ObjecTier: Non-Invasively Boosting Memory Tiering Performance

Vinita Pawar
vinita@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Ankit Bhardwaj
ankitbwj@mit.edu
MIT CSAIL
Cambridge, Massachusetts, USA

Ryan Stutsman
stutsman@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Abstract

Recent research has developed page-based memory-tiering systems that place hot pages in fast tiers and cold pages in slower, more capacious tiers. However, applications place many objects together within pages, and most pages contain some objects that are hot and some that are cold. Our simulations of a key-value workload confirm this; even the hottest pages in the fast tier can contain 50% cold data.

To improve fast tier utilization, we describe the design of a new framework, *ObjecTier*, that uses application knowledge to efficiently consolidate hot data and cold data. This allows ObjecTier-enabled applications to boost fast tier hit rates and improve performance regardless of which underlying memory tiering system they use underneath, even if that system is page based.

With simulations, we show that ObjecTier may improve average memory access time (AMAT) by 2× without adding any memory space overhead for our simulated key-value store workload. We conclude by outlining the next steps to make the ObjecTier framework a reality for easy adaptation of applications like key-value stores and other indexed databases.

CCS Concepts

• **Software and its engineering** → **Memory management**; • **Computer systems organization** → **Distributed architectures**.

Keywords

Tiered Memory; Persistent Memory; Compute Express Link (CXL); Disaggregated Memory; Non-volatile memory

ACM Reference Format:
Vinita Pawar, Ankit Bhardwaj, and Ryan Stutsman. 2025. ObjecTier: Non-Invasively Boosting Memory Tiering Performance. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3680256.3721319

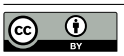
1 Introduction

Machines in data centers have hierarchies of memories that span SRAM (caches), DRAM, NVM [15], and now CXL-attached memories [30]. Disaggregated memory is further deepening these hierarchies. To use these memory tiers most effectively, recent research has developed memory-tiering systems that explicitly attempt to place hot data in fast tiers and cold data in slower, more capacious tiers by adjusting the location of pages of data using address translation hardware [2, 18, 21, 25, 34]. These systems have two main components: page classification and page migration. The page classifier distinguishes frequently accessed (hot) pages from infrequently accessed (cold) pages. Existing systems use software page faults, hardware reference bits in page tables (`ACCESSED` bit), or hardware counter sampling (such as Intel PEBS [14]) to distinguish pages. After classification, hot pages are migrated to faster memory tiers while cold pages are migrated to slower memory tiers. These memory-tiering schemes work in an application-transparent manner since a page's virtual address can remain the same as it is migrated back and forth between fast and slow tiers.

Even though page-based tiering works transparently across all applications, the use of pages forces these systems to make an inherent assumption about the application data in pages — *everything in a hot page is hot.* Studies refute this assumption; many data center workloads have heavily skewed access patterns [3, 6, 11, 22, 35]. When allocating objects, memory allocators reuse space while trying to avoid fragmentation. In turn, this causes them to scatter popular objects all over an application's address space. Hence, often only a few cache lines within a page are hot, while the rest of the data is cold with very few accesses to it [7]. This problem is compounded by the fact that page sizes are determined by hardware, commonly with a 4 KB minimum which is much larger than many hot objects. Even without extreme skew, this problem persists since memory allocators are popularity oblivious (§2).

Our main insight is that with modest reorganization of application objects, fast tier utilization can be substantially improved. In this paper, we describe the design of ObjecTier, a new framework we are developing that uses application knowledge to efficiently consolidate hot data into pages, which improves fast tier utilization. ObjecTier is designed to work synergistically with existing memory-tiering systems without changing them. ObjecTier clusters hot and cold objects together across pages within an application, then any existing memory tiering scheme can more effectively migrate the application's pages between tiers based on the overall temperature of pages.

This is a challenging approach; outside of garbage-collected runtimes like Java, relocating objects is hard because languages like C, C++, and Rust assume allocations' addresses remain fixed until they are freed. Moreover, the system must classify hot and cold objects within pages and continuously update this classification since data center workloads' working sets change over time. Lastly, the system must not require invasive changes to applications since rewriting production applications is risky, and retrofitting applications to work on each new memory hierarchy layer or memory-tiering system is not scalable.



