TierScape: Harnessing Multiple Compressed Tiers to Tame Server Memory TCO

Sandeep Kumar
Processor Architecture Research Lab, Intel Labs
India
sandeep4.kumar@intel.com

Aravinda Prasad
Processor Architecture Research Lab, Intel Labs
India
aravinda.prasad@intel.com

Sreenivas Subramoney
Processor Architecture Research Lab, Intel Labs
India
sreenivas.subramoney@intel.com

Abstract

Tiered memory systems are the norm to effectively tackle the increasing memory total cost of ownership (TCO) in modern data centers. We propose *TierScape* to tame memory TCO through the novel creation and judicious management of multiple compressed memory tiers along with multiple byte-addressable tiers.

As opposed to conventional tiering solutions that employ a single compressed memory tier, we harness multiple compressed tiers implemented through a combination of different compression algorithms, memory allocators for compressed objects, and backing media to store compressed objects. These compressed tiers represent distinct points in the access latency, data compressibility, and unit memory usage cost spectrum, allowing rich and flexible trade-offs between memory TCO savings and application performance impact. A key advantage with TierScape is that it enables aggressive memory TCO saving opportunities by placing warm data in low latency compressed tiers with a reasonable performance impact while simultaneously placing cold data in the best memory TCO saving tiers.

TierScape presents a comprehensive, rigorous, and tunable analytical cost model for performance and TCO trade-off based on continuous monitoring of the application's data access profile. Guided by this model, TierScape takes informed actions to dynamically manage the placement and migration of application data across multiple compressed and byte-addressable tiers. We believe TierScape represents an important server system configuration and optimization capability to achieve the best SLA-aware performance per dollar for applications hosted in production data center environments. For a set of real-world benchmarks, TierScape reduces memory TCO by 15.1–23.6% percentage points while maintaining performance parity or improves performance by 2.61–10.0% percentage points while maintaining memory TCO parity compared to state-of-the-art tiering solutions.

EUROSYS '26, Edinburgh, Scotland Uk

CCS Concepts: • **Software and its engineering → Memory management**.

Keywords: Memory Tiering, Compressed memory, Memory TCO

ACM Reference Format:
Sandeep Kumar, Aravinda Prasad, and Sreenivas Subramoney. 2026. *TierScape:* Harnessing Multiple Compressed Tiers to Tame Server Memory TCO. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3769321

1 Introduction

Memory accounts for 33-90% of the total cost of ownership (TCO) in modern data centers [2, 53, 54]. This cost is expected to escalate further to serve the growing data demands of modern AI/ML applications whose working set already breaks the terabyte barrier [47, 51], thus rendering it imperative for data center operators to tame the memory TCO. Henceforth, TCO savings refers to memory TCO savings.

Memory tiering is a viable and proven solution employed in production data centers to effectively reduce TCO and handle growing data demands of applications [26, 36, 38, 39, 42, 48, 49, 54]. The state-of-the-art solutions compress and place data in a compressed second-tier memory, such as zswap [10, 35] in the Linux kernel, to reduce memory TCO [38]. Placing data in a compressed memory tier reduces the memory footprint of applications, thus reducing the memory TCO, as systems can be provisioned with less memory or can be packed with more applications. However, such TCO savings are *not free* as the data stored in a compressed tier must be decompressed before an application can access it, resulting in a performance penalty. Hence, to trade off memory TCO savings and performance penalties, data center providers only place infrequently accessed or cold data in the compressed tier [38, 54].

We highlight the following critical observations and key limitations of the state-of-the-art solutions. ❶ On average, 20–30% of the data are cold in production systems [26, 38, 41, 42, 54] and hence, placing only cold data in the compressed memory tier has limited memory TCO saving potential. ❷ As shown in Figure 1, aggressively placing more data pages in the compressed tier can increase memory TCO savings

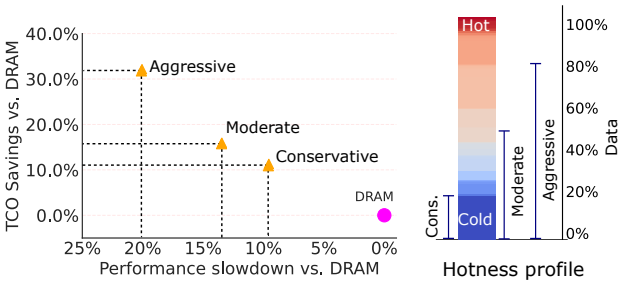


Figure 1. For Memcached on a DRAM + compressed tier system, conservative placement of 20% cold data yields 11% memory TCO savings with 9.5% throughput slowdown. Moderate placement of 50% (cold + some warm) improves savings to 16% with 13.5% slowdown, while aggressive placement of 80% (cold + most warm) achieves 32% savings at 20% slowdown. Color coding reflects access frequency: blue for cold (zero accesses), red for hot (highest accesses), and gradients for intermediate accesses.

but results in a significantly higher and unacceptable performance penalty (> 5% [26]). ❸ Given the high cost of accessing data from a compressed tier, existing tiering solutions do not compress (or tier) warm pages, which account for 50–60% [42, 54] of the data pages, thus leaving significant memory TCO reduction opportunities on the table.

In this paper, we seek to exploit memory TCO saving opportunities beyond the cold data pages with an acceptable performance penalty. We propose TierScape, a novel solution that harnesses multiple compressed memory tiers to aggressively tame server memory TCO. TierScape dynamically tunes and manages placement of data across multiple compressed and byte-addressable tiers to strike the best balance between memory TCO savings and application performance. The compressed tiers can be a combination of different compression algorithms (e.g., lzo-rle, deflate, lz4), memory allocators for compressed objects (e.g., zsmalloc, zbud, z3fold), and backing media to store compressed objects (e.g., DRAM, non-volatile main memory [3], CXL-attached memory [25, 27, 43]). TierScape's compressed tiers are distinct in access latency, unit memory usage cost, and capacity savings (compression ratio), enabling a holistic and flexible option space for hot/warm/cold data placement. TierScape thus compares very favorably to the rigid and restricted data placement and optimization space available in today's state-of-the-art tiering systems.

TierScape enables aggressive memory TCO saving opportunities by placing warm data pages in low-latency compressed tiers with reasonable performance impact while simultaneously placing cold data in the best memory TCO saving tiers. TierScape applies different placement and migration policies for warm/cold data based on the application's dynamic data access profile. For example, in our conservative *waterfall* model, (§6.1) warm pages are initially placed in a low latency compressed tier and eventually moved or aged to compressed tiers with better TCO savings, thus progressively achieving better memory TCO savings.

TierScape introduces a mathematical model that in real-time enables a holistic and flexible option space for hot/warm/cold data placement to balance memory TCO savings and application performance. TierScape periodically recommends *scattering* pages across multiple tiers. The recommendations to move specific groups of pages to specific tiers are based on the access patterns of the application's different memory regions, along with the relative costs of page accessed in different tiers and the real-time memory TCO cost per tier incurred by the application. TierScape's multi-objective optimization enables superior placement and control of hot/warm/cold page sets and calibrated maximization of performance-per-dollar metrics critical for data centers.

