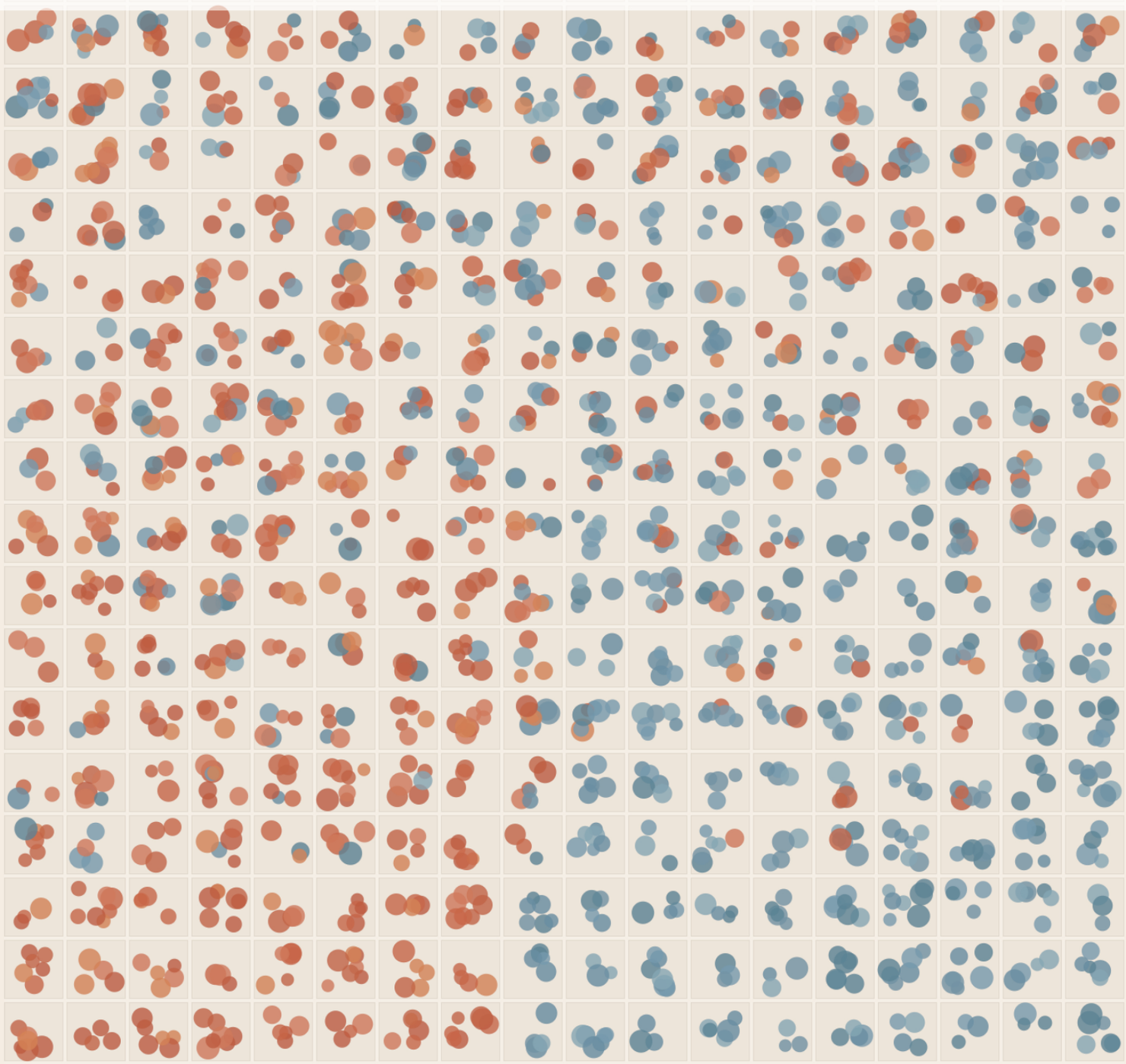
Layout-Aware Data Organization

For Heterogeneous Hierarchies



Vinay Banakar

More great ways to use key-value separation
- Mark Callaghan (Distinguished Database Architect)

That's a lot of platters - Marc Brooker
(Distinguished Engineer at AWS)

WiscSort nicely articulates why thinking of byte-addressable storage as 'slow DRAM' or 'fast disk' is naive - VLDB'23 Reviewer

The idea isn't life-changing, but it is sound and should be published - VLDB'23 Reviewer

This is the first work that I am aware of demonstrating that object-level temporal tracking improves amongst any existent solution... - SOSP DIMES'25 Reviewer

The core idea of object-level tracking for memory tiering is compelling and addresses an important systems problem
- SOSP'25 Reviewer

Targets a real and timely systems problem – OSDI '26 Reviewer

This is really exciting and might be massive on low memory systems - Phoronix community

Layout-Aware Data Organization For Heterogeneous Hierarchies

by

Vinay S. Banakar

B.E., PES University; M.S., University of Wisconsin–Madison

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: Feb 20th, 2026

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Kimberly Keeton, Principal Software Engineer, Google
Michael M. Swift, Professor, Computer Sciences
Shivaram Venkataraman, Associate Professor, Computer Sciences
Dimitris Papailiopoulos, Associate Professor, Electrical Engineering

Dedicated to my mom, who taught me to dream and
my dad, for the courage and support to chase them.

Acknowledgments

I am privileged to be a first-generation computer scientist, hailing from a long line of farmers and artists. A dissertation bears only one name, but such a simplification conceals the sacrifices, the love, and the quiet acts of belief that made it possible. It is these people who deserve the most recognition.

My thanks to my beloved advisors Andrea and Remzi Arpaci-Dusseau have no bounds. Their strive for meaningful impact on the world while insisting on intellectual and fundamental contribution is inspiring. Their attitude toward students as peers and collaborators is empowering; they gave me space to grow and discover myself as a researcher rather than prescribing a path, and in doing so built my confidence and independence. Perhaps most liberating of all, they taught me a philosophy of research rather than a research topic, a gift to someone with varied interests. I am honored to have been their student, their collaborator, and their friend over the last six years. Although I complained that being a teaching assistant for their operating systems and distributed systems courses took me away from research, it quietly sparked an interest in teaching and mentoring that I now hope to carry forward. For this, I am thankful.

I am grateful to my committee members for their feedback on this dissertation. Kim Keeton has been a long-term mentor, and an inspiration to work with; her enthusiasm and dedication are contagious. I had the pleasure of working with her before my PhD, and her belief in my potential and continued encouragement have been invaluable. I hope to work with her long after this as well. Mike Swift is a marvel. His candid feedback and encourage-

ment have been essential to my growth, and it is always interesting to talk to him about anything: research, running, biking, or life itself. He showed me how to hold joy for research and life at the same time. I had the pleasure not only of collaborating with him but also of teaching in one of his courses (CS537). Shivaram Venkataraman and Dimitris Papailiopoulos have taught me so much about AI research and how to weave our ideas into the ML stack.

David Culler has played an influential role in my world view. His first piece of advice to me was: "Step one of doing research is to truly believe one can do research". Spending the last three years with him has been a joy, from cooking together, to going surfing, to sharing bus rides and conversations. I would also like to thank Philip Levis, Hank Levy, and many others in the Systems Research Group at Google for their support and feedback on my work, and especially the many friends I made there. Mike Marty, Dan Gibson, Florentina Popovici, and many others in Google Madison made my time there memorable.

My peers, collaborators, and Madison friends made this journey so much more enjoyable. Sujay Yadalam, Konstantinos Kanellis, Siddharth Suresh, Sambhav Satija, Bobbi Yogatama, Nicholas Roberts, Jihye Choi, Varun Sundar, Anthony Rebello, Ruohong Wang, Ojaswita Lokre, Naman Gupta, Anjali, Ashwin Poduval, Hayden Coffey, Mark Mansi, Karan Grover, Surabhi Gupta, Gowtham Ramesh, Keerthana Desai, John Shwager, Dyah Adila, Johannes Freischuetz, Sahana Upadhya, and many more. Thank you for the conversations, the camaraderie, and the shared experience.

Kan Wu, Suli Yang, and Yuvraj Patel deserve special thanks for their support in the early years of my PhD, when I needed it most. I am lucky to have had other wonderful ADSL lab members like Aishwarya Ganesan, Chenhao Ye, Guanzhou Hu, Jing Liu, Kai Mast, Kaiwei Tu, Ramnatthan Alagappan, Tingjia Cao, Shawn Zhong, Suyan Qu, Yifan Dai, Caeden Whitaker, Junxuan Liao, and many more. Each of them has taught me something, and I am grateful to have interacted with them. ADSL alumnus Vijay Chidambaram

gave me the first opportunity to do academic research as a fellow at the University of Texas at Austin, and I am forever grateful to him for betting on me. My work with Vijay, Supreeth Shastri, Aashaka Shah sparked my interest in pursuing a PhD.

My time at HPE and PES University laid the groundwork for much of what I appreciate in life today. I would like to thank Maneesh Keshavan, Pavan Upadhyay, K.V. Subramanian, Sushanth Hegde, Sujay Kumar, Venkat Datta, Vinayshekar Bannihatti, Akshay Swamy, and Suresh Krishna. They gave me encouragement and opportunities at key decision points. They played an important role in shaping my life as it is today.

Of all the people, my parents, Rekha and Sahadevappa Banakar have given me the most. Their support at every step has been unwavering, always doing their best, and always believing in me before I did. My sister, Arpitha, has always patiently listened to my many endless thoughts and been the one to remind me that there is a world beyond research.

Finally, for the last fifteen years, my wife, Sushma, has been my rock. Her love, support, and belief in me have been the foundation of everything I have achieved. I am grateful for her patience and understanding during the long hours and the ups and downs of this journey. Her support has allowed me to be a perpetual student, pursuing my passion for science and learning.

Contents

List of Tables

List of Figures

Abstract

Modern operating systems manage memory at page granularity, but applications access data at the granularity of objects and fields. This mismatch has always produced inefficiency, but two shifts in the memory hierarchy are making it impossible to ignore: byte-addressable storage devices that sit between DRAM and block storage, and memory tiering systems that migrate pages across capacity tiers to reduce DRAM provisioning. On byte-addressable storage, page-level data movement wastes bandwidth by transferring cold bytes alongside hot ones. Under memory tiering, a single hot object on a page traps its cold neighbors in expensive DRAM. This dissertation argues that efficient use of modern memory hierarchies requires layout-aware organization, and identifies two regimes: static layout when application semantics reveal which data will be hot and which will be cold, and dynamic layout when access patterns depend on workload behavior and must be discovered through observation.

For the static regime, we study external sorting on byte-addressable storage. We introduce the BRAID model, which characterizes five properties of these devices that invalidate the assumptions of conventional storage algorithms: byte addressability, high random-read performance, asymmetric read-write costs, read-write interference, and device-constrained concurrency. We present WiscSort, a sorting system that exploits the semantic structure of sorting by separating keys from values and deferring value retrieval until sorted order is known. Combined with interference-aware scheduling and thread-pool sizing tuned to device characteristics,

WiscSort achieves 2 to 7 times the throughput of competing approaches on industry-standard benchmarks.

When application semantics do not reveal access structure, static layout is insufficient. We quantify this limitation by measuring hotness fragmentation in production workloads at Google, Meta, and Twitter. The fragmentation is pervasive: across six Google workloads, up to 97% of bytes in active pages are cold data trapped by hot neighbors, and production traces confirm that object hotness shifts continuously over time. Because neither the application nor the OS can predict which objects will be hot, dynamic reorganization is necessary.

To address hotness fragmentation, we propose address-space engineering: dynamically reorganizing the virtual address space so that objects of similar access intensity reside together. We present *OBASE*, a compiler-runtime system for unmanaged languages that serves as an object-aware frontend for page-aware OS backends. *OBASE* introduces a guide abstraction that enables object relocation in C++, tracks accesses via lightweight pointer instrumentation, and migrates objects between hot and cold heaps using a lock-free protocol that is safe under concurrency. By reorganizing the address space, *OBASE* enables unmodified backends to tier memory effectively without understanding object boundaries. Across ten concurrent data structures, six tiering backends, and production traces from Meta and Twitter, *OBASE* improves page utilization by 2 to 4 times and reduces memory footprint by up to 70%, with only 2 to 5% performance overhead.

Together, WiscSort and *OBASE* demonstrate that layout-aware data organization, whether guided by application semantics or inferred from observed access patterns, enables modern memory hierarchies to achieve their potential by aligning page boundaries with access intensity.

Chapter 1

Introduction

The page is the dominant unit of address translation, protection, and memory placement in modern operating systems. Since the Atlas computer introduced paging in 1962 [94], operating systems have managed memory in fixed-size units called pages. Paging replaced manual overlays with OS-managed movement between memory and a backing store, enabled virtual memory, and simplified physical allocation by reducing external fragmentation. Every major operating system today manages memory at page granularity, and the 4 KiB page size remains the default nearly six decades later [153].

This choice fixes the OS observation and control granularity at the page. Applications operate on objects, fields, and records, but the OS accounts for pages and makes placement decisions using page-level signals. Most architectures expose accessed and dirty state per page table entry, so activity within a page is aggregated into a single indicator.

This mismatch between object-level access and page-level management becomes more costly under two trends. First, byte-addressable capacity tiers, including persistent memory and CXL-attached memory [13, 8], sit between DRAM and block storage and make data movement costs visible at finer granularity. Second, memory tiering systems migrate pages across tiers based on observed page access patterns, using either page migration to a slower memory node or swapping to an SSD-backed backing

store [116, 101, 152, 131, 99, 114]. These systems aim to reduce DRAM provisioning by placing cold data in cheaper tiers, but page granularity limits reclamation when hot and cold objects share pages. **This dissertation argues that efficient use of modern memory hierarchies requires layout-aware organization. Data layout should align with access patterns, and the mechanism for achieving that alignment depends on whether those patterns are predictable from semantics or must be inferred empirically.**

The work presented here yields several observations that extend beyond the particular systems we build. First, when hardware changes in ways that invalidate existing assumptions, the most productive first step is to model the device, not to modify the algorithm; a clear model constrains the design space and makes it possible to evaluate alternatives against a common set of criteria. Second, reducing the amount of data moved matters more than increasing the rate at which data is moved; bandwidth is finite on every storage and memory device, and the most effective way to conserve it is to avoid spending it on data that does not need to move. Third, the virtual address space is not merely storage; it is a communication channel between applications and the operating system, and by engineering this channel, applications can express object-level semantics in a language that page-based systems already understand. These lessons emerge from the two systems developed in this dissertation, WiscSort and OBASE, and we believe they will remain relevant as memory hierarchies grow deeper and the devices within them grow more diverse.

1.1 The Semantic Gap

Applications access data at object and field granularity. A hash table lookup probes keys but may never touch the corresponding values if the lookup fails. A sorting algorithm compares keys throughout execution but touches values only when writing final output. A key-value store serves requests for

individual records, with some keys accessed millions of times per second and others never touched after insertion [108, 43, 51, 56, 57]. In each case, the application's view of memory is rich with semantic structure: some data is accessed frequently, some rarely, and the application's logic determines which is which.

The operating system, however, manages memory at page granularity. Hardware provides only coarse signals: an *accessed bit* in each page table entry indicates whether any byte on the page was read or written since the bit was last cleared, and a *dirty bit* indicates whether any byte was modified. The OS uses these bits to approximate recency and make placement decisions, but it cannot see finer structure. When a single cache line on a 4 KiB page is accessed, the entire page appears active. When one hot object shares a 2 MiB huge page with thousands of cold objects, the OS sees only that the huge page was touched.

This mismatch creates three problems. First, *bandwidth waste*: fetching or migrating a page moves cold bytes alongside hot ones. On byte-addressable devices where bandwidth is the limiting resource, transferring data that will never be used squanders the device's potential. Second, *trapped cold data*: if even one object on a page is hot, the entire page remains pinned in fast memory. The OS cannot reclaim the cold bytes because they share a page with hot neighbors, leaving theoretically reclaimable memory trapped in expensive DRAM. Third, *noisy signals*: page-level access bits conflate distinct objects with different access patterns. A frequently touched metadata field makes the entire page appear hot, obscuring the fact that most of the page's capacity holds cold payload data.

The granularity gap has always imposed costs, but these costs were difficult to measure and easy to overlook. When memory pressure forced the OS to swap pages to disk, the latency penalty was so severe (millions of cycles for a disk seek) that applications simply avoided memory pressure at all costs. DRAM was provisioned generously, swap was treated as a failure

mode rather than an optimization opportunity, and the inefficiency of page-level management remained hidden. But the costs were real, even when hidden. The term *random-access memory* implies that any part of memory can be accessed as quickly as any other. In practice this is not so. Culler's Law [40] captures the principle that **RAM isn't always RAM**: sometimes randomly accessing the address space, particularly when the number of pages touched exceeds the TLB's coverage, leads to severe performance penalties. The TLB caches recent virtual-to-physical translations, and a hit adds negligible latency; a miss, however, triggers a hardware page table walk costing hundreds of cycles. When hot objects scatter across the address space rather than clustering on a small number of pages, the TLB working set grows unnecessarily, and every miss pays the full walk cost. The page abstraction creates the illusion that all memory is equally accessible, but the reality is that layout determines performance. As memory hierarchies grow richer and the performance gaps between tiers narrow, these layout-dependent costs become impossible to ignore.

1.2 The Rise of Tiered Memory

Two technological shifts are reshaping the memory hierarchy and exposing the costs of page-level management. The first is the emergence of byte-addressable storage devices that sit between DRAM and traditional block storage in both performance and capacity. The second is the growing adoption of memory tiering, where slower memory serves as a capacity tier beneath DRAM to reduce costs. Both shifts create opportunities for more efficient memory use, but both are constrained by the granularity gap.

Byte-addressable storage (BAS) devices include Intel Optane DC Persistent Memory [13], CXL-attached memory [8, 106, 105], and emerging products like Samsung's CXL Memory-Semantic SSD [20]. These devices share characteristics that differ fundamentally from traditional HDDs and

SSDs. Random read performance approaches sequential read performance because there are no mechanical seeks to amortize. Fine-grained access incurs minimal amplification because the devices support byte-level addressing rather than fixed block transfers. Writes are slower than reads, and concurrent writes can interfere with read bandwidth [159]. These properties invert the traditional storage tradeoffs. Algorithms designed for HDDs [112, 38, 124] bundled related data together to amortize seek costs, accepting that some bundled data might never be used. On BAS devices, this bundling wastes bandwidth by transferring bytes that serve no purpose. Chapter 2 provides detailed characterization of these device properties.

Memory tiering extends the hierarchy by treating slower memory as a capacity tier that applications can use transparently. The motivation is economic: DRAM accounts for a substantial fraction of server cost and power consumption [106, 144, 5], yet production workloads access only a small fraction of their allocated memory at any given time. Across six Google production workloads, only 1.7% to 21.3% of bytes are accessed during measurement periods [28]. If cold data could be placed in cheaper tiers, operators could reduce DRAM provisioning significantly. This observation has driven the development of tiering systems including TPP [116], Memtis [101], TMO [152], and kernel mechanisms like zswap [99]. Modern ultra-low-latency SSDs have also made swap viable again for production workloads, enabling memory offloading to storage tiers that would have been unacceptably slow a decade ago [97, 114].

In principle, tiering shows great promise. If 80% to 98% of bytes are cold, that memory could reside in cheaper tiers with minimal performance impact. In practice, realized savings fall short of this potential. Google reported offloading only 20% of infrequently accessed data to a compressed memory tier [99]. Meta achieved 20% to 32% savings with TMO [152]. The limiting factor is not policy sophistication but the granularity gap.

The problem is *hotness fragmentation*: hot and cold objects intermingle

within pages, preventing effective page-level tiering. Modern allocators place objects on pages based on allocation order and size class, with no regard for how those objects will be accessed [165, 69]. A hash table that interleaves key and value allocations will have keys and values sharing pages, even though keys may be probed on every lookup while values are touched only on successful matches. From the OS perspective, these pages appear uniformly active because the hot keys keep the access bits set. The cold values are trapped alongside their hot neighbors.

We quantify this fragmentation in Chapter 4 using a metric called *page utilization*: the fraction of bytes within touched pages that are actually accessed. Across production workloads, page utilization ranges from 8% to 50%. For workloads using 2 MiB huge pages, 85% to 90% of huge pages utilize less than 10% of their capacity. In concrete terms, for every 100 MiB of pages the OS considers active, over 90 MiB may be cold data that the OS cannot reclaim because it shares pages with hot data. This fragmentation explains the gap between theoretical and realized tiering savings: tiering systems can only reclaim pages, and most pages contain at least some hot data.

1.3 Thesis

The granularity gap is a consequence of the page abstraction, not a bug to be fixed in any particular system. The OS will continue to manage memory at page granularity because pages provide the right abstraction for translation, protection, and portability. We cannot change how the OS observes memory. But we can change how data is organized so that page-level observations become meaningful.

This dissertation argues that efficient use of byte-addressable and tiered memory requires *layout-aware organization*: structuring data so that page boundaries align with access intensity. The key insight is that the method for achieving this alignment depends on what is knowable about access

patterns.

> *"The page is a convenient but poor-performance abstraction for modern memory hierarchies. It forces coarse-grained decisions on fine-grained data, wasting bandwidth and trapping cold data in expensive memory. Efficient use of byte-addressable and tiered memory requires layout-aware organization: when access semantics imply stable hot/cold structure, we can separate statically (WiscSort); but when hotness is unknown and evolves, we must reorganize the address space dynamically to make page-based tiering effective (OBASE).*"

The thesis identifies two regimes distinguished by the nature of access patterns. In the first regime, application semantics reveal which data will be hot and which will be cold. For example, sorting algorithms compare keys but do not examine values until producing output. This property is intrinsic to the algorithm, independent of workload or input distribution. When such semantic structure exists, layout can be engineered at design time: separate keys from values, store them in different regions, and access each according to its role in the computation.

In the second regime, access intensity depends on workload characteristics that are unknown at design time and change during execution. A key-value store cannot know at allocation time which keys will be popular. Even if it could, popularity shifts over time as user behavior changes. Production traces show that keys alternate between active bursts and extended dormancy, with reuse distances varying by more than an order of magnitude for the same key [51, 160]. When hotness is dynamic, static layout degrades as access patterns evolve. The system must observe access patterns at runtime and reorganize objects continuously to maintain alignment between page boundaries and access intensity.

Both regimes share the same goal: making page-level decisions effective by ensuring that pages contain data of uniform temperature. They differ

in when and how layout decisions are made. This dissertation contributes techniques for both regimes: WiscSort for the static case where semantics guide layout, and OBASE for the dynamic case where observation guides reorganization.

The two approaches are complementary rather than competing; the choice between them is determined by the nature of the application's access structure, and recognizing which regime a given system falls into is itself a useful design heuristic.

1.4 WiscSort: Static Layout

When application semantics reveal which data will be hot and which will be cold, layout can be engineered at design time. External sorting provides a clear example. A sorting algorithm compares keys to establish order but does not examine values until writing final output. This property is intrinsic to the algorithm: it holds regardless of input distribution, key cardinality, or workload characteristics. The semantic structure of sorting is known before any data is processed.

Traditional external merge sort does not exploit this structure. During run generation, the algorithm reads complete records (keys and values together) into memory, sorts them, and writes sorted runs to storage. During merge, it reads runs back, selects the smallest key across all runs, and writes complete records to output. Values move through every phase even though sorting never examines them.

This design made sense for HDDs and SSDs. On rotational disks, sequential access amortizes seek costs that dominate random access latency. Bundling keys with values enables sequential scans rather than scattered reads. On SSDs, block-level transfer granularity (typically 4 KiB) means that reading a small key still transfers an entire block. Separating keys from values would save nothing if each key read still moved 4 KiB. Under

these device characteristics, moving values along with keys costs little and simplifies buffer management.

Byte-addressable storage inverts these tradeoffs. Random read bandwidth approaches sequential read bandwidth because there are no seeks [159]. Fine-grained access incurs no amplification because the device supports sub-block addressing. Writes are substantially slower than reads and do not scale well with concurrency. On such devices, the traditional approach wastes write bandwidth moving values through intermediate phases where they serve no purpose.

Before redesigning the sorting algorithm, we first characterized these device properties systematically. The result is the BRAID model, a simple enumeration of five properties that distinguish byte-addressable storage from its predecessors: Byte addressability, high Random-read performance, Asymmetric read-write costs, read-write Interference, and Device-constrained concurrency. Once these properties were made explicit, the design of WiscSort followed from checking which properties each design choice respected or violated.

WiscSort exploits the semantic structure of sorting on byte-addressable storage. It separates keys from values, maintaining only key-pointer pairs during run generation and merge. Pointers reference values in the original input file. Only in the final phase, when sorted positions are known, does WiscSort gather values using random reads. This late materialization eliminates value writes during intermediate phases. For workloads where values are larger than keys, which is common in practice [55], the write savings are substantial.

Because WiscSort maintains only keys and pointers in memory, it can sort more keys per run, reducing or eliminating merge phases entirely. A thread-pool controller sizes read and write pools according to device concurrency characteristics. An interference-aware scheduler ensures that reads and writes do not overlap, avoiding the bandwidth degradation that

occurs when concurrent writes interfere with reads on these devices [159].

On Intel Optane Persistent Memory, WiscSort achieves 2 to 3 times the throughput of concurrent external merge sort on industry-standard benchmarks [21]. It outperforms in-place sample sort by 5 times and prior PMEM-optimized sorting by 7 times. The improvement stems not from moving data faster, but from moving less data: by aligning data layout with the semantic structure of sorting (keys are hot, values are cold until the end) and the characteristics of byte-addressable devices (fine-grained access, read-write asymmetry), WiscSort avoids transferring bytes that serve no purpose. WiscSort consumes less device bandwidth than external merge sort yet finishes in less time, precisely because it reads and writes fewer total bytes. The BRAID model itself is likely to remain relevant as new byte-addressable devices emerge, since the five properties it captures reflect physical characteristics of non-volatile media rather than quirks of a particular product generation. The specific concurrency parameters will change as devices evolve, but the need to account for read-write asymmetry, interference, and constrained concurrency is likely to persist. Chapter 3 presents the complete design, implementation, and evaluation.

1.5 When Semantics Are Insufficient

WiscSort succeeds because sorting's access structure is determined by the algorithm, not by external factors. But many applications lack this property. Key-value stores, caches, and databases serve workloads where access patterns depend on user behavior, application logic, and temporal dynamics that cannot be known at design time. For these applications, static layout based on semantic analysis is insufficient.

Consider a key-value store handling user requests. At allocation time, every key-value pair looks the same: a key of some size, a value of some size, inserted by some client. The allocator places the pair in the next available

slot of the appropriate size class. Nothing in the allocation request reveals whether this key will be accessed millions of times per second or never touched again. The semantic structure that made WiscSort possible, the distinction between keys accessed throughout and values accessed only at the end, does not exist here. All keys have the same role in the data structure; their access intensity depends entirely on workload.

Worse, even if access intensity could somehow be predicted at allocation time, it would not remain stable. We analyzed production traces from Meta [51] and Twitter [160] to understand how object hotness evolves. The traces record operations in logical time, allowing us to track access patterns for individual keys over extended periods. If hotness were stable, popular keys would show consistent access throughout the trace. Instead, we observe keys alternating between bursts of activity and extended dormancy. For the majority of keys in the Meta traces, the gap between consecutive accesses varies by more than an order of magnitude: a key accessed every millisecond during one period may go untouched for minutes during another.

This temporal variation means that any static placement degrades over time. Objects allocated together because they appeared hot will eventually include objects that have gone cold. Objects allocated in cold regions will include some that became hot. The result is the hotness fragmentation we described earlier: hot and cold objects intermingled within pages, preventing effective page-level management.

Establishing the severity of this fragmentation required measuring access behavior at a granularity finer than the page. We define *page utilization* as the fraction of bytes within touched pages that are actually accessed during a measurement window. This metric bridges the gap between the OS view (which page was touched) and the application view (which bytes were used), making the problem quantifiable rather than anecdotal. Low utilization indicates that pages contain mostly cold data kept resident by a few hot bytes. Across six Google production workloads [28], median page utiliza-

tion ranges from 8% to 50%. Even at the 4 KiB page granularity, which should be more precise than huge pages, 60% to 80% of pages in some workloads utilize less than 20% of their capacity. For 2 MiB huge pages, fragmentation is severe: 85% to 90% of huge pages utilize less than 10% of their 2 MiB capacity.

This fragmentation has concrete costs. First, memory that the OS considers active is mostly cold. Across the Google workloads, 55% to 98% of bytes within touched pages receive no accesses, yet the OS cannot reclaim them because they share pages with hot data. Second, scattered hot objects inflate the TLB working set. When hot data spreads across many pages, each page requires a TLB entry even though most of its capacity is unused. Microbenchmark measurements show that clustering hot objects can reduce TLB miss rates by up to 2 times compared to scattered placement at the same total data size. Third, the inflated memory footprint forces overprovisioning. Applications reserve memory based on their apparent working set (touched pages), not their actual working set (touched bytes), leaving resources stranded.

Chapter 4 presents detailed analysis of hotness fragmentation across open-source and production workloads, demonstrates the temporal dynamics of object hotness, and quantifies the costs of fragmentation for memory reclamation, address translation, and infrastructure provisioning.

1.6 OBASE: Dynamic Layout

When access patterns are workload-dependent and temporally varying, layout must be inferred from observation and adjusted continuously. We introduce *address-space engineering*: reorganizing the virtual address space so that objects with similar access intensity reside together. By clustering hot objects onto hot pages and cold objects onto cold pages, we enable page-based tiering mechanisms to work effectively without requiring them to understand object boundaries.

OBASE (Object-Based Address-Space Engineering) realizes this approach for C++ applications. Unlike managed languages where garbage collectors can relocate objects freely, C++ programs assume that object addresses remain stable after allocation [53]. Raw pointers may be stored in registers, on the stack, in other heap objects, or serialized to storage. Moving an object without updating every pointer that references it would corrupt the program.

OBASE solves this problem for pointer-based data structures, where elements are accessed through pointer indirection rather than direct offset calculation. Hash tables, trees, skip lists, and linked structures fall into this category, and they dominate memory-intensive applications. OBASE introduces a *guide* abstraction that interposes on pointer dereferences. A guide holds the current address of an object along with metadata for access tracking. When application code dereferences a guide, OBASE resolves it to the object's current location. If the object has moved, the guide transparently redirects to the new address. This indirection enables relocation without modifying application logic or requiring whole-program pointer analysis.

Access tracking is embedded in guide dereferences. Each access sets a bit indicating that the object was touched during the current observation window. The tracking mechanism uses unused bits in 64-bit pointers to store per-object metadata, avoiding external side tables and keeping the common path fast. A background thread called the Object Collector periodically scans this metadata to classify objects by access intensity. Objects touched recently are candidates for the HOT heap; objects untouched for multiple windows migrate to the COLD heap. A spatially-aware allocator ensures that each heap occupies a contiguous virtual address range, so an object's address directly encodes its temperature classification.

Migration must be safe under concurrency. Application threads cannot block waiting for the collector, and the collector cannot move objects while threads hold pointers to them. OBASE uses an optimistic protocol inspired

by optimistic concurrency control. The collector copies an object to its target heap, then atomically attempts to update the guide. If any thread accessed the object during the copy, the commit fails and the object remains in place. Threads never block; concurrent access safely vetoes migration rather than observing inconsistent state. An epoch-based mechanism activates the necessary tracking only during migration windows, eliminating overhead when no migration is in progress.

OBASE operates as a *frontend* that prepares the address space for page-based *backends*. This decomposition reflects a genuine information boundary: the frontend has access to application-level semantics that the OS cannot see, while the backend has access to hardware state that the application cannot see. Rather than replacing existing tiering infrastructure, OBASE respects this boundary. Kernel mechanisms like kswapd and cgroup memory limits, datacenter systems like TMO [152], and research prototypes like TPP [116] and Memtis [101] continue to make page-level decisions using their existing policies. OBASE simply ensures that when these backends examine cold pages, those pages contain only cold data. This decoupling allows independent innovation: improvements to OBASE benefit any backend, and improvements to backends benefit any application using OBASE.

We evaluate OBASE with ten concurrent data structures spanning different synchronization mechanisms: lock-free algorithms, fine-grained locking, and optimistic concurrency control. Using YCSB workloads [63] with skewed access distributions, OBASE improves page utilization by 2 to 4 times across structures and workloads. When combined with reclamation backends, OBASE reduces memory footprint by up to 70% while maintaining throughput within 5% of baseline. On tiered memory configurations with CXL-attached capacity, OBASE achieves equivalent throughput with half the DRAM budget compared to systems without address-space engineering. Replay of production traces from Meta and Twitter demonstrates that OBASE adapts automatically to shifting access patterns, maintaining

high page utilization as the hot working set evolves over multi-hour runs. The principle of address-space engineering extends beyond tiering. The same techniques could improve generational garbage collection by grouping objects by access intensity rather than age, reduce false sharing of pages across NUMA nodes by separating objects with different access patterns, and strengthen isolation in multi-tenant environments by grouping objects by trust level. Chapter 5 presents the complete design, the compiler passes that automate guide instrumentation, and detailed evaluation across backends and workloads.

1.7 Overview

The remainder of this dissertation is organized as follows.

- **Background.** Chapter 2 provides technical foundations. We trace the history of the page abstraction from the Atlas computer through modern page tables, explaining why 4 KiB became the dominant page size and why it has proved so difficult to change. We characterize byte-addressable storage devices and survey memory tiering systems including AutoNUMA, TPP, Memtis, and TMO, identifying the common constraint that all of them share: they observe and act on memory at page granularity. We also review modern memory allocators, whose placement decisions at allocation time are the origin of hotness fragmentation.

- **WiscSort.** Chapter 3 provides external sorting fundamentals and byte-addressable device properties. We then present a sorting system that exploits semantic knowledge for layout-aware organization on byte-addressable storage. We introduce the BRAID device model and describe key-value separation, thread-pool control, and interference-aware scheduling. WiscSort achieves 2 to 7 times the throughput of

competing approaches, demonstrating that aligning data layout with device properties and algorithmic structure can recover performance that conventional designs leave on the table. We evaluate WiscSort on Intel Optane PMEM and project performance on emulated future devices, showing that the design principles hold across a range of device parameters.

- **Dynamic Object Reorganization.** Chapter 4 establishes the need for dynamic address-space engineering when semantic knowledge is unavailable. We introduce the page utilization metric and quantify hotness fragmentation in open-source and production workloads, finding that 55% to 98% of bytes within active pages are cold data that the OS cannot reclaim. We analyze traces from Meta and Twitter to demonstrate that object hotness is neither predictable at allocation time nor stable over an object's lifetime, confirming that allocation-time placement, no matter how sophisticated, cannot solve the fragmentation problem. These findings motivate the principles for address-space engineering that guide the design of OBASE.

- **OBASE.** Chapter 5 presents OBASE, a compiler-runtime system that dynamically reorganizes the address space for C++ applications. We describe the guide abstraction, lightweight access tracking, the Object Collector, and the lock-free migration protocol. OBASE improves page utilization by 2 to 4 times and reduces memory footprint by up to 70%, while eliminating the memory-versus-performance tradeoff that forces datacenter operators to balance aggressive reclamation against throughput. We evaluate OBASE across ten data structures, six tiering backends, and production traces, showing that it enables unmodified backends to achieve near-optimal memory savings.

- **Related Work.** Chapter 6 discusses related work on sorting for non-volatile memory, memory tiering systems, object-level memory man-

agement, garbage collection, and epoch-based reclamation, situating our contributions within the broader research landscape.

- **Conclusion.** Chapter 7 summarizes the contributions of this dissertation, discusses future directions including sorting across heterogeneous storage, direct object tiering across memory hierarchies, and address-space engineering beyond tiering, and reflects on lessons learned during the research.
- **Public Overview.** Chapter 8 provides a public-facing narrative of the dissertation's core ideas, using the warehouse analogy to explain memory hierarchies and layout-aware organization.

Chapter 2

Background

This chapter provides the technical foundations for the dissertation. Each section introduces concepts and mechanisms that later chapters build upon.

We begin with the page abstraction (§2.1), which defines how operating systems observe and manage memory. The page is the unit at which the OS tracks access, enforces isolation, and makes placement decisions. Understanding why page-level management creates a mismatch with object-level access is essential context for every subsequent chapter: WiscSort (Chapter 3) exploits this mismatch by organizing data at finer granularity on byte-addressable storage, Chapter 4 quantifies the costs of the mismatch in production workloads, and OBASE (Chapter 5) reorganizes the address space to reduce it.

We then survey the evolving memory hierarchy (§2.2), focusing on two developments that expose the costs of page-level management: byte-addressable storage devices whose properties motivate the BRAID model and WiscSort (Chapter 3), and memory tiering systems whose effectiveness is limited by hotness fragmentation (Chapters 4 and 5).

Because tiering systems build upon the Linux kernel's page reclamation infrastructure, we devote a full section to Linux memory reclamation (§2.3). This material is directly relevant to Chapter 5, where OBASE interacts with kswapd, cgroup limits, and `madvise` hints to reclaim cold pages, and where we modify the kernel's reclamation path to batch TLB invalidations during

page-out. We then describe four representative tiering systems (§2.4) that serve as backends in the evaluation of OBASE.

Finally, we review modern memory allocators (§2.5), whose placement decisions determine the initial layout of objects in the address space. Understanding why allocators produce hotness fragmentation motivates the dynamic reorganization approach developed in Chapters 4 and 5.

2.1 The Page Abstraction

The *page* is the dominant unit of address translation, protection, and memory placement in modern operating systems. Paging has been enormously successful: it enables each process to run in the illusion of a large, private address space while the OS multiplexes limited physical memory across many processes. However, this success has also entrenched a fixed granularity as the unit at which the OS measures access, enforces isolation, and makes placement decisions. In modern heterogeneous hierarchies, where the performance gap between tiers is often dominated by bandwidth and movement costs, this fixed granularity becomes a liability: *applications access objects and fields, but the OS observes and migrates pages*.

This section reviews the historical motivations for paging, the hardware/software contract that makes it practical, and the consequences of managing memory at page granularity. We trace paging from its origins on the Atlas computer through the x86 translation architecture to explain why 4 KiB became the standard page size, and we identify the *granularity gap* that arises when hot and cold data are intermingled within a page. This gap directly motivates the layout-aware techniques developed in Chapters 3 and 5: WiscSort organizes data to avoid moving unnecessary bytes across the page boundary, while OBASE reorganizes the address space so that page boundaries align with access intensity.

2.1.1 Pages in Hardware and Software

2.1.1.1 From overlays to paging

Early time-sharing systems faced two challenges: (i) *relocation*, to let multiple programs safely coexist in memory, and (ii) *protection*, to prevent one program from corrupting another. A simple starting point is base-and-bounds: hardware adds a base register to each memory reference and checks that the reference stays within a legal range. While efficient, base-and-bounds requires each address space to be physically contiguous, complicating allocation and forcing costly compaction as processes enter and leave the system.

Segmentation generalized this idea by giving programs multiple variable-sized regions (code, heap, stack). Segmentation improves flexibility but introduces *external fragmentation*: over time, free memory splits into unusable holes [40].

Before paging, programmers faced an additional burden: when programs exceeded physical memory, they had to manually partition code and data into *overlays* and explicitly orchestrate transfers between main memory and secondary storage. On the Manchester Mercury computer in the late 1950s, developers spent as much effort managing overlays as implementing algorithms [94].

In 1962, the Atlas computer introduced what its designers called a "one-level storage system" [94]. The key insight was to separate the *address* a program uses from the *physical location* where data resides. Atlas divided memory into fixed-size units of 512 words (approximately 3 KB), automatically moving these units (*pages*) between fast core memory and slower drum storage as needed. Programs could address up to one million words without knowing which pages resided where.

Atlas introduced three innovations that remain foundational: (1) hardware that automatically translates virtual addresses to physical locations, (2) an interrupt mechanism (the *page fault*) triggered when a required page

is absent, and (3) a *replacement algorithm* to select which pages to evict when memory is full [66]. It was estimated that eliminating manual overlays would improve programmer productivity by a factor of three, a prediction borne out as virtual memory spread through the industry. IBM incorporated these ideas into the System/360 series, and by the 1970s paging had become standard.

2.1.1.2 The translation contract

Paging works because hardware and OS agree on a narrow contract:

- **Per-process page tables.** The OS maintains translation metadata that defines which virtual pages are valid and where they reside in physical memory (or whether they are absent).
- **Fast-path translation.** Hardware accelerates translation with a Translation Lookaside Buffer (TLB). On a TLB hit, translation adds negligible latency; on a miss, hardware walks page tables in memory, a costly operation requiring 150–600 cycles compared to roughly 4 cycles for a hit [165].
- **Faults on absence.** If a referenced virtual page is not resident, hardware raises a page fault. The OS handles the fault by bringing the page into memory (possibly evicting another), then restarts the faulting instruction.

Demand paging extends this contract to secondary storage: the OS keeps only a working subset of pages in DRAM and fetches missing pages from disk on demand. This mechanism underlies not only traditional swapping but also modern memory tiering, where the "backing store" is another memory tier (e.g., CXL-attached memory) rather than disk.

2.1.2 Page-Based Memory Management

At the policy level, the OS must decide which pages should remain resident and which should be evicted or migrated. The classic goal is to approximate the process *working set*: retain pages likely to be reused soon, and reclaim those unlikely to be touched.

A core challenge is that the OS cannot directly observe program semantics (objects, fields, or logical records). Instead, it tracks page-level activity through hardware-supported metadata. Most architectures expose an *accessed bit* (set on any read or write) and a *dirty bit* (set on writes) per page table entry, allowing the OS to approximate recency and write intensity without instrumenting every load/store. Operating systems build replacement policies (typically LRU approximations, as described in §2.3) over these coarse signals.

Page-based management implicitly assumes that moving a page is the natural unit of data movement. This assumption aligned well with traditional DRAM+disk hierarchies: disks favor transferring contiguous blocks, and paging amortizes seek and rotation costs. But in heterogeneous hierarchies, especially when lower tiers are byte-addressable and movement is bandwidth-bound, the fixed page size can dominate cost. As latency gaps narrow and bandwidth costs dominate, unnecessary bytes moved with each placement decision become increasingly visible.

2.1.3 The Granularity Gap

Paging introduces a fundamental mismatch between *what applications access* and *what the OS can manage*. Applications allocate and access objects, records, and fields with rich semantics: keys may be probed frequently while payloads are streamed rarely; metadata may be touched on every operation while values remain cold except during compaction. The OS, however, observes only page-level access and can place or migrate data

only at page granularity.

When hot and cold objects share pages, page-based tiering becomes imprecise:

- **Bandwidth waste.** Accessing a small hot subset drags along adjacent cold data when the page is fetched or migrated between tiers.
- **Trapped cold data.** If a page contains even a small amount of hot state, the entire page may be retained in the fastest tier, pinning cold bytes in expensive memory.
- **Noisy signals.** Page-level access bits conflate distinct objects. A frequently touched field makes the entire page appear hot, obscuring which bytes actually matter.

2.1.4 Why 4 KiB?

A recurring question is why 4 KiB became the dominant page size. The answer reflects a balance of engineering tradeoffs that, once standardized, became deeply embedded in hardware and software.