ICPE Companion '25, Toronto, ON, Canada

ACM ISBN 979-8-4007-1130-5/2025/05
https://doi.org/10.1145/3680256.3721319

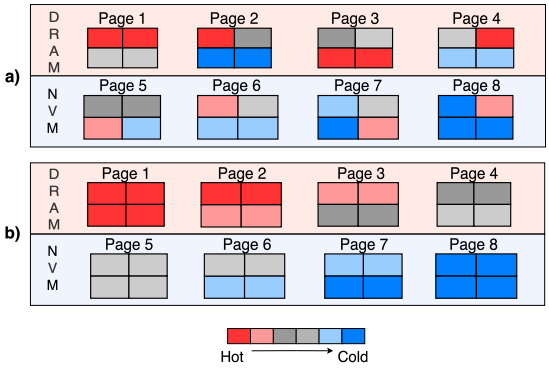


Figure 1: a) In real applications hot objects are scattered around in different pages; they cannot be grouped a priori since they aren't known to be hot when they are allocated. b) Sorting the objects to group like-popularity objects creates pages that better use fast memory.

ObjecTier is designed to work with applications that store large heaps of mixed-popularity objects in memory (100s of GB or more), like memcached, Redis, and in-memory databases. ObjecTier works by identifying application heap regions that have hot and cold data intermixed in them, then it relocates the objects in those regions to separate the hot objects together and the cold objects together in new regions. This requires updating references to the objects in the application with small augmentations to the application logic; for example, in memcached, ObjecTier must update the hash table that maps a key to a relocated object. This separation improves fast memory tier hit rates, improving average memory access times. This paper makes four main contributions:

(1) First, we add to the growing body of evidence that page-granular approaches to memory tiering lead to poor performance [4, 9]; we make the case for hot object consolidation for memory tiering using simulations that show, regardless of skew or page size, scattered hot objects lead to poor fast memory tier utilization (§2).
(2) Second, we assess the performance benefits of hot object consolidation on simple but common access patterns showing that average memory access times can be nearly halved compared to running without consolidation (§2).
(3) Third, we show that small augmentations may provide effective hot object consolidation in memcached, a widely used caching application that houses hundreds of gigabytes of objects per machine in large caching fleets (§4), and we measure the benefits of hot object consolidation in the FlexKVS key-value store [25].
(4) Finally, we discuss our vision for a general framework that can be applied to a wide variety of data center applications to make memory-tiering systems more effective (§4).

2 Motivation: Scattered Hot Data

To make the case for augmenting applications to group hot data into pages, we simulate a very simple application. This application is presented with a set of keys that it looks up in a large array that spans 256 GB mapped with 2 MB huge pages (commonly done to avoid TLB misses in memory tiering [25, 34]). The objects in the

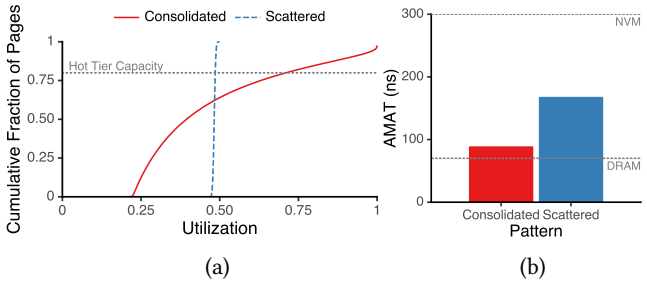


Figure 2: a) The distribution of page utilization when hot objects are Scattered and Consolidated. b) The average memory access time of the two access patterns.

array are 128 B, which models a common object size for memory-intensive web caching workloads like memcached [22]. Keys are chosen according to a random but (Zipf, $\theta = 0.99$) skewed distribution.