The key contributions of the paper are as follows:

1. To the best of our knowledge, we are the first to propose and demonstrate memory TCO savings for warm data with an acceptable performance impact.
2. Highlight the limitations with the state-of-the-art memory tiering systems in saving memory TCO. Specifically, the limited TCO savings with cold data and its incapability to tap TCO saving opportunities for warm data with a reasonable performance penalty.
3. Demonstrate the benefits of harnessing multiple compressed memory tiers that offer a rich and flexible trade-off between memory TCO savings and application performance impact with waterfall and analytical page placement models.

2 Background – Compressed memory tiers

Linux kernel's zswap [11, 35, 38] supports memory compression where pages are compressed and placed in a compressed pool. Whenever a compressed page is accessed, zswap decompresses the data from the compressed pool and places it in the main memory [10]. The Linux implementation of zswap has two key components: (i) the compression algorithm and (ii) the pool manager.

Compression algorithms. The Linux kernel supports different compression algorithms such as deflate, lz4, lzo, and lzo-rle that differ in algorithmic complexity and the ratio of data compression achieved. The deflate compression algorithm offers one of the best compression ratios but consumes comparatively higher CPU cycles to compress and decompress the data [14, 15, 32]. On the other hand, the `LZ4` compression method has a "level of effort" parameter that can trade compression speed and compression ratio [15]. The `lzo` compression method (and its evolved variant `lzo-rle`) offers a balance between compression ratio and decompression overheads [9, 14, 16].

Pool managers: A pool manager manages how compressed pages are stored in a zswap pool. A *pool* is created in physical memory to store compressed data pages by allocating pages using the buddy allocator [1]. The pool dynamically expands to store more compressed objects by allocating more pages or contracts as required. A pool manager manages the compressed objects inside a pool. Linux supports three pool memory managers: zsmalloc [24], zbud [11], and z3fold [17].

1. zsmalloc employs a complex memory management technique that densely packs compressed objects in the pool and thus has the best space efficiency. However, it has relatively high memory management overheads [24].
2. zbud is a simple and low overhead memory management pool management technique that stores a maximum of two compressed objects in a 4 KB region – limiting the maximum amount of total space saved to 50% [11].
3. z3fold is similar to zbud, but it can store three compressed objects in a 4 KB region [17] – allowing for a maximum space savings of ≈ 66%.

Linux allows users to choose a compression algorithm and pool manager for managing zswap. However, only one *active* zswap pool is supported at a time [11]. When a new compression algorithm or manager is configured, the kernel creates a new pool for new compressed pages. The old pool remains until all its data is invalidated [11].

3 Motivation

3.1 Missed opportunities for warm pages.

Data center operators report that around 10–20% of the data are hot and 20—30% of the data are cold [26, 38, 41, 42, 54]. This implies that around 50–70% of the data pages are neither hot nor cold but can be considered as *warm* pages. These warm pages can be (i) pages with relatively fewer accesses than hot pages or (ii) pages that are transitioning from hot to cold – access profiling techniques typically employ gradual cooling of hot pages by calculating the average hotness value from past profiling windows [48]. Hence, hot pages do not become cold instantaneously; rather, they are gradually aged to warm and then cold. Existing tiering solutions do not consider exploiting such warm pages for compression, thus missing significant memory TCO-saving opportunities.

3.2 Drawbacks with aggressive data placement.

A naive approach to aggressively place more data in the single compressed memory tier to increase memory TCO savings results in a significantly higher and unacceptable performance penalty (Figure 1). However, replacing a highly compressible tier with one that has a low compression ratio and low access latency tier can enable aggressive data placement in the compressed tier. However, it severely impacts the memory TCO savings due to low compression ratio.

Employing page prefetching [38] that prefetches or decompresses pages from compressed memory can mitigate high-performance penalty to the extent of prefetching accuracy. However, pages that the prefetcher fails to identify for prefetching still incur high access latency when accessed, and incorrectly prefetched pages result in decreased memory TCO savings. Nevertheless, prefetching can be additionally employed with TierScape and we note it as a future work of interest for the systems community.

3.3 Latency and cost dimensions

In byte-addressable tiers such as CXL and NVMMs, data access latency is purely determined by the memory media. However, in the case of compressed tiers, the latency of the first access to a page depends on decompression latency, which in turn depends on the compression algorithm. Because, the page should be first decompressed and placed in a target byte-addressable memory before accessing. But subsequent accesses to the same page depend on the access latency of the target byte-addressable memory tier.

In addition, for byte-addressable memory tiers, the unit cost of storing the data depends on the memory media, while the unit cost of storing the data in compressed memory tiers depends on both the compression algorithm and compressibility of the actual data. Hence, compressibility introduces an additional dimension to be considered for tiering. For instance, even if the page is cold, it is not beneficial to place it in a compressed tier if the page is not compressible.

3.4 Why multiple compressed memory tiers?

Multiple compressed memory tiers enable application- and data-specific customization. For example, multi-tenant cloud systems host diverse workloads with varying compression ratios—even across different virtual address regions within the same application. A single compressed tier with a fixed algorithm is thus suboptimal. Creating and harnessing multiple tiers with different compression algorithms can cater to the wide range of data hosted in a typical production system.

4 Concept

The core concept behind our proposal is to create and harness multiple compressed tiers in the software. Each compressed tier is created through a combination of (i) compression algorithms, (ii) memory allocator for the compressed pool, and (iii) different backing media – each providing a different access latency and memory cost per byte.

Compression algorithms. Compression algorithms with low compression ratio and, consequently, a low decompression latency are suitable for low latency tiers, but they provide only marginal memory TCO savings. Whereas other compression algorithms, such as deflate with high compression ratio and, consequently, high decompression latency, are suitable for high memory TCO savings tiers but with significantly high memory access latency.

Table 1. Different options available in Linux for setting up a compressed tier

Compression algorithm	Allocators	Backing media
Deflate, LZO, LZO-RLE, LZ4, Zstd, 842, LZ4HC	zsmalloc, zbud, z3fold	DRAM, CXL-attached memory, NVMM

Pool allocators. Allocators that densely pack compressed objects in the pool such as zsmalloc are suitable for high memory TCO saving tiers, but they have high memory management overheads, thus impacting the decompression latency. Allocators with simple and fast pool management such as zbud are suitable for low latency tiers but are less space efficient, resulting in tiers with low memory TCO savings.

Physical media. The access latency of the memory medium storing compressed pages is critical to tier performance. Storing them in DRAM offers the lowest latency [56], making it ideal for low-latency tiers, but limits overall memory TCO savings. In contrast, using cheaper, denser memory like NVMMs or CXL-attached memory improves TCO savings but increases decompression latency, making it suitable for high-TCO-saving tiers. The *key idea* for enabling aggressive memory TCO savings is to harness low-latency tiers for warm pages, balancing performance and savings, while using high-latency, high-compression tiers for cold pages.