The tradeoff space. Choosing a page size trades off competing concerns:

- **TLB reach and fault overhead** (favor larger pages): With a fixed number of TLB entries, larger pages increase the memory footprint covered by the TLB; each fault also transfers more potentially useful nearby bytes when access exhibits spatial locality.
- **Internal fragmentation and precision** (favor smaller pages): Smaller pages reduce wasted space within the last page of allocations and enable finer-grained placement decisions.

- **Page-table structure** (favors powers of two): Page-table layouts are simplest when the number of entries fits neatly in a page-sized chunk of memory.

The 80386 and architectural self-similarity. On 32-bit x86, the Intel 80386 (1985) established 4 KiB pages with a particularly clean decomposition: a 32-bit linear address splits into a 12-bit offset plus two 10-bit indices [1]. With 4-byte entries, a single 4 KiB page holds exactly 1024 entries, so *page tables and directories are themselves page-sized*. This self-similarity simplifies allocation, protection, and even paging out the page tables themselves.

Early empirical studies, on the IBM System/360 Model 67 [79], evaluated 2048- versus 4096-byte pages and characterized when larger pages win (spatial locality amortizes fault cost) versus when smaller pages win (sparser access patterns, reduced fragmentation). These studies informed the design space, though the final choice also reflected the 32-bit address structure.

4 KiB was not universally optimal, but once a page size is standardized and deeply integrated into hardware translation and OS policy, it becomes the unit of observability and control. Modern tiered-memory systems inherit the page as their decision granularity, even when applications and devices would benefit from finer- or coarser-grained placement. The techniques in this thesis work *within* this constraint: rather than changing the page abstraction, we organize data so that page boundaries align with access intensity, making page-based decisions effective.

2.2 The Evolving Memory Hierarchy

For decades, system designers optimized for a two-level hierarchy: fast but limited DRAM and slow but large disk. Applications either fit in memory or paid steep penalties for disk access. This clean separation is dissolving. New device technologies occupy the latency and capacity space between

DRAM and storage, while interconnects like CXL allow memory to be attached, pooled, and tiered across sockets and enclosures. These changes create opportunities for finer-grained data placement but also expose the limitations of page-based management.

This section covers two developments that are central to the dissertation. First, we describe byte-addressable storage devices (§2.2.2), whose properties differ fundamentally from both DRAM and block storage. These properties motivate the BRAID device model and the design of WiscSort in Chapter 3. Second, we introduce memory tiering (§2.2.3), the practice of treating slower memory as a capacity tier beneath DRAM. The limitations of page-granularity tiering motivate the analysis in Chapter 4 and the design of OBASE in Chapter 5.

2.2.1 From DRAM-Centric to Heterogeneous Hierarchies

Traditional system design treated DRAM as the sole working memory. Data either resided in DRAM (fast, byte-addressable, volatile) or on disk (slow, block-addressed, persistent). The performance gap between these tiers was enormous: a disk seek costs millions of CPU cycles, so systems batch I/O into large sequential transfers and cache aggressively in DRAM.

This two-tier model is giving way to richer hierarchies. Several trends drive the shift:

- **Memory capacity pressure.** Dataset sizes grow faster than DRAM density improvements. In-memory databases, machine learning models, and key-value stores routinely exceed single-server DRAM capacity, forcing either distributed architectures or tiered memory.
- **Cost and power constraints.** DRAM accounts for a substantial fraction of server cost and power consumption [106]. Operators seek cheaper capacity tiers that can absorb cold data without sacrificing the performance of hot data.

- **New device technologies.** Storage-class memory (e.g., Intel Optane), Ultra-Low Latency SSDs, and CXL-attached memory provide intermediate price, capacity, and performance points. These devices blur the traditional DRAM/storage boundary.

The result is a hierarchy with multiple tiers, each offering different trade-offs. Efficient use of such hierarchies requires placing data on the tier that matches its access intensity. As Section 2.1.3 argued, when placement decisions occur at page granularity but access patterns vary at finer granularity, mismatches are inevitable.

2.2.2 Byte-Addressable Storage

Byte-addressable storage (BAS) refers to devices that support load/store access at fine granularity, unlike traditional block storage that requires transfers in fixed-size units (typically 512 bytes or 4 KiB). Intel Optane DC Persistent Memory and CXL-attached memory devices are prominent examples. These devices sit between DRAM and SSDs in both latency and capacity. Understanding these device characteristics is essential for Chapter 3, where we formalize them into the BRAID model and design WiscSort to exploit each property.

Device characteristics. BAS devices differ from both DRAM and block storage in several ways:

- **Access granularity.** BAS supports byte-level addressing. On Optane PMEM, the internal access unit is 256 bytes, but applications can issue loads and stores at arbitrary granularity without the amplification penalties of block devices. Sub-256-byte writes still incur an internal read-modify-write at XPLine granularity, but the penalty is far less severe than the sector-level amplification of block I/O.

- **Latency.** Read latencies are roughly 3× higher than DRAM (approximately 300 ns versus 100 ns for local DRAM), write latencies are higher still, primarily due to the slower write physics of the underlying media, with additional overhead from internal wear-leveling and the read-modify-write cycle for sub-XPLine stores [159].

- **Bandwidth.** A single Optane DIMM provides approximately 6.6 GB/s read and 2.3 GB/s write bandwidth. Bandwidth scales with the number of DIMMs but saturates at fewer concurrent threads than DRAM [159].

- **Random versus sequential access.** Unlike HDDs and SSDs, random read bandwidth on BAS approaches sequential read bandwidth. This property fundamentally changes the calculus for data layout: algorithms that avoided random access to amortize seek costs can now reconsider.

- **Read-write asymmetry.** Writes are slower than reads and do not scale as well with concurrency. Concurrent writes can also interfere with read performance, reducing read bandwidth by up to 2× when reads and writes overlap [159].

These characteristics invert several traditional assumptions. On HDDs, sequential access was paramount because seeks dominated; systems bundled data together even if only a subset would be used. On BAS, the cost of reading unnecessary bytes is no longer amortized by avoiding seeks. Similarly, the asymmetry between reads and writes favors algorithms that trade additional reads for fewer writes. Chapter 3 develops a sorting algorithm that exploits these properties by separating keys from values and issuing targeted random reads rather than sequential scans of bundled records.

2.2.3 Memory Tiering

Memory tiering extends the memory hierarchy by treating slower memory (CXL-attached DRAM, Optane, or remote memory) as a capacity tier beneath local DRAM. Unlike traditional swapping to disk, tiered memory remains byte-addressable and is accessed via load/store instructions, albeit with higher latency. The operating system or a user-space runtime migrates pages between tiers based on observed access patterns.

The central question in memory tiering is placement: which pages should reside in fast memory, and which can tolerate the slower tier? An ideal policy would keep the working set in fast memory while demoting cold pages to the capacity tier. In practice, policies must address three interrelated challenges: (1) detecting page hotness with low overhead, (2) setting thresholds that adapt to workload characteristics and tier capacities, and (3) migrating pages without disrupting application performance. Because direct instrumentation of every memory access is prohibitively expensive, tiering systems rely on approximate signals such as page table access bits, hardware performance counters, or induced page faults.

Tiering systems build upon the Linux kernel's memory reclamation infrastructure, which we describe next in §2.3. Several representative tiering systems that serve as backends in the evaluation of OBASE are then surveyed in §2.4.

2.3 Linux Memory Reclamation

Reclamation is a core component of both caching and tiering systems: it moves cold pages from DRAM to a backing store (swap, compressed memory, or disk) to free capacity for new allocations. In a caching model (traditional swap), reclaimed pages must be faulted back into DRAM before they can be accessed; in a tiering model (CXL-attached memory), the CPU can access demoted pages directly, albeit at higher latency. Modern tiering systems

build upon, modify, or work alongside this infrastructure, so understanding its mechanisms is essential context for the tiering systems described in §2.4. More directly, this material is necessary for understanding OBASE (Chapter 5): OBASE interacts with kswapd, cgroup memory limits, and `madvise` hints to reclaim cold pages after reorganizing them, and we modify the kernel's reclamation path to batch TLB invalidations during page-out (§5.2.6).

The description below corresponds to the Linux kernel as of version 6.11, which is the baseline for the kernel modifications presented in Chapter 5. Where specific interfaces differ between cgroup v1 and v2, we note both. The core reclamation architecture has been stable since approximately kernel 4.0, though details such as the multi-generational LRU (MGLRU, merged in 6.1) continue to evolve.

This section covers three aspects of reclamation: the data structures and access-tracking mechanisms the kernel uses to approximate page hotness (§2.3.1), the process by which pages are actually evicted (§2.3.2), and the conditions that trigger reclamation (§2.3.3).

2.3.1 LRU Lists and Access Tracking

To manage which pages to reclaim, Linux uses Least Recently Used (LRU) lists. Pages are ordered by their last access time, from most recent at the head to least recent at the tail. Pages are categorized into distinct LRU lists: one for anonymous pages and one for file-backed pages, as their reclamation properties differ significantly. Clean file pages can be discarded immediately, while anonymous pages must be swapped out. Each LRU list is further divided into active and inactive lists. Active lists contain pages that have been accessed recently, while inactive lists hold pages that are candidates for eviction. An `lruvec` for each NUMA node and cgroup maintains five LRU lists: `INACTIVE_ANON`, `ACTIVE_ANON`, `INACTIVE_FILE`, `ACTIVE_FILE`, and `UNEVICTABLE`.

There are two main types of userspace pages. Anonymous pages, al-

located by `mmap(MAP_ANONYMOUS)` and populated by page faults, must be swapped out if reclaimed. File pages, also known as page cache, are created by file operations or `mmap(..., fd)` and can be immediately discarded when clean or written back to a file when dirty.

Maintaining an exact LRU ordering is prohibitively expensive, so Linux approximates by checking whether pages have been accessed since the last check. This is primarily done using the Access (A) bit in the Page Table Entries (PTEs). When a page is accessed, the hardware sets the A bit. The kernel periodically scans these bits, clearing them after checking. If the A bit is set when checked, it indicates the page was recently accessed. For file-backed pages, the kernel also uses a Referenced (R) bit, which is set when the page is accessed through file operations. Pages are moved between the active and inactive lists based on this access information. The system aims to balance the sizes of the active and inactive lists.

To prevent thrashing and identify working sets, Linux implements a workingset detection mechanism. This uses shadow entries in a radix tree (XArray) to track evicted pages and measure their *refault distance*: the number of other pages accessed between an eviction and a subsequent access to the same page. If this refault distance is less than the size of the active list, the page is considered part of the working set and is immediately activated upon refault. This approach allows the system to distinguish between truly cold pages and those that are part of the active working set but were evicted due to temporary inactivity. Pages are typically reclaimed from the tail of the inactive LRU list, ensuring that the coldest candidates are chosen first.

The PTE access bit, the LRU lists, and the refault mechanism together constitute the signal that page-level tiering systems rely on to classify pages as hot or cold. As Chapter 4 will show, these signals are fundamentally limited by the granularity gap: a single hot cache line on a page makes the entire page appear active, regardless of how much of the page is actually in use.

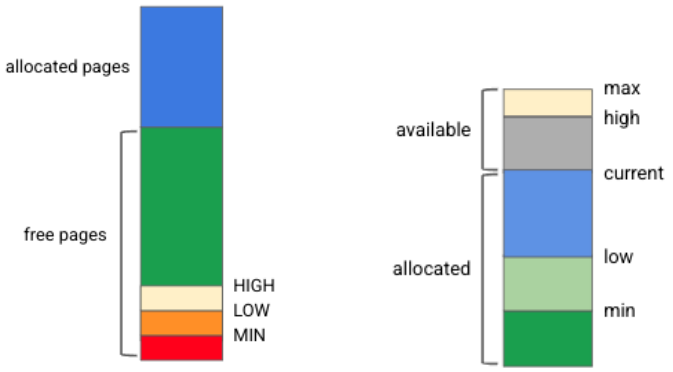
2.3.2 Reclaim Process

The reclaim process involves moving pages between different LRU lists and ultimately evicting them. The main operations include aging (checking if a page has been accessed and moving it accordingly), eviction (removing a page from the inactive LRU list for reclaim), activation (moving a page from the inactive LRU list to the active LRU list), and deactivation (moving a page from the active LRU list to the inactive LRU list).

Balancing the sizes of the active and inactive lists is important. Ideally, inactive LRU lists should be small to minimize reclaim work but large enough to observe page access patterns. The system calculates an `inactive_ratio` based on the square root of the sizes of inactive and active LRU lists. For instance, with 100 MB of memory, the expected size of the inactive list is 50 MB (a 1:1 ratio), but with 100 GB of memory, it is 3 GB (a 31:1 ratio).

The reclamation process uses a scan priority to determine how much of the `lruvec` to scan in one iteration. The default priority is 12, meaning it will scan 1/4096th of the `lruvec` during an aging round. This priority is decremented for each round that does not free enough pages, with a highest priority of 1 resulting in scanning 100% of the `lruvec`. The scan count, which determines how many pages to scan in one iteration, is calculated based on the `lruvec` size weighted by the scan priority. It is set to at least 32 pages and is distinct from the reclaim count. This approach allows the system to adjust its reclamation efforts based on the current memory pressure.

In Chapter 5, we modify the reclamation path invoked by `MADV_PAGEOUT` to batch TLB invalidations across hundreds of pages before issuing a single flush. This optimization reduces inter-processor interrupts by more than 99% when demoting large cold regions, and its design is informed by the scan-and-evict structure described here.



(a) Zone watermarks (b) Memory thresholds

Figure 2.1: **Memory pressure thresholds:** (a) Zone watermarks for system-wide memory pressure. MIN, LOW, and HIGH thresholds determine reclamation triggers. (b) Cgroup memory thresholds, depicting current usage and various limits.

2.3.3 Triggers for Reclamation

Memory reclamation can be triggered in two ways: based on memory pressure or through application guidance.

Memory pressure (reactive). Each memory node consists of zones (e.g., `ZONE_NORMAL`, `ZONE_MOVABLE`), and each zone has three watermarks: `MIN`, `LOW`, and `HIGH`. These watermarks are configured via sysctls, with `MIN` set to a small percentage (typically 0.1%) of total managed pages, and `LOW` and `HIGH` each adding an additional 0.1%. Figure 2.1(a) illustrates these watermarks.

When free memory falls below the `LOW` watermark, it triggers memory pressure. This can lead to two types of pressure-based reclaim: direct reclaim, invoked on the allocation path when free memory is below `LOW` on all eligible zones from all eligible nodes, and kswapd reclaim, where a per-node kswapd thread performs background reclaim when free memory falls below `LOW`, continuing until free memory reaches the `HIGH` watermark.

Additionally, cgroup memory pressure can trigger reclaim within a specific cgroup when its memory usage exceeds its limit. In cgroup v2, this occurs when usage after a charge exceeds `memory.high`, while in v1, it occurs when usage exceeds `memory.soft_limit_in_bytes`. Figure 2.1(b) illustrates these thresholds for cgroup memory management. Cgroup reclamation is initiated when memory usage reaches these thresholds and continues until usage falls below the specified limit.

Application-guided reclaim. Linux also supports application-guided reclaim through various `madvise()` options. These include `MADV_DONTNEED` (unmaps and discards pages), `MADV_FREE` (lazily frees anonymous pages), `MADV_COLD` (deactivates pages without immediate reclaim), and `MADV_PAGEOUT` (immediately reclaims pages). These mechanisms allow applications to provide hints to the kernel about their memory usage patterns.

The Linux page reclamation system employs a complex set of mechanisms that can be triggered through a limited set of controls. The inflexibility to specify exact allocations or data structures to swap per process can result in suboptimal data reclamation. Moreover, application-guided reclamation requires data to be contiguous, which allocations often are not, since allocations are placed next to each other based on their arrival time rather than their access patterns. This limitation is precisely what OBASE addresses: by reorganizing objects so that cold data occupies contiguous regions, OBASE makes application-guided reclamation via `MADV_PAGEOUT` effective where it would otherwise reclaim a mixture of hot and cold data.

2.4 Memory Tiering Systems

Building on the reclamation infrastructure described above, several tiering systems have been deployed or proposed. Each makes different tradeoffs among detection accuracy, overhead, and adaptability. This section surveys

four representative systems. All four serve as backends in the evaluation of OBASE in Chapter 5, where we demonstrate that OBASE improves the effectiveness of each by providing a higher-quality address-space layout. Understanding these systems' detection mechanisms and migration policies is therefore necessary context for interpreting those experiments.

AutoNUMA. AutoNUMA [146] is the Linux kernel's default mechanism for optimizing memory locality on NUMA systems. Although originally designed to migrate pages closer to the CPUs accessing them, AutoNUMA's approach extends naturally to CXL-attached memory tiers that appear as CPU-less NUMA nodes. AutoNUMA periodically unmaps a sample of pages from each process's address space. When the application subsequently accesses an unmapped page, the resulting minor page fault (called a NUMA hint fault) reveals both that the page is active and which CPU accessed it. If the page resides on a remote node, AutoNUMA migrates it to the accessing CPU's local memory. The mechanism relies on recency as its primary hotness signal: a single access is sufficient to trigger promotion. This simple policy works well when most pages are either consistently hot or consistently cold, but it can cause excessive migration when access patterns are transient or when the fast tier cannot hold the entire working set. AutoNUMA also lacks a proactive demotion mechanism; pages migrate to the fast tier on access but are not demoted until memory pressure forces the standard reclamation path.

TPP: Transparent Page Placement. TPP [116] targets CXL-enabled tiered memory systems and was developed at Meta based on production workload characterization. The system introduces three coordinated mechanisms: lightweight demotion, decoupled allocation and reclamation, and hysteresis-based promotion. For demotion, TPP uses migration rather than swapping. When the fast tier is under memory pressure, cold pages identified via the kernel's existing LRU lists are migrated to the CXL-attached

tier rather than paged out to disk. Migration is substantially faster than swapping and preserves byte-addressable access to demoted pages.

A key insight in TPP is that tightly coupling allocation with reclamation causes problems in tiered systems. As described in §2.3.3, standard Linux reclamation halts once free pages reach the `HIGH` watermark, leaving little headroom for new allocations or promotions. TPP decouples these paths by introducing separate watermarks: a higher demotion watermark that triggers background reclamation, and a lower allocation watermark that gates new allocations. This separation maintains free space in the fast tier for incoming hot pages without stalling allocation requests.

For promotion, TPP augments NUMA balancing with an active-LRU check. Rather than promoting any page that triggers a NUMA hint fault, TPP first verifies that the page resides in the active LRU list. Pages found in the inactive list are marked as accessed and moved to the active list but not immediately promoted. Only if the page triggers another fault while still in the active list does TPP promote it. This hysteresis reduces unnecessary migration traffic by filtering out pages with sporadic or one-time accesses.

Memtis. Memtis [101] observes that prior tiering systems rely on static thresholds (such as "promote after two accesses") that do not adapt to workload characteristics or tier capacities. A static threshold may identify a hot set larger than the fast tier can hold, or smaller than the fast tier's capacity, leaving expensive memory underutilized. To address this, Memtis maintains a histogram of page access counts, organized into exponentially scaled bins. Rather than using a fixed threshold, Memtis dynamically computes the hot threshold by finding the highest bin index such that the cumulative size of hotter pages fits within the fast tier. This approach ensures that the fast tier is filled with the hottest pages regardless of the absolute access frequency distribution.

Memtis tracks access counts using Intel's Processor Event-Based Sam-

pling (PEBS), which records the virtual addresses of sampled memory events (such as last-level cache misses) with low overhead. Unlike the PTE access-bit scanning used by the standard LRU mechanism, PEBS can track accesses at sub-page granularity, enabling Memtis to identify skewed access patterns within huge pages. When a huge page contains a mix of hot and cold subpages, Memtis can split the huge page and promote only the hot 4 KiB regions, reducing fast-tier waste. To maintain freshness, Memtis periodically halves all access counts (a process called cooling), which computes an exponential moving average of access intensity. Pages are classified as hot, warm, or cold based on their bin index relative to the dynamically computed thresholds. Warm pages are not migrated, providing hysteresis against short-term fluctuations. All migration occurs in background kernel threads rather than on the allocation or fault paths.

TMO: Transparent Memory Offloading. TMO [152] approaches tiering from a different angle: rather than optimizing placement for performance, it maximizes memory savings subject to a performance constraint. Deployed at Meta across millions of servers, TMO offloads cold memory to compressed memory pools (via zswap) or SSDs, freeing DRAM capacity for other uses. The central innovation in TMO is Pressure Stall Information (PSI), a kernel mechanism that directly measures the fraction of time processes spend stalled due to resource shortage. Unlike indirect metrics such as page fault rates or promotion counts, PSI captures the actual productivity loss caused by memory pressure. This distinction matters because the same fault rate can have different performance implications depending on the speed of the backing device: a high swap-in rate from a fast SSD may cause negligible stalls, while a lower rate from a slow SSD may severely degrade throughput.

A user-space agent called Senpai uses PSI as a feedback signal to control memory reclamation. Senpai periodically writes to the `memory.reclaim`

cgroup interface, requesting that the kernel reclaim a small fraction of each container's memory. The reclaim rate scales inversely with observed pressure: when pressure approaches a configured threshold, Senpai reduces or halts reclamation; when pressure is low, it reclaims more aggressively. This feedback loop automatically adapts to workload sensitivity and backend performance without requiring offline profiling or per-application tuning.

TMO also addresses a historical bias in Linux reclamation: as noted in §2.3, the kernel strongly preferred evicting file cache over swapping anonymous memory, relegating swap to emergency use. TMO rebalances reclamation by tracking file cache refaults (using the workingset detection mechanism described in §2.3.1) and balancing the refault rate against the swap-in rate. This change enables TMO to offload both anonymous and file-backed memory proportionally to their coldness.

Common limitations. Despite their differences in detection mechanisms and policies, all four systems share a fundamental constraint: they observe and migrate memory at page granularity. AutoNUMA and TPP rely on page faults and LRU lists; Memtis samples at cache-line granularity but still migrates entire pages; TMO reclaims pages through the kernel's standard mechanisms. When applications interleave hot and cold data within pages, these systems face the dilemma described in Section 2.1.3: migrating a partially hot page either wastes bandwidth moving cold bytes or leaves hot bytes stranded on the slow tier.

Memory tiering creates both the need and the opportunity for layout-aware organization. The need arises because tiering policies inherit the granularity gap: page-level decisions on object-level access patterns produce suboptimal placement. The opportunity arises because, unlike swapping to disk, tiered memory allows fine-grained access even to demoted data. If hot objects are clustered onto a small number of pages, those pages can remain in fast memory while the remaining pages (now uniformly

cold) migrate to the capacity tier without performance penalty. Chapter 4 quantifies how severe this limitation is in practice, and Chapter 5 presents OBASE, which addresses it by reorganizing the address space so that these backends receive uniformly hot or cold pages.

2.5 Modern Memory Allocators

Applications request memory through allocators such as TCMalloc [165] (used at Google) and Jemalloc [69] (used at Meta). These allocators mediate between fine-grained application requests and coarse-grained OS memory management, optimizing for allocation speed, low fragmentation, and scalability under concurrent access.

Understanding allocator design is necessary for two reasons. First, allocators determine the initial layout of objects in the address space, and this layout is what produces the hotness fragmentation analyzed in Chapter 4. Because allocators place objects based on size class and allocation order rather than access patterns, hot and cold objects end up interleaved on the same pages. Second, OBASE (Chapter 5) builds directly atop allocator infrastructure: its spatially-aware memory allocator (SAMA) extends jemalloc's extent management to maintain separate heaps for hot and cold objects.

This section describes the hierarchical architecture common to modern allocators (§2.5.1), their interaction with the operating system (§2.5.2), and the fundamental placement limitation that motivates dynamic reorganization (§2.5.3).

2.5.1 Allocation Architecture

Modern allocators employ hierarchical structures to minimize contention and improve cache locality. Figure 2.2 illustrates this layered design using jemalloc as an example. At the top, per-thread caches (`tcache`) satisfy most allocations without synchronization. On a cache miss, the allocator con-

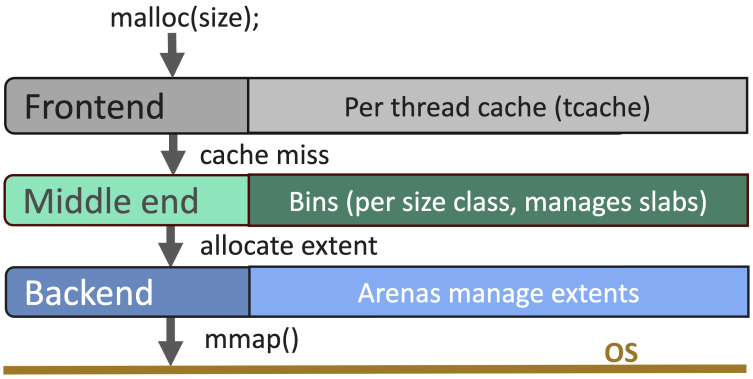


Figure 2.2: **Hierarchical structure of a modern memory allocator (jemalloc).** Allocation requests flow through caching layers. The frontend satisfies most requests from thread-local caches; misses propagate to bins that manage slabs carved from extents. Arenas allocate extents from the OS via `mmap`. Placement decisions occur when extents are created and slabs are carved, before the application reveals which objects will be frequently accessed.

sults the middle layer, where bins manage slabs for each size class. When a bin exhausts its slabs, it requests a new extent from the backend, which manages large memory regions (extents) within arenas. Only when arenas require additional virtual address space does the allocator invoke the OS via `mmap`. TCMalloc follows a similar pattern with per-CPU caches, a transfer cache, and a central page heap. In both systems, the common case (a small allocation that hits the local cache) completes in tens of nanoseconds without acquiring locks.

To reduce internal fragmentation, both allocators round allocation requests up to predefined *size classes*. TCMalloc defines 80–90 size classes for small objects; jemalloc organizes allocations into bins corresponding to size classes. Objects within a size class are packed into contiguous regions (slabs or spans), simplifying bookkeeping and reducing per-object metadata overhead. Large allocations bypass size classes and are satisfied directly from page-aligned regions.

2.5.2 Interaction with the Operating System

Allocators obtain memory from the OS in large chunks, typically via `mmap`, and suballocate to applications. This batching amortizes the cost of system calls and allows allocators to manage memory layout within their reserved regions. When memory is freed, allocators do not immediately return it to the OS; instead, they cache free regions for future allocations, reducing both system call overhead and page fault costs.

Both allocators include mechanisms to return unused memory to the OS over time. TCMalloc periodically releases free spans, potentially breaking up huge pages to return smaller regions. Jemalloc uses a decay mechanism: free extents transition through dirty, muzzy, and retained states, with configurable timers controlling how quickly memory is released. These mechanisms balance memory footprint against allocation performance, but they operate on allocator-internal regions rather than on individual objects.

Huge page management introduces additional complexity. Allocators may request 2 MiB regions from the OS to reduce TLB pressure, but returning portions of a huge page requires either fragmenting it into base pages or tolerating internal waste. TCMalloc's Temeraire subsystem [85] and jemalloc's extent management both attempt to pack allocations densely within huge pages, but these decisions are based on allocation patterns, not access patterns.

2.5.3 The Placement Limitation

Allocators optimize for three goals: minimizing allocation latency, minimizing deallocation latency, and reducing fragmentation (both internal and external). They achieve these goals through careful data structure design, size class selection, and caching policies. What allocators do *not* optimize for is future access patterns. When an application calls `malloc`, the allocator places the object in the next available slot of the appropriate size class. This

placement reflects the temporal order of allocations, not any property of how objects will be used.

Consider a hash table that stores keys and values as separate allocations. Keys may be accessed on every lookup, while values are touched only when a lookup succeeds. If keys and values are allocated in interleaved order (as naturally happens during insertion), they will be placed in interleaved positions within the allocator's slabs. From the OS perspective, pages containing keys also contain values, and vice versa. The page abstraction cannot distinguish the hot keys from the cold values because both share the same pages.

This mismatch between allocation order and access intensity is fundamental. Allocators cannot predict which objects will be hot because hotness depends on application logic and workload characteristics that are unknown at allocation time. Even if an allocator could predict initial hotness, access patterns change over time (as Chapter 4 demonstrates), invalidating any static placement.

Moving garbage collectors can compact heaps and colocate objects because they can rewrite pointers, but this is not available in C/C++ with raw pointers. Even in managed runtimes, compaction is driven by reachability and allocation patterns, not observed access.

Allocators are not broken; they solve the problems they are designed to solve. But their placement decisions are final: once an object is allocated, it remains at that virtual address for its lifetime. Improving page utilization requires either (1) changing allocation to consider predicted access patterns, which is impractical given the unpredictability of hotness, or (2) reorganizing objects after allocation based on observed access. This thesis pursues the second approach. Chapter 5 introduces OBASE, which builds atop existing allocators (specifically, jemalloc's extent management) to migrate objects between hot and cold regions as their access patterns become apparent.

Chapter 3

WiscSort: Static Layout Optimization

Modern memory hierarchies requires organizing data at a finer granularity than the page. The previous chapters established the context: page-based memory management cannot distinguish hot from cold data within a page (§2.1), and the emerging hierarchy of DRAM, CXL-attached memory, and byte-addressable storage creates new placement opportunities that page-granularity decisions cannot fully exploit (§2.2).

This chapter presents a concrete instance of that argument. External sorting is a data-intensive workload whose access structure is well understood: sorting compares keys, not values, yet traditional algorithms move both because sequential access was historically cheap and random access was expensive. Byte-addressable storage inverts this tradeoff. We show that by reorganizing how data moves between memory and storage, matching the layout to the device's strengths and avoiding its weaknesses, a sorting system can achieve 2–7$\times$ higher throughput than conventional approaches. The insight is specific to sorting, but the principle is general: *data layout should reflect the properties of the device that stores it*. Chapters 4 and 5 extend this principle to workloads where the access structure is not known in advance.

External sorting is a critical component of many modern data-intensive applications (web indexing [49], key-value stores [2, 4], data analytics [49],

and relational databases [18, 130, 3]), making use of external storage to sort data that does not fit in DRAM. For example, relational databases (such as MySQL and PostgreSQL) use external sorting for `ORDER BY` queries on non-indexed keys or to handle `TOP-K` queries whose input exceeds the available memory.

Traditionally HDDs or SSDs have been used for external storage; however, Byte-Addressable Storage (BAS) is emerging as an appealing expansion memory or fast storage layer for data-intensive applications [19, 15, 116, 105]. This broad class of storage devices includes the Samsung CXL Memory-Semantic SSD [20] and Intel's Optane DC Persistent Memory (PMEM) [13]. BAS devices provide a larger capacity than DRAM and are much faster than traditional SSDs and HDDs, making them suitable for performance-critical services.

For applications to maximize performance, the unconventional characteristics of BAS devices must be carefully considered. We introduce a generic device model called BRAID that depicts the typical performance characteristics of byte-addressable storage devices. The BRAID model has five crucial properties: **B**yte addressability, **R**andom read performance, **A**symmetric read-write cost, read-write **I**nterference, and **D**evice constrained concurrency. Combinations of two or more of these characteristics can be seen in a BRAID device; for example, PMEM exhibits all five BRAID properties.

This chapter presents *WiscSort*, a BRAID-conscious high-performance external sorting algorithm derived from the popular external merge sort. WiscSort [45] postpones the movement of a (key, value) pair's value until the pair's final sorted position is known (typically, the final merge phase of external merge sort); only the keys are moved between DRAM and the storage device during earlier phases. Except for the final phase, WiscSort maintains keys and pointers in DRAM, whereas previous approaches bundled keys and values. Pointers point to the respective values in the original file on the device. This simple late materialization avoids writes of values

during early sorting phases; for popular workloads with keys smaller than values [55], the savings are significant. Furthermore, because WiscSort only keeps pointers in DRAM, it can generate larger sorted runs during the run-generation phase, reducing the number of merge phases or avoiding the merge phase completely.

BRAID devices can also have peculiar concurrency characteristics. For example, writes do not scale well, and the read bandwidth degrades (up to 2x [159]) when there are overlapping write requests. To alleviate these read-write interference degradation and concurrency constraints, we introduce a *Thread-Pool Controller* and an *Interference-Aware Scheduler*. WiscSort utilizes the thread-pool controller to determine the appropriate concurrency pool size (for read/write) based on the access type, and the interference-aware scheduler schedules reads and writes to the device in a non-overlapping fashion to maximize performance.

We compare the performance of WiscSort with classic external merge sort implementations on both microbenchmarks and standard application-level benchmarks. We show that WiscSort performs 2–3$\times$ better than concurrent external merge sort, 5$\times$ better than state-of-the-art in-place sample sort, and 7$\times$ better than recent PM-based sorting system (PMSort) on sortbenchmark [21] workloads. The interference-aware scheduler and the thread-pool controller reduce total time by at least 50% compared to approaches that overlap reads and writes. Moreover, a system that just separates keys and values but is not aware of these concurrency properties is ~15% slower than one that is interference and concurrency aware. We show that special cases of WiscSort can do better than external merge sort even when the value size is smaller than the key size. As the value size increases, the performance gap between merge sort and WiscSort grows. We also demonstrate that using random reads to reduce unnecessary data loading is better than sequentially reading all data. Finally, we project the performance of WiscSort on emulated BRAID devices and discuss the benefits of our techniques.

3.1 External Sorting Fundamentals

External sorting refers to algorithms that sort datasets too large to fit in main memory. Unlike in-memory sorting, where all data is directly accessible, external sorting must orchestrate data movement between fast but limited memory and slow but capacious storage. This section reviews the classic external merge sort algorithm and examines how its design reflects assumptions about storage device characteristics. These assumptions, while valid for disks and SSDs, do not hold for byte-addressable storage, motivating the techniques developed later in this chapter.

3.1.1 The External Sorting Problem

Many data-intensive applications require sorting datasets that exceed available DRAM. Relational databases use external sorting for `ORDER BY` queries on non-indexed columns and for building indexes over large tables [18, 3]. Key-value stores such as LevelDB and RocksDB sort data during compaction to maintain sorted structure on disk [2, 4]. Analytics pipelines sort intermediate results when shuffle data exceeds memory budgets. In each case, the sorting algorithm must minimize I/O cost while respecting memory constraints.

External merge sort. The dominant approach to external sorting is external merge sort, a two-phase algorithm that extends the classic merge sort to handle out-of-core data [96, 34]. The algorithm proceeds as follows:

1. **Run generation.** Read a memory-sized chunk of input records into DRAM, sort them using any efficient in-memory algorithm (such as quicksort or radix sort), and write the sorted chunk to a temporary file called a *run*. Repeat until all input data has been processed, producing R sorted runs.

2. **Merge.** Open all R runs simultaneously, maintaining a small buffer for each. Repeatedly select the smallest record across all run buffers (using a heap or tournament tree), write it to the output, and refill buffers as they empty. If R exceeds the number of buffers that fit in memory, perform multiple merge passes, each reducing the number of runs by a factor equal to the merge fan-in.

The number of runs R depends on the ratio of input size to available memory. With N bytes of input and M bytes of memory, the run generation phase produces $R = \lceil N/M \rceil$ runs. If R exceeds the maximum fan-in k (determined by buffer sizes), multiple merge passes are required, with the total number of passes being $\lceil \log_k R \rceil$.

I/O complexity. External merge sort is designed to minimize I/O operations. Each record is read once during run generation and written once to a run file; during merge, each record is read from its run and written to the output (or to an intermediate file if multiple passes are needed). For a single merge pass, the total I/O is $4N$ bytes: two reads and two writes per byte of input. This linear I/O complexity, combined with the sequential access patterns described below, makes external merge sort efficient on traditional storage.

3.1.2 Run Generation and Merge Phases

Understanding why traditional external sorting bundles keys with values requires examining the I/O patterns of each phase.

Run generation. During run generation, the algorithm reads input records sequentially, accumulating them in memory until the buffer is full. It then sorts the buffer contents and writes the sorted run sequentially to storage. Both the read and write operations are strictly sequential: the input file is

scanned from beginning to end, and each run file is written from beginning to end.

Sequential access is critical for performance on rotational disks, where seek time (moving the read/write head to the correct track) dominates access cost. A single seek costs 5 to 10 milliseconds, equivalent to millions of CPU cycles. Once positioned, the disk can transfer data at 100 to 200 MB/s. By reading and writing sequentially, external merge sort amortizes seek costs across large transfers, achieving near-peak bandwidth utilization.

Merge phase. The merge phase interleaves reads from multiple run files. Although each individual run is read sequentially, the algorithm alternates between runs as it selects the globally smallest record. On a single disk, this interleaving introduces seeks between runs, potentially degrading performance. Implementations mitigate this cost by using large buffers (reducing the frequency of seeks) and by read-ahead (prefetching data before it is needed).

The output of the merge phase is also written sequentially. As with run generation, sequential writes maximize bandwidth utilization and minimize seek overhead.

Why bundle keys with values? A sorting algorithm compares keys, not values. In principle, one could sort only the keys (along with pointers to their associated values) and then gather values into sorted order at the end. Traditional external sorting does not do this. Instead, it moves complete records (keys and values together) through both phases.

This design reflects storage device characteristics. On block-addressed devices, the minimum transfer unit is a block (typically 512 bytes to 4 KiB). Reading a 10-byte key from a 100-byte record still transfers the entire block containing that record. If keys and values are stored separately, fetching values after sorting requires random reads to scattered locations, each in-

curring a full seek plus rotational delay. The cost of these random reads far exceeds the cost of moving values sequentially along with keys.

Bundling also simplifies buffer management. When keys and values move together, the algorithm needs only one set of buffers; separating them would require coordinating two parallel data flows with different access patterns.

3.1.3 Sorting on Modern Storage

The design of external merge sort embeds several assumptions about storage:

- **Random access is expensive.** Seeks dominate random access cost, so algorithms should maximize sequential access and minimize random access, even if this means moving data that will not be used.
- **Access granularity is coarse.** Block-addressed devices transfer data in fixed-size units. Fine-grained access incurs amplification: reading a single byte requires transferring an entire block.
- **Reads and writes have similar costs.** On HDDs and SSDs, read and write bandwidths are comparable. Algorithms do not strongly prefer one over the other.

These assumptions shaped decades of algorithm design. Systems bundle related data to enable sequential access, prefetch aggressively to hide latency, and batch small requests into larger transfers to amortize per-operation overhead.

Byte-addressable storage invalidates these assumptions. As described in Section 2.2.2, byte-addressable storage devices such as Intel Optane PMEM and CXL-attached memory exhibit fundamentally different characteristics:

- **Random read performance approaches sequential.** Without mechanical seeks, random and sequential reads achieve similar bandwidth. The penalty for scattered access is minimal.
- **Fine-grained access without amplification.** Byte-addressable devices support loads and stores at cache-line or smaller granularity. Reading a 10-byte key does not require transferring a 4 KiB block.
- **Reads are cheaper than writes.** Write bandwidth is lower than read bandwidth, and writes do not scale as well with concurrency. Additionally, concurrent reads and writes interfere, degrading read performance.

These characteristics invert the traditional tradeoffs. Bundling keys with values, which made sense when random reads were expensive, now wastes bandwidth by moving values that sorting does not use. The asymmetry between reads and writes favors algorithms that perform additional reads to avoid writes.

On byte-addressable storage, an alternative approach becomes viable: sort only keys (with pointers to values), then gather values into sorted order using random reads. This approach reduces write traffic during run generation and merge (since only keys move through intermediate files) and replaces sequential value reads with targeted random reads that fetch only the bytes needed.

Whether this tradeoff is beneficial depends on workload characteristics (the ratio of key size to value size) and device parameters (the relative costs of random reads, sequential reads, and writes). The remainder of this chapter develops WiscSort, a sorting algorithm that exploits these properties. WiscSort separates keys from values, manages concurrency to respect device constraints, and schedules I/O to avoid read-write interference.

3.2 Byte-Addressable Storage Characteristics

Chapter 2 introduced byte-addressable storage (BAS) in §2.2.2. Because the sorting techniques in this chapter depend directly on BAS device behavior, we briefly recap the relevant characteristics.

BAS devices such as Intel Optane DC Persistent Memory and CXL-attached memory sit between DRAM and SSDs in both latency and capacity. They differ from traditional block storage in several ways that are consequential for sorting:

- **Access granularity.** BAS supports byte-level addressing. On Optane PMEM, the internal access unit is 256 bytes (the *XPLine*), but applications can issue loads and stores at arbitrary granularity without the amplification penalties of block devices [159, 88]. CXL-attached memory operates at 64-byte flit granularity [141].
- **Latency.** Read latencies are roughly 3× higher than DRAM (approximately 300 ns versus 100 ns for local DRAM on Optane), while write latencies are higher still due to internal wear-leveling and persistence requirements [159].
- **Bandwidth.** A single Optane DIMM provides approximately 6.6 GB/s read and 2.3 GB/s write bandwidth, a 2.9× asymmetry. Bandwidth scales with the number of DIMMs but saturates at fewer concurrent threads than DRAM [159].
- **Random versus sequential access.** Unlike HDDs and SSDs, random read bandwidth on BAS approaches sequential read bandwidth (within 18% on Optane for 256 B accesses [159]). This property fundamentally changes the calculus for data layout: algorithms that avoided random access to amortize seek costs can now reconsider.

- **Read-write interference.** Writes are slower than reads and do not scale as well with concurrency. Concurrent writes can also interfere with read performance, reducing read bandwidth by up to 2× when reads and writes overlap [159, 155].

These characteristics invert several traditional assumptions. On HDDs, sequential access was paramount because seeks dominated cost; systems bundled data together even if only a subset would be used. On BAS, the cost of reading unnecessary bytes is no longer amortized by avoiding seeks. Similarly, the asymmetry between reads and writes favors algorithms that trade additional reads for fewer writes. The next section develops the BRAID model, which formalizes these properties into a framework for algorithm design.

3.3 Motivation

Chapter 3 introduced external sorting (§3.1) and byte-addressable storage characteristics (§2.2.2). This section develops the BRAID model, which captures the device properties most relevant to sorting, and demonstrates why existing approaches fail to exploit these properties.

3.3.1 The BRAID model

We develop a device model for byte-addressable storage that specifies the important properties that distinguish it from other storage media. Being cognizant of these unconventional properties is crucial for maximizing performance. The BRAID model constitutes a device with the following five properties:

1. **Byte Addressability (B).** BAS supports load/store access at fine granularity, avoiding the amplification inherent in block-addressed devices. On Intel Optane PMEM, the internal access unit is a 256-byte

XPLine; applications can issue reads at cache-line (64 B, 256 B) granularity without transferring an entire 4 KiB block [159, 88]. By contrast, reading a 10-byte key from an HDD or SSD requires transferring at least a 512-byte or 4 KiB sector, a 40× amplification for GraySort-sized records.

2. **Higher Random-Read Performance (R).** Random-read performance on BAS approaches sequential read performance for larger accesses. Concurrent random reads on Optane PMEM at only 18% slower than concurrent sequential reads for 256 B accesses [159]. The gap stems from an on-DIMM prefetch buffer (the *XPBuffer*) that benefits sequential patterns; random patterns bypass this buffer but still achieve high throughput because the 3D XPoint media itself supports fast random access [88]. This is a stark contrast with HDDs, where a seek penalty makes random reads orders of magnitude slower than sequential, and with NAND SSDs, where random 4 KiB reads are typically 1.5–2× slower than sequential.