In one run of the application, which we call *Scattered*, objects are scattered randomly in the array regardless of their access popularity. In another run, which we call *Consolidated*, the application is given an oracle that indicates which keys will be accessed in the future, so it ensures that the objects in the array are sorted by access popularity with the hottest object at the start of the array (Figure 1b).

If these two workloads were run separately against today's memory tiering systems, they would result in very different performance. Memory tiering systems will put hotter pages in the fast memory tier. However, the Consolidated workload would use the limited fast tier space better than the Scattered workload since putting a hot object in the fast tier never forces adjacent cold object to be promoted along with it. This improves fast tier hit rates and average memory access times compared to the Scattered workload.

Why is this the case when the objects that the two workloads access are the same size and have the same popularity distribution? Memory-tiering systems use hardware paging as an application-transparent level of indirection to remap pages between the tiers. In practice, applications allocate heap space to objects in a popularity-oblivious fashion, and generally they must do so since they have no way of knowing in advance which objects will be popular. So, over time today's applications end up with hot data intermixed with cold data on pages, similar to the Scattered workload.

How much does this matter? Figure 2 shows the results from our simulation; it shows that consolidation's effect on (a) the amount of hot data in the fast tier and (b) the application's observed average memory access time (AMAT) is substantial. In the simulation, we track every byte that is accessed at least once, then for each page we compute fraction of bytes in each page that were accessed during the run, which we call *page utilization*.

The figure shows the cumulative distribution function of page utilization for each page in the 256 GB array after 5 billion object accesses. The horizontal grey line in Figure 2a indicates the cutoff above which pages would be in the fast tier if a memory-tiering system had an oracle that perfectly picked out the top 20% of hottest pages to place in the fast tier (about 50 GB DRAM compared to 200 GB of NVM Optane). For the Scattered access pattern, the figure

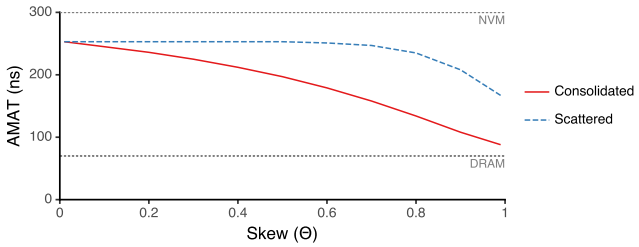


Figure 3: Consolidation's AMAT benefit vs. access skew.

shows that even when the fast tier contains the hottest pages, about half of the data in each page is never accessed. With the Consolidated pattern, hot objects are grouped, improving the utilization of the hottest pages. Consolidation improves the 20% of pages that fit in the fast tier to have to 70 to 100% utilization.

In both workloads, the fast tier still holds the hottest pages, so what is the overall impact on application-observed memory system performance? Figure 2b shows that the average memory access time drops from 167 ns to 87 ns when hot data is consolidated in pages. This nearly halves the delay applications see and bring AMAT much closer to the access time of the DRAM fast tier (70 ns). Memory-intensive applications could see performance losses of 2× or more when their hot data is scattered among pages, and this loss would be higher when tiering against remote memory (remote memory access times via RDMA are more than 3× higher than NVM). Hence, spending CPU time to consolidate hot application data online is promising.

Does this only hold for highly skewed workloads? The above simulations use a θ=0.99 Zipf-skewed access pattern, which is a common workload [3, 11, 22]. At uniform random access patterns, pages have uniform access rates, so consolidation becomes fruitless. So, under what amount of skew is consolidation worthwhile? Figure 3 shows the benefits of consolidation are substantial even with modest skew in object access patterns. Here, modest levels of skew like θ=0.5 result in a 1.3× slowdown for memory accesses.

Would 4 KB pages solve this? Memory-tiering systems use 2 MB pages to reduce TLB misses [2]; however, using 4 KB pages would help break 2 MB pages into smaller units that can be placed separately. Rerunning the same analysis from Figure 2b using 4 KB pages improves AMAT from 167 to 114 ns for scattered accesses, but consolidation still improves that substantially to 88 ns. Hence, even with 4 KB pages, intermixed hot and cold data cause a 30% AMAT overhead for applications. Since 4 KB is the smallest page size on Intel hardware, it isn't possible to reduce this overhead further using smaller pages. Additionally, on top of this 30% overhead from intermixed hot and cold data, using 4 KB pages would add an additional overhead of up to 30% due to increased TLB misses for tiered applications [2].