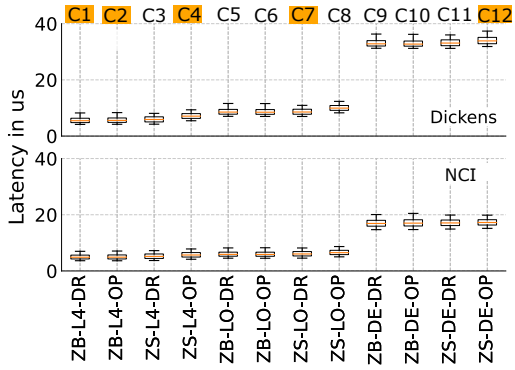
5 Characterization of Compressed Tiers

We start by comparing the access latencies and memory TCO benefits of compressed tiers with different configurations in Linux. The Linux kernel offers only two configuration parameters for a zswap compressed tier (compression algorithm and pool manager) but does not offer any control over on which hardware memory tier the pool (DRAM or NVMM) is allocated. *We extend zswap by adding a new configuration parameter – backing media, which specifies the hardware tier used for pool pages allocation.*

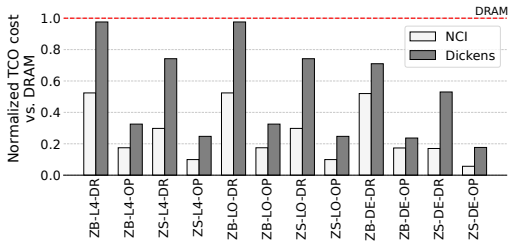
The latency of decompressing a page from zswap is primarily dominated by the compression algorithm, pool manager, backing media, and the compressibility of the data. With the available choices in Linux (as shown in Table 1), we can create a total of 63 different zswap compressed tiers (7×3×3).

To demonstrate the distinct points in the access latency, data compressibility, and unit memory usage cost spectrum we characterize 12 tiers configured based on widely used compression algorithms and pool managers. We initialize 10 GB of data in memory, compress and place them in a compressed memory tier and then measure access latency and compression ratio. We repeat this experiment for all 12 tiers using two Silesia corpus data sets [8], *nci* (highly compressible [22]) and *dickens*.

Access latency. Figure 2a shows the access latency for both *nci* and *dickens* data sets. Access latency with the lz4 algorithm is the fastest, followed by lzo, and lastly, deflate. As



(a) Access latency



(b) Memory TCO savings

Figure 2. Characterization results for 12 different compressed tiers for *dicken* and *nci* data sets. Encoding: **ZS, ZB** refers to zsmalloc and zbud pool managers, respectively. **L4, LO, DE** refers to lz4, lzo, and deflate compression algorithms, respectively. **DR, OP:** refers to DRAM and Optane [3] as the backing storage media, respectively.

expected, the performance of zbud pool manager is better than zsmalloc as zbud employs a simple algorithm that enables faster page lookup. Finally, the access latency of DRAM-backed tiers is better than those backed by the Optane [3] due to the higher media access latency in the latter [20]. For comparison, accessing a page out of DRAM has an average latency of ≈ 33ns.

Memory TCO savings. Figure 2b shows the normalized memory TCO savings of compressed tiers relative to uncompressed data in DRAM. Total TCO savings depend on data compressibility, compression algorithm, and backing media. The cost per gigabyte for storing data on Optane is typically 1/3 ∼ 1/2 of the cost of storing data on DRAM [45]. Hence, the memory TCO for Optane-backed tiers is lower than that of DRAM-backed tiers.

Furthermore, for tiers using the same compression algorithm and backing media the TCO savings depend on the pool manager for the compressed pool. For example, a tier using zsmalloc as its pool manager has a lower memory TCO than a tier using zbud. This is because zsmalloc can pack compressed objects more tightly. Finally, the deflate compression algorithm offers the best compression ratio.

5.1 Compressed tiers selection methodology

For our experiments, we use compressed tiers as used in production data centers: (i) lzo with zsmalloc by Gswap [38] (C7 in Figure 2a), and (ii) zstd with zsmalloc by TMO [54]. It is important to note that prior works such as Gswap and TMO create and use only one compressed tier backed by DRAM memory at all the times. However, to illustrate the rich and flexible placement opportunities and robustness of the TierScape proposal, we also select and experiment with four additional compressed tiers (C1, C2, C4 and C12 in Figure 2a) that are used simultaneously along with byte addressable tiers.

C1 & C12: We pick C1 as it offers the best performance and C12 for its best memory TCO savings. Other tiers with deflate compression algorithms offer a similar performance latency without additional TCO benefits, and hence we do not select any other deflate-based tiers.

C2: We select C2 as it offers the lowest latency for an Optane-backed compressed tier.

C1 & C2: C1 and C2 use zbud and lz4 as their pool manager and compression algorithm – restricting the compression ratio to 2. Hence, we select C4, which uses a fast compression method (lz4), tightly packs compressed objects (due to zsmalloc), and is stored on low-cost Optane.

6 Data Placement Models

In this section, we present two distinct data placement models that fully harness the benefits of multiple compressed tiers. The first is the *Waterfall* model (§6.1), inspired by an existing approach that supports only multiple byte-addressable tiers [36]. We extend it to include compressed memory tiers. The second is the TierScape analytical model (§6.2).

For simplicity, we assume a system with one DRAM tier, N byte-addressable tiers, and M compressed tiers, though the models can be extended to other memory types, such as HBM. Tiers are ordered from low to high latency (and correspondingly, low to high memory TCO savings), with DRAM offering the best performance but the least TCO savings.

6.1 Waterfall model

Prior memory tiering solutions use a static hotness threshold (H_{th}) to decide which pages should be demoted or promoted from the DRAM tier to other tiers. An aggressive threshold value (a high value for H_{th}) demotes more pages to the slow memory tier, saving additional memory TCO but at the cost of high performance penalty due to multiple high latency page accesses from the slow tier.

AutoTiering [36] defines static paths of promotion and demotion across multiple NUMA nodes. We extend this approach to leverage multiple compressed memory tiers. Waterfall model monitors the pages accesses for a fixed duration – a *profile window* – and classifies pages with access count greater than or equal to the threshold (H_{th}) as hot pages and rest as cold pages. The value of profile window may require tuning based on application characteristics.

