[58] 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 (MICRO '23)*. 105–121. https://doi.org/10.1145/3613424.3614256

[59] 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 (EuroSys '20)*. Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[60] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing.Red Hat Summit.

[61] 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 (ASPLOS '24)*. 448–463. https://doi.org/10.1145/3620666.3651326

[62] 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 (ASPLOS '22)*. 609–621. https://doi.org/10.1145/3503222.3507731

[63] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (Sept. 2022), 34 pages. https://doi.org/10.1145/3538532

[64] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[65] 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)*. 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[66] 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 (ASPLOS '24)*. 192–206. https://doi.org/10.1145/3620666.3651350