3 Related Work

Depending on the addressability of the slow memory tier, memory systems can be broadly classified as (a) tiered memory systems, where the memory module with the highest latency is directly addressable and attached to the memory bus or accessed using PCIe or CXL, and (b) remote memory systems, where an RDMA connected memory is treated as the slow memory tier that is not directly addressable. Different schemes have been developed to target either of these systems.

Tiered Memory Systems. These systems use multiple layers of memory devices of different hardware latencies connected using interconnects like CXL. The state-of-the-art memory tiering systems [2, 4, 18, 21, 25, 31] target this type of setup. All of these systems work with 4 KB or 2 MB pages and suffer from the problem of scattered hot objects. HotBox [4] and MEMTIS [18] advocate for the use of 4 KB pages rather than 2 MB pages to reduce hotness fragmentation caused by cold pages taking up space in a hot hugepage; however, they do not solve the problem of hotness fragmentation at a *sub-page* granularity. As we showed in the prior section, this still leaves substantial room for optimization.

Memstrata [38] is a hardware-managed tiering system for virtual machines (VMs) that uses CXL and works at cacheline granularity. Each cacheline in the system is paired with another; the hardware keeps a bit of metadata per pair of cachelines to track which of the two cachelines is in the fast tier and which is in the slow tier. Conflicts occur when both cachelines of a pair are hot. The system includes a specialized memory allocator that helps reduce these conflicts. Memstrata assumes a fixed 1:1 ratio between the size of the hot and cold tier, and it only works for virtual machines since the hypervisor's extended page tables provide an application-agnostic way to remap physical memory locations between VMs to work around these conflicting hot cachelines.

Remote Memory Systems. These systems access data on a remote system connected via RDMA. The slow tier, which is the RDMA-attached tier, is not directly addressable, making it possible for these systems to intercept every remote access via remote "fat" pointers. Atlas [9] uses a hybrid approach of fetching both pages and objects from remote memory using RDMA. Furthermore, when fetching objects instead of pages, Atlas fits consecutively accessed remote objects into a new local page; thus, it groups recently accessed objects in pages. As Atlas is designed to work with remote memory, it intercepts each RDMA request to track accesses to the slow tier. This would be costly for the tiered memory systems that ObjecTier targets, which instead rely on hardware techniques like processor event-based sampling (PEBS) for tracking accesses.

Semswap [12] consolidates hot objects on a single page in a small "swap cache" before swapping out to remote memory by adding extra metadata in each object to track its popularity. The swap cache is limited to Linux's recent page cache victims, which limits the consolidation benefits to this small region. Semswap stores the mapping of the object's new physical address upon relocation, and it primarily relies on page faults to remap these objects, making the initial access to relocated objects expensive.

4 ObjecTier Design

ObjecTier is a framework that improves the performance of applications that run on top of memory-tiering systems. Its insight is that with minor changes, applications can consolidate hot and cold data into separate sets of pages. Then, when these applications use existing memory-tiering systems, fast tier memory utilization is improved since hot and cold data are not intermixed. This lowers the application's memory access times, improving performance.

ObjecTier-enabled applications incrementally reorganize their heap; this small tweak lets existing memory-tiering schemes distinguish hot and cold pages more easily and reduces cold data in the fast tier. Hence, ObjecTier is compatible with all existing page-based memory-tiering systems, it allows memory-tiering systems to evolve independently of ObjecTier-enabled applications, and it ensures that ObjecTier-enabled applications don't need to be changed when memory-tiering systems change.

Challenges. In order to be useful, ObjecTier must work with many existing applications, and using it must not require invasive changes. This raises two key challenges. First, general programs written in non-garbage-collected languages like C, C++, and Rust don't allow objects to be relocated at runtime. When a memory allocator returns a pointer, applications expect the address of that object to remain fixed until it is freed. Unfortunately, applications can't predict object popularity at the time of allocation, so, over time, hot and cold objects are mixed together and cannot be relocated. Second, even if objects can be relocated during their lifetime, consolidating hot objects requires the application to decide what is hot and what is cold.