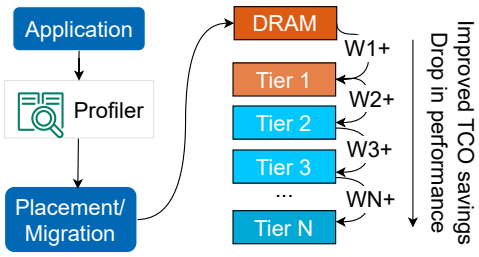


Figure 3. A high-level working of the Waterfall model

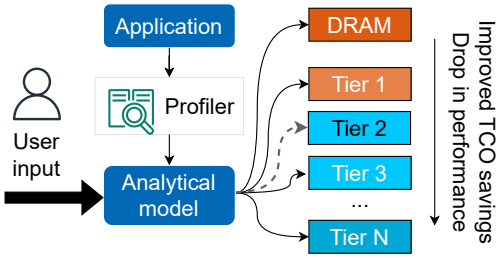


Figure 4. Page placement with the analytical model.

Promotion and demotion: At the end of the first profile window, all the cold pages are moved from DRAM to the next tier (say T1, as shown in Figure 3). This reduces the total memory TCO as some data pages have been placed in a relatively low-cost tier compared to the DRAM tier. After the end of the second profile window, all the hot pages in T1 are moved back to DRAM. All the remaining pages in T1 are then demoted (or waterfalled) to T2. This further improves the memory TCO savings as T2 is better than T1 in terms of memory TCO savings. At the end of each profile window, the model promotes all the hot pages to DRAM. Other pages in all the tiers are waterfalled to one tier below it (to a higher TCO saving tier), except for the last tier. Pages that are moved back to DRAM start their journey from T1 to the last tier again if they become cold.

Discussion: The Waterfall model has an upfront memory TCO savings as all the "cold" pages are immediately placed in a relatively low-cost tier compared to DRAM. Pages are gradually moved to tiers with better memory TCO savings. However, as cold pages are gradually aged to the best TCO saving tier, the model misses the opportunity to aggressively place cold pages directly in best memory TCO saving tiers.

6.2 TierScape's analytical model

We propose *TierScape*, an analytical data placement model that directly distributes pages (see Figure 4) across different memory tiers based on the hotness profile of the data. In addition, the model provides fine control to the users to

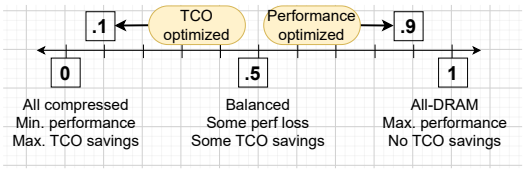


Figure 5. Tuning memory TCO with a knob in TierScape's analytical model.

balance the trade-off between memory TCO savings and performance penalty by exposing a user-guided tunable *"knob"*.

6.3 Memory TCO and performance tuning

As shown in Figure 5, the range of the knob is [0, 1]. A value of 1 indicates the model is tuned for maximum performance, which results in zero memory TCO savings as all data pages are placed in DRAM. On the other hand, a value towards 0 indicates that the model is tuned to maximize TCO savings while striving to minimize performance penalty.

6.4 Data placement modeling

The analytical model is initiated with a knob value – say $\alpha \in [0, 1]$. The theoretical memory TCO savings achievable is the difference between TCO_{max} – when all the data is in DRAM and TCO_{min} – when all the data is in the last tier. The maximum TCO savings (or MTS) can be defined as follows:

$$\text{MTS} = \text{TCO}_{max} - \text{TCO}_{min} \tag{1}$$

The analytical model can be tuned to achieve TCO savings within [0, MTS] by configuring α. At the end of each profile window, the model uses α and the profiled data to solve the following *Integer Linear Program* or ILP:

$$\begin{aligned} &\text{minimize} \quad perf\_ovh \\ &\text{subject to} \quad \mathtt{TCO} \leq (\mathtt{TCO}_{min} + \alpha * \text{MTS}) \end{aligned} \tag{2}$$

To solve Equation 2, we start by formally defining performance overhead ($perf\_ovh$) and the memory TCO.

6.5 Modeling performance overheads

An application executes optimally (in terms of memory accesses) when all its load operations are directly from DRAM (instead of a slow tier). We refer to this performance as $perf\_opt$.

For ease of discussion, consider a system with 3 memory tiers: DRAM (TD), NVMM (TN), and a compressed memory tier (CT). The optimal performance will be when all the memory accesses are from DRAM:

$$perf\_opt = \mathtt{MemAcc}_{tot} * \mathtt{Lat}_{\mathtt{TD}} \tag{3}$$

Here, $\mathtt{MemAcc}_{tot}$ is the total number of memory accesses, and $\mathtt{Lat}_{\mathtt{TD}}$ is the latency of DRAM. However, when all the tiers are in use, the performance can be defined as:

$$\begin{aligned} perf'' = {} & \mathtt{MemAcc}_{\mathtt{TD}} * \mathtt{Lat}_{\mathtt{TD}} + \mathtt{MemAcc}_{\mathtt{TN}} * \mathtt{Lat}_{\mathtt{TN}} \\ & + \mathtt{MemAcc}_{\mathtt{CT}} * (\mathtt{Lat}_{\mathtt{CT}} + \mathtt{Lat}_{\mathtt{TD}}) \end{aligned} \tag{4}$$

Here, $\mathtt{MemAcc}_{\{\mathtt{TD/TN/CT}\}}$ are total memory accesses from DRAM, NVMM, and the compressed tier, respectively, and they sum up to $\mathtt{MemAcc}_{tot}$. $\mathtt{Lat}_{\mathtt{TN}}$ and $\mathtt{Lat}_{\mathtt{CT}}$ are the access latency for NVMM and compressed tier, respectively. A page in a compressed tier needs decompression before it can be accessed. Hence, a page from a compressed tier is always placed in DRAM after decompression (if there is space). Hence, the total cost of accessing a page is the latency of decompressing and reading it from DRAM. However, it is also possible to place a decompressed page in NVMM (when DRAM is full). In that case, the additional latency will be the latency of the NVMM tier. The performance overhead can be defined as:

$$perf\_ovh = perf\_opt - perf'' \tag{5}$$

$$= (\mathtt{Lat}_{\mathtt{TN}} - \mathtt{Lat}_{\mathtt{TD}}) * \mathtt{MemAcc}_{\mathtt{TN}} + \mathtt{Lat}_{\mathtt{CT}} * \mathtt{MemAcc}_{\mathtt{CT}}$$

$$= \delta_{\mathtt{TN}} * \mathtt{MemAcc}_{\mathtt{TN}} + \mathtt{Lat}_{\mathtt{CT}} * \mathtt{MemAcc}_{\mathtt{CT}} \tag{6}$$

Here, $\mathtt{Lat}_{\mathtt{CT}}$ is the latency of accessing a page from the compressed tier and placing it in DRAM.