3. **Asymmetric Read-Write Cost (A).** There is a vast difference between read and write performance on BAS. On Optane PMEM, a single DIMM provides 6.6 GB/s sequential read bandwidth but only 2.3 GB/s write bandwidth, a 2.9× ratio [159]. Write latency is further inflated by internal wear-leveling and the read-modify-write cycle triggered by sub-XPLine stores [88]. By comparison, DRAM exhibits only a 6% read-write bandwidth gap, and HDDs are roughly symmetric.

4. **Read-Write Interference (I).** Read performance degrades when concurrent writes are issued on BAS. This degradation increases with an increased number of concurrent writes. On Optane PMEM, the write pending queue creates head-of-line blocking: concurrent writes can reduce read bandwidth by up to 2× [154, 159]. Sub-256-byte writes are particularly harmful because they trigger implicit reads

(read-modify-write at the XPLine level), compounding the interference [155]. However, the opposite effect is minimal, i.e., there is little to no degradation in write performance when multiple reads are performed concurrently [154].

5. **Device-Constrained Concurrency (D).** BAS exhibits specific concurrency constraints; for example, writes do not scale well, but reads do until the number of read threads matches the total physical cores. On Optane PMEM, write bandwidth saturates at approximately 4 threads across 6 interleaved DIMMs, and adding more threads can actually *decrease* throughput, a pattern DRAM never exhibits [64, 159]. Performing writes with the maximum number of threads can be ~2x slower than peak write performance [64].

Do these properties generalize beyond Optane PMEM? BRAID serves as a comprehensive model encompassing various devices with different characteristics, although not all devices may possess all of these traits. Rather than speculate about hypothetical future devices, we consider three device classes for which published performance data now exists.

CXL-attached DRAM expanders (e.g., Samsung CMM-D, Micron CZ120) are commercially deployed, with measured read latencies of 200–270 ns on ASIC-based devices [141, 142]. These devices exhibit **B** (64-byte CXL flit granularity) and moderate forms of **I** and **D**: tail latencies spike under contention [107], and per-device bandwidth plateaus at roughly 25–30 GB/s due to the PCIe Gen5 link [141]. However, because the underlying media is DRAM, random and sequential reads perform similarly (~20% gap, as in local DRAM), and reads and writes are nearly symmetric, with only a 7–26% write penalty introduced by CXL protocol overhead [161]. CXL-attached DRAM therefore *lacks* the strong forms of **R** and **A**. A sorting algorithm on such a device would benefit from key-value separation (**B**) and thread-pool control (**D**), but interference-aware scheduling and write-avoidance

strategies would provide diminished returns. Our emulated BRD-Device (§3.5.5.2) approximates this profile, and the results confirm this prediction: WiscSort OnePass remains fastest due to reduced data movement, but interference-aware scheduling provides no measurable benefit.

Samsung CMM-H (CXL Memory Module – Hybrid) pairs a 16 GB DRAM cache with a 1 TB NAND SSD backend behind a CXL.mem interface [162]. This device exhibits all five BRAID properties, but through a different mechanism than Optane: byte addressability is a protocol-level abstraction (cache misses fetch 4 KiB pages from NAND), read-write asymmetry arises from slow NAND writes rather than 3D XPoint media physics, and bandwidth saturates at just 2–4 threads. Measured latencies of 1,100–1,475 ns are 9–12× higher than local DDR5 [162]. Our emulated BARD-Device (§3.5.5.3) approximates the asymmetry profile, and the results show that WiscSort's write reduction yields a 2× improvement over external merge sort.

Ultra-low-latency SSDs (e.g., Kioxia FL6 with XL-FLASH [95], Samsung Z-SSD [135]) exhibit strong read-write asymmetry (3.75–4.4× in IOPS) and moderate concurrency constraints, but they are *block-addressed* devices that lack the **B** property entirely. Without byte addressability, key-value separation cannot avoid I/O amplification: reading a 10-byte key still transfers a minimum 4 KiB block. Our emulated BD-Device (§3.5.5.1) models a hypothetical byte-addressable device with poor random reads, and the results show that external merge sort dominates in that regime. No shipping device occupies this exact point in the BRAID space, because the technologies that enable byte addressability (DRAM, 3D XPoint, CXL protocols) inherently support efficient random access.

The BRAID characterizes steady-state performance properties and does not capture transient non-linearities (e.g., performance cliffs when a device-internal DRAM cache fills with dirty data, or latency spikes from SSD garbage collection), nor does it model persistence semantics (crash consistency, durability ordering) of byte-addressable storage.

In short, BRAID properties are not binary and they do not always appear together. The model's utility is that it decomposes device behavior into independent axes, allowing an algorithm designer to predict which optimizations transfer to a new device and which do not. Our evaluation on emulated devices (§3.5.5) demonstrates this concretely: as properties are added or removed, different WiscSort components become more or less valuable, and the model correctly predicts the outcome in each case. In the rest of the paper, we sometimes refer to a BRAID device simply as BRAID for brevity.

3.3.2 The Question: How to Sort on BRAID?

Existing sorting algorithms, unfortunately, do not readily translate into efficient BRAID solutions. In the following, we look at two types of sorting on BRAID: 1) in-place sorting with the BRAID device treated as a slower DRAM, and 2) external sorting with BRAID treated as a faster HDD/SSD. We demonstrate why using these two existing approaches to sort on a BRAID device is inefficient.

3.3.2.1 BRAID As A Slower DRAM

A BRAID device is an order of magnitude slower than DRAM, making direct in-place sorting on PM inefficient. In-place sorting algorithms, such as sample sort, move records around based on pairwise record comparisons. These algorithms produce $\log_2 N$ times the record movement traffic normalized to the dataset size. When sorting in-place directly on BRAID without using DRAM, all of the traffic translates into slow BRAID accesses. In contrast, if we use external sorting algorithms, a significant portion of the traffic will be served by fast DRAM, reducing total sorting time significantly. External merge sort on BRAID, for example, produces $(1+M)$ times the dataset size of BRAID traffic (M is the number of merge phases, $M=1$ in dominant cases).

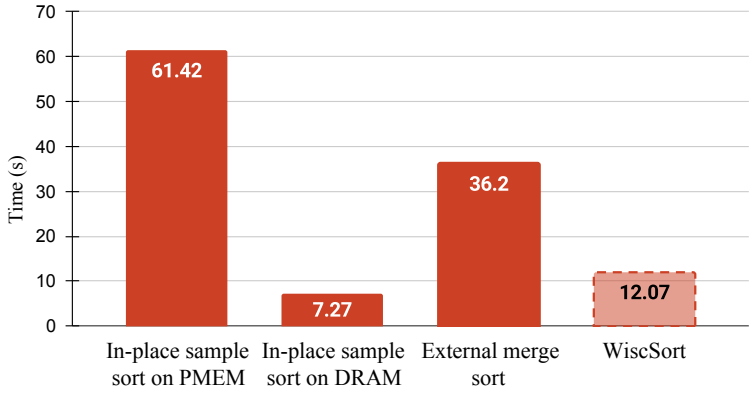


Figure 3.1: **Problems of different sorting approaches on PMEM.** We plot sorting time of 1) In-place sample sort on PMEM, 2) In-place sample sort on DRAM, 3) Traditional external merge sort, and 4) our WiscSort on PMEM for a 20GB workload containing 200M records with 10B keys and 90B values.

A state of the art in-place concurrent sample sort [44] fails to consider the concurrency constraints or the interference properties of the BRAID model, as it was designed for DRAM. As shown in Figure 3.1, external merge sort performs ~2x faster than in-place sample sort, as it requires much less device traffic and respects the device concurrency properties in comparison. Moreover, in-place sorting on DRAM is ~10x faster than in-place sorting on PMEM. Hence, we conclude that in-place sorting on PM is inefficient.

3.3.2.2 BRAID As A Faster Disk.

Previous external sort implementations designed for HDD/SSD are also insufficient on BRAID, particularly in terms of value movement during sorting. Assume the dataset consists of key-value pairs. External sorting implementations typically move values along with keys, even though sorting only involves key comparisons. For example, in external merge sort, both keys and values are read into DRAM and written to temporary BRAID files during run generation; similarly, both keys and values are read and written during merge.

Moving values with keys is effective because it leverages the sequential operation of HDDs/SSDs. Random reads on SSDs are ~1.5x slower and on HDDs are 10–100x slower than sequential reads (unlike (R)). Furthermore, the 4 KB access granularity of HDDs/SSDs is much larger than the value sizes in common sorting workloads (e.g., 100 B in GraySort) (unlike (B)). If we do not move values with keys on HDDs/SSDs, we save sequential reads/writes of values but introduce slow random reads to fetch values into their sorted positions, and each random read yields large amplification (40x = 4 KB/100 B in the case of GraySort). Because of these HDD/SSD characteristics, moving values with keys is advantageous in external sorting (unlike (A)). Traditional concurrent external sorts [112] consider only merge-based or partition-based parallelism and are ignorant of the device-based parallelism required on modern storage (unlike (I,D)) for maximum performance.

However, unlike HDD/SSD, BRAID devices have fundamentally different performance characteristics, necessitating this tradeoff to be reconsidered. BRAID has a limited write bandwidth while providing excellent random-read performance (e.g., a single PMEM DIMM has 2.5GB/s sequential write vs 7GB/s random read bandwidth). Because BRAID supports fine-grained access (256B for PMEM), small random reads required by sorting workloads become significantly more efficient on BAS when compared to HDD/SSD. As we will demonstrate, due to these unique characteristics, existing data movement schemes that do not comply with the BRAID model leave the true potential of BAS largely underutilized.

3.3.2.3 Separating Key from Values.

Separating the key and value to improve sorting is a classic idea. A 1963 CACM paper [84] proposed separating keys from values to perform just the key-pointer sort; however, due to the slow random reads on hard drives, they convert all random reads to sequential reads for gathering the values,

Table 3.1: **Sorting system's compliance with the BRAID model.**

System	B	R	A	I	D
External merge sort (naive)	✗	✗	✗	✗	✗
In-place sample sort [44]	✔	✔	✗	✗	✗
External merge sort	✗	✗	✗	✔	✔
Modified-key sort [84]	✗	✗	✔	✗	✗
PMSort [82]	✔	✗	✔	✗	✗
WiscSort	✔	✔	✔	✔	✔

thus performing more sorts than required. Moreover, they fail to address the I/O amplification of using block accesses when processing small keys. We examine this six-decade-old approach for modern hardware, which is byte addressable and has random bandwidth reaching near-sequential bandwidths.

PMSort [82] performs key-value separation for PMEM to reduce write traffic, focusing on the single-threaded case. However, it does not fully exploit BRAID properties and makes some choices that do not scale. 1) PMSort does not fully take advantage of the random-read bandwidth, as it loads both keys and values to the memory during the RUN phase. 2) They conclude QuickSort is the best approach for sorting on PMEM, but as we will show (see Figure 3.1), this does not scale. 3) PMSort avoids performing random reads (like [84]) and claims bandwidth not to be the bottleneck, which is not true at scale. 4) PMSort focuses on wear-leveling, thus not fully utilizing all the properties of a BRAID device.

Table 3.1 summarizes how different sorting systems adhere to the BRAID model. Traditional external merge sort is not device concurrency constraint aware, but we add a thread pool controller (Sec 3.4.4) and interference-aware scheduling (Sec 3.4.5) for it to be a competitive comparison against WiscSort. Modified-Key Sort makes a conscious decision to avoid random reads due to the exorbitant cost on older devices. PMSort does not com-

pletely take advantage of random-read performance and does not attempt to be device concurrency or interference aware. WiscSort, in contrast, is a practical real-world sorting system that takes advantage of all the properties specified in the BRAID model for maximum performance.

3.3.3 Our target workload

Based on the observations, we aim to design a system that efficiently sorts large volumes of input data using byte-addressable storage. As row-oriented binary data formats become relevant again [9, 7, 11], we rely on the format specified by sortbenchmark [21], a well-known sorting benchmark designed to stress test the I/O subsystem. Specifically, the workload has uniformly random keys, it is read from and written to files on BRAID, and the size of keys and values are fixed for a dataset. For variable length values, we rely on Key-Length-Value encoding [16], where a fixed size key is followed by the length of the value and the value itself. This simple and widely used record format (SQLite [22], PostgreSQL [17], etc.) allows us to make no assumptions about the index data structures. Additionally, we assume the BRAID capacity is large enough to fit the dataset.

3.4 WiscSort

In the previous section, we discussed how byte-addressable storage has different properties compared to slower counterparts such as HDDs or SSDs. Using these properties, we introduce WiscSort, a new algorithm that performs external merge sort on BRAID. WiscSort is a single-machine sorting algorithm that exploits the properties of the BRAID model (3.3.1) to efficiently utilize the higher bandwidth offered. Although the properties of the BRAID model are derived from PMEM, we expect most of them to also be present in future storage devices.

3.4.1 Design goals

Based on the BRAID model, we derive five high-level goals to achieve high throughput:

- **Reduce I/O amplification.** Small accesses to storage devices need no longer be amplified due to the byte granularity offered. Since BRAID bandwidth can be saturated even when making small accesses, reducing total I/O traffic by making byte-level accesses should be preferred.
- **Avoid redundant reads.** Conventionally, systems focus on maximizing bandwidth using sequential accesses; however, this approach is no longer required due to BRAID's high random-read bandwidth.
- **Trade more reads for fewer writes.** Current BRAID devices have asymmetric read-write costs; thus, trading more reads for fewer writes to maximize performance is preferred.
- **Manage access concurrency.** Spawning too few threads or too many threads can hurt performance on BRAID. Hence, we must appropriately size the thread pool based on the access type and the device to maximize I/O bandwidth.
- **Avoid read-write interference.** Overlapping read and write workloads on BRAID can lead to reduced read bandwidth. Thus, we must ensure that reads and writes are not overlapped.

3.4.2 Overview

WiscSort contains four algorithmic innovations to achieve these design goals. First, WiscSort utilizes **key-value separation** to reduce I/O amplification, avoid redundant reads, and trade more reads for fewer writes. Second, WiscSort includes a **thread-pool controller** that determines the number

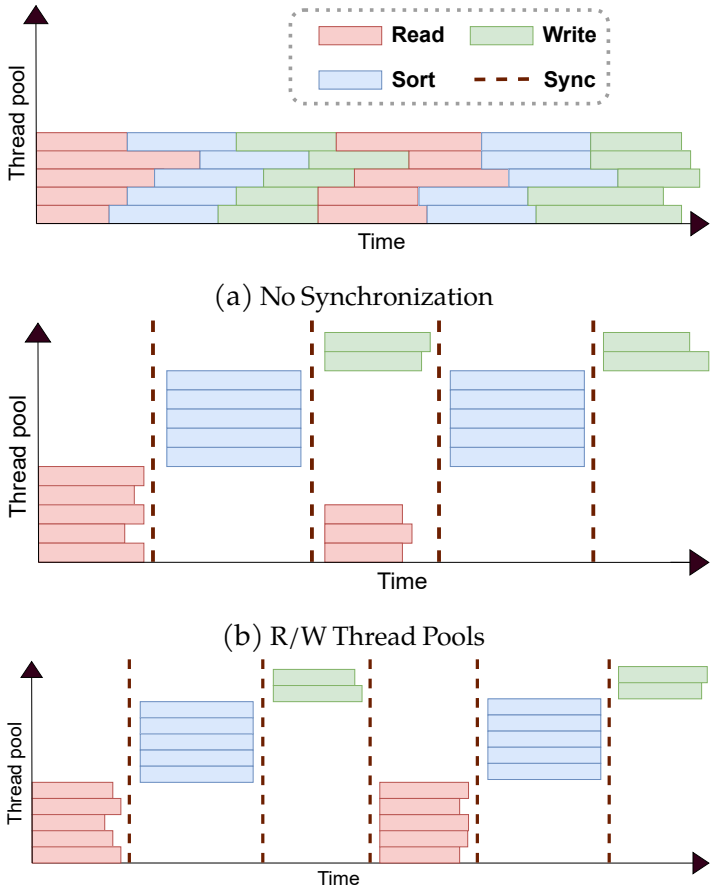


Figure 3.2: **Three different concurrency mechanisms.** To maximize BRAID device bandwidth, WiscSort prefers 3.2c model (interference aware, thread-pool controller) over 3.2b (thread-pool controller) and 3.2a (interference unaware). In all three models, the sort pools are the same size; the size of read and write pools may differ for (b) and (c).

of threads for reads and writes for the device for managing access concurrency. Third, WiscSort introduces an **interference-aware scheduler** that ensures reads and writes are not overlapped to avoid interference. Finally, WiscSort capitalizes on the fact that keys and values are separated to sort larger amounts of data in memory in a single pass without a merge phase

(**OnePass**).

Similar to external merge sort, WiscSort has two phases: the run generation phase and the merge phase. Instead of creating runs of key-value pairs (as in external merge sort), WiscSort creates runs of key-pointer pairs which refer to the values in the original input files. We call these new runs *IndexMaps*. During run generation, multiple threads read disparate partitions of the input file into DRAM and create key-pointer pairs; multiple sorting threads then concurrently sort individual runs; finally, multiple write threads persist the runs as IndexMap files. In merge phase, all the IndexMap files are read concurrently and merged. The final sorted data is then persisted.

Compliance with BRAID model: WiscSort reads only the keys from the device, leaving the values in place. As the value is not utilized to produce the ordered records, there is no motivation to read it to perform the sort. WiscSort reads only the keys, which is facilitated by the (***B***) property. Splitting the records leads to reading keys at strided locations (non-sequential). Due to (***R***), there is a minimal performance impact. Additionally, as the values never follow the keys while sorting, the amount of data written during the run phase is vastly reduced, addressing the (***A***) property.

WiscSort performs interference-aware scheduling of read and write operations addressing the (***I***) property. At any given point, either reads or writes are issued, thereby avoiding the interference created by read-write operations. WiscSort uses the thread pool controller to appropriately size the pool for a given access type (reads or writes) on the device, achieving the (***D***) property.

Lastly, WiscSort can sort in just one pass, bypassing the creation of IndexMap files when the keys and their pointers can fit in the DRAM. We call this version *WiscSort OnePass*. If the total size of the keys and pointers does not fit in the memory, WiscSort, like external merge sort, performs the run and merge phases. We call this version *WiscSort MergePass*.

3.4.3 Key-Value Separation

Maintaining values in the run files and reading and writing values when not used for sorting is one of many external merge sort performance pitfalls. WiscSort is motivated by a simple revelation that keys and values can now be separated because of the byte-level granularity offered by BRAID (B) without massive amplification costs during reads and writes. WiscSort reads only keys from the record in a strided fashion leading to non-sequential reads. Each key read has a pointer associated with it to represent the file offset of the record. We call this key-pointer combination an *index* and the list of key-pointers an *IndexMap*. The IndexMap is stored in the memory during sorting and is later persisted.

To understand the impact of splitting records, let us consider a simple example. Assume a dataset that comprises records with a 10-byte key and a 90-byte value. Traditional external merge sort will read 100 bytes ($10 + 90$), as it focuses on harnessing sequential read performance; WiscSort only reads the 10B key, resulting in a 10x reduction in read I/O traffic. Moreover, the run files are read during the merge phase again, leading to another 10x reduction (or more if multiple merge phases are required). Similarly, there is also a significant reduction in the write traffic also. Instead of the value, only a 5-byte[1] pointer is persisted along with the key. In the example, there will be a ~7x reduction in the write traffic. As the writes are slower than reads (Property A), the write traffic reduction improves the performance.

WiscSort has $2N(V-P)$ less read and write traffic compared to external merge sort in the worst case (MergePass) and $2N(K+V)$ bytes traffic reduction in the best case (OnePass), where N is the number of records and K,V,P are the key, value and pointer sizes.

[1] 5B represents 2^{40} (~1 trillion) record offsets, irrespective of the size of the record. 8B can be used if larger dataset is required, resulting in a 5x write traffic reduction.

3.4.4 Thread-Pool Controller

Traditional high-performance applications want to maximize CPU utilization, so they tend to overlap all operations when possible. In the no-synchronization concurrency model shown in Fig 3.2a, each thread in the pool repeatedly reads some data, sorts it, and writes it to a file. However, in this approach, there is no way to *control* the number of concurrent threads performing a particular action, and straggler threads may overlap read write operations, causing minor read-write interference. In WiscSort, the thread-pool controller determines the thread pool size to be used for a particular operation (read, sort, and write) as shown in Fig 3.2b and 3.2c.

Given that a wide range of BRAID devices will exist in the future, deciding the number of requests to be sent concurrently is a non-trivial task. In our system, a microbenchmark determines the device's peak bandwidth capabilities and scaling behavior. The controller then utilizes this information at run time to determine the thread pool sizes. For example, the pool size of reads is much larger than that of writes (Figure 3.2c). In future Linux versions, one can directly gather the device performance data from Heterogeneous Memory Attribute (HMAT) tables [12].

3.4.5 Interference-Aware Scheduling

As observed in many prior works [128, 64, 159], writes do not scale well on BRAID, and, therefore, one may be motivated to overlap reads and writes as shown in Figure 3.2b. However, BRAID's performance interference between reads and writes ((I)) nullifies the benefit gained by overlapping them. Thus, it is important to isolate read and write operations.

WiscSort relies on interference-aware scheduling, where only read or write operations are issued at any given time, as shown in Figure 3.2c. The majority of the read operations occur at the start of the run phase and merge phase, while the write operations occur towards the end of the phases.

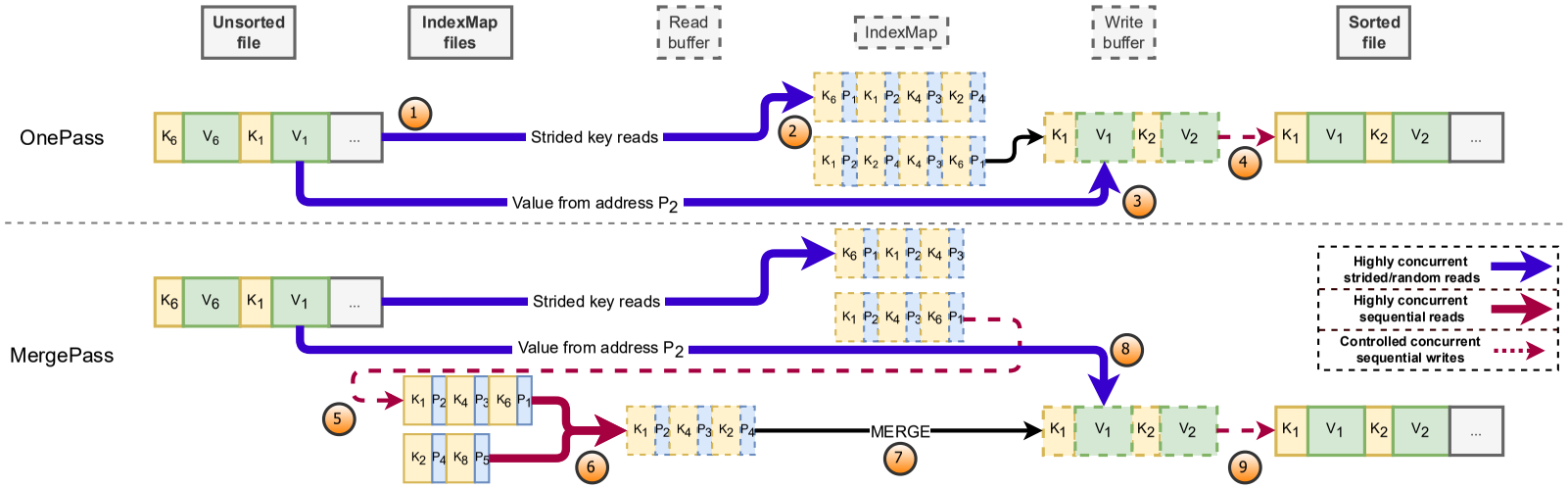


Figure 3.3: **Data flow diagram of WiscSort OnePass and MergePass.** `K, V, and P` represent a Key, Value, and Pointer. Pointer is the offset at which the corresponding value exists in the input file. Solid boxes indicate data persisted on BRAID and the dashed boxes show data in DRAM. There are six different stages a key can be in, as shown at the top of the figure. Solid arrows are reads, and dashed arrows are writes. The red color represents sequential accesses and the blue non-sequential. The thickness of the arrow is proportional to the concurrency in the corresponding operation.

While the data is being read in the run phase, WiscSort uses a write buffer to temporarily store the sorted data in the memory and periodically flushes the write buffer while stalling the reads. Similarly, the values gathered during the merge phase are isolated from the writes to the device through a write buffer. These buffers act as a logical barrier between different kinds of operations. The write buffers are essential for adding a control point to separately size the number of reader and writer threads and to avoid any interference between these operations.

3.4.6 More Keys in One Pass

Traditional sorting algorithms sort keys in a single pass (without writing intermediate results) when all keys and values fit into main memory, but must perform a second pass merge phase when the keys and values exceed memory capacity. In contrast, splitting keys from values (via the IndexMap) reduces the memory footprint of WiscSort, and enables it to sort keys in a single pass when keys and pointers fit in main memory.

In the one-pass version of WiscSort, the IndexMap is concurrently loaded into memory through strided key reads and then sorted in-place. If we ignored the (I) property, the values could be moved directly from the input file to the output file. However, WiscSort-OnePass performs thread-pool and interference-aware scheduling with a write buffer to optimize performance.

3.4.7 Algorithm

Fig 3.3 represents the operation of WiscSort, including how data flows and state changes across steps for fixed-size records. Steps 1-2 are performed regardless of whether one or multiple passes will be needed. Section 3.4.7.3 will describe the minor changes required to handle variable length records.

❶ ***RUN read:*** For a given input file, WiscSort determines the maximum IndexMap size that can fit in memory while aligned to the input file size.

Consecutively the appropriate portion of the input file is evenly partitioned amongst the reader threads. Each reader thread performs a strided read of the keys in its partition to the unsorted IndexMap, generating record-id pointers on the fly to reference the offset of the value in the original input file. Since the record sizes are fixed and contiguous (Sec 3.3.3), each pointer is a hex address, calculated as (`start_address+record_id*record_size`).

❷ ***RUN sort:*** Once all threads finish reading, a concurrent in-place sample sort is performed on the IndexMap. If the IndexMap fits into DRAM, WiscSort will perform just one pass as in steps 3-4; otherwise, WiscSort performs two passes as in steps 5-9.

3.4.7.1 WiscSort OnePass

❸ ***RECORD read:*** The sorted IndexMap is divided across a predetermined number of read threads provided by the thread pool controller; each thread performs a random read for the value from the unsorted file and places it into a write buffer.

❹ ***RUN write:*** Once the write buffer is full, it is written sequentially to the output sorted file. The write buffer enables interference-aware scheduling; however, without it, WiscSort would still be faster than external merge sort simply because of finishing the entire sorting in a single pass.

3.4.7.2 WiscSort MergePass

As seen in **Steps 1 and 2**, the keys are gathered and sorted in chunks equal to that of the IndexMap size.

❺ ***RUN write:*** Since a single IndexMap does not contain all the keys of the input file, it is temporarily written to a file on BRAID in a sequential and concurrent manner; this does not require an output buffer. **Steps (1, 2, and 5)** are repeated until all the keys of the input file are read, where a set of sorted IndexMap files are generated. This marks the end of the run phase.

❻ ***MERGE read:*** In the merge phase, the read buffer is split evenly amongst the number of IndexMap files. Reader threads then sequentially load a chunk of each IndexMap to its appropriate area in the read buffer. Once a set of keys from all the IndexMaps fill the read buffer, a set of cursor pointers indicates the current and the end for the space allocated to the IndexMap file. The current cursors indicate the keys to be compared across the IndexMap files, and the end cursor pointer points to the last key in the space allocated for that IndexMap. These pointers are reset every time a new set of keys is read from the IndexMap file to its space in the read buffer.

❼ ***MERGE other:*** WiscSort finds the minimum of the keys pointed to by the current pointers. The minimum key is then enqueued to an offset queue that maintains a list of pointers whose values must be read into the write buffer. WiscSort does not fetch the value after finding each min key because single thread random read bandwidth is poor.

❽ ***RECORD read:*** The size of the offset queue is determined by the size of the write buffer. Once the offset queue is full, WiscSort performs concurrent random reads of records to retrieve the values from the input file and update the write buffer.

❾ ***MERGE write:*** Once the write buffer is full, it is sequentially written to the output file. The write thread-pool is controlled as per the device characteristics. If any current pointer reaches the end pointer, WiscSort will read the next set of keys from the respective IndexMap file to its allocated space in the read buffer. If all the keys are already read from that IndexMap file, the read buffer space allotted to this IndexMap will be transferred to a neighboring IndexMaps evenly and the number of keys compared to find the minimum key will be reduced by one. The pointers are also updated accordingly when keys are read for the neighboring IndexMap.

Finally, if only one IndexMap file remains, it is loaded completely to the read buffer. If WiscSort has finished traversing of all the IndexMap files and the write buffer is still not full, the partially full buffer is flushed to the

byte-addressable storage concurrently, marking the end of the merge phase. Throughout all of these steps WiscSort carefully ensures that the reads and writes never overlap through interference-aware scheduling.

3.4.7.3 WiscSort for variable length values.

Sorting the variable length records with fixed size keys (KLV - Sec 3.3.3) requires only two changes, the IndexMap layout, and the random read processing. The IndexMap file will now contain one additional attribute, the length of the value. So an IndexMap is now a list of `(<key, pointer, vlength>)` entries, and `pointer` now points to the byte-offset of the corresponding value. The following steps indicate the changes to the previously described algorithm,

❶ ***RUN read:*** Since the key byte offsets are unknown, concurrently reading the key and vlength is impossible. A single reader thread must serially read the key + vlength of each record to determine the next address to read from. The next key to be read is determined by appending the vlength to the existing key byte offset. Hence, the IndexMap file must be loaded serially in the RUN phase when dealing with the KLV format. This is restriction is shared by other sorting algorithms as well.

❸ & ❽ ***RECORD read:*** The offset queue used to perform concurrent random reads now maintains vlength along with the list of pointers (sorted) that must be read to the write buffer. Once this queue is full, the thread-pool controller evenly partitions the queue among an optimal set of reader threads. Each reader thread now reads the value of vlength size from the input file to the write buffer. Once the write buffer is full, it is written to the BRAID device.

3.4.8 Implementation

To saturate the BRAID bandwidth, we must determine the right number of read and write threads and access granularity. The thread-pool controller relies on this information to decide the pool sizes. Thus, we developed a microbenchmark suite to characterize the device's performance. In our setup (Sec 3.5), read bandwidth scales up to 16 threads (#physical cores) and saturates after that. Therefore, our implementation uses 16 to 32 threads (sequential and random) for reading data to the read buffers and 5 threads for writing from the write buffer, since writes do not scale with more concurrency.

External merge sort slightly benefits from a large read buffer during the merge phase, as the number of times the read operations are to be performed is reduced. In the case of WiscSort, the read buffer sizes determine the number of passes required. If the read buffer is small and the entire IndexMap does not fit into the buffer, then MergePass will have to be used. So when possible, larger read buffers and smaller write buffers are preferred in WiscSort. The size of the write buffer has no performance significance.

To perform in-place sort of keys and index/pointers, we employ the state-of-the-art sorting implementation IPS^{4}o [44]. The concurrency is implemented using C++'s standard threading module `std::thread`. The synchronization between threads performing an operation is achieved through a condition variable indicating if all the threads have finished their portion of the work. We employ concurrent operations whenever possible; for example, loading keys from the read buffer to the key array (`<key,read_buff_ptr>`) and moving keys from the key array and values from the read buffer to the write buffer are all performed concurrently. Finally, for optimized I/O accesses, we use `AVX512` non-temporal stores followed by a `clflushopt` for writes and `AVX256` instructions for reads.

3.5 Evaluation

In this section, we evaluate the benefits of the design choices made in WiscSort. We compare WiscSort to well-established sorting algorithms on standard sorting benchmarks and show how WiscSort effectively utilizes a real byte-addressable storage device – Intel Optane DC PMEM. Finally, we show the effectiveness of our techniques on emulated BRAID devices with varying properties.

All the experiments are run on a test machine with one Intel(R) Xeon Gold 5218 2^{nd} Gen CPU with scaling governor set to `performance`. There are two 16GB @2400MHz DRAM and four 128GB Intel Optane DC PMEM 100 @2666MHz placed on six distinct memory channels, as advised by Intel [14]. The operating system is 64-bit Linux 5.0, and the PMEM devices are configured to App Direct mode with `fsdax` namespace and ext4 as the file system.

To evaluate the effect of Key-Value Separation, Thread-Pool Controller, and Interference-Aware Scheduling,
we ask:

1. How does WiscSort perform on popular application benchmarks (e.g., sortbenchmark)? Does WiscSort utilize the BRAID device bandwidth effectively?

2. What is the benefit of concurrency optimizations? How does PMSort compare against WiscSort?

3. What is the benefit of key-value separation? How does the benefit vary with different key:value ratios? Should sequential reads be preferred over random reads for all key-value sizes when generating the IndexMap from the input?

4. How do WiscSort and other sorting methods perform on future BRAID devices (e.g., CXL) with different characteristics?

We answer the above questions using a series of microbenchmarks and industry-standard sorting benchmarks, and employ well-established techniques to emulate future BRAID devices [105].
We find that:

1. WiscSort OnePass is 3x, and MergePass is 2x better than concurrent external merge sort on the sortbenchmark. Moreover, WiscSort saturates the device bandwidth for a given operation, demonstrating the benefits of conforming to the *BRAID* model.

2. Being device concurrency aware provides up to 50% improvement in total time of WiscSort compared to I/O overlapping WiscSort counterparts. Also, WiscSort OnePass is 7x and
MergePass is 4x faster than PMSort.

3. WiscSort offers more improvements over external merge sort with larger $V:K$ ratios. WiscSort OnePass outperforms external merge sort regardless of the $V:K$ ratios, and MergePass outperforms external merge sort when $V:K > 1$. In addition, loading IndexMap via strided reads is always beneficial regardless of $V:K$ ratios.

4. Experiments on future CXL BRAID devices show external merge sort and WiscSort OnePass are most favorable amongst others on devices with poor random-read performance, large asymmetric read-write costs, and symmetric costs.

3.5.1 SortBenchmark

The sortbenchmark, first introduced in AlphaSort [124], is the de facto industry standard for stress testing I/O architectures [65, 90, 38, 156]. We evaluate WiscSort on sortbenchmark workloads as a representation of application workloads. The benchmark is to sort binary records with 10B keys and 90B values. The input file to be read and the generated output file must be placed

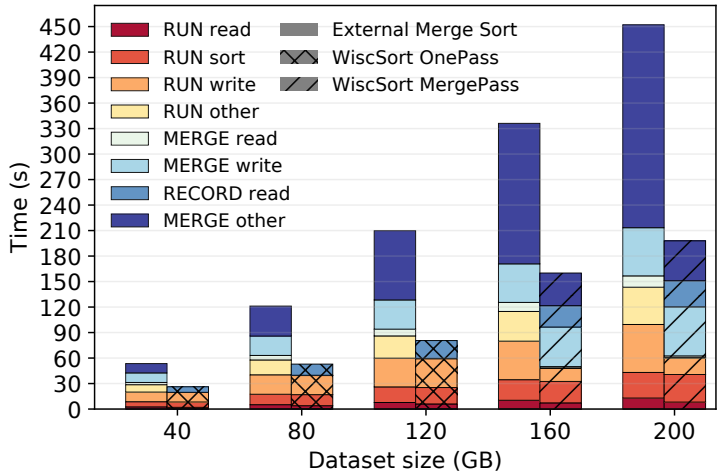
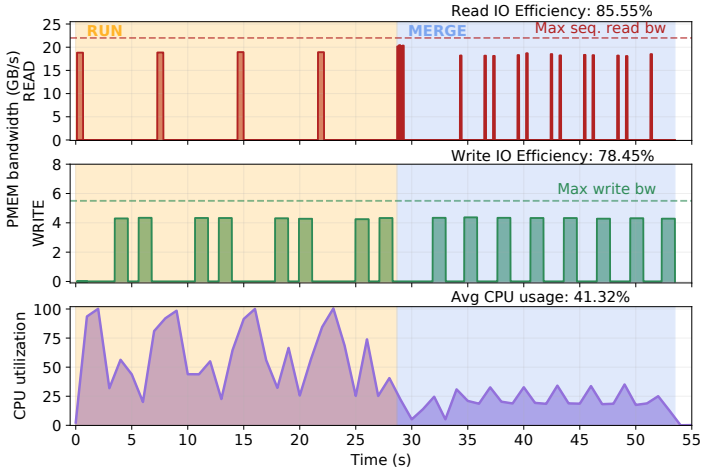


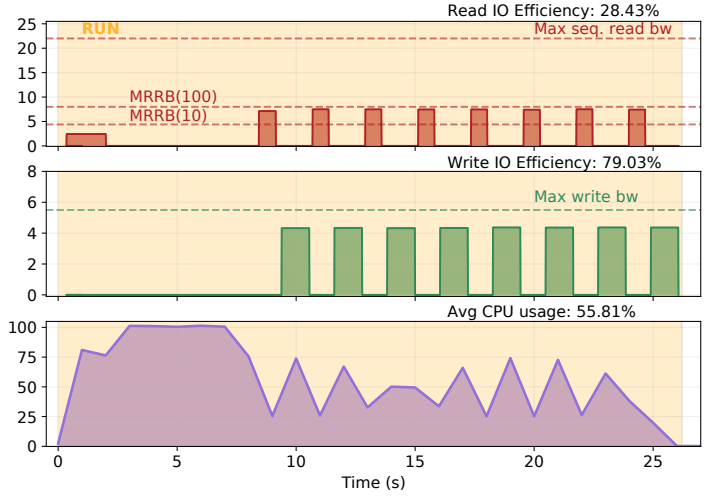
Figure 3.4: **WiscSort and external merge sort performance on sortbenchmark workload.** Since the key values sizes are fixed, the speedup between WiscSort and external merge sort is consistent for varying file sizes.

on BRAID (e.g., PMEM). The keys are of uniformly random distribution, and the output file must be a permutation of the input file, sorted in key ascending order. As shown in Figure 3.4, WiscSort consistently outperforms (2-3x) a competitive external merge sort implementation (using thread-pool controller and interference aware scheduling) for all sortbenchmark dataset sizes due to WiscSort's design choices (conforming to BRAID properties). The legend in the figure maps to the operations described in Sec 3.4.7.

WiscSort can sort up to 200 GB of input data on 448 GB of usable PMEM using a 5-byte pointer. The 200GB dataset benchmark is the largest workload we can run with our PMEM capacity. The benchmark requires 30 GB for IndexMap files and another 200 GB for the output file. When evaluating WiscSort MergePass, we limit the available DRAM capacity (32 GB) to 20 GB so that the IndexMaps of input files larger than 140GB do not fit entirely in the DRAM. The external merge sort uses a 10 GB read buffer and 5 GB write buffer. WiscSort only uses a 5 GB write buffer. The buffer size choice has no effect on either sorting system.



(a) External merge sort. 10GB read buffer, 5 GB write buffer



(b) WiscSort OnePass. With 5 GB write buffer

Figure 3.5: **Resource usage of external merge sort** (I+D) **and WiscSort OnePass** (B + R + A + I + D) **for sorting a 40 GB file.** The dotted lines represent the peak bandwidth possible. As reported by microbenchmarks, the Max Random-Read Bandwidth(*MRRB*) changes with access size. I/O efficiency compares actual time to ideal time for data operation. Ideal time = operation size / peak bandwidth. for example, the ideal time to read 20 GB on our setup is 0.90s (read size / max read bandwidth).

WiscSort OnePass is up to 3x faster than the external merge sort for the 40GB/80GB/120GB datasets. This is due to the 50% reduction in read/write traffic and the avoidance of the merge phase computation. The external merge sort is at least 25% slower than WiscSort during RUN read, with larger datasets (more value read traffic), the gap can reach up to 60%. The total in-memory sort times (RUN sort) are the same as the key array, and IndexMap is of similar size. In the 40 GB case, the RUN other in external merge sort accounts for 12% of total time; this includes the time taken to copy records from the read buffer to the key array and values from the read buffer to the output buffer concurrently. In contrast, WiscSort does not have the RUN other overheads because it does not have related operations, further reducing computation overhead. The total write time during the run phase is the same between the external merge sort and OnePass since external merge sort has to write all the key values to the run files, whereas WiscSort OnePass writes all the key values in the final sorted order to the output file.

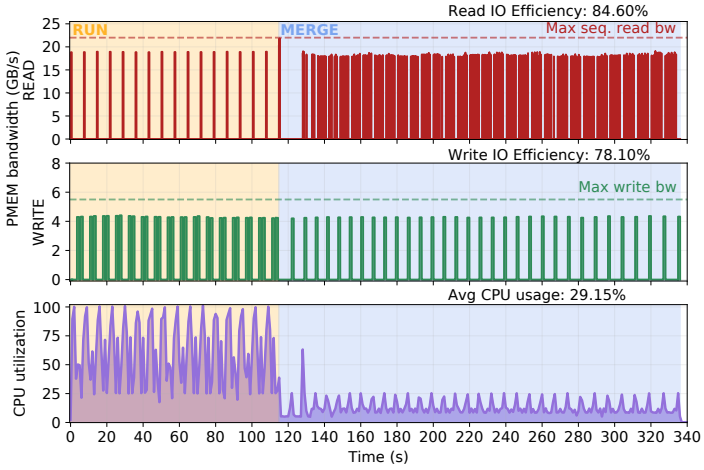
Figure 3.5 and 3.6 show the PMEM bandwidth and CPU resource usage of external merge sort, OnePass, and MergePass. As shown in Figure 3.5, WiscSort consumes less bandwidth than external merge sort due to its strided key reads and random reads of values. Figure 3.5b shows the repeated random reads made to fetch the records from the input file pointed by the indexes in the sorted IndexMap file (RECORD read). The strided key reads have lower bandwidth due to the smaller accesses performed (10B key compared to 100B records). Although reading the entire dataset sequentially to memory (MERGE read) is faster than RECORD read, OnePass avoids all other merge phase operations warranting the use of slower random reads (B+A+R property). The thread-pool controller sizes the pool appropriately to ensure all I/O operations (sequential and random reads, writes) perform at peak bandwidth as shown in Figure 3.5 & 3.6.

WiscSort MergePass is up to 2x faster than external merge sort for the 160GB/200GB datasets, due to the total reduction in read/write traf-

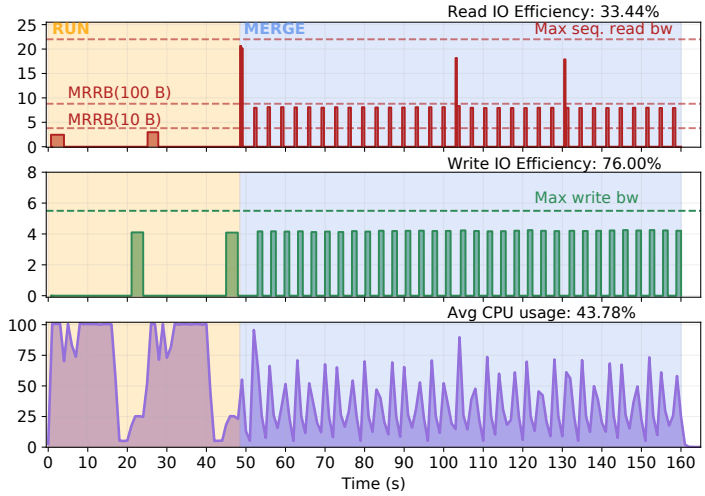
fic (42.5%). Like OnePass, MergePass takes (considerably) less time to load IndexMap files, comparing to external merge sort. Unlike OnePass, MergePass persists IndexMap files on the device (as in Figure 3.6b). During the merge phase, MergePass saturates device read bandwidth due to its sequential loads of the portion of the IndexMap file; in MergePass, there are fewer IndexMap loading reads (compared to external merge sort), since more keys can fit in DRAM due to key/value separation. As a result, for example, with the 160 GB dataset, WiscSort MERGE read time is 7x smaller than that of the external merge sort. In the merge phase, WiscSort MergePass generates random record value reads once keys are ordered after merge. Due to interference-aware scheduling, in WiscSort MergePass, no two types of I/O operations overlap.