4.1 Case Study: Key-Value Stores

Two important characteristics of key-value store applications make them a good fit for ObjecTier. First, these applications use large regions of memory and keep references to data objects in a small set of structures. For example, objects in cache servers and databases are frequently referenced by a hash table or B-tree. Updating the reference in the hash table is sufficient to relocate an object. These objects are also often protected by locks or reference counts, making it easy to synchronize with other code that may be temporarily holding references to them. Second, some of these applications that keep large heaps of objects already track information about the relative utility and popularity of objects. For example, caching servers commonly keep eviction queues to track frequency and recency of accesses to rank objects for eviction. In-memory databases rank objects for eviction as well [19]. By tapping into this information and consolidating objects among pages, applications can use information they already collect to improve the decisions that memory-tiering systems make at a low cost. Examples of these applications include FlexKVS [17], Redis [26], TAO [6], FASTER [8], which are used in hyperscalars such as Microsoft and Meta across thousands of machines.

As an example, memcached [1] is a good fit for ObjecTier. It keeps massive sets of objects in its heap, the objects are generally sub-page sized [22], they are referenced by a few structures (a hash table and a set of eviction queues), and it already tracks the relative popularity of objects. Meta famously caches hundreds of terabytes of objects that occupy large regions of memory in thousands of machines [22, 23]; they have also experimented with page-based memory-tiering [21], so their workloads stand to benefit from ObjecTier.

Figure 4 shows memcached's key data structures. Each request it services updates an object associated with a key (a SET) or returns the value most-recently associated with a key (a GET). When a new object is inserted into memcached, it makes space for the object by evicting less popular objects. To facilitate this, each object is linked

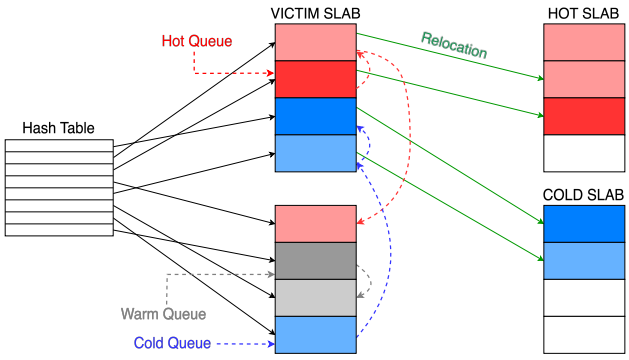


Figure 4: memcached stores objects in slabs that are indexed by a hash table. Objects are on one of three queues that track eviction ranking: the hot, warm, and cold queues. ObjecTier adds two fresh slabs into which hot and cold objects are incrementally consolidated.

into one of three queues. Objects are initially placed at the head of the *hot queue*. As the hot queue grows, objects that have been accessed while on the hot queue are moved to the *warm queue*; otherwise, they are demoted to the *cold queue*. Objects at the tail of cold queue are evicted to make space for new objects as needed.

When inserting a new object memcached links the object to the head of the hot queue, and it adds a reference in its hash table that maps the object's key to the object's location in memory. In most workloads, most requests are GETs rather than SETs. A GET finds an object's key in the hash table, follows the reference to the object, marks the object as being recently accessed, and sends the object back over the network. Over time, memcached's heap becomes intermixed with objects of differing popularity.

4.2 Integrating ObjecTier

Integrating ObjecTier in a key-value store application requires three changes.