Generalization: Say there are N byte-addressable tiers and M compressed tiers in a system. In such systems, the total performance overhead will be:

$$\boxed{perf\_ovh = \sum_{x=1}^{N} (\delta_{\mathtt{TN}_x} * \mathtt{MemAcc}_{\mathtt{TN}_x}) + \sum_{y=1}^{M} (\mathtt{Lat}_{\mathtt{CT}_y} * \mathtt{MemAcc}_{\mathtt{CT}_y})} \tag{7}$$

Here, $\delta_{\mathtt{TN}_x}$ is the latency difference between DRAM and $\mathtt{TN}_x$. Note that the latency of DRAM will always be smaller as, by definition, it is the fastest tier in our setup. Hence, δ will always be a positive number.

6.6 Modeling TCO

Again, for ease of discussion, consider a 3-tier system with the same tier configuration as earlier. The total memory TCO can be reduced by placing some pages in the NVMM and compressed tier.

$$\begin{aligned} \mathtt{TCO} = {} & \mathtt{P}_{\mathtt{TD}} * \mathtt{USD}_{\mathtt{TD}} + \mathtt{P}_{\mathtt{TN}} * \mathtt{USD}_{\mathtt{TN}} \\ & + \mathtt{P}_{\mathtt{CT}} * \mathtt{C}_{\mathtt{CT}} * \mathtt{USD}_{\mathtt{CT}} \end{aligned} \tag{8}$$

$$\mathtt{P}_{total} = \mathtt{P}_{\mathtt{TD}} + \mathtt{P}_{\mathtt{TN}} + \mathtt{P}_{\mathtt{CT}} \tag{9}$$

Here, $\mathtt{P}_{total}$ is the total number of pages. $\mathtt{P}_{\mathtt{TD}}$, $\mathtt{P}_{\mathtt{TN}}$, and $\mathtt{P}_{\mathtt{CT}}$ are the number of pages in the DRAM, NVMM, and compressed tier respectively. $\mathtt{C}_{\mathtt{CT}} \in (0, 1]$ is the compression ratio[1] of the tier. The cost of storing a page in the compressed tier is reduced by the compressibility of the data on that tier.

[1]Ratio of compressed size to original size. It cannot be more than 1 as zswap implementation rejects highly uncompressible objects.

Generalization: Say a system has N byte-addressable tiers and M compressed tiers. In such systems, total memory TCO

$$TCO = P_{TD} * USD_{TD} + \sum_{x=1}^{N}(P_{TN} * USD_{TN}) + \sum_{y=1}^{M}(P_{CT} * C_{CT} * USD_{CT}) \quad (10)$$

As in prior work [38, 42, 48, 54], the model assumes that the number of memory accesses to pages in the next profile window remains proportional to the accesses seen in the last profile window.

6.7 Discussion

We account for the cost of page migration in a separate filter that pre-processes the output of the ILP model before triggering migrations. We avoid adding constraints in the ILP model to handle such scenarios as it makes ILP solving more time-consuming and computationally intensive. The filter ensures that the number of pages placed in a tier is bounded by the tier capacity and also considers resource contention on memory tiers and avoids moving pages to already pressured tiers. We briefly discuss the key benefits of the TierScape's ILP-based analytical model.

Fine tuning memory TCO and performance penalty trade-off: Analytical model offers a tunable knob (α) which can be configured as per the requirement of the user. The "knob" conveys the amount of TCO targeted to be saved while striving to minimize performance loss.

Quick convergence: The model quickly converges to optimal data placement based on the profiled hotness of the data. Cold data are directly placed in the most optimal tier as per the constraints instead of "waterfalling" on multiple tiers.

7 Implementation

7.1 Linux kernel changes

Tier's backing media: As discussed in Section 2, the Linux kernel configures a compressed memory tier using the compression algorithm [35]. We augment the zswap subsystem to add an additional parameter to specify a *backing media* which can be NVMM or CXL-attached memory. With this compressed objects can be allocated on any hardware-based memory tiers allowing more flexibility in performance and memory TCO savings tradeoff.

Multiple active compressed tiers: Linux kernel only supports a single active zswap tier. Upon creation of a new compressed tier, all new data compression requests are directed to the newly created tier. The kernel deletes the old tiers if they are empty. We modify the zswap subsystem to support multiple active compressed zswap tiers and also allow multiple compressed tiers to coexist. We augment the `struct page` structure with a *tier_id* field which is updated by a modified `madvise()` function. During page compression, the zswap module reads this field and places the compressed page in the intended tier. During decompression, the swap entry contains the tier information, including the pool details and other relevant information to handle the fault [35].

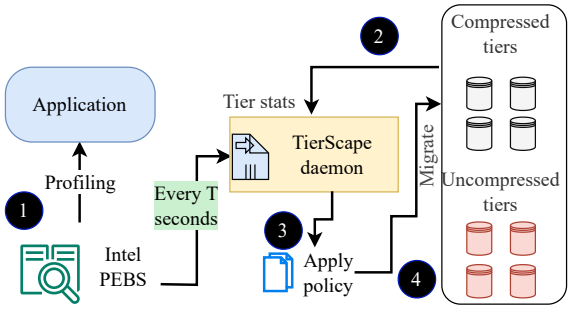


Figure 6. A high-level working of TierScape.

Page migration between tiers: We enhance the kernel to allow the migration of pages between two compressed tiers. Currently, we follow a naive approach while migrating pages between compressed tiers by first decompressing the page from the source tier and then compressing it again and placing it in the destination tier. This can be further optimized by skipping the decompression step if the source and destination tiers use the same compression algorithm.

Tiers statistics: We added support in the zswap subsystem to collect per-tier statistics such as the number of pages in the tier, the size of the compressed tier, and total faults.

7.2 TS-Daemon

As shown in Figure 6, we implement our TierScape logic as a daemon (TS-Daemon). TS-Daemon uses the hardware counters to profile the memory access pattern of an application for a fixed time window (profile window). Specifically, it uses Intel PEBS [34] to monitor `MEM_INST_RETIRED.ALL_LOADS` and `MEM_INST_RETIRED.ALL_STORES`. These events report the virtual address of the page on which the event was generated [7]. We use a sampling rate of 5K, which provides a good balance of sample quality and performance overhead [48].

Based on the collected profile, TS-Daemon applies the Waterfall or TierScape placement model on the collected hotness profile to decide the destination tiers for the memory regions. Based on the model's outcome, TS-Daemon uses the kernel APIs described above to manage memory placement.

Regions. In order for efficient management of the address space of an application, TS-Daemon operates at a granularity of 2 MB regions instead of 4 KB pages as commonly followed in other memory tiering solutions [48]. The hotness of 2 MB region is an accumulated value of the hotness of each 4 KB page in it. TS-Daemon performs data migration to and from compressed tiers at the granularity of 2 MB regions.

Table 2. Description of the workloads and configurations.