MERGE writes dominate the overall time of WiscSort MergePass. The total write time of external merge sort is 2x of the total write time of WiscSort OnePass and 1.5x that of WiscSort MergePass, satisfying the (A) property. MERGE other time indicates operations other than reads and writes in the merge phase. In external merge sort, a single thread finds the minimum between keys from each run file and copies the record from the read buffer to the write buffer. This cannot be made concurrent since all the RUN files are merged in a single merge phase. On the other hand, the WiscSort MergePass performs concurrent copies of records to the output buffer directly since the read offsets are accumulated and submitted at once, as depicted by the better CPU utilization in Figure 3.6. A similar optimization cannot be applied to external merge sort because the records at input buffer offsets can potentially change before they are concurrently copied to the output buffer.

Overall, we show that WiscSort maximizes random-read bandwidth (R), reduces the amount of writes (A), takes advantage of the byte addressability (B) while maximizing CPU utilization, and is aware of the concurrency constraints of the device (I+D), thus making it a *BRAID compliant* algorithm. On the other hand, our implementation of external merge sort is only aware



(a) External merge sort. 10 GB read buffer, 5 GB write buffer



(b) WiscSort MergePass. 12 GB read buffer, 5 GB write buffer

Figure 3.6: **Resource usage of external merge sort** (I+D) **and WiscSort MergePass** (B+R+A+I+D) **for sorting a 160 GB file.**

of (I+D), making it a *non BRAID compliant* system.

3.5.2 Concurrency & Interference Optimizations

Figure 3.7 demonstrates the benefits of paying attention to the constrained concurrency and the read-write interference of a device. We compare multiple concurrency models (Figure 3.2) of traditional external merge sort, PMSort, and WiscSort against each other.

To differentiate the benefits gained by separating key and value, we compare the external merge sort No Sync (Figure 3.2a) and No I/O overlap (Figure 3.2c). During the merge phase of No Sync, we use a write buffer to make the output writes concurrent. The buffer enforces an order between read-write operations, which avoids interference, but it still suffers from write degradation due to uncontrolled pool sizes. The external merge sort No Sync has worse run time performance than any other multi-threaded sorting system, due to larger amounts of contention. Due to interference-aware scheduling and with thread-pool control, the No IO overlap performs 25.7% faster than No Sync. Indicating that the thread-pool control and interference-aware scheduling can improve performance for any device-unaware algorithm.

The published version of PMSort separates keys and values, but does not perform strided gather of keys during the run phase; additionally, it avoids value writes at the end of the run phase. It ignores device concurrency characteristics, does not investigate effective resource utilization, and avoids random reads whenever possible. Since the PMSort codebase is not available, we implemented the single-threaded version as specified [82]; we also built PMSort multi-threaded versions based on traditional concurrency models (Figure 3.2a and 3.2b) calling it PMSort+. Our PMSort+ implementation queues the random read offsets in the merge phase, so that the value gathering can be concurrent as done in WiscSort.

The merge phase of PMSort+ No Sync has no write buffer (unlike external merge sort No Sync). The values can be moved directly and concurrently from the input file to the output file once the offset vector is filled. This

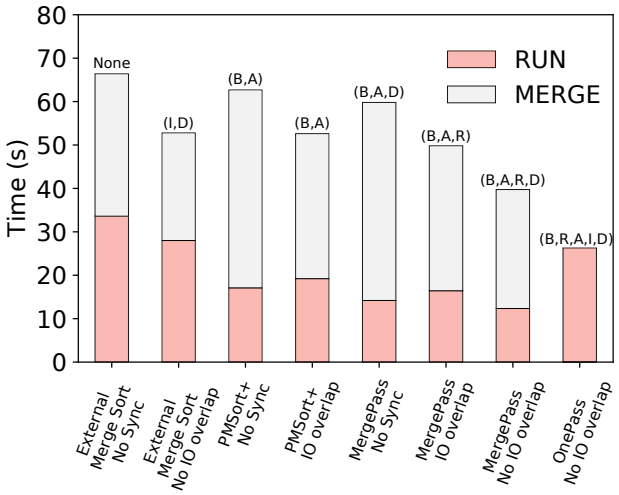


Figure 3.7: **Sorting systems using different concurrency models and the BRAID properties they fulfill.** Sorting 400M records of 10B K: 90B V each.

method, however, causes serious read-write interference coupled with poor write performance degradation, hence making No Sync 16% slower than I/O overlap. During the merge phase of all I/O overlap systems, we maintain two write buffers and two offset vectors to ensure that the random reads and writes always overlap; this helps quantify the effect of thread-pool control alone – the I/O overlap merge is 36% faster than No Sync.

We implement all three concurrency variants of WiscSort to study the effects of BRAID compliance. The run phases of WiscSort, unlike PMSort, perform strided gather, thus reducing the overall run time in comparison. However, the merge phase time between the two remains the same. Because of this, both WiscSort I/O overlap and No Sync perform better than PMSort+ equivalents. Moreover, the MergePass no I/O overlap, which performs interference-aware scheduling and uses thread-pool controller, performs 33% faster than the hypothetical best case of PMSort+ and ~**4x** faster than the actual (single thread). If the IndexMap fits into memory, WiscSort OnePass is **7x** faster than single-threaded PMSort. The single-threaded

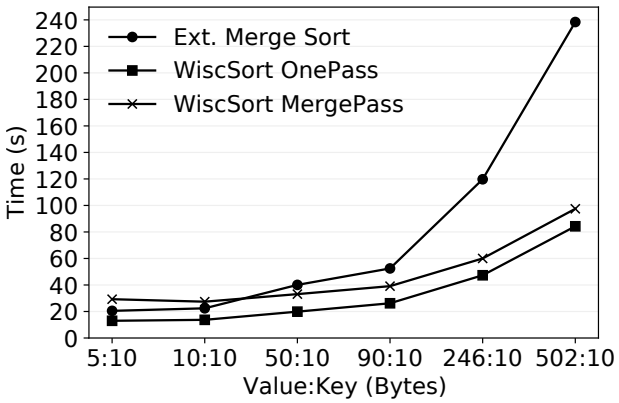


Figure 3.8: **Key Value splitting benefits for 400M records.** The key size is 10B, Pointer is 5B, and the value size varies.

WiscSort MergePass has no performance regression, although the single-thread random reads are bad. This slowdown is because of the redundant reads (of values) PMSort performs during the run phase and retaining only keys, causing two copies rather than one. Nevertheless, since intermediate writes are avoided, WiscSort OnePass will still be faster than PMSort.

Overall, we show that even a concurrency optimized external merge sort (I+D) can outperform sorting systems with naive concurrency, even if they separate key and values (B+A+R). Even amongst systems that comply with (B+A+R), choosing the appropriate concurrency model can result in huge gains. We also highlight the importance of utilizing the concurrency feature of the device itself (PMSort single-thread vs. multi-thread). Finally, the considerable benefits of interference-aware scheduling and thread-pool control shown in Figure 3.7 will only grow with larger datasets.

3.5.3 Random-Read Optimizations

To better understand the key-value separation impact, we conduct experiments where we vary the Value:Key ratios. It's worth noting that most production KV workloads (like Facebook[55]) have large values and small

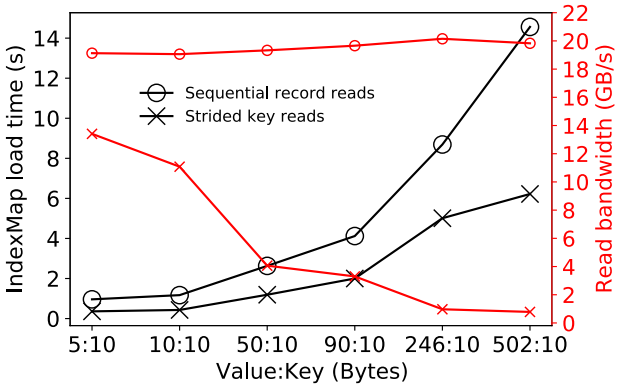


Figure 3.9: **Load IndexMap by Strided vs. Sequential reads for 400M records.** The key size is 10B, and the value size varies.

keys, making the key-value separation idea a good fit. In addition, we evaluate whether random/strided reads are more efficient than sequential reads for loading the IndexMap file with varying value-key ratios. We generate the datasets required using a custom tool built on top of Gensort [10].

As seen in Figure 3.8, WiscSort performs up to 3x (OnePass) and 2x (MergePass) better than external merge sort (I+D) when value sizes are larger than 90B. The improvement is due to the reduced traffic volume in WiscSort caused by the key-value separation. For WiscSort OnePass, the traffic reduction percentage is constant (50%) regardless of the value size. For MergePass, there is more traffic reduction with larger values; for example, MergePass has 48.5% traffic reduction with 502B value but 37.5% reduction with 50B value (compared to external merge sort). Figure 3.9 demonstrates the impact of such reduced traffic on IndexMap load time.

With values of medium sizes (i.e., 50B and 90B), Figure 3.8 tells a different story, where although there is a benefit of WiscSort MergePass, it is small in comparison. This decline is because of the lower traffic reduction and lesser impact of improved CPU usage of WiscSort during the merge phase. Since the key size is close to the value size, the benefits of splitting are small. Nevertheless, OnePass performs 2x better. The difference in

loading the IndexMap file through sequential vs. random is also reduced. Similarly, PMSort performs sequential record reads to memory and then gathers keys and pointers from it; as shown in Figure 3.9 it is up to ~3x worse than performing strided gather of keys.

For cases where the value size is the same as the key size or even smaller, MergePass performs worse than the traditional external merge sort. This regression is because smaller record sizes exhibit poor random-read performance compared to sequential read on PMEM during value gathering in the merge phase. Even though strided reads have good read bandwidth because multiple records can fit the 256B cache line (17 15B records and 12 20B records), the random read during the merge phase does not make use of this. However, external merge sort can reach peak read bandwidth even for small records because they are sequential. For values of 10B and smaller ($V:K < 1$), the write reduction will be minimal or none in the case of MergePass, thus splitting key-value unsuitable. However, OnePass does better because of property (A). Due to the disparity of read-write costs, multiple concurrent writes are still costlier than a single concurrent random read.

Overall, we demonstrated that irrespective of the $V:K$ ratio separating them on a BRAID device is beneficial, given that the IndexMap file fits in the DRAM (OnePass). For larger $V:K$, MergePass benefits further from the write reduction (A). Due to PMSort access patterns, it always underperforms in comparison to MergePass and OnePass for any $V:K$. This implication was derived from Figure 3.9 where strided gather performs better than sequential reads irrespective of the $V:K$ (R property).

3.5.4 Background I/O interference effects

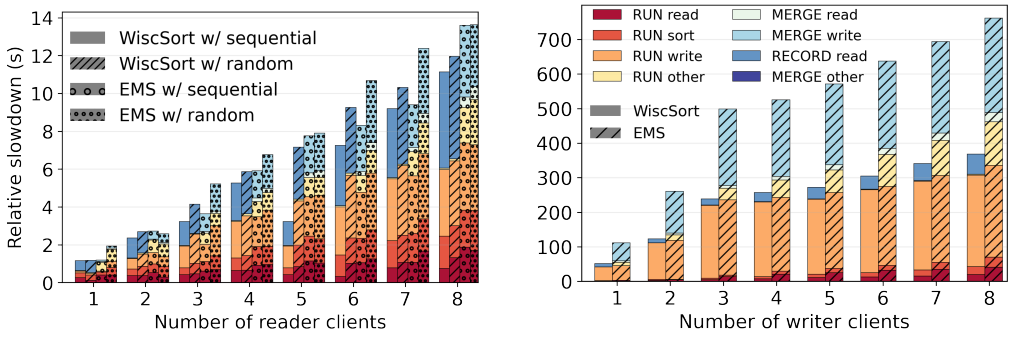
Thus far, we demonstrated the benefits of interference and concurrency constraint awareness ($I + D$). However, at the OS level, the BRAID device will be utilized by multiple processes for which we do not have any control over the requests made. Moreover, in the context of a database, it may not be

desirable to delay a write from a short transaction while a long read phase for another query is underway. While efficient BRAID utilization with multiple processes is not the focus of this work, we will demonstrate the robustness of WiscSort with varying degrees of I/O interference intensity.

In Figure 3.10, we observe the slowdown of WiscSort and merge sort as they are subjected to read and write heavy background workloads with varying concurrency. Each thread/client executes a 4KiB read (Fig 3.10a) or write (Fig 3.10b) operation on a large file. None of the background clients share cores with themselves or the sorting workload. Although the size of accesses made by the background clients can impact the interference effect, we keep it constant at 4KiB to facilitate direct comparison between WiscSort and merge sort in the common case.

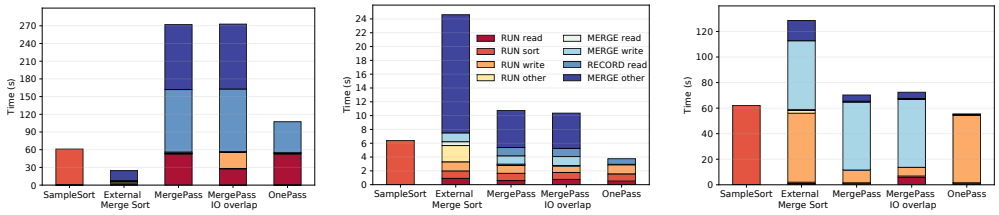
The impact of background read workloads on WiscSort and merge sort is minimal when compared to background write-heavy workloads. The primary sources of the slowdown in the case of background readers are random reads and writes (Fig 3.10a), whereas writes are the primary cause of the slowdown in the presence of background writers (Fig 3.10b). Random reader threads in the background affect sequential reads due to on-device prefetching overhead and limited device scratch memory. Overlap of random read requests can lead to significant overhead, while sequential background reads have a negligible effect. For example, WiscSort experiences a slowdown of 45% when executing 8 concurrent random reader threads, whereas merge sort's slowdown is only 25%.

The presence of background write-heavy clients poses a significant challenge for the sorting workloads due to the poor scalability of writes on PMEM, leading to a substantial impact on write times. Nonetheless, WiscSort, which is BRAID-compliant, is always twice as fast as merge sort, regardless of write intensity. We observe a slowdown of up to 14x in both WiscSort and merge sort with eight overlapping writer threads. Random reads are considerably slower than sequential reads when overlapped with



(a) Read interference overhead (b) Write interference overhead

Figure 3.10: **Interference effects of WiscSort OnePass and External Merge Sort (EMS) against multiple I/O intensive clients.** Sorting 400M records of 100B each. Each background thread performs 4KiB requests of read or writes to a different file on the device.



(a) BD-Device (b) BRD-Device (c) BARD-Device

Figure 3.11: **Future BRAID device devices through CXL emulation.** Sorting 100M records with 10B Key and 90B value each. (a) Random reads are 500ns slower than sequential reads. (b) Random read is equal to sequential read. (c) Writes 500ns slower than reads.

background writes. Overall, WiscSort outperforms merge sort for both read and write-heavy background workloads; however, a userspace IO scheduler is essential to manage multi-tenant workloads on BRAID devices.

3.5.5 Other BRAID Devices

Our previous experiments examined WiscSort's performance on Optane PMEM, which exhibits all five BRAID properties. However, emerging BAS

devices occupy different points in the BRAID model space. In the following experiments, we show the sensitivity of WiscSort and its optimizations on devices with various BRAID characteristics. We also compare in-place sample sort [44] (Sec 3.3.2.1) and external merge sort on these new devices to determine what sorting technique works best on them.

For each emulated device, we identify the closest real-world device class, describe which BRAID properties it exhibits, and explain what the sorting results imply for algorithm design on that class of hardware.

We emulate a single-socket system with a BAS device on a two-socket server by disabling all cores in one socket, while keeping its memory accessible from the other socket. This memory now mimics a CXL byte-addressable storage device. This is a well established emulation technique [105, 35] that provides close to real CXL device performance [19]. The emulation test bed comprises two 20-core Intel Xeon Gold 5218R CPU with scaling governor set to *performance* and hyperthreading disabled. The testbed constitutes 128 GB @2933 MHz DRAM split between two NUMA nodes equally. The operating system is 64-bit Linux 5.4, and the emulated BAS device is accessed through a file system interface created through `tmpfs`. Hence, the max BAS device capacity on this testbed is 64 GB. We inject delays through unoptimized `for` loops that busy-loop until the desired wait time in nanoseconds is met. Each added delay is per cache-line access (64 B).

3.5.5.1 BD-Device: Byte Addressable, Device Concurrency

This device is inspired by traditional SSDs, which have *symmetric* sequential read-write costs, but sequential reads are much faster than random reads. Thus this device does not exhibit the (A) and (R) properties of BRAID. BD-Device is an emulated byte-addressable 'disk' where random reads are 500 ns slower than sequential reads.

Real-world analogs. No shipping byte-addressable device matches this profile. Ultra-low-latency block SSDs such as the Intel Optane SSD P5800X [87] and Kioxia FL6 [95] have low random-read latency (6–29 µs) relative to conventional NAND, but they remain block-addressed (4 KiB minimum), so the **B** property does not apply. Moreover, their random-to-sequential read ratios are high: the Samsung Z-SSD achieves random reads within 92% of sequential bandwidth [135], and the Kioxia FL6 reaches 95% [95], leaving little room for the large random-sequential penalty modeled here. BD-Device therefore represents a hypothetical worst case: a byte-addressable device that inherits disk-like sequential biases. No real device has taken this path, because the technologies that enable byte addressability (DRAM, 3D XPoint, CXL protocols) inherently support efficient random access.

Sorting results. Figure 3.11a shows the performance of different sorting strategies; it highlights the pitfalls of a WiscSort-like design that relies completely on the random-read performance of a device. The sample sort performs direct in-place record movement to generate the sorted output without copying the entire input to local DRAM, hence paying a one-time cost of random access. External merge sort, as designed, performs best on BD-Device as it avoids random reads altogether. In WiscSort, MergePass and OnePass pay a huge price due to their reliance on random reads during both the RUN phase (to generate the IndexMap file) and the merge phase (to gather values from the input file).

Implication. On a byte-addressable device with poor random-read performance, WiscSort's key-value separation becomes a liability rather than an advantage. The BRAID model correctly predicts this outcome: without the **R** property, the random reads introduced by separating keys from values are not offset by the write savings. External merge sort, which relies entirely on sequential access, is the appropriate algorithm for this regime.

3.5.5.2 BRD-Device: Byte Addressable, Higher Random Read, Device Concurrency

BRD-Device is an emulated BAS with equal random read, sequential read, and write performance. Our emulation testbed does not need any modifications to emulate this device.

Real-world analogs. CXL-attached DRAM expanders are the closest real-world match and are now commercially available. ASIC-based devices such as the Samsung CMM-D 2.0 and Micron CZ120 exhibit read latencies of 200–270 ns, roughly 2–3$\times$ local DRAM [141, 142]. Because the underlying media is DRAM, random and sequential reads perform nearly identically (the gap mirrors standard DDR at roughly 20%), and read-write performance is close to symmetric, with only a 7–26% write penalty from CXL protocol processing rather than media physics [161]. These devices have no meaningful read-write interference at the media level, though tail latency can spike under high contention [107]. NVDIMM-N devices (DRAM with flash-backed persistence) also match this profile during normal operation: reads and writes both complete at DDR4 speeds with no asymmetry or interference [138]. The original text cited NVDIMM-N, larger on-device DRAM caching, and XL-FLASH [15] as motivation for this device class; CXL-DRAM expanders have since validated that prediction in production deployments [42].

Sorting results. As expected, Figure 3.11b shows that WiscSort OnePass performs the best among the sorting systems. Since BRD-Device does not exhibit concurrency constraints such as interference, sample sort can interact with the device in an uncontrolled manner. Hence sample sort performs better than external merge sort and MergePass as it avoids repeated data copies to DRAM. However, WiscSort OnePass is still faster due to smaller data movement by dealing with keys, unlike sample sort, which

moves records in place. Due to its reliance solely on sequential reads, merge sort is forced to write the record twice, making it the slowest. Due to the lack of the (I) property, we observe that MergePass without Interference-Aware Scheduling (I/O overlap) performs similarly to MergePass with Interference-Aware Scheduling. BRD-Device performance results suggest that the improved random-read bandwidth alone is enough to warrant redesigning the sorting system.

Implication. On CXL-attached DRAM, WiscSort's core advantage persists: key-value separation (**B**) reduces total data movement, and this benefit is independent of the device's read-write symmetry. However, interference-aware scheduling provides no benefit, and the thread-pool controller matters primarily because the PCIe Gen5 link imposes a bandwidth ceiling (25–30 GB/s per x8 device [141]) that can be saturated. Even modest improvements in random-read bandwidth over traditional storage are sufficient to warrant redesigning the sorting system around key-value separation.

3.5.5.3 BARD-Device: Byte Addressable, Asymmetric Read-Write, Higher Random Read, Device Concurrency

Flash-backed CXL devices exhibit read-write asymmetry that requires systems to adapt accordingly. To mimic a device with such a property, we emulate BARD-Device to have writes 500 ns slower than reads (A). However, its sequential and random-read bandwidth is the same (R), and it is byte-addressable (B).

Real-world analogs. The Samsung CMM-H (CXL Memory Module – Hybrid) is the closest characterized device. CMM-H combines a 16 GB DRAM cache with a 1 TB NAND flash backend behind a CXL.mem interface [162]. Measured read latency is 1,100–1,475 ns (9–12× local DDR5),

maximum bandwidth peaks at 4.5 GB/s, and load bandwidth saturates at just 4 threads [162]. The read-write asymmetry is strong because cache-miss writes must propagate to NAND flash, while cache-hit reads are served from DDR4. CMM-H also exhibits interference (**I**) and concurrency constraints (**D**), making it closer to a full BRAID device than to BARD alone; our emulation captures the asymmetry dimension but not the cache-boundary effects. On CMM-H, accesses within the 16 GB DRAM cache perform at DDR4 speeds, while misses incur microsecond-scale NAND latencies, producing a bimodal latency distribution that uniform delay injection cannot represent. Samsung has announced an ASIC-based production variant (CMM-H PM) with PCIe Gen5 and CXL 2.0 [162] that may reduce the interference and concurrency penalties while preserving the underlying flash asymmetry, moving closer to the BARD profile.

Sorting results. From Figure 3.11c, we can observe that writes dominate the overall run time. Since sample sort does not suffer from (I), it performs better than WiscSort MergePass; OnePass does slightly better due to its reduced sorting time (sorting only key-pointer). As expected, the difference in performance between external merge sort and WiscSort is 2x due to the reduced writes during the run phase. The MergePass I/O overlap sees similar performance as that of MergePass no I/O overlap, indicating that there will be benefits of Interference-Aware Scheduling only in the presence of read-write interference property. As the writes become more costly, the benefits of WiscSort OnePass degrade compared to sample sort, external merge sort, and even WiscSort MergePass. Nevertheless, OnePass still achieves the lowest execution time due to reducing the total amount of data movement.

Implication. On devices with strong read-write asymmetry, WiscSort's write reduction is the dominant source of improvement. The 2× speedup over external merge sort arises almost entirely from halving write traffic during the run phase. For CMM-H and similar flash-backed CXL devices, al-

gorithms that minimize writes will see the largest gains. As the cost of writes increases further, the gap between WiscSort and merge sort will only grow.

3.6 Summary

Byte-addressable storage requires application redesign for peak performance. We introduced the BRAID model to characterize the important properties of byte-addressable storage that applications must account for. We showed that conventional access strategies, such as performing sequential reads and overlapping reads and writes, are no longer appropriate. We proposed WiscSort, a high-performance concurrent sorting system that complies with BRAID using three features: key-value separation, thread-pool control, and interference-aware scheduling. Our results show that WiscSort is 2–7$\times$ faster than competing approaches.
Several observations from this work extend beyond the immediate sorting problem.

Device-aware data organization pays for itself. The central lesson of WiscSort is that *the layout of data should reflect the properties of the device that stores it*. On Optane PMEM, separating keys from values and issuing targeted random reads rather than sequential scans of bundled records improved throughput by 2–3$\times$. This improvement did not come from a better sorting algorithm in the comparison-theoretic sense; it came from organizing data to match the device's strengths (byte addressability, high random-read bandwidth) while avoiding its weaknesses (slow writes, read-write interference). The same principle applies beyond sorting: any data-intensive application that bundles hot and cold data together because "sequential access is fast" should reconsider that assumption on byte-addressable storage.

The BRAID model decomposes device complexity into actionable axes. Our evaluation on emulated devices (§3.5.5) showed that different BRAID properties call for different optimizations. Key-value separation (**B**, **A**) helps whenever byte-addressable access and write avoidance are beneficial. Interference-aware scheduling (**I**) helps only on devices where reads and writes contend for shared internal resources. Thread-pool control (**D**) helps whenever per-device bandwidth saturates below the number of available cores. By decomposing a device's behavior into these five axes, a designer can predict which WiscSort components transfer to a new device and which become unnecessary. CXL-attached DRAM, for instance, benefits from key-value separation and thread-pool control but not from interference-aware scheduling, a prediction our BRD-Device results confirm (§3.5.5.2).

BRAID properties persist across device generations, though their intensity shifts. Intel discontinued Optane PMEM in 2022, but the device properties that motivated BRAID have not disappeared. CXL-attached DRAM expanders (Samsung CMM-D, Micron CZ120) are now deployed in production at Microsoft Azure [42] and exhibit moderate forms of **B**, **I**, and **D**. Flash-backed CXL devices such as the Samsung CMM-H exhibit all five properties, with asymmetry arising from the NAND backend rather than 3D XPoint media physics [162]. CXL 3.0 pooled memory adds switch-induced latency [106] that further amplifies the **D** property. As the memory hierarchy grows deeper, the calculus of data placement only becomes more consequential: devices with varying latency, bandwidth, and concurrency profiles coexist, and algorithms that treat storage as a uniform sequential medium will leave performance on the table.

Conclusion. WiscSort addresses the *static* regime of the thesis: when the application's access semantics reveal a stable structure (keys are consistently hotter than values during sorting), we can organize data at allocation time to match device characteristics, without runtime reorganization. The

BRAID model formalizes the device side of this equation, while key-value separation is the layout technique that bridges application semantics and device properties.

The remaining chapters address the *dynamic* regime. In many workloads, access patterns are not known in advance and change over time. Chapter 4 quantifies this problem by measuring hotness fragmentation in production workloads, showing that allocators intermingle hot and cold objects on the same pages. Chapter 5 introduces OBASE, which reorganizes the address space at runtime so that page-granularity placement decisions remain effective as access patterns evolve. Where WiscSort separates keys from values because the programmer knows which fields are hot, OBASE separates hot objects from cold objects because the runtime has learned which allocations are hot. Both techniques share a common insight that motivates this dissertation: *efficient use of modern memory hierarchies requires organizing data at a finer granularity than the page*.

Chapter 4

Case for Object Reorganization

WiscSort demonstrated that data layout can be tailored to device characteristics when the application's access structure is known at design time: keys are consistently hotter than values during sorting, so separating them at allocation time suffices. Many workloads do not enjoy this luxury. In key-value stores, caches, and databases, which objects are hot depends on user traffic and application logic that change over time. Allocators place objects by size class and allocation order, without regard for future access patterns, and the OS tracks activity only at page granularity. The result is *hotness fragmentation*: hot and cold objects intermingle on the same pages, preventing the OS from managing memory effectively.

This chapter quantifies the severity and consequences of hotness fragmentation and establishes the need for dynamic reorganization. We begin by introducing *page utilization*, a metric that captures the fraction of bytes within touched pages that are actually accessed (§4.1). Using this metric, we measure fragmentation in open-source key-value stores (§4.1.1) and in six publically available workload traces from Google (§4.1.2). We then show that object hotness changes continuously over time, ruling out allocation-time solutions (§4.2). Next, we trace the consequences of fragmentation through three pitfalls: trapped memory, translation overhead, and infrastructure cost (§4.3). We discuss the challenge of object mobility in unmanaged languages (§4.4) and articulate the principles of address-space

engineering that guide the design of *OBASE* (§4.5).

4.1 Page Utilization: A Metric

Existing memory management systems observe activity at page granularity: a page is either active or inactive. This binary view cannot distinguish a page where every byte is hot from one where a single cache line is hot and the rest is cold. To reason about the gap between the OS view and the true access pattern, we need a metric that captures how much of each active page is genuinely in use. We define *page utilization* for this purpose.

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.

4.1.1 Fragmentation in Open-Source Workloads

To quantify hotness fragmentation in widely deployed systems, we instrument three popular key-value stores: Redis 6.2 (in-memory hash table), Memcached 1.6 (slab-based caching), and MongoDB 5.0 (document store with WiredTiger backend). Each system runs on a single-socket server with an Intel Xeon Gold 6248 processor (20 cores, 2.5 GHz), 192 GiB DDR4 DRAM, and linux 6.11. We load each store with 10 million 1 KB key-value pairs and drive it with a YCSB-C (read-only) workload using a Zipfian request distribution ($\theta = 0.99$). We use Intel PinTool [26] to intercept ev-

ery memory load and store instruction during a 360-second measurement window, recording the virtual address and size of each access. From these traces, we compute per-page utilization by counting the number of unique bytes accessed within each 4 KB page.

Figure 4.1 shows the cumulative distribution of per-page utilization for all three systems. The results reveal severe fragmentation: 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%. This scattered placement creates an illusion of high memory activity when only a small fraction of bytes receives regular access [46].

4.1.2 Fragmentation in Production Workloads

Open-source benchmarks under synthetic workloads provide a controlled setting, but production services at scale may exhibit different allocation patterns, object size distributions, and access behaviors. To understand hotness fragmentation across a broader range of applications, we analyze memory access traces from six production workloads at Google [28], collected using DynamoRIO's drmemtrace framework [27].

The six workloads, identified by NATO phonetic codenames, span the major categories of warehouse-scale computing: web search and indexing, storage and database infrastructure, large-scale data analytics, and distributed serving. Together they represent a cross-section of Google's fleet, covering services with varied object sizes, allocation rates, and concurrency patterns. The traces record every user-mode memory load and store at cache-line granularity. We process each unique access, annotate pages as 4 KB or 2 MB using `/proc/self/smaps`, and count unique 64 B cache lines touched per page. Based on instruction counts, core counts, and typical warehouse-scale CPU characteristics [93], we estimate the traces capture up to ~30 s of steady-state execution.

Note that publicly released traces may not fully reflect typical produc-

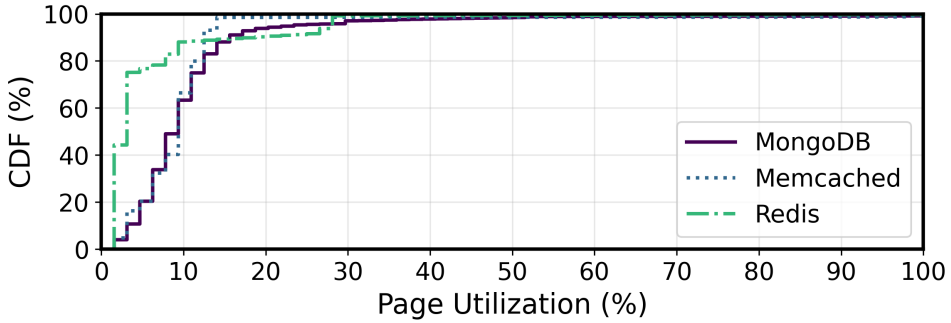


Figure 4.1: **Low Per-Page Utilization in Open-Source Workloads.** Redis, Memcached, and MongoDB Page Utilization for 360s epochs running a YCSB-C (read-only) workload with a Zipfian distribution. A small fraction of bytes per page are accessed, and more pages are touched than necessary.

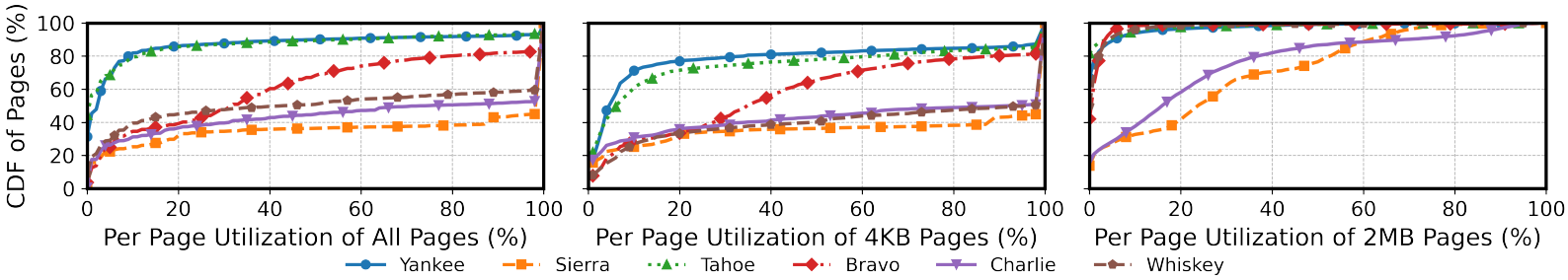


Figure 4.2: **Low Per-Page Utilization in Google Production Workloads.** CDFs of page utilization for six widely deployed applications, shown for all pages combined (left), separately for 4KB pages (center), and 2MB pages (right). All applications exhibit low page utilization, indicating substantial hotness fragmentation.

tion conditions (e.g., they may capture load tests or workloads selected for ease of instrumentation). The workloads may also be instrumented in idealized environments. However, the traces are the best available window into real-world access patterns at scale. Google's traces remain the only publically available memory access traces from production services.

Figure 4.2 shows the cumulative distribution of per-page utilization for all workloads, ordered as: (a) aggregated across both page sizes, (b) 4KB pages only, and (c) 2MB pages only.

Aggregated view. As shown in the first graph, aggregated for 4KB and 2MB pages, all workloads have low per-page utilization. Median utilization ranges from 8% for Tahoe to approximately 50% for Sierra. For most workloads, half of all touched pages waste over 70% of their capacity on cold data.

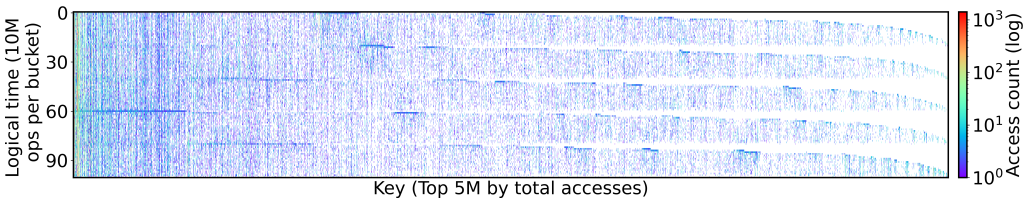
4KB pages. Fragmentation is visible even at the smallest page size. Even with 4KB pages, for Tahoe and Yankee, ~80% of pages have less than 20% utilization; for Bravo, 60% of pages have below 40% utilization; and Charlie, Whiskey, and Sierra have 35–40% of pages below 40% utilization. Thus, hotness fragmentation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

2MB pages. Fragmentation worsens dramatically for 2MB huge pages. Tahoe, Bravo, and Yankee all exhibit extreme fragmentation: 85–90% of their huge pages utilize less than 10% of their 2MB capacity; even Sierra, the best-performing workload, has 40% of its huge pages below 20% utilization. The vast majority of huge pages (each consuming 512× the capacity of a base page) holds over 1.8MB of cold data alongside a few accessed cache lines.

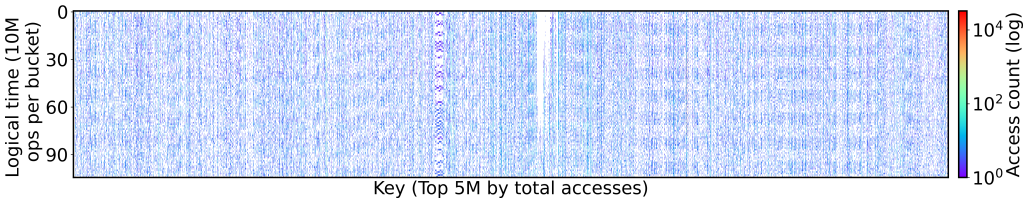
Finding 1: *Active pages are mostly cold: 70–90% of bytes in pages the OS considers hot receive no accesses.*

4.2 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. This strategy



(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 4.3: **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.

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 should remain hot throughout their lifetimes.

We show that these assumptions do not hold. We analyze object-level

traces from Meta [51] and Twitter [160], which record operations in logical time (10M operations per bucket). Figure 4.3 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 remain consistently hot, but the majority exhibit brief access bursts separated by long idle gaps, even among the top 5 million keys.

To quantify this churn, we examine Meta KV production traces [51] and measure, for every key, the spread between its 25th- and 75th-percentile operation reuse distances (i.e., the number of operations between accesses to the same key). Figure 4.4 groups keys by value size and reports the fraction whose reuse-distance spread exceeds 5×, 10×, or 30×. A high spread indicates that a key alternates between short gaps (hot) and long gaps (cold). For mid-sized objects (64B–4KB), which represent 94% of Meta keys and 98.2% of Twitter keys, 75% of keys have a reuse spread exceeding 5×, and approximately 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 hotness change is required.*

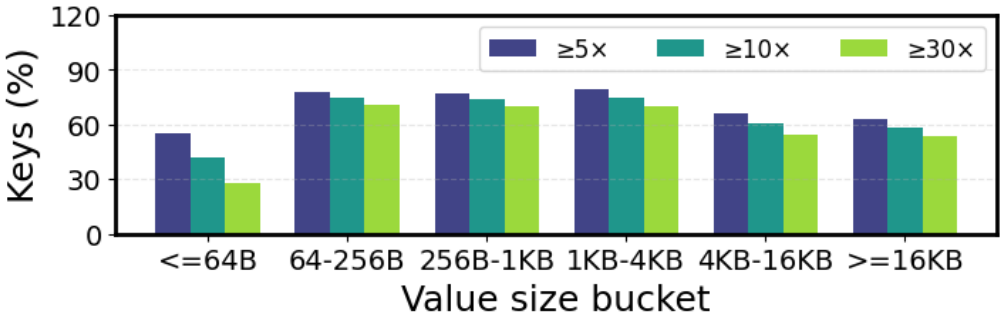


Figure 4.4: **Intra-key reuse-distance spread in Meta KV traces.** Bars show the fraction of keys in each value-size bucket whose reuse-distance interquartile range (p75/p25) exceeds 5×, 10×, or 30×. A larger spread indicates frequent transitions between hot and cold phases for the same key.

4.3 Pitfalls of Hotness Fragmentation

Hotness fragmentation—the intermingling of hot and cold objects within the same pages—creates a cascade of inefficiencies that span from individual CPU cycles to datacenter-scale resource provisioning. We identify three major pitfalls: trapped memory that cannot be reclaimed despite being cold, translation overhead from scattered hot objects inflating the TLB working set, and infrastructure costs from memory overprovisioning. Each pitfall compounds the others, making hotness fragmentation a systemic problem that demands architectural solutions.

4.3.1 Trapped Memory

Operating systems manage memory in page-sized units. To move a page to a slower tier—compressed memory, SSD, or remote DRAM—the OS requires confidence that the entire page is cold. A single hot cache line anywhere within a 4KB or 2MB page forces the OS to keep the entire page resident, trapping all the cold data alongside it. Low page utilization therefore translates directly into reduced reclamation efficiency.

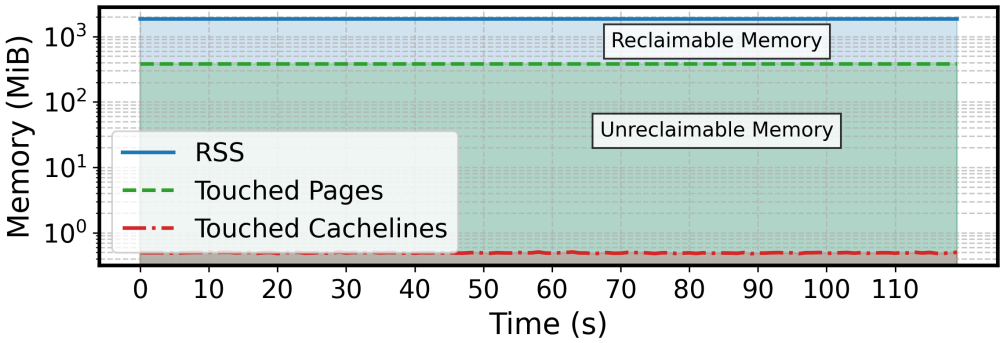


Figure 4.5: **Unreclaimable Memory in Redis.** YCSB-C with Zipfian distribution running on Redis shows memory used (RSS), pages needed (Touched Pages), and cachelines needed (Touched Cachelines). Only 0.5 MiB of actual data is required whereas 1.2 GiB remains resident. The gap represents theoretically reclaimable memory that current systems cannot efficiently recover.

Figure 4.5 illustrates this problem concretely with Redis running a YCSB-C workload under a Zipfian distribution. Despite requiring 1.2 GiB of resident memory (RSS), 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. The gap between RSS and actual working set represents memory that is cold but unreclaimable—trapped by the few hot objects scattered across pages.

To understand trapped memory at scale, we analyze the six Google production workloads introduced in §4.1.2. Figure 4.6 breaks down each workload's Resident Set Size by page type (4KB or 2MB) and access status (accessed or unaccessed). Accessed portions (hatched) represent pages touched at least once during the measurement epoch; these pages are unreclaimable by standard OS mechanisms regardless of their internal utilization. Unaccessed portions (solid) represent pages within RSS that received no accesses and could potentially be reclaimed.

For Tahoe, Yankee, and Bravo, 2MB huge pages dominate the RSS, with

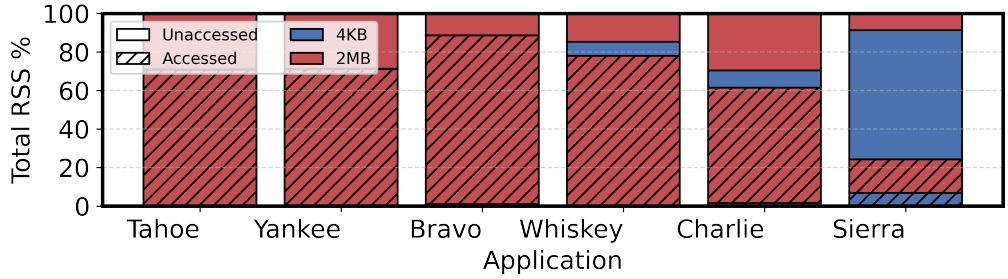


Figure 4.6: **RSS Composition in Google Workloads.** Breakdown of total Resident Set Size for each application. Hatched areas indicate memory within pages that were accessed during the epoch (split by 4KB and 2MB); solid areas represent memory within pages that were not accessed at all.