Finding regions with intermixed hot and cold data. First, ObjecTier must find candidate regions of memory with intermixed hot and cold data. Reorganizing these regions will best help separate hot and cold data into different sets of pages. For example, memcached stores objects in slabs that can be augmented to track which slab has intermixed data; similarly, FlexKVS has segments that can serve the same purpose. Augmenting the allocator to track which slab or segment has intermixed objects only requires one additional word of metadata (a counter) that tracks fraction of hot objects. In case of memcached, the data in the slab is referred to by the hot and warm eviction queues. Small changes in the eviction code increment and decrement this counter as objects in the slab are promoted and demoted between eviction queues, which adds negligible overhead. Whenever ObjecTier needs candidate regions, it can randomly sample these counts to find pages where the fraction of hot data in the slab is neither near 0 nor 1. A slab or segment whose hot and cold data is intermixed is selected as a victim.

Classifying objects into hot and cold regions. Next, ObjecTier must iterate the objects in the victim region and move them into one of two new regions: one holding objects ObjecTier believes are hot and one holding objects that ObjecTier believes are cold. The tiering system monitors accesses and decides tiering placement for

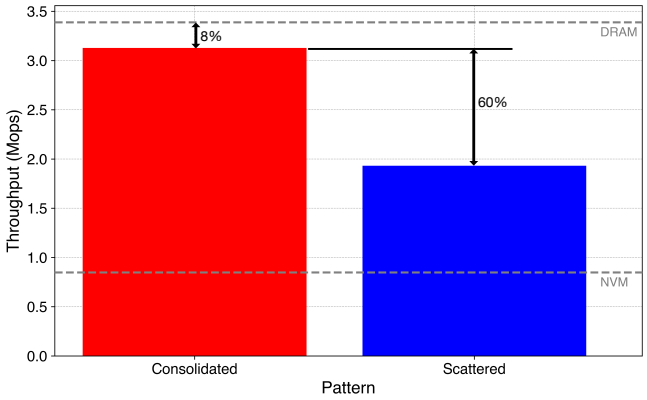


Figure 5: FlexKVS throughput when hot objects are Scattered and Consolidated. Horizontal lines show the throughput of FlexKVS when run completely in DRAM and NVM. Consolidated hot objects achieve 60% higher end-to-end throughput compared to Scattered hot objects.

these newly generated regions. ObjecTier can reuse the memory allocators of these key value stores to acquire new slabs or segments and fill them up with objects such that their fraction of hot objects becomes close or equal to 0 or 1. After all objects in a victim have been relocated, the victim region is freed and returned to the memory allocator.

In memcached, which maintains eviction queues to track object popularity, we can add a two-bit field to each object to track its eviction queue. This can be used to efficiently decide which slab to copy the object into if its slab becomes a victim.

Relocating objects. Finally, when an object is relocated, the references to it must be updated, and ObjecTier must ensure this is safe despite concurrent SET and GET operations. This requires adding a callback. The callback is passed a pointer to an object and a pointer to a new location in a new hot or cold region. The callback uses the key embedded in the object to find and update its reference in the hash table, The callback must acquire and release the appropriate locks while doing so to make this safe.

4.3 Expected Performance Gains

We now evaluate performance gains that could be achieved by integrating ObjecTier into key-value stores. We experiment with FlexKVS [17], a key-value store that is memcached compatible and uses similar data structures for its memory management. We run FlexKVS with 4 threads and a heterogeneous memory setup consisting of 8 GB DRAM and 8 GB Optane NVM. We configure the working set size of FlexKVS to 16 GB. We load the hash table with 15 M key-value pairs of 1 KB each, and access them in a distribution such that 25% of the keys are hot. We compare the performance of FlexKVS against two workloads — a consolidated workload, where the hot keys are consolidated into a set of pages, making up a contiguous hot-set of 4 GB, and a scattered workload, where the hot keys are scattered uniformly across the entire working set. We report the performance of FlexKVS in Figure 5. We observe a 60% increase in the throughput in the consolidated version compared to the scattered version. Compared to the ideal case, when the entire working set fits in the DRAM, the consolidated version is merely 8% slower.

In practice, we expect many key-value-based applications to benefit even more significantly, since many real in-memory key-value store workloads have much smaller values [3]; here with 1 KB objects a single page can only contain four different objects each of varying popularity, but many practical workloads have 100 B objects or smaller leading to dozens to hundreds of varying popularity objects within each page.