Workloads	Description	RSS
Memcached [12]	A commercial in-memory object caching system.	42GB/ 58GB
Redis [13]	A commercial in-memory key-value store.	90 GB
BFS [50]	Traverse graphs generated by web crawlers. Use breadth-first search.	30 GB
PageRank [50]	Assign ranks to pages based on popularity (used by search engines).	30 GB
XSBench [52]	A key computational kernel of the Monte Carlo neutron transport algorithm	119 GB
GraphSAGE [30]	A framework for inductive learning on large graphs.	40 GB

7.3 Data placement models

Waterfall model. We implement the waterfall model in the TS-Daemon. The input to the model is a hotness threshold value – H_{th}. The value controls the pages that are to be evicted from, for example, DRAM to Tier 1. TS-Daemon maintains the tier data for all the regions and uses it to waterfall (demote to the next tier) the regions at the end of a profile window.

Analytical model. We implement the analytical model in C++ using the OR-Tools from Google [46]. The input to the model is the hotness profile of the application, tier stats (e.g., compressibility ratio, cost of the media backing the compressed tier, and access latency), list of regions, and a value for the knob (α). The model outputs a recommendation with a destination tier for each region. We discuss the overhead of the ILP solver in Section 8.4.

8 Evaluation

We demonstrate the capability of TierScape for holistic and flexible data placement across multiple tiers with two distinct tier configurations.

1. Standard mix of tiers: We use two byte-addressable tiers (DRAM and Intel Optane) along with two compressed tiers. Compressed tier CT-1 is based on Gswap [38] and CT-2 is based on TMO [54](§ 5.1). CT-1 is a low-latency, low-compression tier with physical backing media as DRAM – ideal for storing warm pages. CT-2 is a high-latency, high-compression tier with Optane as the physical backing media – ideal for storing cold pages.

2. Spectrum of compressed tiers: To demonstrate the capability, TierScape we experiment with six memory tiers: one byte-addressable tier (DRAM) along with five compressed tiers: C1, C2, C4, C7 and C12 (refer to Section 5.1 and Figure 2a for details).

8.1 Experiment setup

We have implemented a modified version of HeMem [48], Gswap [38], and TMO [54] – henceforth referred to as HeMem*, GSwap*, TMO*, respectively – to compare against TierScape. We use telemetry from Intel PEBS [34] as the common page access profiling data for a fair comparison. HeMem*, GSwap*, and TMO* star uses NVMM, CT-1, and CT-2 as the slow memory tier, respectively.

Hotness threshold: Instead of a static threshold, which is not optimal for different applications and may cause unexpected performance overhead [26], we use a *percentile*-based tiering threshold of 25^{th} percentile for the prior works. In a profile window, any region with an access count value exceeding the 25^{th} percentile will be promoted to DRAM. All the other regions will be pushed to the slow memory tier. For the Waterfall model, we use the same threshold of 25^{th} percentile to determine promotion/demotion between memory tiers. For the analytical model, we evaluate with two different configurations for the tunable knob (α); a small value indicates the TCO-preferred setting (AM-TCO) and a large value indicates preference for performance (AM-perf).

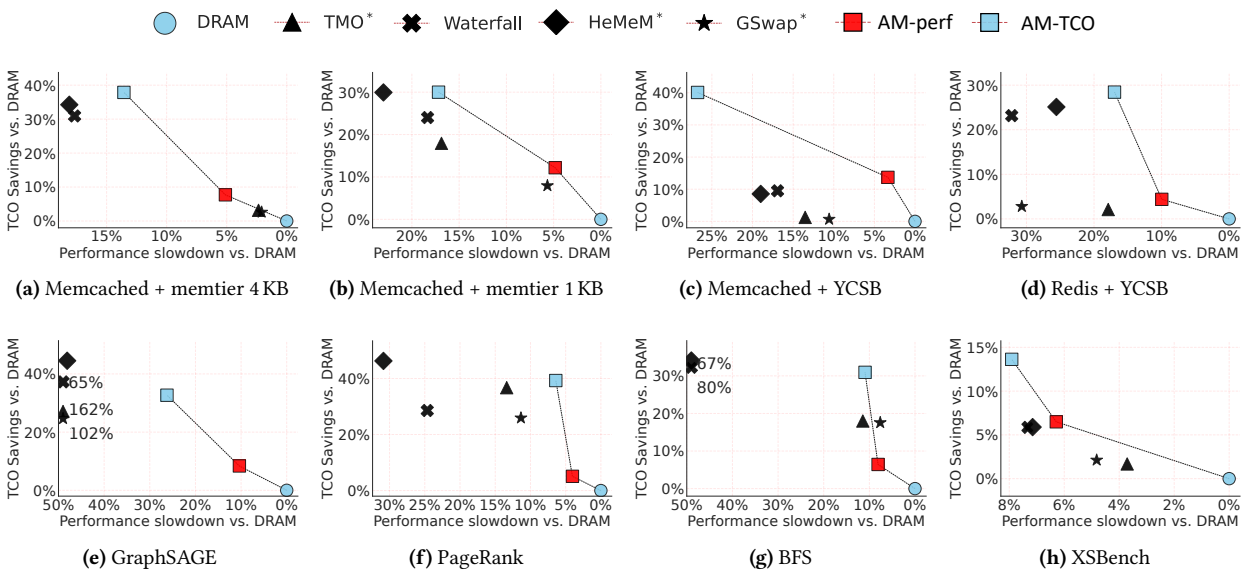
System setup. We use a tiered memory system with Intel Xeon Gold 6252N with 2 sockets, 24 cores per socket, and 2-way HT for a total of 96 cores. It has a DRAM-based fast memory tier with 384 GB capacity and an NVMM memory tier with Intel's Optane DC PMM [3] configured in flat mode (i.e., as volatile main memory) with 1.6 TB capacity. We run Fedora 30 and use a modified version of Linux kernel 5.17.

Workloads: Table 2 shows the real-world benchmarks and their configuration used to evaluate TierScape. For Memcached we use two different workload generators: memtier [5, 13] and YCSB [23]. We initialize Memcached with ≈42 GB of key-value pairs with a size of 1 KB for YCSB and 1 KB and 4 KB for memtier. We use "*workloadc*" to load the data and generate access requests using a Zipfian distribution for YCSB. For memtier, we generate the requests in a Gaussian distribution [6]. We report the throughput and latency numbers as reported by memtier and YCSB. We use PageRank and BFS from the Ligra suite of graph benchmarks [50]. Input graphs for both graph workloads are generated using the standard rMat graph generator [4]. We report the geometric mean of the time taken to execute multiple rounds. For XSBench, we use the XL option as provided by the workload [52] and for GraphSAGE, we use the ogbn-products dataset [33] as the input.

TCO calculation: We use Equation 8 to calculate the memory TCO. We capture the size of DRAM, NVMM, and all the compressed memory tiers to compute the cost of storing data in the respective compressed tiers. We set the per-GB cost of NVMM as 1/3 of DRAM [45].