70–90% of memory residing in accessed huge pages. Whiskey shows a mixed footprint with both 4KB and 2MB pages, but over 80% of its RSS consists of accessed pages. Charlie has approximately 70% accessed memory split between page sizes. Sierra is an outlier: roughly 60% of its RSS consists of entirely unaccessed pages, making it more amenable to conventional page-based reclamation. However, even Sierra suffers from hotness fragmentation within its accessed pages.

To isolate memory trapped specifically by hotness fragmentation, we compute the *unreclaimable fraction*: the ratio of cold bytes within touched pages to the total capacity of those pages. Figure 4.7 presents this metric, with stacked bars showing contributions from 4KB pages (blue) and 2MB pages (red). Because both components share the same denominator (total touched-page capacity), they sum to the labeled percentages.

Tahoe, Yankee, Whiskey, and Bravo all exceed 96% unreclaimable memory within their touched pages. In concrete terms, for every 100MB of pages the OS considers *active*, over 96MB is actually cold data trapped by a few hot cache lines scattered across those pages. Charlie shows 74.3% unreclaimable, while Sierra—despite its large pool of entirely unaccessed pages—still wastes 54.9% of its touched-page capacity. The dominance of the red (2MB) component in Figure 4.7 confirms that huge pages, while

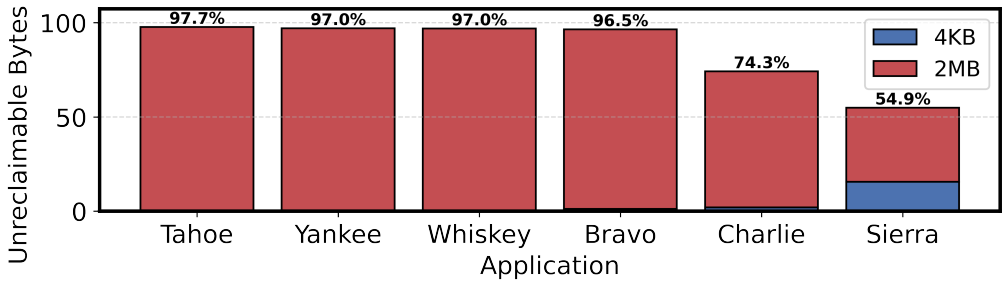


Figure 4.7: **Unreclaimable Memory Due to Hotness Fragmentation.** Percentage of total touched-page capacity that contains cold (unaccessed) data, shown as stacked contributions from 4KB and 2MB pages. The labeled percentages represent the total unreclaimable fraction. For Tahoe, Yankee, Whiskey, and Bravo, over 96% of memory within touched pages cannot be reclaimed despite being cold.

beneficial for TLB coverage, exacerbate hotness fragmentation.

Observation 1

Segregating hot and cold objects within the address space can unlock the 55–98% of touched-page capacity currently trapped by hotness fragmentation, enabling the OS to reclaim or tier pages more effectively.

4.3.2 Translation Overhead

Hotness fragmentation wastes CPU cycles through increased address translation overhead. When hot objects scatter across many pages, the application's working set—from the MMU's perspective—becomes unnecessarily large. This inflated page working set increases pressure on the Translation Lookaside Buffer (TLB). Each TLB miss triggers a page table walk costing 150–600 cycles, compared to roughly 4 cycles for a TLB hit [165]. Google reports that data TLB misses consume up to 11% of fleet-wide CPU cycles; more broadly, 20% of cycles see at least one active page walk in the memory pipeline [165, 85]. Hotness fragmentation exacerbates this overhead by keeping pages in the translation working set solely because they contain

a few active cache lines.

Page Table Architecture and Virtual Address Distance. Modern processors employ hierarchical approaches to page table caching that make virtual address locality critical for translation performance. Intel processors use individual caches for each page table entry (PTE) level, while AMD uses a unified cache across levels [48]. Applications with dense virtual address space usage benefit more from the PTE L2 cache's partial translations than from PTE L3 or PTE L4 caches, which are typically smaller at higher levels. Consequently, larger virtual address distances between accesses can result in page walks taking up to six times longer than walks for nearby addresses.

Beyond the TLB itself, virtual address locality affects multiple microarchitectural optimizations. Minimizing virtual address distance enhances both stride and next-line prefetching [24], as well as next-PAGE prefetching (pre-loading TLB entries for sequential access patterns). In essence, decreasing virtual address distances between hot objects can significantly reduce page walk times and increase TLB hit rates, even when consecutive accesses span different page boundaries.

Theoretical Analysis. To quantify the potential impact of object placement on translation overhead, consider the following scenario: 256 GiB of data containing 256-byte objects (1.07 billion objects total), with a skewed access pattern touching only 1% of the data (10.73 million hot objects) and a page size of 4 KiB. We examine two extreme placement strategies:

1. **Lower bound (clustered):** All hot objects packed together, requiring only 671K pages to cover the hot working set.

2. **Upper bound (scattered):** No two hot objects share a page, requiring 21.46 million unique pages, with each hot object potentially spanning two pages.

For sequential accesses to objects placed in temporal order, the clustered case achieves approximately 10 million TLB hits, while the scattered case achieves zero TLB hits. Using conservative estimates for page walk latency (150 cycles for L1D hit, 600 cycles for memory access) and TLB lookup latency (4 cycles), we compute the total translation cycles as:

$$\text{Total cycles} = (\text{Accesses} \times 4) + 600 + (\text{Pages Accessed} \times 150)$$

This yields 143 million cycles for the clustered case versus 3.2 billion cycles for the scattered case. On a 2.8 GHz processor (Intel Cascade Lake), the clustered case completes in 0.051 seconds while the scattered case requires 1.116 seconds—a **23× difference** attributable solely to object placement.

Empirical Validation. To validate these theoretical estimates, we implemented a microbenchmark that isolates the effect of object clustering on TLB and cache behavior. We allocate N objects adjacently in a large memory region. In the *scattered* configuration, we randomly select X% of objects across the entire region as the hot set. In the *clustered* configuration, we select X% of contiguous objects as the hot set. Once a configuration is chosen, objects within the hot set are accessed in random order.

Figure 4.8 shows the percentage decrease in TLB and L3 miss rates when comparing clustered versus scattered placement, both using 4 KiB pages. Accessing clustered objects achieves up to a 2× improvement in TLB efficiency. This improvement decreases as the hot set size increases due to higher TLB evictions caused by random object selection within the larger hot set. The vertical lines indicate the TLB coverage size (red) and L3 cache size (blue); benefits are most pronounced when the clustered hot set fits within these hardware structures.

If hot objects can be identified and clustered, their address range becomes an ideal candidate for huge page promotion. Figure 4.9 compares the

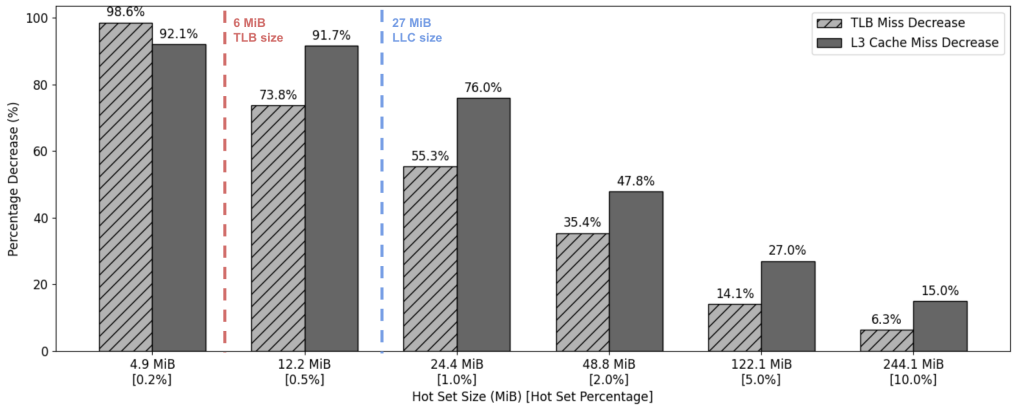


Figure 4.8: **TLB and Cache Benefits of Clustering (4KB Pages).** Percentage decrease of TLB and L3 miss rate between clustered and scattered hot sets. Both configurations use 4 KiB pages. The red line indicates the TLB coverage size and the blue line indicates the L3 cache size.

percentage decrease in TLB and L3 miss rates between clustered objects on 2 MiB huge pages versus scattered objects on 4 KiB base pages. The larger TLB coverage results in dramatically decreased miss rates, even with larger hot sets. This demonstrates that the combination of clustering and targeted huge page promotion can recover substantial CPU efficiency lost to hotness fragmentation.

The Transparent Huge Page Paradox. Transparent Huge Pages (THPs) are commonly deployed to mitigate TLB pressure by increasing the memory range covered by each TLB entry. However, applying THPs to address spaces suffering from hotness fragmentation creates a paradox. Consider Tahoe from our production analysis: 97.7% of its touched huge-page capacity is cold, yet each 2MB page remains pinned in fast memory and occupies a TLB entry. The THP provides no benefit—it covers mostly cold data—while consuming 512× more physical memory than necessary. Prior work reports that indiscriminate THP usage can inflate memory footprints by up to 69% in server workloads [98].

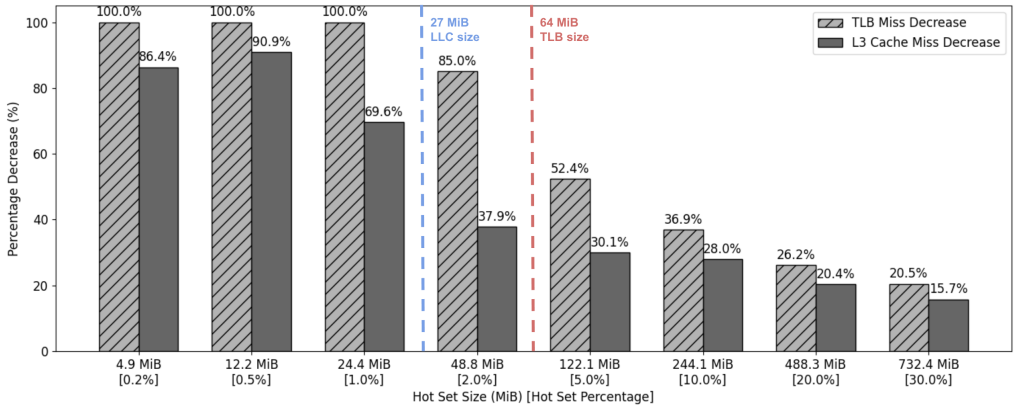


Figure 4.9: **TLB and Cache Benefits of Clustering with Huge Pages.** Percentage decrease of TLB and L3 miss rate between clustered hot sets on 2 MiB huge pages and scattered hot sets on 4 KiB base pages. The red line indicates the TLB coverage size and the blue line indicates the L3 cache size.

Improving page utilization resolves this tension. By packing hot objects densely, applications can concentrate their active data into fewer pages. These dense regions become ideal candidates for huge page promotion: a 2MB page filled with hot objects genuinely benefits from the expanded TLB coverage. Meanwhile, regions dominated by cold data can use 4KB pages, preserving fine-grained reclaimability.

Observation 2

Improving page utilization enables targeted huge page promotion, reducing TLB misses without the memory bloat caused by applying huge pages to sparsely-utilized regions.

4.3.3 Infrastructure Cost

The inflated memory footprints caused by poor page utilization contribute to datacenter-level inefficiencies that extend far beyond individual application performance. Because the OS perceives a larger active memory set than is truly necessary, applications reserve more memory than their

actual working set demands. This prevents dense packing of workloads onto servers, forcing operators to spread jobs with small working sets but large allocation footprints across multiple machines to avoid CPU stranding [144, 105]. Even when actual memory needs are much smaller than allocated, the apparent memory pressure prevents co-locating additional workloads on the same host, leaving CPU resources idle.

When address-space engineering consolidates hot data onto fewer pages, the remaining cold pages become safely reclaimable (via swap, CXL tiering, or compression) without risking performance. The resulting reduction in resident set size enables denser workload packing: jobs whose true (granular) working sets fit in fast memory can share machines that their inflated RSS would otherwise preclude.

The financial implications are substantial. DRAM accounts for up to 50% of server costs [106], and unlike flash storage, its cost per bit is not decreasing rapidly [117]. Memory overprovisioning due to hotness fragmentation directly translates to capital expenditure on DRAM that provides no useful work.

The environmental implications are equally concerning. DRAM carries a higher carbon footprint per bit than SSDs, producing approximately 12$\times$ more emissions [117]. Manufacturing and operating unnecessary DRAM capacity contributes to datacenter carbon emissions that could be avoided through more efficient memory utilization.

By reorganizing virtual address spaces to reflect true memory requirements, operators can achieve multiple benefits simultaneously: accurate memory provisioning based on actual working sets, efficient huge page deployment in hot regions, and fine-grained reclaimability in cold regions. This enables stable, skewed workloads to run on fewer machines, reducing both capital expenditure and environmental impact.

Observation 3

Improved address-space layout enables more accurate memory provisioning, reducing infrastructure costs, resource stranding, and environmental impact.

4.4 Object Mobility in Unmanaged Languages

The analysis in the preceding sections establishes that hotness fragmentation is pervasive, dynamic, and costly. A natural response is to reorganize the virtual address space so that hot objects reside together on hot pages and cold objects reside together on cold pages. However, the feasibility of such reorganization depends critically on the language runtime.

Managed runtimes, such as the JVM and Go, can relocate objects during garbage collection by updating all references atomically. The runtime maintains complete knowledge of the object graph and can rewrite pointers as part of its normal operation, making object mobility relatively straightforward. In unmanaged languages like C++, however, programs assume that an object's address remains stable after allocation [53]. Raw pointers may be stored in registers, on the stack, in other heap objects, or even serialized to disk; the runtime has no mechanism to discover and update them. This assumption renders dynamic object migration significantly more challenging.

The challenge, therefore, is not merely to reorganize memory layout dynamically, but to do so while preserving pointer semantics. A moved object must remain accessible through any pointer that previously referenced it. Ideally this should require minimal source-level changes without imposing application logic changes.

We focus on unmanaged languages precisely because of this difficulty. By demonstrating that address-space engineering is feasible even when object addresses are expected to be stable, we establish the broad applicability of the approach. Techniques proven in C++ generalize naturally to

managed runtimes, where the runtime already provides mechanisms for pointer updates.

Pointer-Based Data Structures. To make the problem tractable in an unmanaged context, we focus on **pointer-based data structures**, where elements are accessed indirectly through pointers rather than by direct offset calculation. Hash tables, linked lists, trees, and skip lists fall into this category. These structures are ubiquitous in memory-intensive applications; Table 5.2 surveys their prevalence in widely deployed systems.

By instrumenting pointer dereferences, a system can intercept accesses and, when an object has moved, transparently redirect the access to the object's new location. This interception must operate safely in highly concurrent environments. Coarse-grained locks or stop-the-world pauses would negate the performance benefits of improved memory layout; a practical system must support concurrent access and migration with minimal synchronization overhead.

4.5 Principles of Address-Space Engineering

The pervasive hotness fragmentation documented in this chapter stems from a semantic gap between application-level object management and OS-level page management. Allocators optimize for speed and spatial fragmentation, making placement decisions at allocation time without knowledge of future access patterns. The OS, meanwhile, tracks activity at page granularity using access bits in page table entries. This mismatch scatters frequently accessed objects across memory, intermixing them with rarely accessed data.

To bridge this gap, we advocate for **address-space engineering**: dynamically reorganizing the application's virtual address space to align object placement with observed access patterns. Figure 4.10 illustrates the con-

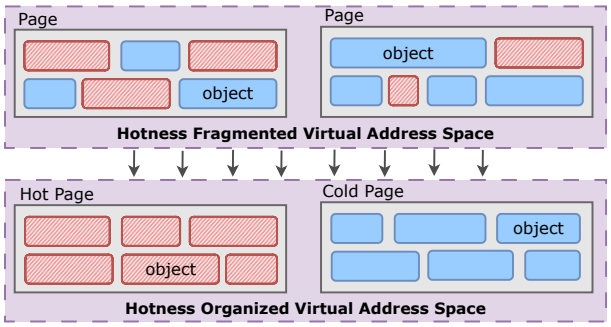


Figure 4.10: **Address-Space Engineering.** Hotness fragmentation (top) intermixes hot (red) and cold (blue) objects, making pages difficult to reclaim. Address-space engineering (bottom) reorganizes objects into uniformly hot and cold regions, enabling efficient page-level management.

cept. Under conventional allocation (top), hot and cold objects intermingle within pages, making the entire page appear active to the OS even when most bytes are cold. With address-space engineering (bottom), objects are grouped by access intensity, creating uniformly hot and uniformly cold regions that the OS can manage effectively.

This approach rests on three core principles: enabling object mobility in unmanaged languages, decoupling address-space organization from page reclamation, and dynamically grouping objects by observed access intensity.

4.5.1 Enabling Object Mobility

The fundamental requirement for address-space engineering is the ability to move objects after they have been allocated. As discussed in §4.4, this is straightforward in managed runtimes but challenging in C++ where addresses are assumed to be stable.

For pointer-based data structures, mobility can be achieved by instrumenting the indirection mechanism. When an access occurs through a pointer, the system checks whether the target object has moved and, if so,

redirects the access to the new location. Two properties are essential for this mechanism to be practical:

- **Transparency:** Applications should require minimal modification to benefit from reorganization. The mobility mechanism must be compatible with existing code and preserve the semantics of pointer operations.
- **Concurrency:** In multi-threaded applications, accesses and migrations may occur simultaneously. The system must handle concurrent operations without serializing access or introducing data races.

4.5.2 Decoupling Layout from Reclamation

Significant engineering effort over the past decade has produced sophisticated page-based memory tiering infrastructure. Kernel mechanisms like kswapd and zswap [99] handle local reclamation. Datacenter-scale systems like TMO [152] orchestrate fleet-wide memory offloading. Emerging CXL memory pooling [106, 140] promises hardware-assisted tiering with systems like Memstrata [164] managing placement across memory tiers. Research prototypes such as TPP [116], Memtis [101], and Hemem [131] explore sophisticated page placement policies.

Rather than replace this infrastructure, we **decouple object-level address-space organization from page-level reclamation**. This separation suggests a frontend/backend architecture, illustrated in Figure 4.11.

- **Frontend** Responsible for engineering the virtual address space. It monitors object accesses, identifies hot and cold objects, and migrates objects so that each page contains data of uniform temperature. The frontend operates at object granularity and understands application-level access patterns.

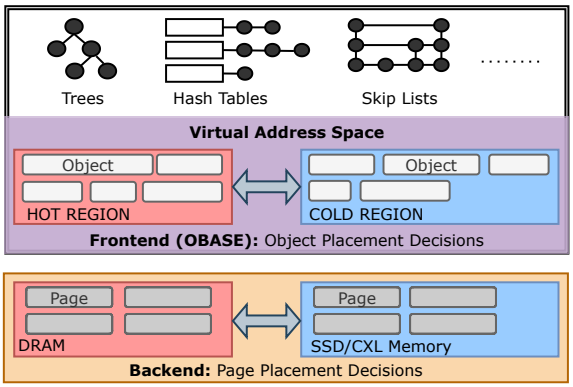


Figure 4.11: ***OBASE* Overview.** As a frontend, *OBASE* makes object placement decisions to organize the virtual address space into hot and cold regions. This enables any backend to make more effective page placement decisions between DRAM and tiered storage.

- **Backend** Consists of existing OS-level reclamation mechanisms or specialized tiering agents that operate on pages. The backend decides which pages to reclaim, when to reclaim them, and where to place them in the memory hierarchy. It operates at page granularity using standard access-bit tracking.

Crucially, the frontend does not decide which pages to reclaim or when; it simply ensures that when the backend examines a cold region, that region contains *only* cold data. Similarly, the backend need not understand object boundaries or application semantics; it operates on pages that have been prepared by the frontend.

This decoupling yields two benefits. First, it enables **independent innovation**: improvements to object-management techniques (frontend) and advances in memory-tiering policies or hardware (backend) can proceed separately without tight coupling. A better frontend benefits any backend; a better backend benefits any frontend. Second, it provides the backend with a **higher-quality signal**. When cold regions are uniformly cold, backend

policies can reclaim memory more aggressively without risking performance degradation from unexpectedly paging in hot data.

4.5.3 Grouping Objects by Access Intensity

Given object mobility and the decoupled architecture, a policy must guide placement decisions. One approach is to segregate hot and cold objects at allocation time using static hints or profiling [60, 67, 33]. This strategy assumes that hotness is both identifiable at allocation time and relatively stable afterward.

As demonstrated in §4.2, neither assumption holds in practice. The same allocation site produces objects with vastly different access patterns: a single `SET` handler in Redis allocates memory for session tokens accessed every millisecond and user profiles never touched again. Static analysis cannot distinguish these cases. Moreover, even among heavily accessed objects, hotness is transient: keys alternate between active bursts and extended dormancy, with reuse distances varying by more than an order of magnitude.

Static allocation-time decisions therefore degrade as access patterns evolve, gradually recreating the fragmentation they were designed to prevent. Instead, the system must **dynamically group objects based on their observed access intensity**. This requires lightweight runtime monitoring to distinguish hot objects from cold ones. Grouping by observed temperature creates regions with uniform access characteristics:

- **Hot regions** contain densely-packed, frequently-accessed objects. These regions become candidates for huge page promotion, improving TLB efficiency without wasting memory on cold data.
- **Cold regions** contain objects that have not been accessed recently. These regions become safe targets for backend reclamation policies, as demoting or swapping them will not cause performance degradation.

The access-tracking mechanism must impose minimal overhead to remain viable in latency-sensitive production environments. Heavy instrumentation that doubles request latency would negate any benefits from improved memory layout. §5.1 describes how *OBASE* achieves lightweight tracking through careful integration with the data structure's access path.

4.6 Summary

This chapter has established the case for dynamic object reorganization through four key findings:

1. **Hotness fragmentation is pervasive.** Both open-source workloads (§4.1.1) and Google production applications (§4.1.2) exhibit low page utilization, with 70–90% of bytes in touched pages receiving no accesses.

2. **Hotness is dynamic.** Analysis of Meta and Twitter traces (§4.2) reveals that object access patterns shift continuously over time. Keys alternate between active bursts and extended dormancy, with reuse distances varying by more than an order of magnitude. Static allocation-time placement cannot prevent fragmentation.

3. **Fragmentation has significant costs.** Hotness fragmentation traps 55–98% of touched-page capacity in unreclaimable cold data (Observation 1), wastes CPU cycles on unnecessary TLB misses with up to 23× overhead from scattered placement (Observation 2), and forces memory overprovisioning that increases infrastructure costs and environmental impact (Observation 3).

4. **Unmanaged languages present unique challenges.** C++ programs assume stable object addresses, making dynamic reorganization difficult. However, pointer-based data structures provide an opportunity:

by instrumenting pointer dereferences, a system can transparently redirect accesses to moved objects.

Based on these findings, we have articulated three principles for address-space engineering: enabling object mobility through transparent pointer redirection, decoupling object-level layout from page-level reclamation to leverage existing tiering infrastructure, and dynamically grouping objects by observed access intensity rather than static allocation-time decisions.

The following chapter presents *OBASE*, a system that instantiates these principles for C++ applications. We describe how *OBASE* achieves transparent, concurrent object mobility within the C++ memory model, and how it organizes the address space to improve page utilization while imposing minimal runtime overhead.

Chapter 5

OBASE: Dynamic Layout Optimization

The previous chapter established that hotness fragmentation is pervasive, that it is dynamic, and that it exacts measurable penalties: trapped cold data, inflated TLB working sets, and memory overprovisioning. It also showed that allocation-time placement cannot prevent fragmentation, because the same allocation site produces objects with vastly different access lifetimes, and individual objects alternate between hot and cold phases with wide variation in reuse distance. Static decisions, no matter how sophisticated, cannot solve a problem whose inputs change continuously.

These findings point toward a runtime system that reorganizes objects after their access behavior has been observed. The goal is *address-space engineering*: restructuring the virtual address space so that objects with similar access intensity reside together, yielding uniformly hot and uniformly cold pages. Once pages are uniform, existing tiering backends can reclaim the cold ones effectively.

The potential gains are substantial. Figure 5.1 illustrates the gap for six Google production workloads [28]: only 1.7%–21.3% of bytes are accessed, yet 64%–96% of pages are touched, because a few hot objects pin entire pages in DRAM. Closing this gap requires moving from page-level observation to object-level action.

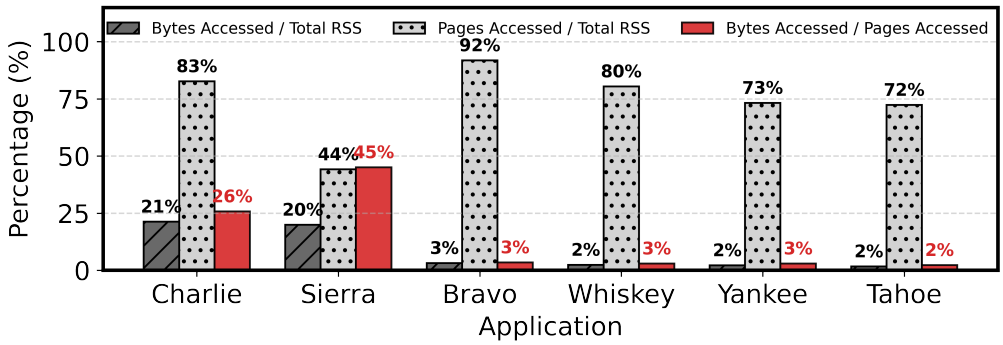


Figure 5.1: **The granularity gap in memory access of Google workloads.** Dark Grey: The percentage of total memory *bytes* actually accessed. Light Grey: The percentage of total memory *pages* accessed. Red: The page utilization (calculated as Bytes Accessed / Pages Accessed, see §4.1.2 for more details).

A key design choice is how to relate object-level layout to page-level reclamation. One approach is to build a monolithic system that both reorganizes objects and manages tier placement. We instead decouple the two concerns. The *layout* problem (frontend) organizes objects to produce high-quality page candidates for reclamation. The *reclamation* problem (backend) migrates pages across tiers according to its own policy. This separation has three practical advantages. First, it allows reuse of existing page-based tiering infrastructure, both swap-based [99, 152] and byte-addressable [116, 101, 131], without modification. Second, it ensures that layout improvements remain applicable to future, currently nonexistent memory tiers (e.g., CXL 3.0 fabrics). Third, it lowers the adoption barrier: rather than replacing their tiering backends, operators gain a better memory layout that makes those backends work more effectively.

This chapter presents ***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* [47] acts as a frontend for memory organization: it manages pointer-based data structures, profiles object access using lightweight instrumentation, and employs a lock-free protocol to

migrate objects safely, clustering hot and cold objects onto separate pages. As a result, *OBASE* prepares the memory layout for any OS backend. Crucially, *OBASE* preserves standard language semantics, requiring minimal developer intervention.

We evaluate *OBASE* with ten concurrent pointer-based data structures and six state-of-the-art reclamation and tiering backends [30, 116, 101, 146, 152]. 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 achieves equivalent savings with negligible performance impact. For tiered-memory configurations, *OBASE* achieves the same throughput with half the DRAM capacity.

The rest of this chapter is organized as follows. §5.1 presents the design of *OBASE*, including the guide abstraction for object mobility, lightweight access tracking, and our pauseless lock-free migration protocol. §5.2 details key implementation choices. §5.3 evaluates *OBASE* across data structures, backends, and production traces.

5.1 *O*bject-*B*ased *A*ddress-*S*pace *E*ngineering

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: it dynamically clusters frequently-accessed ("hot") objects into dense regions and segregating inactive ("cold") objects into separate regions, *OBASE* increasing page utilization and exposing large pools of reclaimable memory to existing tiering mechanisms.

OBASE operates in environments where the operating system manages memory in page-sized units and may migrate pages across tiers [97, 99, 140]. The design does not assume a particular tiering policy or memory hierarchy; it only assumes that backends observe per-page activity. By presenting

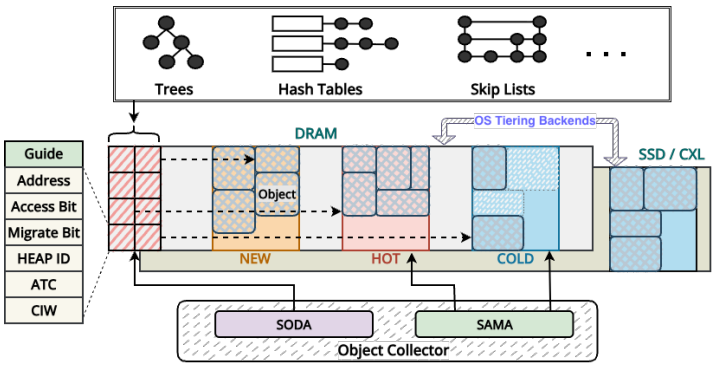


Figure 5.2: ***OBASE* Overview.** *OBASE* acts as a frontend that organizes the virtual address space into hot and cold regions. The Object Collector monitors access via Guides and SODA, migrates objects using SAMA, and presents a re-organized address space to the OS backend.

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 [152, 99, 116, 101, 131].

Achieving object-level placement with page-level backends requires overcoming four fundamental challenges:

1. **Object mobility in unmanaged languages.** C++ ties object identity to a physical location (address); relocation must be transparent and safe under arbitrary aliasing. *OBASE* addresses this with a *guide* abstraction (§5.1.2) that interposes a level of indirection on pointer dereferences, decoupling an object's logical identity from its physical location.

2. **Low-overhead access tracking.** Hotness classification must run on the common path without degrading performance. *OBASE* uses a lightweight, compiler-inserted tracking mechanism (§5.1.3) that records per-object access activity with a single test-and-set operation per dereference.

3. **Dynamic adaptation.** As §4.2 showed, hotness shifts continuously; layout must evolve accordingly. *OBASE* employs a feedback-driven controller (§5.1.4) that adjusts the cold-classification threshold in response to observed promotion rates, keeping the layout aligned with the current working set.

4. **Safe concurrent migration.** Objects must move without global pauses, even while threads hold pointers to them. *OBASE* uses a lock-free protocol based on active thread counts (§5.1.5) that relocates objects between heaps using optimistic concurrency control, avoiding stop-the-world coordination.

5.1.1 System Overview

OBASE realizes the frontend/backend separation: it continuously reorganizes object placement while leaving reclamation decisions to unmodified OS mechanisms. The following subsections describe how *OBASE* enables object mobility (§5.1.2), tracks accesses efficiently (§5.1.3), organizes the address space (§5.1.4), and migrates objects safely under concurrency (§5.1.5).

Figure 5.2 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 in the current time window. 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 ap-

plication's shifting working set while presenting page-based backends with clear hot and cold targets.

5.1.2 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 §5.1.3). 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). Guides are enforced through three compiler passes. 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.

5.1.3 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 [23, 99]. Software profilers such as DynamoRIO and LLVM's memory profiling provide fine-grained information but impose prohibitive overheads for always-on deployment [31, 27]. Hardware sampling via PEBS offers lower overhead but provides statistical coverage rather than precise per-object tracking [25].

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 [32]. 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 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.

5.1.4 Dynamically Engineering the Layout

The access signals from §5.1.3 feed into a layout policy that continuously reorganizes the virtual address space. OBASE groups objects by observed

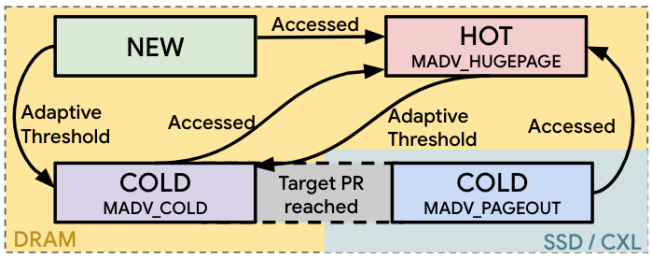


Figure 5.3: **Object Migration State Diagram.** Objects transition between heaps based on access intensity. The Object Collector promotes accessed objects to HOT and demotes inactive objects to COLD, allowing SAMA to apply differential `madvise` policies to each region if required.

temperature so that page-based backends see large, uniform regions rather than interleaved hot and cold data. Three logical 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 windows. Each heap occupies a dedicated virtual address range, so an object's address directly encodes its temperature.

Figure 5.3 shows the conceptual state machine. Objects transition between heaps as their observed access intensity changes over time: initial allocations start in NEW, recent activity promotes objects into HOT, and extended inactivity demotes objects into COLD. Objects in COLD that become active again are promoted back to HOT. This continuous reclassification tracks the application's evolving working set instead of 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 when inactivity is treated as "cold enough" for demotion, and different workloads may require different thresholds to balance reclaim opportunities against performance. *OBASE* adjusts C_t gradually based on observed access behavior, raising it when demotions are frequently reversed (indicating intermittent re-use) and lowering it when few recently demoted objects are touched (indicating deeper coldness). Adjustments are additive to avoid large swings. The goal is for the threshold to converge to a value where COLD contains deeply inactive data, while HOT tracks the active working set. The details of our adaptive strategy are described more in § 5.2.

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:

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

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

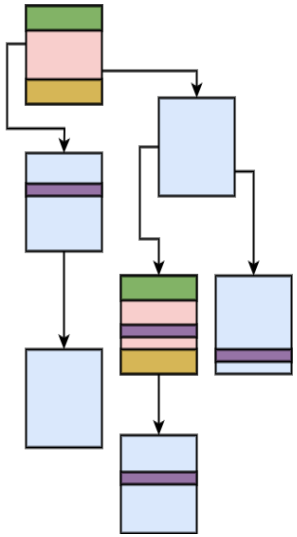
converges to a workload-specific regime. The goal is not to minimize COLD-heap accesses, but to ensure that pages classified as COLD are *safe targets* for any backend policy. § 5.2 details initialization and tuning.

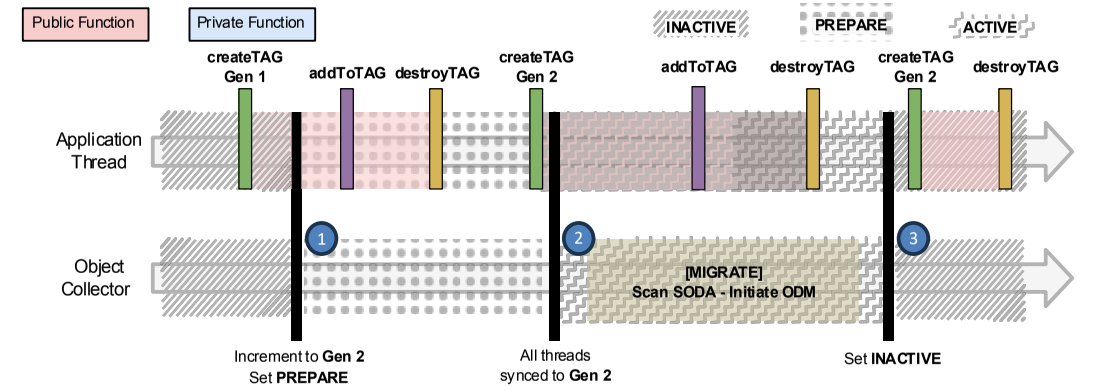
By design, *OBASE* is decoupled from any specific backend. The base system reorganizes the address space but issues no reclamation hints. This separation is intentional: the value of address-space-engineering lies in making pages uniformly hot or cold, which improves the precision of any page-based mechanism. 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.

5.1.5 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 §5.1.2, 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.





Start of Public Function	Guide Access	End of Public Function
createTAG: publicNestingLevel++ TAI[slot] = 1 threadLocalGen = GlobalGen	if (TAG_state != INACTIVE && addToTAG(Guide)): incrementATC(Guide)	destroyTAG: publicNestingLevel-- if (!publicNestingLevel): decrementAndClear(TAG) TAI[slot] = 0

Figure 5.4: **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.

Lightweight Scoping. An object can be migrated only when no thread is actively using it. Classic approaches like stop-the-world pauses or per-access load barriers are unsuitable for C++ systems code [158, 70, 118]. 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. 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, ATC is decremented. Migration is permitted only when ATC reaches zero, indicating that no thread is currently executing an operation that could read or modify the object.

ATC must be updated efficiently. Rather than incrementing 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 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 obtained them, not indefinitely. Compiler instrumentation (detailed in § 5.2.2) 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 (§5.1.3) 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. § 5.2.5 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.

5.2 Implementation

This section details *OBASE*'s concrete realization: the guide metadata encoding (§5.2.1), scope-guard data structures (§5.2.2), SODA bitmap layout (§5.2.3), controller parameters (§5.2.4), and the epoch-based migration protocol (§5.2.5).

5.2.1 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 [6], 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 sub-allocates objects within them, returning unused extents to the OS as objects migrate or are freed.

5.2.2 Efficient Scope Guard Tracking

Figure 5.4 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 call private helpers that also dereference guides; the TAG tracks all such accesses but only decrements ATC when the *outermost* public function returns. This nesting-aware design ensures that 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. 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.

5.2.3 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 [68, 99].

5.2.4 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 [36, 72, 73]. The target promotion rate of 1% is based on budgets used in production compressed-memory and CXL tiering deployments [99, 68]. C_t is bounded between 1 and 32 windows.

5.2.5 Epoch Protocol and Optimistic Migration

As shown in Figure 5.4 the OC coordinates migration through three states (`INACTIVE`, `PREPARE`, and `ACTIVE`) using a global epoch counter and a Thread Activity Index (TAI).

Epoch Transitions. In `INACTIVE`, ATC updates are disabled; dereferences only set access bits. When the OC initiates migration, it increments the epoch counter and enters `PREPARE` (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 `ACTIVE`. After migration completes, the OC returns to `INACTIVE` (3). Crucially, threads never block—they simply record epoch participation, while the OC performs convergence checks.

Optimistic Data Migration. Within `ACTIVE`, the OC migrates objects using a Optimistic Data Migration (ODM) protocol. The ODM mechanism (Algorithm 1) is 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 (first CAS). (2) Allocates space in the target heap using SAMA and copies the object. (3) Constructs a new guide (new address, heap ID, reset CIW, cleared lock) and attempts to publish it (second CAS). If either CAS fails, the migration is abandoned. Note that each guide dereference not only sets the access bit but also clears the migration lock of the corresponding guide. Through this series of steps, OC safely moves data with the assistance of ATC, SAMA, and atomic operations by following the ODM protocol.

Race Resolution. Table 5.1 illustrates a race between migration and access. Every dereference performs an atomic RMW that sets the access bit and *clears* the migration-lock bit. If a thread accesses object x while the OC is copying it (t_1), the guide changes, the commit CAS fails (t_2), and x remains at its original address. The thread always sees valid data; concurrent access vetoes migration.

5.2.6 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

Algorithm 1 Optimistic Data Migration (ODM)

```
 1: procedure MigrateObject(ptr, targetHeap)
 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)                             ▷ Mark as being migrated
 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)
17:     ClearMigrationLock(newGuide)
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)
25:         return false                                  ▷ Migration failed, guide changed
26:     end if
27: end procedure
```

efficient memory tiering. The default reclamation 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 for swap-out. This fine-grained approach creates substantial overhead when reclaiming large memory regions, as the cumulative cost of per-page TLB invalidations and IPIs degrades application performance even for threads accessing entirely unrelated hot data.

We modify `shrink_folio_list()` to batch these operations. Our optimization aggregates multiple pages—up to a full PMD spanning 512 base

Time	Actor	Action	ATC	Lock	Outcome
t_0	OC	read guide	0	0	eligible
t_0'	OC	CAS: set lock	0	1	copying begins
t_1	Thread	dereference	1	0	lock cleared
t_2	OC	CAS: commit	*mismatch*		**aborted**
If no thread intervenes between t_0' *and* t_2*:*					
t_2'	OC	CAS: commit	0	0	**success**

Table 5.1: **Race between migration and concurrent access.** A dereference at t_1 modifies the guide, causing the OC's commit CAS to fail. When no thread intervenes, both CAS operations succeed and the object moves.

pages—before issuing a single TLB flush, and groups pages for I/O submission. The modified path clears each page's PTE and marks it for pageout but defers the TLB invalidation until a complete batch is prepared, then issues one flush covering the entire range and submits all batched pages together for I/O. By amortizing the cost of TLB shootdowns and I/O submissions, this approach reduces IPIs by more than 99% when demoting 10 GiB of memory compared to the unmodified kernel.

Beyond local memory tiering, batched TLB invalidation addresses a critical scalability bottleneck in RDMA-based far-memory systems [133, 115, 114]. When evicting pages to remote memory, per-page TLB shootdowns require inter-processor interrupts to every application core; as thread counts increase, the resulting IPI storms cause shootdown latencies to grow superlinearly, throttling eviction throughput and forcing synchronous fallbacks that stall the fault-in path. By aggregating invalidations across hundreds of pages, our batching optimization amortizes these costs and enables the kernel's reclamation path to scale efficiently for both CXL and RDMA backends.

5.2.7 Compiler Passes for Guide Management

OBASE uses three complementary compiler passes (Figure 5.6) to automate the mechanical work of converting raw pointers to guides and insert-

```
void HashMap::get(char* key) {
    if (*key > 37) {
        return;
    }
    *key = 42;
    std::cout << "Value: " << *key << std::endl;
}
```

(a) Original C++ function using a raw pointer.

```
void HashMap::get(Guide<char>& key) {
    createTAG(); // Public function entry

    addToTAG(&key); // Register before dereference
    if (*key > 37) {
        destroyTAG(); // Early return
        return;
    }

    addToTAG(&key); // Register before dereference
    *key = 42;

    addToTAG(&key); // Register before dereference
    std::cout << "Value: " << *key << std::endl;

    destroyTAG(); // Public function exit
}
```

(b) After transformation: the pointer becomes a `Guide`, TAG lifecycle calls bracket the function, and each dereference is preceded by `addToTAG`. ATC increments occur inside `addToTAG` when the TAG state is not `INACTIVE`.

Figure 5.5: LLVM pass transformation: raw pointer to Guide with TAG instrumentation. The developer marks `key` for conversion; the compiler handles the rest.

ing the runtime instrumentation for TAGs and active reference counting. This automation confines developer effort to marking which pointer fields should be managed—the compiler handles the rest.