Finally, these numbers do not capture the overhead that ObjecTier will introduce due to consolidating objects; however, we expect these overheads to be very small. This is because the relocation process can correlate the placement of objects with similar popularties and lifetimes to avoid most relocations just as other log-structured cleaning approaches have shown [10, 28, 29, 36]. We qualitatively discuss the expected CPU overheads in consolidation in the next section.

5 Toward a Generic Framework

Now that we have explored application-specific changes required for ObjecTier, we can look at how all this fits together as a generic framework for other applications. ObjecTier targets applications with two main properties. First, they should be memory-intensive with smaller objects where multiple objects are needed to fill a page. Many data center applications fit in this category [6, 35]. Second, they should have a simple memory layout where objects and metadata (e.g. memory allocator metadata) are cleanly separated. This simplifies object relocation without worrying about intertwined objects and metadata state. Currently, our design is tailored to key-value stores, but we plan to extend our framework for other applications with similar characteristics. A generic framework for integrating ObjecTier into an application requires the following two main components:

Object Classifier: The classifier records memory accesses to pages, and it identifies pages with mixed-temperature objects. ObjecTier can use existing application-specific information for classifying objects, like eviction queues in memcached. However, it can also be extended to gather this information with an independent approach. Existing sub-page sampling-based (e.g. PEBS) methods used by memory-tiering schemes can be repurposed for this goal.

Object Relocator: ObjecTier needs application-specific logic to relocate objects to hot and cold pages (as shown in Figure 4). It involves a subcomponent that allocates and maintains a hot/cold page for object relocation. ObjecTier relocates each object using an application-provided callback function (e.g. `bool (*relocate)(void* object, char* dst, size_t bytes_available)`). The callback can refuse to relocate an object if `dst` points to a region with insufficient space available for the object, in which case ObjecTier allocates a fresh region and reattempts the relocation. Finally, the callback function also updates all the references from the old object to the relocated object.

Relocation Policy Manager: Besides the mechanism to identify and relocate hot and cold objects, the last key piece that ObjecTier adds are control functions that decide whether the application can benefit by performing additional consolidation or whether

it makes sense to leave objects where they are (for example, if hot objects are already sufficiently consolidated). ObjecTier is designed as a set of procedure calls that can make some incremental progress on consolidation each time they are called, or they can be used to run consolidation on its own thread.

Overall, different applications will use ObjecTier differently. Some applications will rely on it to discover objects to relocate and others will do this themselves. Some will have complex relocation callbacks while others may be a simple `memcpy`. Our current effort is finding a small and orthogonal set of components that works for many applications.

6 Future Work

While we have augmented memcached data structures to support ObjecTier, we have yet to complete its implementation and evaluate its performance. There are a number of important questions and directions that we hope to explore after we have a few initial working applications.

Other Candidate Applications. There are many other applications that can benefit from using ObjecTier. As explained earlier, good candidate applications will have large sets of sub-page-sized objects and a small number of references that are easy to enumerate, since they need to be changed if an object is relocated.

In-memory object stores like Redis [26] are a good fit for ObjecTier. Row-store, OLTP-focused in-memory databases are also good candidates [16, 19, 20, 32, 33]; they similarly keep scattered hot and cold objects primarily under a single index. Silo is a transactional store commonly used in evaluating memory tiering schemes [25]; we have begun looking at augmenting it using ObjecTier in addition to FlexKVS.

Analytical applications like Spark and vector databases [5, 24, 37] are seemingly good candidates. However, they often scan large contiguous runs of objects or large vectors. These objects are frequently larger than a page, avoiding intermixing of hot and cold data in pages. We plan to validate by running a similar analysis as we performed in Figure 2 for a wide variety of real-world applications.

Finally, languages that use garbage-collected runtimes like Java and Go could apply ObjecTier to all applications that they run. Generational garbage collectors already perform a form of hot and cold separation. One area of future work we plan to investigate is the extent to which these runtimes already help perform similar consolidation to what ObjecTier does. While it is possible existing garbage collectors provide some of the benefits of ObjecTier when used on memory-tiered applications, one thing that is worth investigating is whether adding additional tracking of hot and cold objects (e.g. via PEBS) can be used to augment their existing garbage collectors to enhance hot object consolidation.