8.2 Standard mix of tiers

Figure 7 compares relative performance and memory TCO savings for different tiering techniques. Note that values on the x-axis are on a decreasing scale. the analytical model, in its TCO-preferred setting (AM-TCO), outperforms both traditional tiering solutions and Waterfall model in terms



(a) Memcached + memtier 4 KB **(b)** Memcached + memtier 1 KB **(c)** Memcached + YCSB **(d)** Redis + YCSB

(e) GraphSAGE **(f)** PageRank **(g)** BFS **(h)** XSBench

Figure 7. Performance slowdown and memory TCO savings w.r.t to DRAM for different tiering techniques. Points towards the top-right corner of the plots are optimal (indicating high memory TCO savings and a low performance overhead). The lines connecting AM-TCO and AM-perf are for ease of readability and comparison.

of performance and memory TCO savings. AM-perf outperforms traditional tiering solutions and Waterfall by 73% and 108% in terms of performance, and by 45% and 24% in terms of TCO savings, respectively. In AM-perf, the performance of the workloads is improved by an average of 319.39% and 403% compared to traditional tiering solutions and the Waterfall model, respectively.

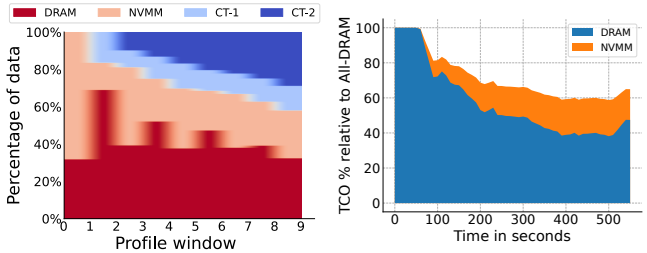
Better TCO savings with similar performance: In Memcached (memtier with 1K key value size), the maximum memory TCO savings is 29.94% with HeMem*. But it suffers a ≈ 23.01% performance slowdown. The Waterfall model achieves a TCO saving of ≈ 24.01% while incurring a performance loss of ≈ 18.34%. AM-TCO achieves a TCO savings of 30.00% while incurring a performance loss of 17.18%. AM-perf achieves a TCO savings of 12.17% while incurring a performance loss of just 4.84%.

Better performance with similar TCO savings: In PageRank, in a tiering system, GSwap* offers the least performance slowdown of 11.35% while saving 25.91% memory TCO. HeMem* offers the highest amount of TCO savings, 46.27%, but a high performance cost of 30.87%. Waterfall offers a TCO savings of 28.57% with a performance overhead of 24.65%. Whereas, AM-TCO incurs a performance slowdown of 6.04% while saving 39.27% of memory TCO. AM-perf incurs a performance slowdown of only 4.04% and offers a memory TCO savings of 5.09%.

8.2.1 Page placement recommendations.

We analyze the TierScape's placement recommendation.

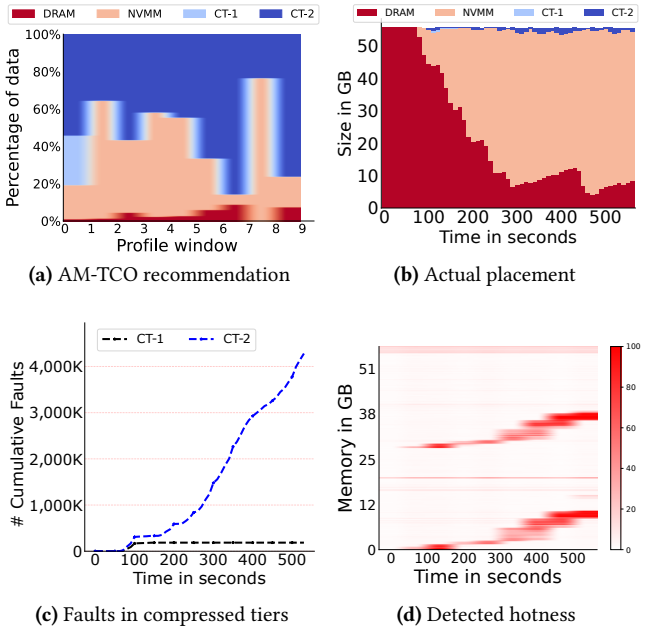
Waterfall model: Figure 8a shows the Waterfall recommendation in each profiling window for Memcached with YCSB. We see a good utilization of all the memory tiers. This is because data is demoted from DRAM and is "waterfalled" between the tiers, eventually reaching the last tier. Here, it can be observed from Figure 8a that pages are first "waterfalled" to the NVMM tier, and then they are gradually aged to better memory TCO-saving tiers, resulting in a reduction in memory TCO (see Figure 8b). Note that the TCO plot just shows DRAM and NVMM as compressed tiers are also stored on these memory tiers. A compressed tier on DRAM will result in lower memory TCO due to compression.



(a) Waterfall recommendation **(b)** Waterfall Memory TCO

Figure 8. Data placement recommendations for Memcached with YCSB by Waterfall and the corresponding memory TCO savings.

AM-TCO model: AM-TCO, based on the hotness profile, number of faults to the compressed tiers, data compressibility, and access latency of each tier, it recommends placing



(a) AM-TCO recommendation **(b)** Actual placement

(c) Faults in compressed tiers **(d)** Detected hotness

Figure 9. Figure showing (a) AM-TCO model recommendations for page placement, (b) actual page placements in different tiers, (c) faults in the compressed tiers, and (d) memory TCO trend for Memcached with YCSB with AM-TCO.

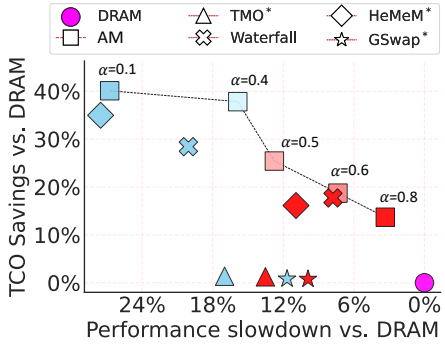


Figure 10. Performance of TierScape's analytical model for with different values for the tuning parameters (α) – indicated with different colors, for Memcached with YCSB. Figure also shows performance of HeMem*, GSwap*, TMO* and Waterfall model for two different values of hotness threshold indicated by two different colors – and .

less than 5% of data in DRAM and the rest in other tiers (Figure 9a). It consistently recommends placing most of the pages either in the NVMM tier or CT-2 the tier. Figure 9a shows the recommendation for placing Memcached (with YCSB) pages in DRAM, NVMM, and compressed tiers using AM-TCO configuration. The rationale behind the recommendation decision can be understood based on the hotness pattern in Figure 9d. Only a small set of pages are hot or warm, and the rest are cold. Hence, AM-TCO recommends placing most of the data either in the NVMM or CT-2 as CT-1 is stored on DRAM and has a higher cost associated with it.