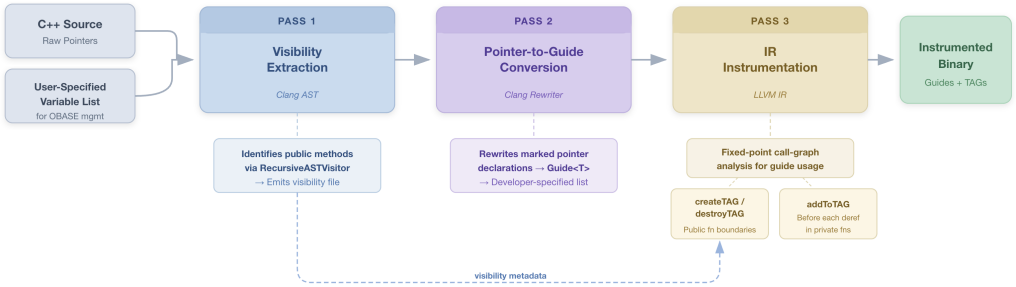


Figure 5.6: Compiler pipeline overview. Three passes progressively transform C++ source into an instrumented binary. Pass 1 extracts method visibility from the Clang AST. Pass 2 rewrites developer-marked pointers into guides. Pass 3 performs a fixed-point call-graph analysis on LLVM IR to insert `createTAG/destroyTAG` at public function boundaries and `addToTAG` before each dereference in private functions. Visibility metadata flows from Pass 1 to Pass 3 (dashed arrow).

5.2.7.1 Visibility Extraction

A Clang front-end pass inspects C++ class definitions using a `RecursiveASTVisitor` to identify public methods that serve as entry points for data-structure operations (e.g., `get`, `set`, `delete`). The pass records method visibility in a file consumed by later stages, enabling the runtime to create TAGs only at public boundaries rather than on every function call.

5.2.7.2 Pointer-to-Guide Conversion

A second Clang pass rewrites developer-marked pointer variables into guides. Developers provide a list of pointer names to convert, and the pass uses the Clang rewriter to transform declarations and usages. This targeted approach gives developers precise control over which objects are managed by *OBASE* while automating the syntactic conversion.

5.2.7.3 IR Instrumentation

The third pass operates on LLVM IR and determines where to insert TAG and ATC instrumentation. It performs a fixed-point analysis that:

1. Scans all functions to identify direct guide usage (conversions, destructor calls, operator overloads, assignments).
2. Propagates guide usage through the call graph to find indirect usage.
3. Combines visibility data with usage analysis to classify each function.

Based on this classification, the pass inserts:

- `createTAG/destroyTAG` at entry and all exit points of public functions that use guides directly or indirectly.
- `addToTAG` before each guide dereference in private functions, without creating new TAGs.

This tiered approach restricts full TAG creation to public function boundaries while still tracking every guide access throughout the call stack, minimizing runtime overhead.

Figure 5.5 illustrates the transformation on a simple function: raw pointers become guides, and the pass injects TAG lifecycle calls at function boundaries and guide registration before each dereference.

5.3 Evaluation

We evaluate *OBASE* along four dimensions:

E1: Effectiveness. Does *OBASE* reduce hotness fragmentation and memory footprint across different data structures and workloads? (§5.3.2)

E2: Backend synergy. How much does *OBASE* improve the effectiveness of existing page-based reclamation and tiering backends? (§5.3.3)

E3: Overhead and scalability. What runtime overheads do tracking and migration introduce, and how do they scale with thread count? (§5.3.4)
E4: Dynamic behavior. How does *OBASE* adapt to changing hotsets in long-running, production workloads? (§5.3.5)
E5: Multi-tenant Consolidation. Can *OBASE*'s memory savings enable practical consolidation of multiple memory-bound applications on a single machine? (§5.3.6)

5.3.1 Experimental Setup

All experiments run on an Intel Xeon Gold 5218 (32 cores) 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 maximize bandwidth. 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 [106, 116]. The Optane tier provides approximately 2.5× higher access latency than local DRAM [159], consistent with first-generation CXL memory characteristics [140]. 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 5.2), from lock-free algorithms to global locks. Many structures are borrowed from ASCYLIB [74], 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

Structure	Concurrency Control	Used In
HashTable Harris [78]	Lock-free algorithm	NGINX
HashTable Pugh [129]	Fine-grained r/w lock	Redis, Memcached
HashTable Java CHM [29]	Segmented bucket locks	Linux kernel, HAProxy
SkipList Coarse	Global lock	LevelDB/RocksDB
SkipList Fraser [71]	Lock-free algorithm	Redis Sorted Sets
SkipList Herlihy [80]	Optimistic fine-grained	Cassandra, CockroachDB
B+Tree Coarse	Global lock	SAP HANA
B+Tree OCC	OCC w/ epoch reclaim	VoltDB index
MassTree [113]	OCC + RCU	MICA, Silo
Adaptive Radix Tree [102]	Fine-grained r/w lock	DuckDB, PostgreSQL

Table 5.2: **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.

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 [51], DBbench [55]) and Twitter (Cluster 7, Cluster 23) [160]. We evaluate *OBASE* using CrestDB (§5.2), an in-memory key-value store configured with ten concurrent data structures spanning diverse synchronization mechanisms (Table 5.2). Unless otherwise noted, CrestDB runs with six server threads and six client threads communicating over Unix domain sockets.

Controller parameters. Unless otherwise noted, *OBASE* uses the controller described in §5.1.4: the Object Collector scans access metadata every 120 s, targets a promotion rate (PR) of 1%, and adjusts the cold threshold C_t by ± 1 window after each scan while keeping $1 \leqslant C_t \leqslant 32$. We choose a conservative 1% target following production compressed-memory deployments [152, 99], which has proven effective across tiering backends with

different latency characteristics. The optimal promotion rate is hardware-dependent: faster secondary tiers can tolerate higher cold-access fractions, while much slower tiers may require more conservative thresholds; exploring such hardware-aware tuning is left to future work. These defaults apply across all workloads; §5.3.5 examines the dynamics of C_t adaptation over time.

5.3.2 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 5.7(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 (§4.1.2).

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,

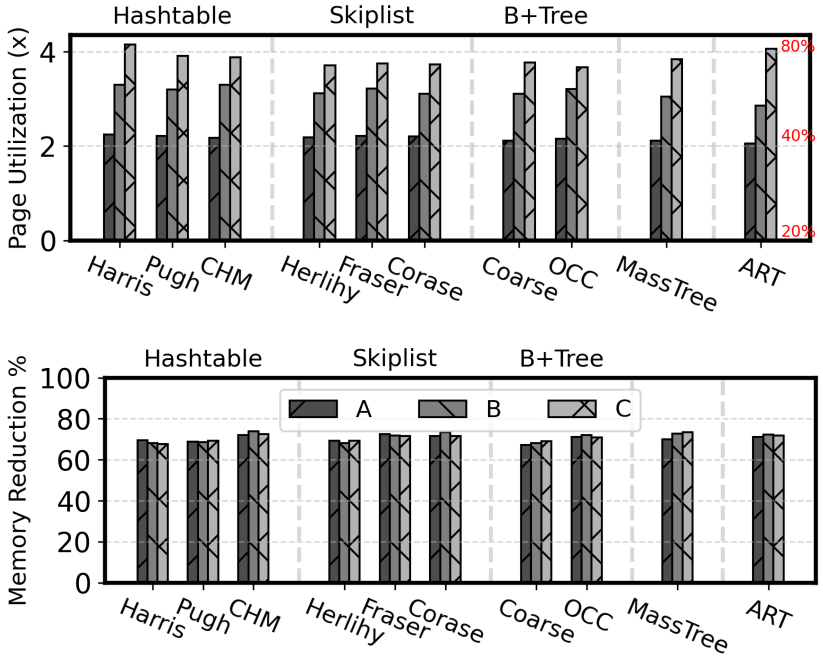


Figure 5.7: ***OBASE* effectiveness (YCSB, 10M keys).** Top: Page utilization improvement relative to the baseline allocator; *OBASE* increases utilization by 2–4× across workloads and data structures. Bottom: RSS reduction after *OBASE* pages out the COLD heap (via Kswapd). Memory footprint shrinks by 65–72%.

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 `MADV_PAGEOUT` on the COLD heap. Figure 5.7(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 §5.3.4).

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.*

5.3.3 Backend Synergy (E2)

The memory savings demonstrated in §5.3.2 are valuable only if OS 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.

5.3.3.1 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.

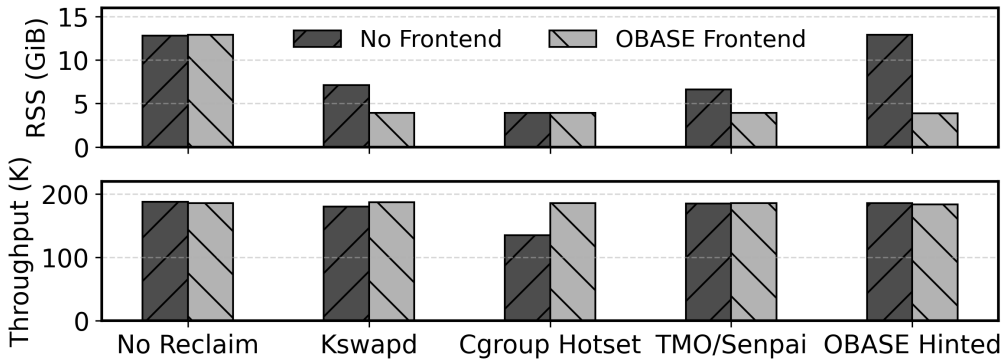


Figure 5.8: ***OBASE* with reclamation backends (YCSB-C, MassTree).** Top: RSS after convergence. Bottom: throughput. Without *OBASE*, backends face a trade-off between memory savings and performance. With *OBASE* (hatched bars), all backends achieve near-optimal memory savings with minimal throughput loss.

- **TMO:** Meta's PSI-based proactive reclaimer [152].
- ***OBASE* Hinted:** Proactive `MADV_PAGEOUT` on the COLD heap after convergence.

For all configurations except *OBASE* Hinted, we disable *OBASE*'s proactive paging to isolate the benefit of address-space reorganization from the hinting mechanism.

Figure 5.8 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 6.5 GiB (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 eliminates the memory-vs-performance trade-off that forces datacenter operators to balance aggressive reclamation against SLO compliance.*

5.3.3.2 Tiering Backends

To demonstrate the memory tiering benefits of reduced hotness fragmentation, we evaluate *OBASE* with three page based migration systems: TPP [116], AutoNUMA [146], and Memtis [101].

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 [101], 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 5.9 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

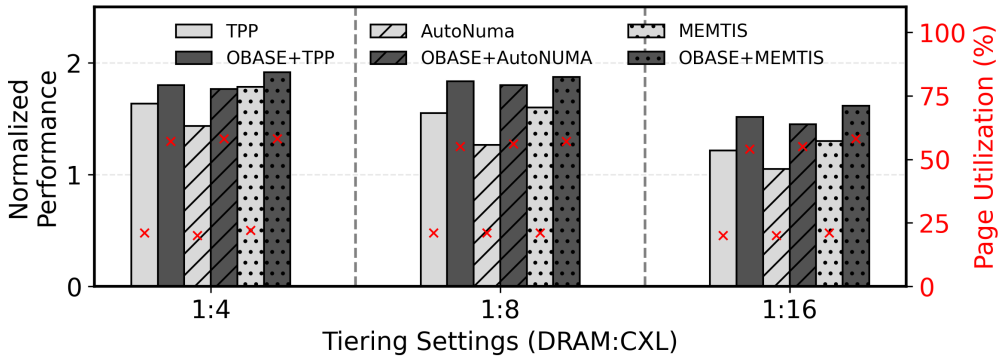


Figure 5.9: ***OBASE* with tiering backends (YCSB-B, MassTree, 50M keys).** Throughput normalized to CXL-only baseline (higher is better). Without *OBASE*, performance degrades as DRAM shrinks because the hot set cannot fit. With *OBASE*, the hot set compacts, enabling stable performance even at 1:16.

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.

From Figure 5.9, *OBASE* at ratio **1:X** performs comparably to baseline at **1:(X/2)**: *OBASE*+TPP at 1:16 (1.45×) approaches TPP alone at 1:8 (1.55×). This arises because *OBASE* reduces the effective hot set by 2.5×, allowing backends to achieve equivalent DRAM residency with half the capacity. For operators, this translates directly to cost savings: *OBASE* enables the same performance with half the DRAM, or doubles effective memory capacity of existing tiered deployments.

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

5.3.4 Overhead and Scalability (E3)

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

5.3.4.1 Steady-State Overhead

Figure 5.10(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.

5.3.4.2 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).

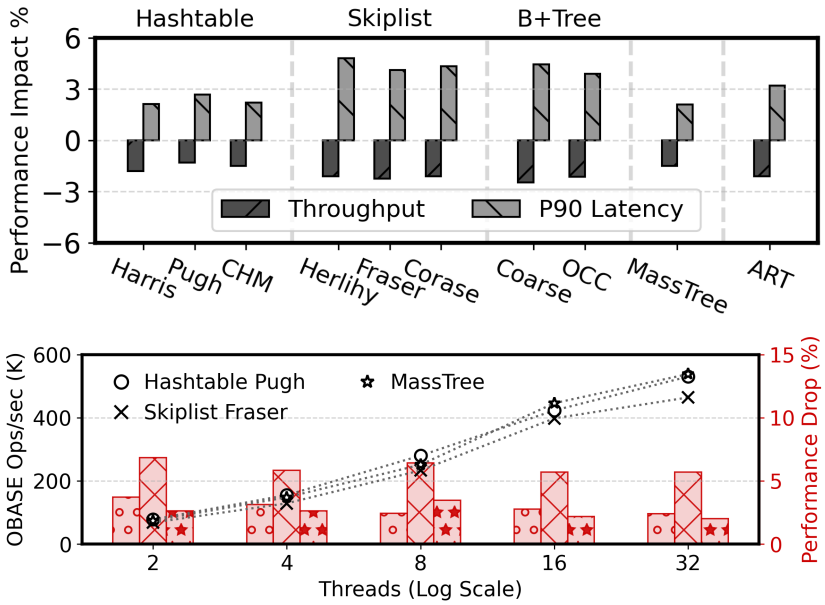


Figure 5.10: ***OBASE* overhead and scalability (YCSB, 10M keys).** Top: Throughput and p90 latency overhead across data structures. Overhead ranges from 1.5–5% depending on structure. Bottom: Scalability from 2 to 32 threads. Bars show absolute throughput; markers show overhead relative to baseline. Overhead remains bounded at 1–8% with no upward trend.

Figure 5.10(b) shows that throughput scales nearly linearly 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 §5.3.3 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 2–5% overhead that remains constant as thread count increases, a modest cost relative to backend improvements demonstrated in §5.3.2–5.3.3.*

5.3.5 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** [51]: Read-heavy (83% GET) with gradually shifting popular keys.
- **DBench Mixgraph** [55]: Models Meta's ZippyDB with key-range locality across 30 prefixes. Read-heavy (85% GET, 14% PUT, 1% SEEK).
- **Twitter Cluster 7** [160]: High skew (α=1.07) with small, concentrated working set of reads and writes.
- **Twitter Cluster 23** [160]: 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$\times$ across traces. Cluster 23 shows the highest gain (3.4$\times$) because its low skew disperses

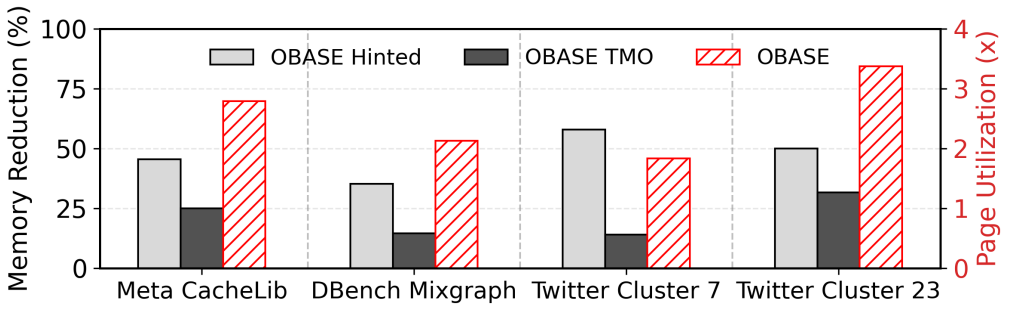


Figure 5.11: **Production trace results.** Memory reduction (left axis): *OBASE* Hinted vs. no-reclaim, and *OBASE*+TMO vs. TMO-alone. Page utilization improvement (right axis, hatched) from address-space reorganization.

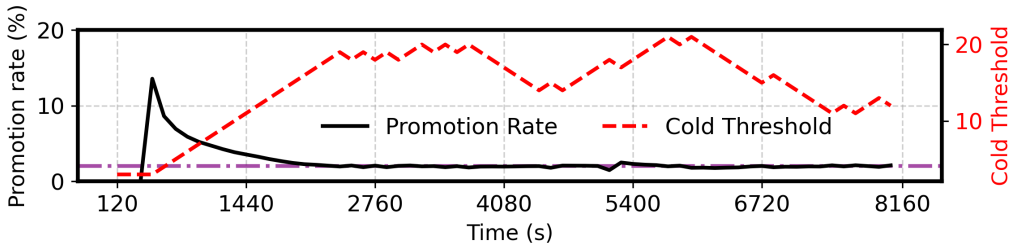


Figure 5.12: **Cold threshold adaptation (Meta CacheLib).** Promotion rate (black) and cold threshold C_t (red) over time. The controller automatically adjusts C_t to maintain the 1% target (purple).

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 ($1.8\times$).

Memory reduction (E1, E2 & E4). Figure 5.11 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, demonstrating that *OBASE* and TMO are complementary: TMO identifies reclaimable pages, while *OBASE* ensures those pages contain uniformly cold data.

Adaptive cold threshold (E4). Figure 5.12 demonstrates *OBASE*'s abil-

ity 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. Note that a high promotion rate during convergence *does not indicate degraded performance*: COLD heap pages remain in DRAM until *OBASE* (or any backend) explicitly pages them out. The promotion rate reflects workload behavior, not backend decisions; *OBASE* measures COLD-heap accesses regardless of where those pages physically reside. This decoupling is intentional: *OBASE* cannot determine which tier a page occupies, so it takes a backend-agnostic approach, keeping the promotion rate low to ensure that COLD pages are safe targets for any backend policy.

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 target, demonstrating that *OBASE* tracks workload evolution.

Takeaway #5: *OBASE delivers consistent memory savings across real world workloads with diverse characteristics, even provides benefits over state-of-the-art backends like TMO, and dynamically adapts to shifting access patterns.*

5.3.6 Multi-Tenant Consolidation (E5)

Datacenter operators seek to maximize machine utilization by co-locating multiple applications, but memory-intensive workloads often resist packing due to their large footprints. We evaluate whether *OBASE*'s memory savings translate into practical consolidation benefits.

Setup. We run three concurrent CrestDB instances on a single 32 GiB machine, each serving a distinct workload: A1 (ART with YCSB-A, 50% writes), A2 (HashTable CHM with YCSB-C, read-only), and A3 (MassTree

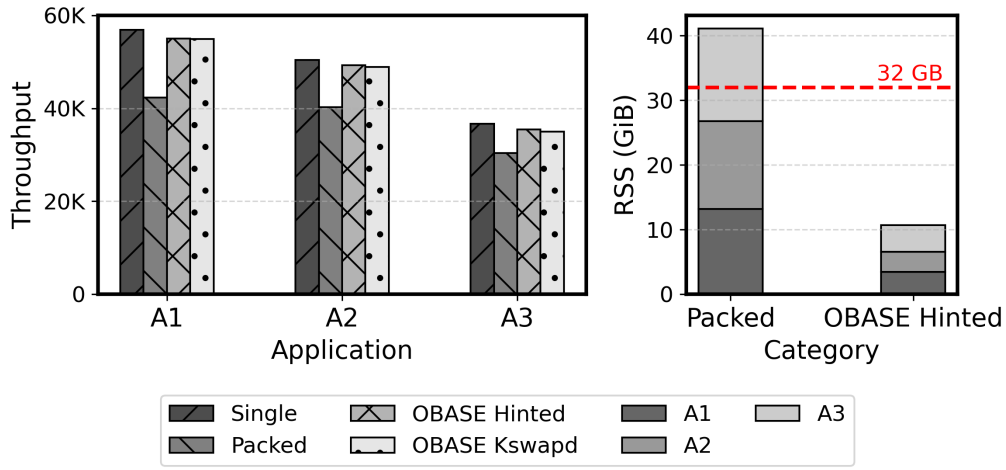


Figure 5.13: **Multi-tenant consolidation.** Left: Per-application throughput. Right: Combined RSS. Conventional packing exceeds the 32 GiB limit and causes thrashing. *OBASE* reduces total memory by 75%, enabling all three applications to run at near-isolation performance.

with YCSB-B, 5% writes). Each instance loads 10M keys, and we measure throughput and total RSS after 10 minutes of concurrent execution.

Figure 5.13 compares four configurations. *Single* runs each application in isolation as a performance baseline. *Packed* runs all three simultaneously without *OBASE*; their combined footprint of 41 GiB exceeds physical memory, triggering severe swap thrashing that degrades throughput by 15–22% across all applications.

With *OBASE* (*OBASE Hinted*), each instance independently reorganizes its address space and reclaims cold pages. Total RSS drops from 41 GiB to 10.2 GiB, a **75% reduction**, comfortably fitting within the 32 GiB budget. Throughput remains within 3–4% of single-tenant performance for all three applications. The *OBASE Kswapd* configuration demonstrates that even without proactive hinting, *OBASE*'s address-space reorganization enables kswapd to reclaim the right pages: RSS reaches 11.5 GiB with similar throughput preservation.

Takeaway #6: *These results suggest that OBASE can increase effective ma-*

chine density by 3–4× for memory-bound workloads, directly translating to infrastructure cost savings in tiered-memory deployments.

5.4 Summary

OBASE demonstrates that memory tiering improves substantially when applications cooperate with the operating system rather than work around it. By reorganizing the virtual address space so that page boundaries align with object temperature, *OBASE* makes existing backends more effective without modifying them: page utilization improves by 2–4×, memory savings reach up to 70%, and tiered configurations match DRAM-only throughput at half the DRAM capacity.

A central lesson from this work is that decomposing the problem along information boundaries yields a more effective and more adoptable design than a monolithic alternative. The frontend has access to application-level semantics that the OS cannot see: which objects exist, how they are linked, and how frequently each is accessed. The backend has access to hardware state that the application cannot see: which physical tier a page occupies, how much capacity remains on each tier, and what the migration cost will be. Neither layer can do the other's job well. By respecting this boundary, each layer can be designed, tested, and improved independently, and a better frontend benefits any backend.

This decomposition also connects *OBASE* to the earlier chapters of this work. WiscSort and *OBASE* share a common principle: ***reducing the amount of data moved matters more than increasing the rate at which data is moved.*** WiscSort achieves this statically, exploiting the semantic structure of sorting to avoid writing values through intermediate phases. *OBASE* achieves it dynamically, reorganizing the address space so that backends migrate only pages that are uniformly cold rather than pages polluted by a few hot neighbors. The two systems operate in different regimes, static when application

semantics reveal access structure, dynamic when access patterns must be inferred from observation, but the underlying principle is the same.

The approach has limitations. It requires developer annotation of relocatable pointer types, and it applies only to pointer-based data structures where indirection can be interposed without violating language semantics. Workloads that do not use pointer-based containers, or that require raw-pointer arithmetic across object boundaries, fall outside *OBASE*'s current scope.

The approach has limitations. It requires developer annotation of relocatable pointer types, and it applies only to pointer-based data structures where indirection can be interposed without violating language semantics. Workloads that do not use pointer-based containers, or that require raw-pointer arithmetic across object boundaries, fall outside *OBASE*'s current scope. *OBASE*'s classification accuracy also depends on precise, per-access tracking through the guide abstraction and cannot operate on sampled telemetry alone (e.g., PEBS-based profiling), because statistical sampling can cause correctness errors. Finally, our evaluation focuses on key-value workloads with well-characterized access skew. General-purpose applications allocate heterogeneous objects whose spatial and temporal locality may differ; validation on such workloads remains future work.

Looking forward, the principle of address-space engineering extends beyond tiering. The same techniques could inform generational garbage collection (grouping objects by observed access intensity rather than allocation age), reduce false sharing across NUMA nodes (separating objects whose access patterns map to different sockets), and strengthen isolation in multi-tenant environments (grouping objects by trust domain so that memory pressure in one tenant reclaims only that tenant's cold pages). The virtual address space is not merely storage; it is a communication channel between applications and the operating system. By engineering this channel, applications can express object-level semantics in a language that page-based systems already understand.

Chapter 6

Related Work

In this chapter, we discuss prior work related to this dissertation. We begin with sorting algorithms and systems designed for memories with asymmetric read-write costs (§6.1), covering both theoretical models and PMEM-specific implementations. We then describe the lineage of key-value separation techniques in storage (§6.2). We next turn to page-level memory management (§6.3), discussing tiering systems and page-level optimizations that share OBASE's goal of improving placement but operate at a coarser granularity. We then cover object-level memory management (§6.4), including far-memory systems and allocation-time placement strategies. Finally, we discuss garbage collection and object relocation (§6.5), which provides the closest analogue to OBASE's migration mechanism, and epoch-based reclamation (§6.6), which inspires OBASE's quiescence protocol.

6.1 Algorithms for Asymmetric Memory

A substantial body of work has been developed around persistent memory, much of it predating the commercial availability of the devices themselves. New file systems [92], data structures [123, 100], key-value stores [50, 163], and databases [145, 41] have all been proposed for PMEM. These systems, however, give primary emphasis to the persistence properties of the device (crash consistency, durability, and recovery) rather than taking full advan-

tage of the broader set of performance characteristics that we capture in the BRAID model. In particular, few of them account for the read-write asymmetry, the interference between concurrent reads and writes, or the constrained write concurrency that characterize byte-addressable storage devices.

Algorithms for asymmetric read-write costs. A separate line of work has studied algorithms that explicitly account for the cost disparity between reads and writes on persistent memory [54, 59, 148, 147]. The general strategy in these works is to perform multiple reads in order to reduce the total number of writes issued to the device. For example, the B-tree variant of Chen et al. [59] neither sorts the keys within a leaf node nor repacks a leaf after a key deletion, thereby avoiding the write cost of sorting and compaction at the expense of additional reads during search. AEM-sort [54] introduced a theoretical model for the asymmetric external memory setting and showed how to asymptotically reduce the number of writes performed by multi-way merge sort, sample sort, and heap sort relative to their standard counterparts. Viglas [148] introduced a family of "write-limited" sorting and join algorithms that prefer to scan the input multiple times rather than materialize intermediate results, thereby reducing writes at the cost of additional read passes. Viglas also proposed a B^+-tree variant [147] adapted for asymmetric I/O in a similar spirit.

Sorting on persistent memory. MONTRES-NVM [52] and NVMSorting [61] introduce techniques to leverage partially sorted inputs on persistent memory. Both systems detect naturally sorted portions ("natural runs") within the input data, which can be skipped during the run generation phase, reducing the total number of writes. These natural runs are then incorporated during the merge phase, where they are merged on the fly with the generated sorted runs. WiscSort, by contrast, does not make any assumptions about the structure or distribution of the input data. More importantly, none of the above algorithms—whether theoretical or

empirical—were designed for a real byte-addressable storage device such as Intel Optane PMEM. As a consequence, they all fail to consider the full range of performance characteristics captured by the BRAID model: the random-read performance (R), the read-write interference (I), and the device-constrained concurrency (D) properties are absent from their designs. WiscSort is the first sorting system that accounts for all five BRAID properties on a real device. Nevertheless, WiscSort is orthogonal to all the above solutions; combining WiscSort's BRAID-aware scheduling with the write-reduction techniques of AEM-sort or the natural-run detection of MONTRES-NVM could further improve sorting performance.

Read-write interference on PMEM. Several prior works have observed and characterized the read-write interference behavior on Optane PMEM. NyxCache [154] proposed an interference-effect estimation mechanism for multi-tenant persistent memory caching, enabling tenants to estimate the performance impact of co-located workloads. Dicio [125] further developed interference-source locating techniques that identify which tenants are responsible for bandwidth degradation in shared PMEM environments. Both systems focus on the multi-tenant setting, where separate applications sharing a PMEM device must coordinate to avoid mutual degradation. Unlike these works, WiscSort focuses on scheduling and sizing the read-write concurrency *within* a single application. The thread-pool controller and the interference-aware scheduler in WiscSort ensure that reads and writes within the sorting pipeline do not overlap, avoiding the bandwidth degradation that occurs when concurrent writes interfere with reads on byte-addressable devices. None of the prior systems perform this kind of intra-application interference-aware scheduling or thread-pool control.

6.2 Key-Value Separation in Storage

Separating keys from values is a classic idea that has been explored in multiple contexts across six decades of systems research [84, 124, 109].

The Modified Key Sort [84], introduced by Hubbard in 1963, was the first system to propose key-value separation for sorting. Rather than moving complete records through each phase of external merge sort, it maintained only key-pointer pairs during sorting and deferred value gathering until the final sorted order was known. However, the hard drives of that era imposed severe penalties on random reads: each value retrieval required a mechanical seek costing milliseconds. To avoid these random reads, the Modified Key Sort converted them to sequential reads by performing additional sorting passes over the pointers, effectively trading compute for I/O locality. This workaround made key-value separation viable in 1963 but also limited its appeal. On modern byte-addressable storage, where random read bandwidth approaches sequential read bandwidth, the additional sorting passes are unnecessary—random reads to gather values are efficient without conversion.

AlphaSort [124] adopted a related strategy for in-memory sorting: it maintained only key-pointer pairs during the sorting phase to improve cache efficiency. By keeping the working set of the sort small enough to fit in cache, AlphaSort reduced cache misses during comparisons. However, AlphaSort operated in a world of block-addressed disks. Since the storage devices of its era did not provide byte-addressability, reading a small key still transferred an entire disk block, and the I/O amplification this caused did not warrant performing late materialization of values from disk. The key-pointer approach thus remained a cache optimization rather than an I/O optimization.

WiscKey [109] applied key-value separation to LSM-tree-based key-value stores. In an LSM-tree, compaction is the dominant source of write amplification: sorted runs are repeatedly merged, and each merge rewrites

all keys and values. WiscKey stores values separately in a value log and keeps only keys and value pointers in the LSM-tree. Because compaction now merges only the smaller key-pointer entries, write amplification drops substantially. Value retrieval requires a random read to the value log, but on SSDs this cost is tolerable because SSD random read latency is orders of magnitude lower than disk seek time. WiscKey demonstrated that key-value separation becomes broadly practical once the underlying device supports efficient random reads.

PMSort [82] examined key-value separation specifically for sorting on PMEM, with a focus on reducing write traffic to improve device endurance through wear-leveling. PMSort studied only the single-threaded case and does not fully exploit the properties of byte-addressable storage. We identify four specific limitations. First, PMSort does not fully take advantage of the random-read bandwidth: it loads both keys and values into DRAM during the run-generation phase, whereas WiscSort loads only keys and pointers, keeping values on the device until the final gather phase. Second, PMSort concludes that QuickSort is the best sorting algorithm for PMEM; however, this conclusion does not hold at scale because QuickSort is neither aware of read-write interference (I) nor does it perform thread-pool control (D) to match the device's concurrency constraints. Third, PMSort avoids performing random reads during value gathering and claims that bandwidth is not the bottleneck, which is not true at scale when multiple threads contend for device bandwidth. Fourth, PMSort's primary optimization target is wear-leveling rather than performance, and it consequently does not utilize all the properties of a BAS device. WiscSort addresses each of these limitations: it is a high-performance, multi-threaded sorting system that complies with the full BRAID model by combining key-value separation with thread-pool control and interference-aware scheduling to use all device resources optimally.

6.3 Page-Level Memory Management

Chapter 2 described the functional design of several page-level tiering systems, including AutoNUMA [146], TPP [116], Memtis [101], and TMO [152]. Here we focus on how these systems and related page-level optimizations relate to OBASE's contributions, rather than repeating their mechanisms.

Tiering systems as backends. Despite their differences in detection mechanisms and migration policies, all four tiering systems share a fundamental constraint: they observe and act on memory at page granularity. AutoNUMA [146] and TPP [116] rely on page faults and LRU lists to detect hot and cold pages. Memtis [101] samples memory accesses at cache-line granularity using hardware performance counters, but it still migrates entire pages between tiers. TMO reclaims pages through the kernel's standard shrink path, using Pressure Stall Information to calibrate reclamation aggressiveness. When applications interleave hot and cold objects within the same pages, all of these systems face the dilemma described in §2.1.3: migrating a partially hot page either wastes bandwidth moving cold bytes or strands hot bytes on the slow tier. OBASE is designed to enhance these backends, not replace them. By reorganizing the address space so that cold pages contain only cold objects, OBASE provides each backend with a higher-quality signal, allowing it to make better decisions using its existing policy without modification. This decoupling allows independent innovation: improvements to OBASE benefit any backend, and improvements to any backend benefit applications using OBASE.

Hardware-aware page placement. A separate class of systems optimizes how pages interact with hardware caching and tiering mechanisms, without attempting to understand the objects within those pages. Johnny Cache [103] manipulates physical page allocation to minimize address conflicts in the hardware sets of direct-mapped DRAM caches. In systems

where CXL-attached memory serves as a capacity tier behind a direct-mapped DRAM cache, page placement determines whether frequently accessed pages collide in the same cache set, causing unnecessary evictions. Johnny Cache addresses this problem by controlling the physical addresses assigned to virtual pages, reducing conflict misses without modifying the application or the caching hardware. Memstrata [164] solves a related problem in virtualized environments running Intel Flat Memory Mode, where hardware transparently tiers cache lines between local DRAM and CXL-attached memory. Memstrata manages host-physical page mappings to isolate tenants and optimize per-tenant performance within this hardware tiering layer. Both Johnny Cache and Memstrata treat the contents of each page as opaque: they optimize *where* pages are placed in the physical address space but have no visibility into *what* those pages contain. OBASE's approach is orthogonal. By ensuring that the cache lines within each virtual page are more uniformly hot or cold, OBASE makes the hardware's per-cache-line tiering decisions more effective. A page whose cache lines are all frequently accessed will be promoted and retained in fast memory by the hardware cache; a page whose cache lines are all cold will be evicted without performance penalty. Combining OBASE with Johnny Cache or Memstrata could yield compounding benefits: OBASE prepares page contents for uniform temperature, and the page-placement systems ensure those pages land in optimal physical locations.

Page size and tier selection. HawkEye [127] improves huge page management by performing fine-grained page-level access tracking to make better decisions about when to promote base pages to huge pages and when to split huge pages back. The goal is to improve TLB utilization: huge pages reduce TLB pressure when their contents are uniformly accessed, but waste memory when only a small region within a huge page is hot. HawkEye makes these decisions at page granularity; it does not distinguish between

hot and cold objects that share a page.

Memtis [101] goes further by dynamically determining both the page tier and the page size in heterogeneous memory systems. Using hardware sampling (Intel PEBS), Memtis detects access skew within huge pages: if only a subset of a huge page's base-page-sized regions are hot, Memtis can split the huge page into base pages and migrate only the hot subpages to the fast tier, balancing tiering benefits against the increased address-translation cost of using smaller pages. This approach is more precise than systems that treat each huge page as an atomic unit, but it still operates at the page level. If a base page itself contains a mix of hot and cold objects, Memtis cannot separate them: the entire base page must be classified as either hot or cold, and any hot object on an otherwise cold page forces the page to remain in fast memory.

OBASE addresses precisely this remaining gap. By organizing objects so that each page is more likely to be homogeneously hot or cold, OBASE ensures that when Memtis examines a base page, that page's access pattern accurately reflects its contents. Huge-page splitting becomes more effective because the subpages that Memtis migrates to the fast tier are densely packed with hot objects, while the subpages it demotes contain only cold data. Similarly, HawkEye's decisions about huge-page promotion improve when the candidate pages have uniform access intensity. In this sense, OBASE serves as a preparatory layer that improves the precision of any page-level mechanism, whether it selects pages for migration (TPP, TMO), manages page sizes (HawkEye, Memtis), or controls physical placement (Johnny Cache, Memstrata, by ensuring that the per-page signals these mechanisms observe are faithful representations of the data they contain.

6.4 Object-Level Memory Management

Several systems manage memory at object granularity rather than page granularity. We discuss them in two groups: systems that provide object-level management for far-memory environments, and systems that attempt to improve placement through allocation-time hints or profiling.

Far-memory systems. AIFM [133] and MIRA [77] both operate at object granularity but target disaggregated far-memory scenarios over RDMA, a fundamentally different deployment setting from the local tiered-memory environment that OBASE addresses.

AIFM replaces OS memory management entirely with a user-space runtime built on Shenango's green threads [126], kernel-bypass networking, and custom evacuation mechanisms tightly coupled to its thread scheduler. This architecture enables AIFM to hide remote access latency through lightweight context switches (approximately 50 ns) while application threads wait for data to arrive from remote memory. However, adopting AIFM requires applications to use its complete runtime stack, including its threading model, networking stack, and data structure library with `DerefScope` lifetime guards that demarcate regions of far-memory access. AIFM's evacuator is also functionally different from OBASE's Object Collector: AIFM's evacuator moves objects *to remote memory* when local memory is exhausted, whereas OBASE's Object Collector reorganizes objects *within local memory* to improve page-level uniformity.

OBASE takes a complementary approach: rather than replacing OS mechanisms, it acts as a *frontend* that prepares the address space for unmodified OS *backends*. This architectural choice reflects a key insight: the problem in tiered-memory systems is often not the OS's tiering policy itself, but the quality of the signal it receives. When pages contain uniformly hot or cold objects, existing mechanisms such as kswapd, TPP [116], and Memtis [101] make correct decisions without modification.

Direct performance comparison between AIFM and OBASE is not meaningful because they target different deployment scenarios and solve different problems. AIFM optimizes for the *latency* of accessing remote memory once data has already been tiered; OBASE optimizes the *decision* of what to tier by eliminating hotness fragmentation. These approaches are orthogonal: AIFM's local memory cache could benefit from OBASE's address-space engineering to improve its evacuation decisions, as AIFM faces the same hotness fragmentation problem when selecting which local objects to evict.

MIRA [77] focuses on program-behavior-guided prefetching for RDMA-based far memory, sharing AIFM's requirement for specialized hardware interfaces and custom runtime support. Neither AIFM nor MIRA addresses the problem OBASE targets: that page-based mechanisms cannot distinguish hot from cold objects when they are interleaved on the same pages within local memory.

Handle-based indirection. Alaska [151] uses additional indirection to enable heap compaction in unmanaged languages, reducing RSS through defragmentation. Alaska's indirection is inspired by the MacOS Resource Manager [86], which introduced movable memory blocks accessed through handles, allowing the OS to compact memory without invalidating pointers. Alaska's handles serve a purpose similar to OBASE's guides in that both decouple an object's logical identity from its physical location. However, the two systems differ in what motivates relocation and how they decide which objects to move. Alaska does not track object hotness or organize memory by access patterns; it addresses physical memory fragmentation—the accumulation of unusable holes in the heap—rather than the semantic mismatch between object-level access and page-level management that OBASE targets. OBASE's guides serve a different purpose: enabling safe concurrent migration based on observed access intensity, so that pages presented to OS tiering mechanisms contain data of uniform temperature.

Allocation-time placement. A number of systems attempt to improve memory placement by making decisions at allocation time, using either programmer-supplied hints or profile-guided information [67, 60, 119, 165, 91]. X-Mem [67] provides a data-tiering framework for heterogeneous memory where programmers annotate data structures with placement preferences. ATMem [60] uses offline profiling to guide data placement in graph applications on heterogeneous memories. TCMalloc [165] supports hot/cold hints that allow allocation sites to request placement in different memory regions. Systems like PGHO [91] can apply these hints automatically based on profile data collected through LLVM's memory profiling infrastructure, removing the need for manual annotation.

Despite their differences in how hints are generated—manually, through offline profiling, or through compiler-driven instrumentation—all of these approaches share a fundamental limitation: they make placement decisions only at allocation time. As Chapter 4 demonstrates, object hotness is neither knowable at allocation nor stable over time. The same allocation site produces objects with vastly different access patterns: a single `SET` handler in a key-value store allocates memory for both session tokens accessed every millisecond and user profiles never touched again. Even among objects that are initially hot, access intensity shifts continuously, with reuse distances varying by more than an order of magnitude for the same key. Allocation-time approaches cannot capture these transitions between hot and cold states, nor can they distinguish between objects from the same allocation site that develop different runtime access patterns.

Furthermore, the instrumentation-based profile collection that systems like PGHO require imposes prohibitive overhead for always-on deployment in production workloads, and the resulting placement decisions depend on the availability of representative workloads for offline profiling, which are often unavailable or unrepresentative of the temporal dynamics that drive hotness changes. OBASE instead tracks access patterns at runtime,

enabling migration based on observed usage rather than static predictions. As access patterns shift, OBASE continuously reclassifies objects and reorganizes the address space, preventing the gradual re-emergence of hotness fragmentation that static approaches inevitably suffer.

6.5 Garbage Collection and Object Relocation

Garbage collectors have long exploited object relocation to improve memory layout, and the techniques they employ provide the closest precedent for OBASE's migration mechanism. We discuss generational and profile-guided collectors, GC-OS cooperation, and the concurrent relocation protocols of modern production GCs, drawing out the distinctions in both mechanism and purpose.

Generational and profile-guided collection. Generational garbage collectors partition the heap by object age, exploiting the weak generational hypothesis: most objects die young. By clustering recently allocated objects in a nursery and copying survivors to an older generation, generational collectors concentrate allocation and reclamation in a small, frequently scanned region. This organization improves cache locality for young objects and reduces the frequency of full-heap scans.

Profile-guided approaches go further by using observed access patterns to guide object placement during copying collection. Online Object Reordering [83] groups hot objects together when the collector copies them, so that frequently accessed objects end up on the same cache lines and pages after collection. This reordering improves spatial locality for the application's working set, reducing cache and TLB misses.

OBASE's Object Collector shares the high-level intuition of these profile-guided approaches (group objects by access intensity) but differs in the classification criterion and the target outcome. Generational collectors

group by age, which correlates with but does not directly measure access intensity. Profile-guided collectors group by observed access, but their goal is to improve cache locality for the application. OBASE groups by access intensity specifically to improve *page utilization*: the fraction of bytes within each page that are actually accessed. This distinction matters because OBASE's goal is not to speed up individual accesses through better cache behavior, but to ensure that page-level tiering mechanisms see uniformly hot or cold pages, enabling them to reclaim or migrate memory with high confidence.

GC-OS cooperation. The interaction between garbage collectors and the operating system's memory management has also been studied. The Bookmarking Collector [81] addresses a specific failure mode that arises when the OS pages out heap memory during garbage collection: if the collector must trace pointers through an evicted page, it triggers a page fault that stalls collection and defeats the purpose of paging. The Bookmarking Collector avoids this problem by tracking outgoing pointers from evicted pages in "bookmarks," allowing the collector to skip evicted pages during tracing without losing reachability information.

This work shares OBASE's concern with the interaction between object-level and page-level memory management, but approaches it from the opposite direction. The Bookmarking Collector adapts the GC to cope with the OS's paging decisions; OBASE adapts the address space so that the OS's paging decisions are more effective. In the Bookmarking Collector, pages are evicted by the OS without regard for their object-level contents, and the GC must work around the resulting inconsistency. In OBASE, the address space is organized so that when the OS examines a page, its contents faithfully represent a uniform temperature, and eviction or migration decisions based on page-level signals are unlikely to strand hot data or waste bandwidth on cold data.