Consolidation Overhead. Consolidation is a cleaner similar to those in log-structured file systems [28]; cleaning costs were a notoriously contentious point of debate in the academic community [27]. However, this consolidation only works on memory rather than disk, making low overhead [10, 29, 36]. For example, supposing that each object needed to be relocated once per hour, then consolidating a 256 GB heap would only require about 150 MB/s of memory bandwidth, which would consume about 0.06% of the memory bandwidth of a modern machine. In practice, costs would be much lower because ObjecTier only relocates regions of memory that are known to intermix hot and cold objects; very hot and very cold objects would rarely be relocated.

Disaggregation. CXL 2.0 memory pooling [13] and RDMA-based remote memory pooling schemes [7] can also benefit from ObjecTier, especially since remote memory access costs are commonly higher for remote tiers than for local NVM tiers. We plan to investigate the benefits of ObjecTier for remote memory tiers, though decreased bandwidth between the local and remote tier may place additional constraints on how aggressively we reorganize memory in order to limit churn between the tiers.

7 Conclusion

Recent years have yielded numerous memory tiering systems and algorithms, but most work at page granularity to preserve application transparency. However, for many workloads this hurts effective fast tier utilization since applications generally organize their objects in a page-oblivious fashion.

ObjecTier seeks to overcome that limitation without giving up a decoupled design where applications need not integrate with the tiering system on which they are run. Our initial explorations suggest many memory-intensive applications with simple indexing structures like key-value stores and databases may be easy to adapt with our approach, and our simulations suggest the benefits can be substantial (near 2× improved average memory access times compared with 2 MB-page-based tiering or 1.3× when 4 KB page are used).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2245999. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by VMware.

References

[1] 2024. Memcached. https://memcached.org/, accessed 03/16/2024.

[2] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 631–644.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64.

[4] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management* (San Diego, CA, USA) *(ISMM 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3520263.3534650

[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (dec 2008), 77–85.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60.

[7] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual,

USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–92.

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

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

[10] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 321–334.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. Association for Computing Machinery, New York, NY, USA, 143–154.

[12] Siwei Cui, Liuyi Jin, Khanh Nguyen, and Chenxi Wang. 2022. SemSwap: semantics-aware swapping in memory disaggregated datacenters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems* (Virtual Event, Singapore) *(APSys '22)*. Association for Computing Machinery, New York, NY, USA, 9–17.

[13] CXL Consortium, Inc. 2024. CXL® 3.1 Specification. https://computeexpresslink.org/cxl-specification/, accessed 03/16/2024.

[14] Intel Corporation. 2024. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, accessed 03/16/2024.

[15] Intel Corporation. 2024. Intel® Optane™ Persistent Memory — Work. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html, accessed 03/16/2024.

[16] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. doi:10.14778/1454159.1454211

[17] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. *SIGPLAN Not.* 51, 4 (March 2016), 67–81. doi:10.1145/2954679.2872367

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

[19] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*.

[20] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. 2014. Indexing on modern hardware: Hekaton and beyond. In *SIGMOD*. 717–720.

[21] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Association for Computing Machinery, New York, NY, USA, 742–755.

[22] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[23] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 29–41.

[24] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984.

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

[26] Redis, Ltd. 2024. Redis. redis.io, accessed 03/16/2024.

[27] John Regehr and Peter Bailis. 2017. Vigorous Public Debates in Academic Computer Science: Expert-curated Guides to the Best of CS Research. *Queue* 15, 3 (jun 2017), 80–84.

[28] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (feb 1992), 26–52.

[29] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 1–16.

[30] Samsung. 2024. Expanding the Limits of Memory Bandwidth and Density: Samsung's CXL Memory Expander | Samsung Semiconductor Global. https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/, accessed 03/16/2024.

[31] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 283–297.

[32] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[33] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[34] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. doi:10.1145/3297858.3304024

[35] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[36] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 503–518. https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28.

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