Concurrent relocation in production GCs. The most relevant comparison is with the concurrent relocation protocols of modern production garbage collectors, particularly ZGC [158] and Shenandoah [70]. Both systems relocate objects concurrently with application execution, avoiding stop-the-world pauses for compaction. Their mechanism is the *load barrier*: every pointer load is intercepted by a small code sequence that checks whether the referenced object has been relocated and, if so, redirects the access to the object's new location via a forwarding pointer. This design allows relocation to proceed even while application threads hold stale pointers to the old location, because any subsequent dereference will be transparently redirected. RAMCloud's log-structured memory [134] achieves concurrent relocation in a storage context using a different mechanism: all objects are accessed through a hash table that serves as a single point of indirection, and the cleaner atomically updates this entry after copying an object. Old segments are freed only after all in-flight RPCs complete, using an RCU-like quiescence check that exploits the natural request boundaries of a storage API. OBASE's design generalizes this idea to arbitrary C++ data structures, where no hash table mediates access and no RPC boundary provides a natural quiescent point; guides and compiler-inserted TAGs fill these roles respectively.

OBASE cannot use load barriers because C++ has no managed runtime to intercept arbitrary pointer loads. Instead, OBASE adopts a *quiescence-based* approach: it tracks how many threads are actively using each object through the Active Thread Count (ATC) and only relocates an object when its ATC reaches zero, indicating that no thread is currently executing an operation that could dereference a pointer to it. If any thread accesses the object during the copy phase, the migration is aborted via a CAS failure on the guide, and the object remains at its original address. Application threads never block on the collector, and concurrent access safely vetoes relocation rather than requiring forwarding-pointer resolution.

This design reflects OBASE's operating constraints: in C++ systems code, adding a check to every pointer load would impose unacceptable overhead and require pervasive code changes, whereas scoping ATC tracking to public API boundaries (via compiler-inserted TAGs) confines the instrumentation to well-defined entry and exit points. The tradeoff is that OBASE may fail to migrate objects that are continuously accessed, since their ATC rarely reaches zero. In practice, this is the desired behavior: frequently accessed objects belong in the HOT heap and should not be moved. Cold objects, by definition, have extended periods of inactivity during which ATC reaches zero, making them amenable to migration.

The key distinction from garbage collection is purpose. GCs relocate objects to reclaim memory (by compacting live objects and freeing the space occupied by dead ones) or to improve cache locality (by clustering related objects). OBASE relocates objects to improve *page utilization* clustering objects by access temperature so that OS tiering and reclamation mechanisms see uniformly hot or cold pages. The relocation itself does not free memory; it reorganizes the address space so that existing page-based backends can free memory more effectively.

6.6 Epoch-Based Reclamation

OBASE's migration protocol draws on a family of concurrent memory reclamation techniques that defer resource cleanup until it is safe to do so. We describe the principal members of this family and explain how OBASE adapts their coordination mechanism for a different purpose.

Read-Copy-Update. Read-Copy-Update (RCU) [118] allows readers to access shared data structures without acquiring locks. Writers create a modified copy of the data and atomically install it, but defer freeing the old version until all readers that could have observed it have completed

their critical sections. The key abstraction is the *grace period*: a duration after which no thread can still hold a reference to the old version. RCU identifies grace period boundaries by observing quiescent states—points in each thread's execution where it is guaranteed not to hold references to RCU-protected data (e.g., context switches in the Linux kernel, or explicit quiescent-state reports in user-space RCU implementations).

Epoch-based reclamation. Epoch-based reclamation [71, 56] generalizes the grace-period concept. The system maintains a global epoch counter. When a thread begins an operation that accesses shared data, it records the current epoch; when it completes, it clears its registration. Resources retired during epoch e can be reclaimed once every active thread has advanced past e, because no thread that observed the old version can still be executing. Fraser [71] used this mechanism in lock-free data structures to safely reclaim nodes removed by concurrent operations. FASTER [56] employed a similar epoch framework to coordinate state transitions in a concurrent key-value store, using epochs to determine when log pages can be safely truncated and when thread-local operations can be committed.

OBASE's adaptation. OBASE adapts epoch-based coordination for a different purpose: rather than deferring memory reclamation until a grace period completes, OBASE uses epoch transitions to activate and deactivate per-object Active Thread Count (ATC) tracking. Outside migration epochs, guide dereferences record only access activity (an accessed bit is set via an atomic read-modify-write) with no ATC overhead. When the Object Collector initiates a migration window, it increments the global epoch counter and enters a PREPARE state. Threads entering public data-structure operations record the new epoch in a Thread Activity Index (TAI); threads exiting clear their slots. Once every active slot in the TAI reflects the new epoch, the Object Collector knows that all threads have enabled ATC tracking, and it transitions to the ACTIVE state, during which objects with ATC=0 can

be migrated. After migration completes, the system returns to INACTIVE, and ATC tracking is disabled until the next migration window.

This design confines the synchronization cost of ATC tracking to brief, infrequent windows rather than imposing it on every pointer dereference throughout execution. The epoch mechanism provides the same safety guarantee as in classical epoch-based reclamation, no thread can be in a state inconsistent with the current protocol phase, but the protected resource is not a retired memory block awaiting deallocation. Instead, the protected invariant is that all threads are aware of the migration window and are correctly maintaining ATC for the objects they touch.

The relationship between OBASE's epoch protocol and classical epoch-based reclamation is thus one of mechanism reuse rather than direct application. In RCU and Fraser's scheme, epochs gate when memory can be freed. In FASTER, epochs gate when state transitions can be committed. In OBASE, epochs gate when objects can be moved. The underlying coordination pattern of announcing participation, observing global convergence, and performing the deferred action is the same in all cases. OBASE's contribution is not a new reclamation scheme but a novel application of epoch-based coordination to safe object migration in unmanaged languages, where load barriers are unavailable and stop-the-world pauses are unacceptable for production systems.

Chapter 7

Conclusion and Future Work

In this closing chapter, we summarize the contributions of the dissertation (§7.1), discuss directions for future work (§7.2), reflect on lessons learned during the research (§7.3), and conclude (§7.4).

7.1 Summary

This dissertation comprises three interwoven parts that together advance the state of the art in data organization for modern memory hierarchies. The unifying observation is that the page, while a convenient unit of translation and protection, is a poor unit of data placement. Applications access data at object and field granularity, yet operating systems and allocators arrange data without regard for how it will be accessed. The resulting mismatch wastes bandwidth on byte-addressable storage and traps cold data in expensive memory tiers. We address this mismatch along two axes: when application semantics reveal access structure at design time, layout can be engineered statically; when access patterns depend on workload behavior and change over time, layout must be inferred from observation and adjusted dynamically. We summarize each part below.

7.1.1 The BRAID Model and WiscSort

Byte-addressable storage devices such as Intel Optane PMEM and CXL-attached memory exhibit properties that differ fundamentally from both DRAM and block storage. We identified five such properties and organized them into the BRAID model: Byte addressability, high Random read performance, Asymmetric read-write costs, read-write Interference, and device-constrained concurrency (D). Together, these properties invalidate the assumptions underlying conventional storage algorithms. Sequential access no longer amortizes seek costs, because there are no seeks. Bundling keys with values no longer avoids random-access penalties, because random reads are nearly as fast as sequential reads. Overlapping reads and writes no longer improves throughput, because concurrent writes degrade read bandwidth.

We applied the BRAID model to external sorting, a fundamental operation in databases and data-intensive systems. The key observation is that sorting compares keys but does not examine values until the final output phase. On block-addressed devices, exploiting this observation is impractical: gathering values via random reads at the end would incur prohibitive seek costs. On byte-addressable storage, the calculus changes. Random reads of individual values are inexpensive, while writing unnecessary data through intermediate phases wastes the scarce write bandwidth.

WiscSort separates keys from values, maintaining only key-pointer pairs during sorting and deferring value retrieval until sorted order is known. This late materialization reduces write traffic in proportion to the ratio of value size to key size. WiscSort further incorporates a thread-pool controller that sizes read and write pools according to device characteristics, and an interference-aware scheduler that avoids overlapping reads and writes on the same device.

We evaluated WiscSort on real Optane PMEM hardware across a range of workloads. WiscSort achieves a 2–3$\times$ speedup over a competitive external

merge sort for the standard sort benchmark workload, and the performance gap widens as value sizes increase relative to key sizes. We also projected WiscSort's behavior on future BRAID devices through CXL emulation, confirming that the design principles hold across a range of device parameters.

7.1.2 Hotness Fragmentation in Practice

When application semantics do not reveal access structure at design time, static layout is insufficient. Key-value stores, caches, and databases serve workloads where access patterns depend on user behavior, application logic, and temporal dynamics that cannot be known in advance. For these applications, we turned to the question of how existing systems manage data placement and what costs arise from their current approach.

Modern memory allocators place objects based on size class and allocation order, with no consideration for how those objects will be accessed. The operating system, in turn, manages memory at page granularity using coarse access bits that cannot distinguish a few hot bytes from an entire cold page. We introduced page utilization as a metric to quantify the resulting fragmentation: the fraction of bytes within touched pages that are actually accessed during a measurement window.

Across open-source key-value stores and six Google production workloads, we found that page utilization is consistently low. In Redis, 75% of accessed pages utilize 3% or fewer of their bytes. Across the Google workloads, median utilization ranges from 8% to 50%, even at 4 KiB granularity. The consequences are threefold. First, cold data becomes trapped alongside hot neighbors, preventing the OS from reclaiming pages that contain mostly unused bytes; across the Google workloads, 55–98% of touched-page capacity holds cold data that cannot be reclaimed. Second, the inflated page working set increases TLB pressure, with scattered placement causing up to 23$\times$ more TLB misses than clustered placement for the same number of hot objects. Third, the resulting memory overprovisioning translates directly

to higher infrastructure costs.

We further analyzed production traces from Meta and Twitter to examine the temporal stability of object hotness. The traces reveal that hotness is neither predictable at allocation time nor stable over the lifetime of an object. The same allocation site produces objects with vastly different access lifetimes, and individual objects alternate between hot and cold phases with wide variation in reuse distance. These findings establish that allocation-time placement decisions, no matter how sophisticated, cannot solve the fragmentation problem. Dynamic reorganization is necessary.

7.1.3 Object-Based Address-Space Engineering

To address hotness fragmentation in workloads with unpredictable and shifting access patterns, we introduced address-space engineering: reorganizing the virtual address space so that objects with similar access intensity reside together. By clustering hot objects onto hot pages and cold objects onto cold pages, page-based tiering and reclamation mechanisms can operate effectively without understanding object boundaries or application semantics.

We developed OBASE as a realization of this idea for C++ applications, where object addresses are assumed to be stable and no managed runtime exists to intercept pointer loads. OBASE is structured as a frontend that makes object placement decisions, decoupled from any page-level backend that makes tier placement decisions. This separation allows each layer to improve independently: a better frontend benefits any backend, and a better backend benefits any frontend.

OBASE introduces three ideas to make dynamic reorganization practical in unmanaged languages. First, a guide abstraction interposes a level of indirection on pointer dereferences, enabling objects to be relocated transparently. Second, a lightweight access tracking mechanism records which objects are accessed during each observation window without the

overhead of full software instrumentation. Third, a lock-free migration protocol relocates objects between hot and cold heaps without pausing application threads, using optimistic concurrency control to handle races between migration and concurrent access.

A set of compiler passes automates the conversion of raw pointers to guides and the insertion of tracking instrumentation, confining the developer's role to marking which pointer types should be managed.

We evaluated OBASE on ten concurrent data structures spanning lock-free, fine-grained, and coarse-grained synchronization mechanisms. OBASE improves page utilization by 2–4$\times$ across YCSB workloads and reduces resident set size by 65–72% when the OS reclaims cold pages. When paired with existing tiering backends (AutoNUMA, TPP, Memtis, TMO), OBASE enables each backend to reach near-optimal memory savings with minimal throughput loss, even in configurations where the backend alone cannot distinguish hot pages from cold ones. Replay of production traces from Meta and Twitter confirms that OBASE adapts to shifting access patterns over multi-hour runs, maintaining high page utilization as the working set evolves. Throughout, OBASE imposes modest overhead: throughput decreases by 2.5% on average, with no upward trend as thread counts increase.

7.2 Future Work

We recognize that WiscSort, OBASE, and the broader principles developed in this dissertation all have potential for further extension. We discuss six directions that build on the contributions presented here.

7.2.1 Sorting Across Heterogeneous Storage

WiscSort is currently designed for datasets that fit within a single BRAID device. If the dataset exceeds the device capacity and must spill to secondary storage such as SSDs or HDDs, new design questions arise. The challenge is

not simply to use the secondary device as overflow, but to use both devices simultaneously in a manner that respects the distinct properties of each. A BRAID device offers fast random reads and penalizes writes; an SSD offers higher capacity and tolerates writes more readily but penalizes random access at small granularities. An effective design would partition the sorting workflow across devices according to their strengths: for example, retaining the IndexMap and frequently accessed key-pointer pairs on the BRAID device while placing intermediate run files or archived values on the SSD.

This heterogeneous setting also introduces scheduling complexity. The interference and concurrency constraints that WiscSort manages for a single device now extend across devices with different characteristics. A thread-pool controller must account for the distinct scaling behavior of each device, and an interference-aware scheduler must coordinate access patterns across storage tiers rather than within a single one. Exploring these tradeoffs for sorting, and for data-intensive operations more broadly, is a natural next step.

7.2.2 Compression of Intermediate Representations

WiscSort's IndexMap files contain key-pointer pairs that are written to the BRAID device during the run generation phase. Compressing these files before writing them would reduce write traffic, directly addressing the asymmetric read-write cost property. However, compression introduces a tradeoff: the reduced write cost must be weighed against the increased read cost of decompression during the merge phase, as well as the additional CPU load imposed by both operations.

Whether compression is worthwhile depends on the specific device parameters and the compressibility of the key space. Highly sequential or repetitive key distributions would compress well, yielding substantial write savings at modest decompression cost. Random or high-entropy key distributions would compress poorly, adding CPU overhead without meaningful

I/O reduction. A practical approach would adapt compression decisions to the observed key distribution, applying compression selectively when the expected write savings exceed the decompression penalty. This direction is orthogonal to WiscSort's core design and could be incorporated without altering the sorting algorithm itself.

7.2.3 BRAID-Aware Operations Beyond Sorting

WiscSort demonstrates the benefits of redesigning a single operation for byte-addressable storage. However, a database comprises many fundamental operations, each with its own access patterns and I/O characteristics. Operations such as `JOIN`, `GROUP BY`, `MERGE`, and `FILTER` all move data between storage and memory, and all were designed under assumptions that BRAID devices invalidate.

The IndexMap structure that WiscSort produces during sorting opens a path toward broader changes. Because the IndexMap separates keys from values and records the location of each value in the original file, it effectively converts a row-oriented layout to a column-oriented one on the fly. This conversion enables late materialization for operations beyond sorting. For instance, a range query over sorted keys can retrieve only the keys that satisfy the predicate and defer value retrieval until the result set is known. Similarly, a join between two relations can proceed over their respective IndexMap files without moving the associated values, fetching values only for the records that survive the join condition.

These observations suggest that the row-to-column conversion need not be performed in advance and stored redundantly, as current HTAP systems require. Instead, the byte-addressability and random read performance of BRAID devices make it practical to maintain a single copy of data in row-oriented format and derive columnar views dynamically as queries demand. This would avoid the capacity overhead and synchronization complexity of maintaining dual formats, a well-known source of difficulty

in existing HTAP designs. Exploring how far this principle extends across the space of database operations is a substantial direction for future work.

7.2.4 Address-Space Engineering Beyond Tiering

This dissertation presented address-space engineering in the context of memory tiering, where the goal is to separate hot objects from cold ones so that page-level backends can place them on appropriate tiers. The same principle of reorganizing the address space by observed access intensity applies to several adjacent problems.

Garbage collectors in managed runtimes already move objects between generations based on age. Address-space engineering offers a complementary signal: rather than relying solely on the generational hypothesis that young objects die quickly, a collector could use observed access intensity to inform promotion and tenuring decisions. Objects that are old but cold could be separated from objects that are old and hot, enabling the collector to prioritize compaction of cold regions and avoid disturbing hot regions that would incur cache and TLB disruption. The guide abstraction developed for OBASE is unnecessary in managed languages, where the runtime already controls pointer dereferencing, but the classification policy and the feedback-driven cold threshold controller transfer directly.

Beyond tiering and collection, address-space engineering can inform NUMA placement decisions. On multi-socket systems, the OS must decide which NUMA node should host each page. Current mechanisms such as AutoNUMA rely on page-level fault sampling to detect which node accesses a page most frequently, but they cannot distinguish pages that are uniformly hot from pages that contain a mix of hot objects accessed by one node and cold objects that happen to share the same page. If the frontend has already separated hot and cold objects, NUMA placement decisions become more precise: hot pages can be placed close to the node that accesses them, and cold pages can be placed on any node without performance concern.

A related application is memory isolation in multi-tenant environments. When multiple tenants share a machine, the operating system must ensure that one tenant's memory pressure does not degrade another's performance. If each tenant's address space is organized so that hot and cold regions are clearly delineated, the OS can enforce memory limits by reclaiming cold regions first, reducing the likelihood of evicting a hot page belonging to one tenant to satisfy another's allocation. These extensions share a common structure: the frontend organizes the address space, and different backends exploit that organization for different purposes.

7.2.5 Direct Object Tiering Across Memory Hierarchies

In OBASE's current design, the frontend organizes the virtual address space and the backend decides where physical pages reside. The frontend has no knowledge of which tier a page occupies, and the backend has no knowledge of which objects a page contains. This separation is a deliberate design choice that enables compatibility with existing OS infrastructure. However, it also means that placement decisions pass through two layers of indirection, each operating at a different granularity and with incomplete information.

An alternative design would collapse these layers, allowing the frontend to place objects directly on specific memory tiers. If the system includes DRAM, CXL-attached memory, and a BRAID device, the frontend could allocate hot objects in DRAM, warm objects on CXL memory, and cold objects on the BRAID device, bypassing the page-level backend entirely. The guide abstraction already provides the indirection needed to relocate objects transparently; extending it to manage cross-tier placement is a natural progression.

This direct approach introduces new constraints. Placement decisions on a BRAID device must respect the same properties that informed WiscSort's design: writes should be minimized, reads and writes should not

be overlapped carelessly, and the concurrency characteristics of the device must be considered. An object that migrates from DRAM to a BRAID tier incurs a write; if that object is later accessed and must migrate back, it incurs both a read from the BRAID device and a write to DRAM. The migration policy must therefore account for the asymmetric costs and interference properties of each tier, not merely the access frequency of the object. In other words, object tiering on heterogeneous memory must itself be BRAID-compliant.

The potential benefit is precision. Page-level backends must classify entire pages as hot or cold, and a single hot object can pin an otherwise cold page in fast memory. Direct object tiering eliminates this granularity mismatch entirely: each object is placed according to its own access pattern and the properties of the target tier. The cost is complexity. The frontend must now understand the characteristics of each tier, maintain per-tier allocation pools, and make migration decisions that balance access latency, write endurance, and interference. Whether this added complexity yields sufficient benefit over the two-layer approach, and under what workload and hardware conditions, remains an open question.

7.2.6 Address-Space Engineering for Vector Search Indices

Large language models (LLMs) have made vector search a critical piece of production infrastructure. Retrieval-augmented generation pipelines [104] query a vector store at every inference step, fetching the nearest embeddings to a prompt so that the model can ground its output in external knowledge rather than relying solely on its parameters. As models are deployed at scale, the associated embedding collections grow to billions of vectors: enterprise document corpora, web-scale knowledge bases, and multimodal indexes all produce datasets whose working sets exceed DRAM capacity on a single machine [149, 75]. The natural response is to spread the index across heterogeneous memory tiers, placing some data in DRAM, some in CXL-attached memory, and some on SSD. How data is assigned to tiers

has a direct effect on query latency and throughput, yet the assignment strategies used by current systems are almost entirely static.

Two families of approximate nearest neighbor (ANN) indices dominate the literature. Graph-based indices build a proximity graph over the embedding space and answer a query by greedy traversal from an entry point. DiskANN [139] stores a Vamana graph and full-precision vectors on SSD while caching product-quantized vectors in DRAM, achieving high recall at billion scale on a single node. HM-ANN [132] extends HNSW [111] to heterogeneous memory by placing the upper layers of the hierarchical graph in DRAM and the large bottom layer in persistent memory. CXL-ANNS [89] places the graph in CXL-attached memory and caches nodes that lie within a fixed hop radius of the entry point in local DRAM. Starling [150] co-locates graph neighbors within the same disk blocks to reduce I/O amplification on SSD. FreshDiskANN [137] supports streaming inserts through a merge-based two-tier design, and FusionANNS [143] introduces CPU/GPU collaborative filtering across GPU HBM, main memory, and SSD. In every one of these systems, the decision of which data resides on which tier is determined by a structural property of the index: layer membership in HNSW, hop distance from the entry point in Vamana, or block adjacency on disk. The actual access frequency imposed by the query workload plays no role.

Partition-based indices take a different approach. They cluster the embeddings, store cluster centroids in memory for coarse-grained navigation, and keep posting lists of full vectors on slower storage. SPANN [58] established this template at billion scale, and ScaNN [76] introduced anisotropic vector quantization to improve compressed distance estimates within each partition. SPFresh [157] solved the update problem for partition indices through lightweight incremental rebalancing that splits and merges clusters as the data distribution shifts, without requiring a global index rebuild. More recently, a line of work has begun to incorporate workload awareness at the partition level. Quake [121] maintains a cost model that includes

the observed access frequency of each partition and dynamically splits partitions that receive disproportionate query traffic while merging those that receive little. Under skewed workloads, this yields substantial latency reductions compared to HNSW, DiskANN, and ScaNN. Ada-IVF [120] prioritizes maintenance of frequently read partitions and observes that the large majority of index updates affect partitions that are never touched by search. CrackIVF [110] takes the idea further by building the index progressively as queries arrive, so that regions of the embedding space with heavy query traffic receive finer-grained partitioning while untouched regions remain coarse. These systems represent meaningful progress, but their adaptation mechanism is to restructure the index itself: Quake splits and merges clusters, and CrackIVF refines partition boundaries. The granularity of adaptation is the partition, which typically numbers in the hundreds or low thousands, not the individual embedding.

The approach suggested by OBASE differs in a fundamental way. Rather than restructuring the index, OBASE would reorganize the address space in which the embeddings reside while leaving the index structure entirely intact. The graph edges of an HNSW or Vamana index, the cluster assignments of an IVF index, and the quantization codebooks used for distance estimation would all remain unchanged. What changes is the physical grouping of embeddings in the virtual address space. Embeddings that are frequently accessed by queries are relocated so that they share pages with other frequently accessed embeddings, and embeddings that are rarely or never accessed are allowed to coalesce on pages occupied by other cold embeddings. The page-level backend then places hot pages on DRAM, warm pages on CXL-attached memory, and cold pages on SSD or a BRAID device, with each migration decision respecting the cost and interference properties of the target tier. Because the index structure is not modified, this reorganization is invisible to the search algorithm: a graph traversal or a posting list scan proceeds exactly as before, but the memory accesses

it generates now hit faster storage for the vectors that matter most.

This separation between the index and the placement of the data it references is what distinguishes the OBASE approach from all prior vector-index tiering work. HM-ANN decides placement based on which HNSW layer a node belongs to. CXL-ANNS decides placement based on hop distance from the entry point. Quake restructures the index so that hot partitions become smaller and more numerous, which improves query performance but conflates the indexing concern with the placement concern. Coleman et al. [62] profile edge traversal frequencies across queries and reorder the graph layout for cache locality, but the reordering is a one-time offline transformation that cannot adapt to a changing query distribution. None of these systems cleanly separates the question of how the index organizes search from the question of where the underlying embeddings physically reside in the memory hierarchy.

Applying the OBASE frontend to vector indices requires integrating the guide abstraction with the index's pointer representation. In a graph index, each node stores a neighbor list of node identifiers that are used to look up embedding vectors. The guide would interpose on this lookup, redirecting the identifier to the embedding's current location in the address space after a relocation. In an IVF index, each centroid's posting list contains vector identifiers; the same indirection applies. The cost of this indirection is a pointer dereference per access, which is small relative to the distance computation that follows. The classification policy itself can reuse the approximate counting and feedback-driven cold-threshold mechanisms developed for heap objects in OBASE, since the underlying problem is the same: given a large population of objects with heterogeneous access rates, identify the hot fraction with low overhead.

The empirical motivation for this approach is strong. Query distributions over embedding spaces are skewed in practice: certain topics, entities, or document clusters are queried far more frequently than others, creating

regions of the embedding space that are accessed by a disproportionate share of searches. Quake's measurements confirm that under realistic workloads the majority of partitions receive negligible query traffic [121]. Coleman et al. [62] find sufficient skew in per-edge traversal frequencies within graph indices to justify layout reordering. Recent analysis of HNSW structure shows that a small set of hub nodes is traversed by a large fraction of queries, functioning as highways through the graph [122]. Measurements of SSD-based HNSW indices report that the bottom layer, which contains the vast majority of nodes, achieves only a 59% buffer cache hit ratio even when half the data is cached, while upper layers achieve near-perfect hit rates [136]. These observations all point to the same conclusion: within the large bottom layer or within the posting lists of an IVF index, access intensity varies by orders of magnitude, and the current practice of treating all nodes or all vectors at the same tier uniformly wastes fast memory on cold data while stranding hot data on slow storage.

7.3 Lessons Learned

Throughout this research on data organization for modern memory hierarchies, we have gathered lessons that may be applicable beyond the specific systems presented here. We share them in the hope that they prove useful to others working at the intersection of storage, memory, and systems software.

7.3.1 Model the Device Before Redesigning the Algorithm

When we began studying sorting on byte-addressable storage, the natural starting points were to treat the device as either slower DRAM or faster SSD and to apply existing algorithms accordingly. Both approaches performed poorly, but the reasons were not immediately obvious. Progress came only after we stepped back and characterized the device properties systematically. The BRAID model was the result: a simple enumeration of

five properties that, taken together, distinguish byte-addressable storage from its predecessors. Once these properties were made explicit, the design of WiscSort followed almost directly from checking which properties each design choice respected or violated. The lesson is that when hardware changes in ways that invalidate existing assumptions, the most productive first step is not to modify the algorithm but to model the device. A clear model constrains the design space and makes it possible to evaluate alternatives against a common set of criteria. **Model, Then Build.**

7.3.2 Moving Less Data Outweighs Moving Data Faster

A recurring theme in both WiscSort and OBASE is that reducing the amount of data moved matters more than increasing the rate at which data is moved. WiscSort achieves lower read bandwidth utilization than external merge sort on the same device, yet it finishes in less time because it reads and writes fewer total bytes. Similarly, OBASE does not make page migration faster; it ensures that backends migrate the right pages by presenting them with regions that are uniformly hot or cold. In both cases, the performance gains come not from doing the same work more efficiently, but from avoiding unnecessary work entirely. This principle is not new, but it is easy to lose sight of when optimizing for device throughput. Bandwidth is a finite resource on every storage and memory device, and the most effective way to conserve it is to avoid spending it on data that does not need to move.

7.3.3 Measure the Problem at the Right Granularity

The case for OBASE rested on demonstrating that hotness fragmentation is both severe and unavoidable under static placement. Establishing this required measuring access behavior at a granularity finer than the page, which is the level at which operating systems observe and act. The page utilization metric we introduced captures the fraction of bytes within touched pages

that are actually accessed, bridging the gap between the OS view and the application view. Without this metric, the fragmentation problem would have remained anecdotal rather than quantifiable. Similarly, analyzing production traces from Google, Meta, and Twitter revealed temporal patterns that synthetic benchmarks do not reproduce: object hotness shifts over time, the same allocation site produces objects with different lifetimes, and reuse distances vary widely for individual keys. These measurements shaped OBASE's design, in particular the feedback-driven cold threshold controller, which exists precisely because hotness is not static. The general lesson is that the right metric and the right workload can reveal problems that are otherwise invisible, embodying the principle **"Measure, Then Build"** [39].

7.3.4 Decompose Along Information Boundaries

OBASE is structured as a frontend that understands objects and a backend that understands pages. The frontend knows which objects are hot and cold but does not know which physical tier a page occupies. The backend knows which pages to promote or demote but does not know what those pages contain. This decomposition is not merely an implementation convenience; it reflects a genuine information boundary. The frontend has access to application-level semantics that the OS cannot see, and the backend has access to hardware state that the application cannot see. Respecting this boundary allowed each layer to be designed, tested, and improved independently. The same principle applies in WiscSort, where the sorting algorithm is separated from the device-specific scheduling logic. The sorting phase determines the order of keys without concerning itself with thread-pool sizing or interference avoidance, and the I/O scheduling phase manages device access without concerning itself with sort order. In both systems, the decomposition improved not only modularity but also clarity of design: each component's responsibilities are defined by the information available to it. The principle of building layered services that combine ap-

plication knowledge with OS algorithmic knowledge, without modifying the OS itself, was formalized in the context of gray-box systems [37] and is exemplified by OBASE's design.

7.3.5 Different Approaches Serve Different Regimes

This dissertation began with a static approach (WiscSort) and ended with a dynamic one (OBASE). A natural question is whether one subsumes the other. Our experience suggests that it does not. When application semantics clearly identify which data will be hot and which will be cold, as in sorting where keys are compared and values are not, static layout is simpler, more predictable, and incurs no runtime overhead. Attempting to discover this structure dynamically through observation would add complexity without benefit. Conversely, when access patterns depend on workload behavior and shift over time, as in key-value stores under production traffic, static layout at design time is impossible because the necessary information does not yet exist. The two approaches are complementary rather than competing, and the choice between them is determined by the nature of the application's access structure. Recognizing which regime a given system falls into is itself a useful design heuristic.

7.4 Closing Remarks

The page abstraction succeeded for decades not because it was the right granularity for data placement, but because the cost of getting placement wrong was hidden. When all of main memory was DRAM and all of storage was disk, a misplaced page meant a few wasted cache lines in the common case and a catastrophic disk seek in the worst case. Systems avoided the worst case through generous DRAM provisioning, and the common case was cheap enough to ignore. The emergence of byte-addressable storage and tiered memory changes this arithmetic. Every unnecessary byte read

from a BRAID device wastes scarce write bandwidth that will not be replenished by faster hardware. Every cold object trapped on a hot page in DRAM occupies capacity that could serve a useful purpose on a cheaper tier. The costs are no longer hidden; they are continuous and cumulative.

This dissertation has argued that closing the gap between how applications access data and how systems place it requires layout-aware organization at a granularity finer than the page. We developed techniques for both ends of the spectrum: static layout when application semantics reveal access structure, and dynamic layout when access patterns must be discovered through observation. Neither approach alone is sufficient for all workloads, but together they cover a broad range of practical systems. As memory hierarchies grow deeper and the devices within them grow more diverse, the question of where data resides will only become more consequential. We hope that the models, mechanisms, and principles presented here provide a useful foundation for that work.

Chapter 8

Warehouses Inside Your Computer

In this chapter, I present my research in a form geared towards a general public audience. I hope to make the results of our work more accessible. Access to knowledge is a powerful tool in developing an engaged and compassionate society, and we are proud to contribute in our own small way. Many thanks to the Wisconsin Initiative for Science Literacy (WISL), for providing this opportunity to communicate my work to a broader audience.

8.1 The Warehouse Problem

One number changed how I thought about computers. While analyzing data from Google's servers, I discovered that for every dollar spent on fast memory, as little as four cents was going toward data anyone actually used. The other ninety-six cents paid for dead weight: data sitting in expensive storage that no program had touched in hours, days, sometimes ever. I had expected some waste. I had not expected that kind of waste.

To understand how this happens, think about a warehouse. You tap "buy" on your phone, and a package shows up the next day. That speed is possible because the warehouse stored products in the right places. Bestselling items sit on shelves right next to the shipping dock, ready to be grabbed and loaded onto a truck in seconds. Rarely ordered items live in back storage, where shelf space is cheaper and nobody minds the longer walk.

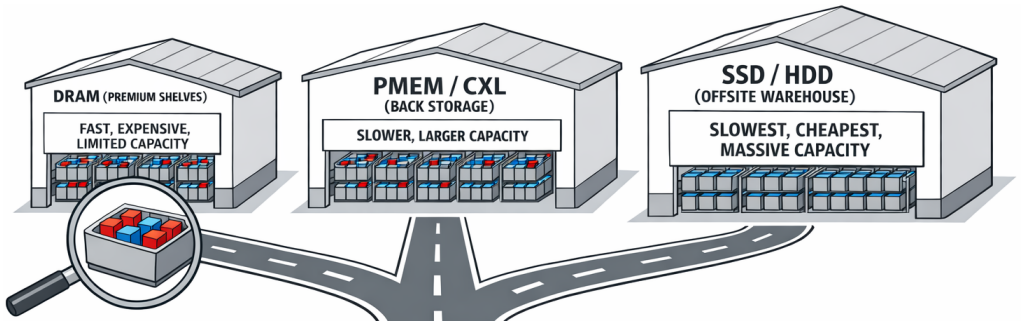


Figure 8.1: **The warehouse problem.** Your computer's memory works like a chain of warehouses, from a premium facility next to the shipping dock (fast, expensive, limited) to back storage and offsite warehouses (slower, cheaper, larger). The operating system moves data between them in fixed-size bins called *pages*. The magnified bin in the lower left shows the core problem: because the manager can only move whole bins, a single popular product (red) traps an entire bin full of unpopular products (blue) in premium space, wasting expensive shelf real estate on data nobody is using.

This simple principle, put popular things where they are easy to reach, seems obvious. But what happens when the warehouse is disorganized? Imagine that products are shelved based on *when* they arrived, not *how often* they are ordered. A best-selling phone case ends up in the same bin as a novelty item nobody wants. The warehouse manager can only move *entire bins*, never individual products inside them. So a single popular product traps its entire bin in premium space near the dock, wasting expensive shelf real estate on products nobody is ordering. Meanwhile, popular items stuck in back-storage bins take ages to retrieve (Figure 8.1).

This is exactly the problem your computer's memory faces. Modern computers do not have just one kind of memory. They have a hierarchy, ranging from small, fast, and expensive memory at the top to large, slow, and cheap memory at the bottom. The operating system, which manages all of this memory, works like the warehouse manager: it can move data around, but only in fixed-size chunks called *pages*. It cannot reach inside a page to move one piece of data without moving everything else on that page. The

premium shelves correspond to your computer's fast memory (a technology called DRAM), and back storage corresponds to slower, cheaper memory technologies. Each bin is a memory page, and each product is a piece of data.

These different kinds of memory do not merely differ in speed. They differ in character. Some can hand you a single byte; others insist on giving you a whole block whether you want it or not. Some read quickly, but write data slowly. Some slow down when reading and writing happen at the same time. An approach that works beautifully on one kind of storage can perform terribly on another, the way a shelving strategy designed for a walk-in closet falls apart in a three-story warehouse. Any serious attempt to organize data must account for the specific properties of the hardware it lives on.

When popular data, data your programs touch all the time, gets mixed with unpopular data on the same pages, the result is waste on a staggering scale. This is the problem I mentioned at the start. Across real workloads at companies like Google and Meta, I found that for every 100 megabytes of memory the operating system considers "active," up to 96 megabytes is actually cold data that nobody is using. It sits trapped in expensive fast memory by a few hot neighbors, just like that novelty gift trapping its whole bin next to the shipping dock.

My dissertation asks a simple question: ***can we reorganize how data is laid out in memory so that the operating system's fixed-size view of the world actually works well?*** The answer, it turns out, is yes. But the right approach depends on what you know about the data or how it will be accessed.

Sometimes you know in advance which data will be popular and which will not. A sorting algorithm, for instance, compares small labels (called *keys*) over and over again but only touches the full records (called *values*) at the very end. In that case, you can plan ahead: keep the labels close at hand and leave the bulky records in cheap storage until you need them. But planning ahead also means understanding the hardware you are planning for. A new kind of computer storage arrived in recent years with a set of

properties no previous device had combined: it could read tiny amounts of data from random locations almost as fast as reading sequentially, but writes were far slower than reads, and reading and writing at the same time caused both to degrade. I built *WiscSort* [45], a sorting system designed around these specific properties. It separates the lightweight labels from the bulky products and sorts only the labels. Then it exploits the fast random reads of modern storage to fetch the products at the very end. It also carefully schedules reads and writes so they stay out of each other's way. The result is two to three times faster than the best prior approaches.

Other times, you have no idea what will be popular. A social media platform cannot predict which posts will go viral. A database does not know which records a user will search for next. In these cases, the only option is to watch and reorganize: monitor which data is being accessed, and periodically move popular data to fast memory and unpopular data to slow memory. Think of a warehouse crew that tracks which products are selling and re-sorts the shelves every night.

In a physical warehouse, this reorganization is straightforward. You pick up a box, carry it to a new shelf, and set it down. If a coworker asks where the box went, you tell them. In a computer, the situation is fundamentally harder. Dozens of programs may be reading a piece of data at the same instant, and each of them holds an address, a note that says "this data lives at location 7042." If you move the data to location 2058 while a program is in the middle of reading location 7042, that program gets nonsense. If you freeze every program while you move the data, you destroy performance. The reorganization must happen while the warehouse is open and busy, without anyone ever following an address to the wrong place. I built a system called *OBASE* [46, 47] that solves this problem. It observes which data is hot and reorganizes the layout continuously. At the old address of every moved object, it installs a forwarding label, so any program still holding the old address is silently redirected to the new one. No program

ever pauses, and no program ever finds stale data. The result is up to 70% less wasted memory, with almost no slowdown.

In the rest of this chapter, I will walk you through how your computer's memory actually works, why the current approach wastes so much of it, and how WiscSort and OBASE fix the problem. All you need is the warehouse in your mind.

8.2 Why Bins?

If mixing popular and unpopular products together causes so much waste, the obvious question is: why not track individual products instead of bins? If the warehouse manager knew the exact shelf location of every single product, she could move just the popular ones to premium shelves and leave everything else in back storage. No bins, no wasted space, no problem.

The answer is the same in a warehouse and in a computer: keeping track of everything individually is too expensive. A large warehouse might stock tens of millions of products of different sizes. A master catalog listing the exact shelf location of every single one would fill an entire room, and a worker would spend more time flipping through the catalog than actually retrieving products. The solution is the same one librarians have used for centuries: group items into sections, and keep a much smaller catalog that tracks sections rather than individual books.

In a computer, this smaller catalog is a piece of hardware called the *translation lookaside buffer*, or TLB for short. You can think of it as a small index-card box bolted to the front desk of the warehouse. Every time a worker (the processor) needs a piece of data, it first checks the catalog to find which bin the data lives in and where that bin sits on the shelves. If the bin is listed, the worker gets an answer almost instantly, in about a billionth of a second. If not, the worker has to walk to the back office and search through a much larger set of records. That detour is about a hundred times slower.

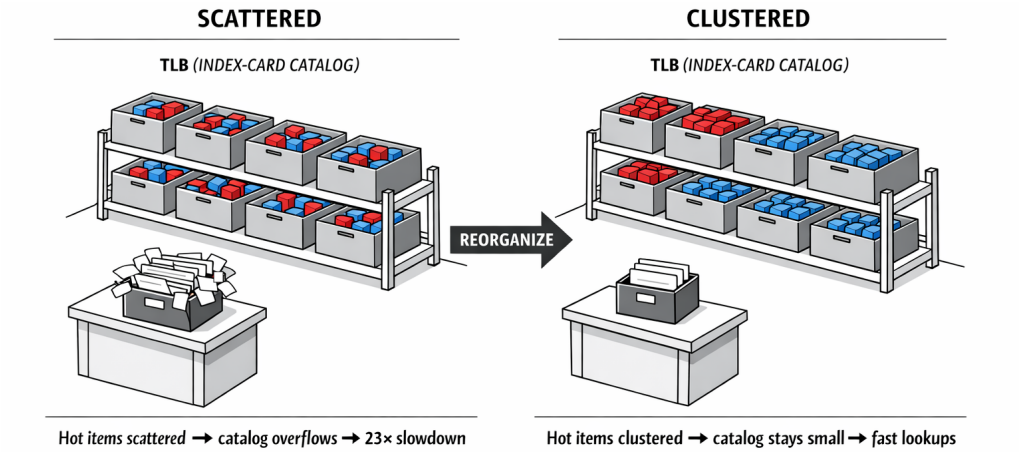


Figure 8.2: **The catalog problem.** When popular products (red) are scattered across many bins, the index-card catalog must track every bin and overflows. When popular products are clustered into just a few bins, the same catalog holds everything it needs with room to spare. The data and the work are identical in both panels; only the *layout* has changed.

The catalog can only hold a limited number of entries, typically a few thousand. When the data a program is actively using fits into a small number of bins, the catalog works beautifully: it holds every bin the worker might need, and lookups are nearly free. But when popular data is scattered across many bins, each containing just a few hot items mixed with cold ones, the catalog overflows. The worker constantly looks up a bin, finds it missing, walks to the back office, retrieves the location, evicts an old entry from the full card box, and repeats. In one of our analysis, this kind of catalog overflow caused a program to run *twenty-three times slower* than it did when the same data was clustered into fewer bins. The program was doing the exact same work on the exact same data. The only difference was where the data had been placed.

This is the hidden cost of the bin system. Bins make the catalog small enough to be practical, but they also create a rigid constraint: the operating system sees bins, not products. If popular products are scattered across

many bins, two bad things happen at once. First, the premium shelves fill up with bins that are mostly empty space, the memory waste I discussed in the last section. Second, the catalog overflows, and the processor grinds to a halt looking up bin locations. The two problems reinforce each other, and both share the same root cause: popular and unpopular data mixed together on the same pages.

This also means that any system that clusters popular data into fewer bins earns a double benefit. It frees up premium shelf space *and* it shrinks the catalog back to a manageable size. Both WiscSort and OBASE do exactly this, and both reap the rewards. I will show how in the next two sections.

8.3 Not All Warehouses Are the Same

In Sec. 8.1, I described a sorting algorithm that separates lightweight labels from bulky products and sorts only the labels. That is the core idea behind WiscSort. But to understand *why* it works so well, you need to understand the hardware it was designed for, because that hardware breaks nearly every assumption older sorting systems relied on.

For decades, computer storage meant hard drives: spinning metal platters with a tiny mechanical arm that reads data by physically sliding across the surface, the way a record player's needle traces a groove. Reading data in order, one piece right after the next, was fast because the arm barely had to move. But jumping to a random location meant the arm had to swing across the platter, wait for it to spin to the right spot, and only then start reading. That random jump was roughly a *hundred thousand times* slower than simply reading the next piece in sequence. Because of this, every sorting algorithm built for hard drives followed the same basic strategy: always read and write sequentially, and never, ever jump around.

Then a new kind of storage arrived, one that blurred the line between memory and disk. Intel's Optane was the most prominent example. In ware-

house terms, imagine a new storage facility where the layout has changed in five unexpected ways. First, you can grab a single product off a shelf without pulling the entire bin (*byte addressability*). Second, sending a worker to grab one item from a random shelf is nearly as fast as having them walk down the aisle picking up items in order (*fast random reads*). Third, putting products *onto* shelves is much slower than taking them off; reads are about three times faster than writes (*asymmetric costs*). Fourth, if workers are simultaneously loading shelves and unloading them, they get in each other's way and both slow down (*read-write interference*). Fifth, adding more workers helps only up to a point, especially for loading; beyond that, they start bumping into each other (*constrained concurrency*).

No previous device had combined all five of these properties. Hard drives were slow at everything but predictable. Solid-state drives were faster but still insisted on handing you a whole block at a time. This new storage was fast and flexible in some ways, painfully constrained in others. The key insight is that designing for one or two of these properties is not enough. A system that exploits fast random reads but ignores the interference between reads and writes will sabotage itself. WiscSort was designed around all five simultaneously. (In our technical work, I named this combination of properties the *BRAID* model, after the initials of each property, but the name matters less than the idea: you have to respect the whole package.)

Here is how it works in practice (Figure 8.3). Suppose you have a dataset of one hundred million records, each consisting of a 10-byte label and a 90-byte product. A traditional sorting algorithm reads the full 100 bytes of every record, shuffles them around repeatedly during sorting, and writes them all back. WiscSort reads only the 10-byte labels, each paired with a tiny pointer noting where the corresponding product lives in storage. It sorts these label-pointer pairs entirely in fast memory. Only at the very end does it use the device's fast random reads to fetch each product from its original location and write the final sorted result.

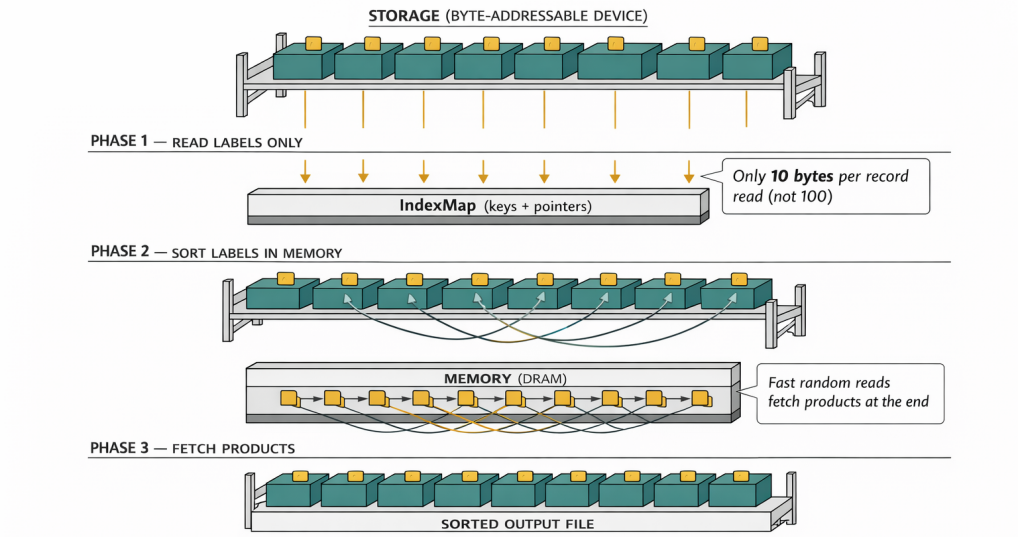


Figure 8.3: **How WiscSort works.** Traditional sorting carries each product (value) alongside its shipping label (key) through every stage. WiscSort separates the two. In the first phase, it reads only the lightweight labels from storage using fine-grained byte accesses, and sorts them in memory. In the final phase, it uses the fast random reads of modern storage to fetch the corresponding products and assemble the sorted output. Because labels are far smaller than products (often one-tenth the size), this dramatically reduces the data that must be read and written.

The difference in data movement is stark. For those hundred million records, traditional sorting reads and writes around 19 gigabytes of data, most of it products being carried along for the ride. WiscSort reads only 1.9 gigabytes of labels during sorting and fetches the products in one final pass. That is a tenfold reduction in intermediate traffic. Because writing is the slowest operation on this hardware, moving less data translates directly into faster completion.

But reducing traffic is only half the story. WiscSort also manages *how* it accesses the device. At startup, it runs a brief calibration step to measure the device's actual performance: how many workers can read at once before throughput plateaus, how many can write, and when interference kicks in.

Based on these measurements, WiscSort assigns separate pools of workers for reading and writing, each sized to match the device’s sweet spot. Crucially, it never lets reading and writing happen at the same time. When data is being loaded from storage, no writing occurs; when sorted data is being written back, all reading pauses. This is the interference-aware scheduling that the intro alluded to, and it prevents the mutual degradation that costs other systems up to fifty percent of their throughput.

I tested WiscSort on the industry-standard sorting benchmark, which stress-tests storage systems by sorting records with small labels and large values. WiscSort’s one-pass version finished three times faster than a competitive traditional approach. Even the two-pass version, needed when the dataset is too large to sort in one shot, was twice as fast. Compared to the best prior system designed for this kind of hardware, WiscSort was seven times faster. The gap comes from treating the device’s five properties as a unified design constraint rather than addressing them one at a time.

8.4 Reorganizing While the Warehouse Is Open

WiscSort works because the task itself reveals which data is hot and which is cold: labels are always hot, products are cold until the end. But most software does not come with that kind of road map. A key-value store (a system that looks up records by name, like a phone book) serving a social media feed, a caching layer at a major web company, a database answering search queries: in all of these, any piece of data might suddenly become popular or fall out of use, and the pattern shifts constantly. I looked at real usage logs from Meta and Twitter and found that for most items, the time between one use and the next swings a lot; over five times for three-quarters of items, and over thirty times for nearly two-thirds. Nothing about when an item was created or where it was allocated tells you how popular it will be next week.

When you cannot predict popularity, you must observe it. OBASE does

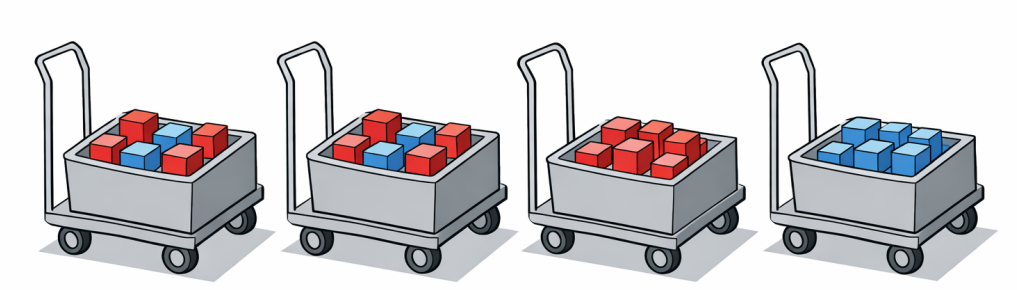


Figure 8.4: **Before and after OBASE.** The left two bins show the typical state of memory: popular products (red) and unpopular products (blue) mixed together on every page. Because the warehouse manager can only move whole bins, every bin must stay in premium space as long as it contains even one popular item. The right two bins show the same data after OBASE has reorganized it: popular products are clustered into one bin, unpopular products into another. Now the cold bin can be safely moved to back storage, freeing premium shelf space without losing any popular items.

this by continuously watching which data is being accessed, classifying it as hot or cold, and physically reorganizing the layout so that hot objects cluster onto the same pages and cold objects cluster onto others (Figure 8.4). The warehouse manager, who can still only move whole bins, now finds that cold bins are *genuinely* cold: every product inside is unused, so the entire bin can be safely moved to back storage. Hot bins are densely packed with popular products, making the premium shelves far more productive.

This sounds straightforward in principle. In practice, it requires solving four problems that have no easy parallels in a physical warehouse.

Forwarding labels. The first problem is that the most widely used programming languages for building fast, performance-critical software (languages called C and C++) give programmers direct control over where data lives in memory. That power comes at a cost: every part of the program that uses a piece of data holds a note saying "this lives at address 7042." If you move the data to address 2058, every one of those notes becomes wrong,

and there is no built-in mechanism to fix them. Some newer programming languages (like Java or Python) trade away that direct control in exchange for a safety net. They insert a manager between the program and memory, and when data moves, the manager quietly updates all the notes for you. C and C++ have no such manager. The programmer is talking directly to the hardware, which is exactly what makes these languages fast, and exactly what makes reorganizing data so hard.

OBASE introduces a lightweight intermediary I call a *guide*. Instead of pointing directly at the data, every reference points at the guide, and the guide holds the data's current address. When the data moves, only the guide needs to be updated: one single value in one place. Every part of the program that follows the guide is silently redirected to the new location. The guide is the forwarding label from our warehouse analogy: a small card left at the old shelf location that says "this product is now on shelf B-14." Any worker who arrives at the old location reads the card and walks to the new one without even realizing the product was moved.

The cost of this indirection is small. Each time a program accesses data through a guide, it performs one extra lookup, around five billionths of a second, comparable to reading a value already sitting in the processor's fastest cache. For most programs, this adds between two and five percent to the total running time.

Tracking popularity. The second problem is knowing which data is hot and which is cold. OBASE cannot afford to maintain a separate log of every access; at the rate modern processors touch data, such a log would itself become a bottleneck. Instead, OBASE hides a tiny tracking flag inside the guide itself. Here is the trick: every guide contains an address, a string of digits that tells the processor where to find the data. But the address is longer than it needs to be, the way a phone number might have unused digits at the end. OBASE repurposes a few of those spare digits to record

whether the data has been accessed recently. Every time a guide is followed, one of those spare digits is flipped. No separate data structure, no external log, no additional memory. The tracking rides along with the access itself, like a turnstile counter built into the warehouse doorframe that clicks each time a worker walks through.

A background process I call the *Object Collector* periodically reads these flags, about once every two minutes by default, and classifies each object as hot or cold based on recent activity. Objects that have not been accessed for several consecutive windows are marked cold. Objects accessed in any window are marked hot. The threshold for "cold enough to move" adjusts automatically: if the system finds that it is moving objects that soon get accessed again, it becomes more conservative; if almost nothing it moves is ever revisited, it becomes more aggressive. This self-tuning means OBASE adapts to each workload without manual configuration.

Three zones. Based on these classifications, OBASE maintains three zones in memory: NEW, for freshly created data whose popularity is not yet clear; HOT, for the current working set; and COLD, for data that has been idle long enough to be considered inactive. Each zone occupies a contiguous stretch of memory, so the operating system sees clean, uniform regions. A cold zone is a cold zone: every object inside is genuinely idle, and the entire region can be moved to slower memory or reclaimed with confidence. This is the key difference from the status quo, where the operating system looks at a page and sees a mix of hot and cold data, unable to act without risking the hot items.

Data flows between zones as its popularity changes. A freshly created object starts in NEW. If the Object Collector sees it being accessed, it migrates to HOT. If it sits idle for several windows, it migrates to COLD. And if a cold object becomes popular again, it moves back to HOT. The layout is never static; it continuously tracks the application's shifting working set.

Moving products without closing the warehouse. The hardest challenge is the last one: how do you move data while programs are actively reading it? Pausing every program while you move an object, the way a garbage collector might in Java, is unacceptable in the performance-critical systems OBASE targets. But moving data out from under an active reader could corrupt the program's state.

OBASE uses a "try and check" approach (database researchers call it *optimistic concurrency control*, but the intuition is simple). Assume the move will succeed, do the work, and verify at the end that nothing went wrong. Concretely, the Object Collector copies the data to its new location in the target zone. Then it attempts to update the guide in a single all-or-nothing operation: either the update succeeds completely, or it fails completely, with no in-between state. If any program accessed the data while the copy was happening, the guide will have changed, the update fails, and the move is simply abandoned. The data stays where it was, perfectly intact. The program that was reading it never noticed a thing.

This "try and check" approach has a beautiful property: programs never wait on the reorganization crew, and the reorganization crew never corrupts data. Frequently accessed objects naturally resist being moved, because someone is always reading them, and their guide changes before the move can commit. Cold objects, by definition, are not being read, so their moves almost always succeed. The system migrates exactly the objects that can safely be migrated, without anyone having to coordinate.

Letting the manager do her job. One final design choice deserves emphasis, because it is the reason OBASE works with so many different systems. OBASE does not decide what to do with cold data. It does not page it to disk, compress it, or send it to slower memory. All it does is reorganize the address space so that cold data and hot data are no longer mixed together. The existing warehouse manager, whether that is Linux's built-in memory

reclaimer, Meta's TMO system, or a hardware-aware tiering engine like Memtis, continues to make the actual decisions about which bins to move where. But now, when the manager looks at a bin and asks *is everything in here cold?*, the answer is reliably yes.

This separation is what makes OBASE practical to deploy. A large company does not need to replace its existing memory management infrastructure. It plugs OBASE in as a frontend that improves the layout, and every backend it already uses becomes more effective. I tested OBASE with six different memory management systems and every single one improved. Some improved dramatically. Linux's built-in reclaimer, without OBASE, can reduce memory usage from 13 gigabytes to about 7 by reclaiming only the pages it is confident are cold. With OBASE preparing the layout, the same reclaimer reaches 4 gigabytes, the theoretical minimum, with no loss in performance. The reclaimer did not get smarter. It just finally had accurate information to work with.

In tiered memory configurations, where a computer has a small amount of fast memory and a larger pool of slower memory, OBASE is equally effective. Without OBASE, the popular data is spread across so many pages that it cannot fit in the fast tier, even when the actual amount of popular data is small. With OBASE compacting the popular data into fewer pages, the fast tier has room to spare. In our experiments, OBASE allowed the system to achieve the same performance with *half* the fast memory, effectively doubling the capacity of the expensive tier for free. Across all workloads and configurations, the overhead of running OBASE amounted to between two and five percent of total running time. That includes the guides, the tracking, and the migration. The overhead stayed flat whether the system had two workers or thirty-two.

For a representative workload, a key-value store holding ten million records, the baseline system used 12.4 gigabytes of memory. After OBASE reorganized the layout, the same workload ran in 3.5 to 4 gigabytes, a re-

duction of about seventy percent. The application did the same work on the same data. Only the arrangement changed.

8.5 The Messy Middle

Research, as it appears in papers and dissertations, looks like a clean march from question to answer. It is not. The path from "memory is organized badly" to WiscSort and OBASE was full of dead hardware, impossible measurements, and a problem that an entire field had avoided for decades. I want to describe some of that messiness, because it shaped the work in ways the technical sections cannot convey.

The hardware vanished. WiscSort was designed for Intel Optane, a pioneering persistent memory technology that combined the persistence of an SSD (a solid-state drive, the kind of storage that holds your files even when the computer is off) with something approaching the speed of fast memory. I spent years studying its properties, building our five-property model around its quirks, and tuning WiscSort to squeeze every bit of performance from it. Then Intel discontinued Optane. The hardware our entire sorting system was designed for was no longer being manufactured.

This could have been a disaster. Instead, it forced us to ask a more interesting question: which of WiscSort's design principles are specific to Optane, and which generalize to future hardware? To find out, I built emulated devices with different combinations of properties. I could dial up or down the asymmetry between reads and writes, change the interference behavior, adjust the random-read penalty. The result was a map of when each design choice matters. Separating labels from products helps whenever products are larger than labels, regardless of the device. Interference-aware scheduling matters enormously on devices where reads and writes compete for the same internal resources, but adds little when they do not. The worker-pool

controller adapts automatically to whatever concurrency constraints the device imposes.

When CXL memory, a next-generation technology for connecting pools of slower memory to a computer, began to emerge, I found that our model described its properties just as well as it had described Optane's. WiscSort's principles transferred. Losing the hardware I had built for turned out to be the push I needed to build something less fragile.

Measuring the invisible. Before I could fix the mixing problem, I had to prove it existed, and that turned out to be far harder than we expected. Measuring which bytes of memory are actually accessed requires tracking every single load and store instruction a processor executes. For a production workload running for minutes or hours, this generates terabytes of raw trace data. Existing tools could not keep up: they either slowed the workload to a crawl, making the trace unrepresentative, or sampled too sparsely to give accurate per-page utilization numbers.

I had to build my own analysis infrastructure. My trace processing tools were themselves concurrent systems. They parsed billions of memory accesses, annotated each one with its page address and size, and computed utilization statistics across millions of pages, all while keeping their own memory usage and runtime manageable. The engineering required to *study* the problem was nearly as involved as the engineering required to solve it.

When the numbers finally came in, they were more extreme than I had imagined. The Google production traces I analyzed showed that the median page, the typical page in a running workload, used only eight to fifty percent of its capacity. With huge pages, the numbers were even more stark: eighty-five to ninety percent of pages utilized less than ten percent of their space. I had expected some waste. I had not expected that for every dollar spent on fast memory, as little as four cents was going toward data anyone was actually using. These numbers became the foundation of the argument

for OBASE, but reaching them required months of tooling work that never appeared in the final paper.

I analyzed Google's traces because they were among the few production memory traces available to the research community. I expect similar fragmentation in most large-scale deployments (Meta, Amazon, Microsoft, etc), since the root cause, allocators grouping objects by allocation time rather than access pattern, is universal. But confirming this across the industry would require more organizations to share their traces, which remains rare.

The problem nobody had solved. The core technical challenge of OBASE, relocating objects in C and C++ while programs are actively using them, is not new as a concept. Garbage collectors in managed languages like Java have moved objects for decades. They can do this because the language runtime controls every reference to every object. When the garbage collector moves an object, it updates all the references automatically, and the program never sees raw memory addresses to begin with.

C and C++ offer no such infrastructure. A pointer in C is a raw memory address, nothing more. The language provides no way to find every pointer to a given object, no way to intercept when a pointer is followed, and no runtime that could update references on your behalf. These languages have been in widespread use for over fifty years. In all that time, while significant work had gone into automatic *freeing* of memory (detecting when objects are no longer needed), no practical system had been proposed for automatic *relocation* of objects based on how frequently they are accessed.

This was the problem we stared at for a long time. The guide abstraction, the mechanism that finally made relocation possible, emerged from the realization that we did not need to find and update every pointer to an object. We only needed to ensure that every path to an object went through a single stable intermediary. If the intermediary always knew the current address, and if updating the intermediary was atomic (all-or-nothing, with

no half-finished state), then we could move the object and update one value in one place. The compiler could enforce that all access went through guides, and the "try and check" approach could guarantee that no program ever observed a stale address. It sounds straightforward in retrospect. Arriving at it was not.

Making it real. Even after the core mechanism worked, demonstrating that OBASE was practical required building far more than a prototype. Real systems use a remarkable variety of data structures, and each one has its own strategy for handling the fact that multiple workers may try to read or change the same data at the same time. Some systems solve this by closing the entire warehouse while one worker makes a change. Others lock only the specific shelf being touched. Still others use clever tricks to let everyone work simultaneously without locks at all. If OBASE's reorganization conflicted with any of these strategies, it would be useless in practice. I implemented and tested ten different concurrent data structures, from lookup tables used in systems like Redis and NGINX to tree-structured indexes used in databases like PostgreSQL and DuckDB. OBASE had to work with all of them without disrupting their existing coordination strategies.

Proving that the benefits held under realistic conditions meant testing OBASE against recordings of real activity from Meta and Twitter's production caching systems. These recordings captured the full messiness of real use: popularity shifting from one item to another, data being read, updated, and deleted in unpredictable combinations, and item popularity evolving over hours. It is easy to show that a system works on a tidy, artificial test. The real question is whether it works when the patterns are as chaotic as they are in situations closer to practice.

Finally, when OBASE successfully reorganized memory and produced clean cold regions ready for reclamation, I discovered a bottleneck I had not anticipated: the Linux kernel, the core piece of software that sits be-

tween every program and the hardware (PC, phones, etc), acting as the ultimate warehouse manager. When the kernel reclaims a large region of memory by moving it to slower storage, it processes each page one at a time. For every single page, it sends a separate notification to every processor core, telling each one to discard its cached location for that page. For a ten-gigabyte cold region, this meant millions of notifications. The reclamation path, which should have been the easy part, became the slowest step in the pipeline. I modified the kernel to batch these operations, bundling hundreds of pages into a single notification. The fix reduced the number of these inter-processor notifications by over ninety-nine percent and unblocked the entire system. It was a reminder that improving one layer of a complex system often reveals bottlenecks in the next.

8.6 What It All Means

Memory is the single most expensive component in a modern data center, accounting for close to half the cost of a server and a substantial share of its energy consumption. Fast memory carries about twelve times the carbon footprint per bit compared to flash storage, both in the energy needed to manufacture it and the electricity needed to keep it powered around the clock. When companies overprovision fast memory, paying for gigabytes that hold data nobody is accessing, the cost is not just financial. Every wasted gigabyte draws power continuously, adds heat that must be removed by cooling systems that draw still more power, and will eventually be replaced by new hardware that required energy and raw materials to produce. As data centers draw more power worldwide, wasting memory wastes energy too.

The work in this dissertation offers a different path. WiscSort and OBASE are software systems. They require no new hardware, no changes to the operating system's core memory manager, and in the case of OBASE, minimal changes to application code. They work by being smarter about *arrangement*:

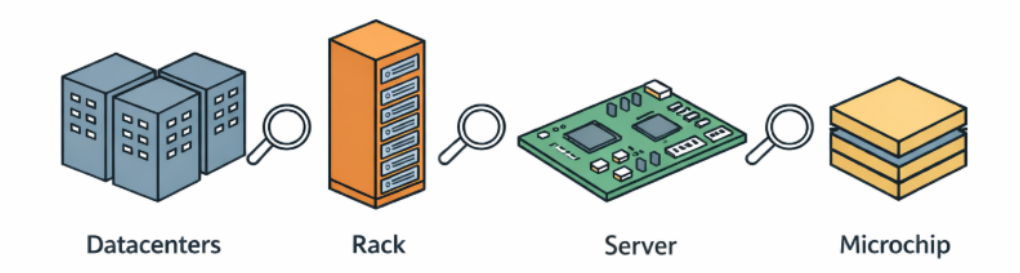


Figure 8.5: **Data organization at every scale.** From datacenters down to individual microchips, the speed of light is constant: accessing nearby data is always faster than reaching for something far away. At every level of this hierarchy, the same principle applies. If you can keep the data that is popular right now physically close to where it is needed, performance improves and resource waste decreases. WiscSort and OBASE demonstrate this principle at the memory and storage layers, but the idea applies wherever data lives.

ensuring that the data the system actually uses is stored where it can be reached quickly, and the data it does not use is stored where it is cheap to keep. WiscSort does this at design time, by understanding the hardware well enough to separate hot data from cold before the work begins. OBASE does it at run time, by watching what the application touches and continuously reshaping the layout in response.

Underneath both systems lies a single insight that I think extends well beyond memory management. The address space, the map that tells the processor where to find each piece of data, is not just bookkeeping. It is a communication channel between the application and the operating system. Today, that channel is mostly silent: applications allocate memory and the operating system manages pages, but neither side tells the other anything about what the data means or how it is used. OBASE shows that when applications speak up, when they organize their data so that pages reflect actual usage patterns, the operating system's existing tools become dramatically more effective without any modification.

This principle could reshape how we think about memory well beyond

the specific problems in this dissertation. Any system that manages objects in memory, whether it is a database, a web browser, or a programming language's garbage collector, faces the same mismatch between what it knows about its data and what the operating system can see. Bridging that gap, letting the software that *understands* the data communicate with the hardware that *moves* it, is a lever that has barely been pulled.

There is more to do. Better hardware support for tracking which data is actually being used could shrink even the small overhead our software-based approach introduces. And the measurement challenge persists: we need more organizations to share real traces of how their systems use memory, so researchers can study the problem as it actually exists rather than as we assume it does.

But even today, the numbers tell a clear story. For every dollar the industry spends on fast memory, the vast majority is wasted on data nobody is touching. WiscSort [45] and OBASE [46, 47] show that we can recover that waste using nothing but smarter arrangement, no new hardware, just a better understanding of what goes where. In an era when data centers consume a growing share of global electricity, using the memory we already have more wisely is not just a scientific challenge. It is a responsibility.

Bibliography

[1] Intel 80386 programmer's reference manual, 1986. URL https://pdos.csail.mit.edu/6.828/2018/readings/i386.pdf.

[2] Level db, 2011. URL https://github.com/google/leveldb.

[3] Sqlite virtual database engine external sorter, 2011. URL https://github.com/sqlite/sqlite/blob/master/src/vdbesort.c.

[4] Rocks db, 2012. URL http://rocksdb.org/.

[5] Cxl and gen-z iron out a coherent interconnect strategy, 2020. URL https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherentinterconnect-strategy/.

[6] Road to 128-bit linux, 2022. URL https://lwn.net/Articles/908026/.

[7] Apache Avro. https://avro.apache.org/, Sep 2022. [Accessed 26-Sep-2022].

[8] Compute express link. https://www.computeexpresslink.org/, Sep 2022.

[9] Data Lakehouse Platform by Databricks. https://www.databricks.com/product/data-lakehouse, Sep 2022. [Accessed 26-Sep-2022].

[10] gensort data generator, Sep 2022. URL http://www.ordinal.com/gensort.html.

[11] Google cloud data lake modernization solutions. https://cloud.google.com/solutions/data-lake, Sep 2022. [Accessed 26-Sep-2022].

[12] Heterogeneous memory attribute table, Sep 2022. URL https://lwn.net/Articles/724562/.

[13] Intel optane dc pmm, Sep 2022. URL https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[14] Intel optane dc persistent memory start up guide, Sep 2022. URL https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[15] Kioxia launches second generation of high-performance, cost-effective xl-flash storage class memory solution, Sep 2022. URL https://www.businesswire.com/news/home/20220801005862/en/Kioxia-Launches-Second-Generation-of-High-Performance-Cost-Effective-XL-FLASH%E2%84%A2-Storage-Class-Memory-Solution/. [Accessed 26-Sep-2022].

[16] Key-length-value format., Sep 2022. URL https://en.wikipedia.org/wiki/KLV.

[17] Postgresql storage format, Sep 2022. URL https://www.postgresql.org/docs/current/storage-toast.html#STORAGE-TOAST-ONDISK.

[18] Postgress parallel external sort, Sep 2022. URL https://wiki.postgresql.org/wiki/Parallel_External_Sort.

[19] Samsung shows off cxl server memory expander, Sep 2022. URL https://www.nextplatform.com/2022/08/23/samsung-shows-off-cxl-server-memory-expander/.

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

[21] Sort benchmark home page., Sep 2022. URL http://sortbenchmark.org/.

[22] Sqlite record format, Sep 2022. URL https://www.sqlite.org/fileformat2.html#record_format.

[23] Damon: Data access monitor, 2024. URL https://sjp38.github.io/post/damon/.

[24] Battling the prefetcher: Exploring coffee lake, 2024. URL https://abertschi.ch/blog/2022/prefetching/.

[25] Advanced profiling topics. pebs and lbr, 2024. URL https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR.

[26] Pin - a dynamic binary instrumentation tool, 2024. URL https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[27] Dynamorio: Dynamic binary instrumentation framework, 2025. URL https://dynamorio.org/.

[28] Google workload traces version 2. https://console.cloud.google.com/storage/browser/external-traces-v2, 2025.

[29] Overview of package util.concurrent, 2025. URL https://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[30] Vm linux kernel doc. https://docs.kernel.org/admin-guide/sysctl/vm.html, 2025.

[31] Llvm-dev rfc:sanitizer-based heap profiler. https://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html, 2025.

[32] Tagged pointers. https://en.wikipedia.org/wiki/Tagged_pointer, 2025.

[33] Tcmalloc leveraging hot/cold hints. https://google.github.io/tcmalloc/temeraire.html#leveraging-hotcold-hints, 2025.

[34] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. 31(9):1116–1127, sep 1988. ISSN 0001-0782. doi: 10.1145/48529.48535. URL https://doi.org/10.1145/48529.48535.

[35] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, DaMoN'22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393782. doi: 10.1145/3533737.3535090. URL https://doi.org/10.1145/3533737.3535090.

[36] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM*, 62(11):114–120, October 2019. ISSN 0001-0782. doi: 10.1145/3318163. URL https://doi.org/10.1145/3318163.

[37] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 43–56, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133898. doi: 10.1145/502034.502040. URL https://doi.org/10.1145/502034.502040.

[38] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, page 243–254, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919114. doi: 10.1145/253260.253322. URL https://doi.org/10.1145/253260.253322.

[39] Remzi Arpaci-Dusseau. Measure, then build. Renton, WA, July 2019. USENIX Association.

[40] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[41] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, 2017. URL https://db.cs.cmu.edu/papers/2017/p1753-arulraj.pdf.

[42] Astera Labs. Astera Labs' Leo CXL smart memory controllers on Microsoft Azure M-series virtual machines overcome the memory wall. https://www.asteralabs.com/news/astera-labs-leo-cxl-smart-memory-controllers-on-microsoft-azure-m-series-virtual-machines-overcome-the-memory-wall/, November 2025. Published Nov. 18, 2025.

[43] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310970. doi: 10.1145/2254756.2254766. URL https://doi.org/10.1145/2254756.2254766.

[44] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (ips^{4}o), 2017. URL https://arxiv.org/abs/1705.02257.

[45] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wiscsort: External sorting for byte-addressable storage. *Proc. VLDB Endow.*, 16(9): 2103–2116, May 2023. ISSN 2150-8097. doi: 10.14778/3598581.3598585. URL https://doi.org/10.14778/3598581.3598585.

[46] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. Tidying up the address space. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems*, SOSP '25, page 63–72. ACM, October 2025. doi: 10.1145/3764862.3768179. URL http://dx.doi.org/10.1145/3764862.3768179.

[47] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. Obase: Object-based address-space engineering to improve memory tiering, 2026. URL https://arxiv.org/abs/2603.00378.

[48] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *SIGARCH Comput. Archit. News*, 38 (3):48–59, jun 2010. ISSN 0163-5964. doi: 10.1145/1816038.1815970. URL https://doi.org/10.1145/1816038.1815970.

[49] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Third Edition*. 2018. URL http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024.

[50] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proc. VLDB Endow.*, 14(9):1544–1556, may 2021. ISSN 2150-8097. doi: 10.14778/3461535.3461543. URL https://doi.org/10.14778/3461535.3461543.

[51] 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. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/berg.

[52] Mohammed Bey Ahmed Khernache, Arezki Laga, and Jalil Boukhobza. Montres-nvm: An external sorting algorithm for hybrid memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 49–54, 2018. doi: 10.1109/NVMSA.2018.00013.

[53] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1):25–36, jun 2004. ISSN 0163-5999. doi: 10.1145/1012888.1005693. URL https://doi.org/10.1145/1012888.1005693.

[54] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. *CoRR*, abs/1603.03505, 2016. URL http://arxiv.org/abs/1603.03505.

[55] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[56] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196898. URL https://doi.org/10.1145/3183713.3196898.

[57] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/chen-jiqiang.

[58] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: highly-efficient billion-scale approximate nearest neighbor search. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

[59] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*, January 2011. URL https://www.microsoft.com/en-us/research/publication/rethinking-database-algorithms-for-phase-change-memory/.

[60] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 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, page 293–304, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377922. URL https://doi.org/10.1145/3368826.3377922.

[61] Zhaole Chu, Yongping Luo, and Peiquan Jin. An efficient sorting algorithm for non-volatile memory. *International Journal of Software Engineering and Knowledge Engineering*, 31(11n12): 1603–1621, 2021. doi: 10.1142/S0218194021400143. URL https://doi.org/10.1142/S0218194021400143.

[62] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[63] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[64] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 339–351, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457292. URL https://doi.org/10.1145/3448016.3457292.

[65] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL https://doi.org/10.1145/1327452.1327492.

[66] Peter Denning and Roland Ibbett. The atlas milestone. *Commun. ACM*, 65(9):26–29, August 2022. ISSN 0001-0782. doi: 10.1145/3548781. URL https://doi.org/10.1145/3548781.

[67] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342407. doi: 10.1145/2901318.2901344. URL https://doi.org/10.1145/2901318.2901344.

[68] 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. 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, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582031. URL https://doi.org/10.1145/3582016.3582031.

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

[70] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 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, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341356. doi: 10.1145/2972206.2972210. URL https://doi.org/10.1145/2972206.2972210.

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

[72] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.*, 26(4):63–68, December 1997. ISSN 0163-5808. doi: 10.1145/271074.271094. URL https://doi.org/10.1145/271074.271094.

[73] Jim Gray and Franco Putzolu. 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, page 395–398, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912365. doi: 10.1145/38713.38755. URL https://doi.org/10.1145/38713.38755.

[74] Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with optik. *SIGPLAN Not.*, 51(8), February 2016. ISSN 0362-1340. doi: 10.1145/3016078.2851146. URL https://doi.org/10.1145/3016078.2851146.

[75] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. Manu: a cloud native vector database management system. *Proc. VLDB Endow.*, 15(12):3548–3561, August 2022. ISSN 2150-8097. doi: 10.14778/3554821.3554843. URL https://doi.org/10.14778/3554821.3554843.

[76] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[77] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613157. URL https://doi.org/10.1145/3600006.3613157.

[78] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3540426051.

[79] D. J. Hatfield. Experiments on page size, program access patterns, and virtual memory performance. *IBM Journal of Research and Development*, 16(1):58–66, 1972. doi: 10.1147/rd.161.0058.

[80] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. SIROCCO'07, page 124–138, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540729181.

[81] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 143–153, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065028. URL https://doi.org/10.1145/1065010.1065028.

[82] Yifan Hua, Kaixin Huang, Shengan Zheng, and Linpeng Huang. Pmsort: An adaptive sorting engine for persistent memory. *Journal of Systems Architecture*, 120:102279, 2021. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2021.102279. URL https://www.sciencedirect.com/science/article/pii/S1383762121001922.

[83] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. *SIGPLAN Not.*, 39(10): 69–80, October 2004. ISSN 0362-1340. doi: 10.1145/1035292.1028983. URL https://doi.org/10.1145/1035292.1028983.

[84] George U. Hubbard. Some characteristics of sorting computing systems using random access storage devices. *Commun. ACM*, 6(5): 248–255, may 1963. ISSN 0001-0782. doi: 10.1145/366552.366586. URL https://doi.org/10.1145/366552.366586.

[85] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/hunter.

[86] Apple Computer Inc. *Inside MacIntosh: Memory*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1992. ISBN 0201632403.

[87] Intel Corporation. Intel Optane SSD DC P5800X product brief, 2021.

[88] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. 2019. doi: 10.48550/arXiv.1903.05714. URL https://arxiv.org/abs/1903.05714.

[89] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/jang.

[90] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. Tencent sort. *sortbenchmark.org*, 2017.

[91] Teresa Johnson, Snehasish Kumar, , and David Li. Rfc: Ir metadata format for memprof, 2021. URL https://groups.google.com/g/llvm-dev/c/aWHsdMxKAfE/m/WtEmRqyhAgAJ.

[92] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359631. URL https://doi.org/10.1145/3341301.3359631.

[93] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015. ISSN 0163-5964. doi: 10.1145/2872887.2750392. URL https://doi.org/10.1145/2872887.2750392.

[94] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, 1962. doi: 10.1109/TEC.1962.5219356.

[95] KIOXIA Corporation. FL6 Series enterprise NVMe SSD. https://americas.kioxia.com/en-us/business/ssd/enterprise-ssd/fl6.html, 2023.

[96] Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973.

[97] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. 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)*, pages 216–227, Los Alamitos, CA, USA, November 2019. IEEE Computer Society. doi: 10.1109/IISWC47752.2019.9042009. URL https://doi.ieeecomputersociety.org/10.1109/IISWC47752.2019.9042009.

[98] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon.

[99] 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. Software-defined far memory in warehouse-scale computers. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *ASPLOS*, pages 317–330. ACM, 2019. ISBN 978-1-4503-6240-5.

[100] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359635. URL https://doi.org/10.1145/3341301.3359635.

[101] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613167. URL https://doi.org/10.1145/3600006.3613167.

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

[103] Baptiste Lepers and Willy Zwaenepoel. 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)*, pages 519–534, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL `https://www.usenix.org/conference/osdi23/presentation/lepers`.

[104] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[105] 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. Pond: Cxl-based memory pooling systems for cloud platforms, 2022.

[106] 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. 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, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL `https://doi.org/10.1145/3575693.3578835`.

[107] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. *Systematic CXL Memory Characterization and Performance Analysis at Scale*, page 1203–1217. Association for Computing Machinery, New York, NY, USA, 2025. ISBN 9798400710797. URL `https://doi.org/10.1145/3676641.3715987`.

[108] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017. doi: 10.1109/BigData.2017.8258257.

[109] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[110] Vasilis Mageirakos, Bowen Wu, and Gustavo Alonso. Cracking vector search indexes. *Proc. VLDB Endow.*, 18(11):3951–3964, July 2025. ISSN 2150-8097. doi: 10.14778/3749646.3749666. URL https://doi.org/10.14778/3749646.3749666.

[111] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, April 2020. ISSN 0162-8828. doi: 10.1109/TPAMI.2018.2889473. URL https://doi.org/10.1109/TPAMI.2018.2889473.

[112] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. *Parallel External Sorting*, pages 209–218. Springer US, Boston, MA, 2000. ISBN 978-1-4419-8590-3. doi: 10.1007/978-1-4419-8590-3_10. URL https://doi.org/10.1007/978-1-4419-8590-3_10.

[113] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312233. doi: 10.1145/2168836.2168855. URL https://doi.org/10.1145/2168836.2168855.

[114] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/al-maruf.

[115] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/al-maruf.

[116] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered memory, 2022.

[117] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. FairyWREN: A sustainable cache for emerging Write-Read-Erase flash interfaces. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 745–764, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/mcallister.

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

[119] Svetozar Miucin and Alexandra Fedorova. Data-driven spatial locality. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 243–253, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364751. doi: 10.1145/3240302.3240417. URL https://doi.org/10.1145/3240302.3240417.

[120] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Umar Farooq Minhas, Jeffery Pound, Cedric Renggli, Nima Reyhani, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Incremental ivf index maintenance for streaming vector search. 2024. URL https://arxiv.org/abs/2411.00970.

[121] Jason Mohoney, Devesh Sarda, Mengze Tang, Shihabur Rahman Chowdhury, Anil Pacaci, Ihab F. Ilyas, Theodoros Rekatsinas, and Shivaram Venkataraman. Quake: adaptive indexing for vector search. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

[122] Blaise Munyampirwa, Vihan Lakshman, and Benjamin Coleman. Down with the hierarchy: The 'h' in hnsw stands for "hubs". 2025. URL https://arxiv.org/abs/2412.01940.

[123] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/nam.

[124] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628, oct 1995. ISSN 1066-8888.

[125] Jinyoung Oh and Youngjin Kwon. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 97–105, 2021.

[126] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[127] Ashish Panwar, Sorav Bansal, and K. Gopinath. 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, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304064. URL https://doi.org/10.1145/3297858.3304064.

[128] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304–315, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372060. doi: 10.1145/3357526.3357568. URL https://doi.org/10.1145/3357526.3357568.

[129] William Pugh. Concurrent maintenance of skip lists. Technical report, USA, 1990.

[130] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. doi: 10.1145/3299869.3320212. URL https://doi.org/10.1145/3299869.3320212.

[131] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 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, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483550. URL https://doi.org/10.1145/3477132.3483550.

[132] Jie Ren, Minjia Zhang, and Dong Li. Hm-ann: efficient billion-point nearest neighbor search on heterogeneous memory. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[133] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ruan.

[134] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble.

[135] Samsung Electronics. Samsung Z-SSD SZ985. https://semiconductor.samsung.com/news-events/tech-blog/samsung-z-ssd-sz985/, 2018.

[136] Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyoung Do, and Sang-Won Lee. Turbocharging vector databases using modern ssds. *Proc. VLDB Endow.*, 18(11):4710–4722, July 2025. ISSN 2150-8097. doi: 10.14778/3749646.3749724. URL https://doi.org/10.14778/3749646.3749724.

[137] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search. 2021. URL https://arxiv.org/abs/2105.09613.

[138] SMART Modular Technologies. Persistent memory: NVDIMM-N and Optane DC DIMM. White Paper M-WP007, 2021.

[139] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. Diskann: fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2019. Curran Associates Inc.

[140] 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. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL https://doi.org/10.1145/3613424.3614256.

[141] 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. Demystifying cxl memory with genuine cxl-ready systems and devices. MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL https://doi.org/10.1145/3613424.3614256.

[142] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. Exploring performance and cost optimization with ASIC-based CXL memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22–25, 2024*, pages 818–833. ACM, 2024. doi: 10.1145/3627703.3650061. URL https://doi.org/10.1145/3627703.3650061. Best Runner-up Paper Award.

[143] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 171–185, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/tian-bing.

[144] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387517. URL https://doi.org/10.1145/3342195.3387517.

[145] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. SIGMOD '18, page 1541–1555, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196897. URL https://doi.org/10.1145/3183713.3196897.

[146] Rik van Riel and Vinod Chegu. Automatic numa balancing.red hat summit, 2014.

[147] Stratis D. Viglas. Adapting the b + -tree for asymmetric i/o. In Tadeusz Morzy, Theo Härder, and Robert Wrembel, editors, *Advances in Databases and Information Systems*, pages 399–412, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33074-2.

[148] Stratis D. Viglas. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.*, 7(5):413–424, jan 2014. ISSN 2150-8097. doi: 10.14778/2732269.2732277. URL https://doi.org/10.14778/2732269.2732277.

[149] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457550. URL https://doi.org/10.1145/3448016.3457550.

[150] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. volume 2, New York, NY, USA, March 2024. Association for Computing Machinery. doi: 10.1145/3639269. URL https://doi.org/10.1145/3639269.

[151] Nick Wanninger, Tommy McMichen, Simone Campanoni, and Peter Dinda. 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, page 448–463, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651326. URL https://doi.org/10.1145/3620666.3651326.

[152] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 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, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507731. URL https://doi.org/10.1145/3503222.3507731.

[153] P. Weisberg and Y. Wiseman. Using 4kb page size for virtual memory is obsolete. In *Proceedings of the 10th IEEE International Conference on Information Reuse & Integration*, IRI'09, page 262–265. IEEE Press, 2009. ISBN 9781424441143.

[154] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association. ISBN 978-1-939133-26-7. URL https://www.usenix.org/conference/fast22/presentation/wu.

[155] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519556. URL https://doi.org/10.1145/3492321.3519556.

[156] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. Graysort on apache spark by databricks. *sortbenchmark.org*, 2014.

[157] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. Spfresh: Incremental in-place update for billion-scale vector search. SOSP '23, page 545–561, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613166. URL https://doi.org/10.1145/3600006.3613166.

[158] Albert Mingkun Yang and Tobias Wrigstad. Deep dive into zgc: A modern garbage collector in openjdk. *ACM Trans. Program. Lang. Syst.*, 44(4), September 2022. ISSN 0164-0925. doi: 10.1145/3538532. URL https://doi.org/10.1145/3538532.

[159] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/yang.

[160] Juncheng Yang, Yao Yue, and K. V. Rashmi. 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)*, pages 191–208. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/yang.

[161] Yiwei Yang, Yusheng Zheng, Yiqi Chen, Zheng Liang, Kexin Chu, Zhe Zhou, Andi Quinn, and Wei Zhang. CXLAimPod: CXL memory is all you need in AI era. *arXiv preprint*, 2025. doi: 10.48550/arXiv.2508.15980. URL https://arxiv.org/abs/2508.15980.

[162] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. Performance characterizations and usage guidelines of samsung cmm-h. *IEEE Micro*, 45(5):94–102, 2025. doi: 10.1109/MM.2025.3591577.

[163] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456237. URL https://doi.org/10.1145/3447786.3456237.

[164] 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. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong.

[165] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 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, page 192–206, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651350. URL https://doi.org/10.1145/3620666.3651350.