8.2.2 Deep dive. We pick Memcached/YCSB, a unique case where the ground reality does not reflect the model's recommendation (Figure 9b), to do a deep dive. Figure 9b shows the actual distribution of the pages across the tiers after migrating the pages as per the recommendations. Here, we see that most of the pages are placed in the NVMM tier, although the model recommends placing a significant number of pages in CT-2. To understand this behavior, we look at the trend of faults to the compressed tiers in Figure 9c which shows the cumulative faults incurred in both the compressed tiers. Here, we can see that the total number of faults to CT-2 keeps increasing. This indicates that pages are being moved to CT-2; however, due to the shifting memory access pattern (see Figure 9d), they are immediately faulted upon and moved to DRAM/NVMM. Therefore, despite TierScape's migration of pages to CT-2, as advised by the AM-TCO model (Figure 9a), due to the application's access pattern continuously shifting (Figure 9d), the application's access pattern almost immediately triggers a fault on the compressed tier. However, if the access pattern remains stable, which is the case in other benchmarks, the ground reality almost reflects the model recommendation.

8.2.3 Multi-objective tuning with TierScape. To demonstrate the capability of the knobs we tune it with five different knob values that achieve different memory TCO savings with corresponding performance overhead (as shown in Figure 10) We also execute Waterfall and other tiering solutions with two different hotness threshold values (25^{th} and 75^{th} percentile). TierScape consistently outperforms two-tier models and Waterfall model in terms of achieved TCO savings and incurred performance overhead. In addition, TierScape also demonstrates the achievable spectrum of performance and TCO benefits points.

8.2.4 Impact on the tail latencies. One of the key requirements in a data center is to maintain an SLA guarantee on the tail latencies of an application. In TierScape systems, the total number of faults in the slowest tier can impact the tail latency of the application. Figure 11 shows the average, 95^{th} and 99.9^{th} percentile latency for Redis as reported by YCSB. It can be observed that both the configurations of TierScape outperform all tiering solutions placement models and Waterfall. TierScape carefully scatters the pages across tiers based on the hotness values of the pages. Hence, it performs better than tiering and Waterfall.

TMO* vs. HeMem*: The average latency for TMO* is lower than HeMem*, even though the former uses a compressed tier on NVMM, whereas the latter directly uses NVMM. This

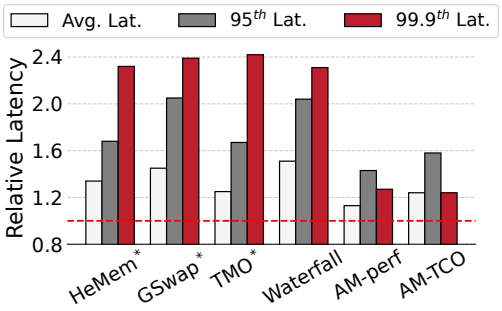
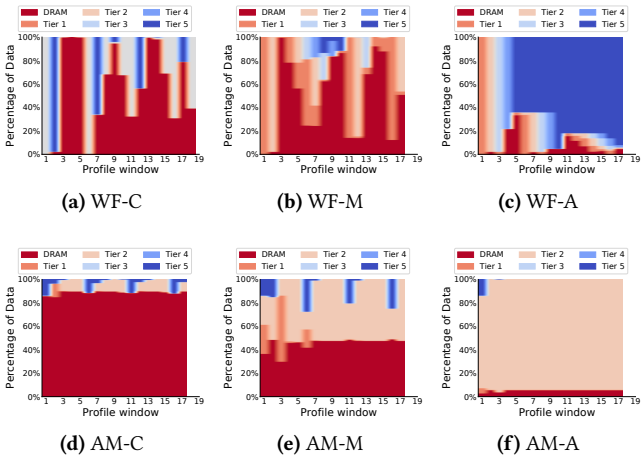


Figure 11. Latency data for Redis normalized to DRAM.

is because, in TMO*, the page is promoted from the compressed tier after the first fault and placed in DRAM. As a result, all the subsequent accesses to the page will be from the fast DRAM tier. Hence, the average latency of TMO* is less than that of HeMem*.

8.3 Spectrum of compressed tiers

We demonstrate TierScape's performance when six tiers (DRAM with five compressed tiers) are used. We compare the performance of TierScape's Waterfall (WF) and analytical model (AM) with GSwap* tiering (GS). Each tiering solution is executed in three settings: conservative (-*C*), moderate (-*M*) and aggressive (-*A*) settings by varying the hotness threshold (25, 50, and 75 percentile and α value as 0.9, 0.5, and 0.1). Figure 12 shows the data placement recommendation for TierScape's placement model.



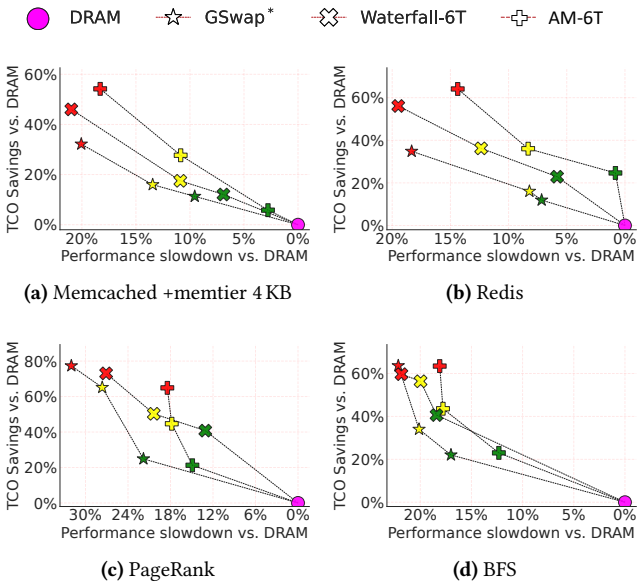
(a) WF-C **(b)** WF-M **(c)** WF-A

(d) AM-C **(e)** AM-M **(f)** AM-A

Figure 12. Data placement recommendations for Memcached by waterfall and analytical models with three different aggressiveness for memory TCO savings.

8.3.1 TCO savings and Performance overheads.

TierScape's outperforms GSwap* by saving more memory TCO



(a) Memcached +memtier 4 KB **(b)** Redis

(c) PageRank **(d)** BFS

Figure 13. Performance slowdown and TCO savings w.r.t. DRAM, GSwap*, Waterfall, and analytical model with six memory tiers. Colors ●, ●, and ● indicate different hotness thresholds with increasing aggressiveness, respectively.

with similar or better performance or with better performance at a similar or better TCO (see Figure 13).

TCO savings at similar performance: For example, the maximum TCO savings GS-A offers for Redis is 34.84% with a performance slowdown of 18.33%. On the other hand, WF-A achieves a TCO saving of 56.11% (21.27 percentage points better) while incurring a performance loss of 19.48% (only 1.15 additional percentage points).

Better performance with similar or more TCO savings: In PageRank, GS-C offers the least performance slowdown of 21.82% while saving 24.86% memory TCO. Whereas, WF-C offers a better trade-off than GS-C with 13.09% performance slowdown and 40.78% memory TCO savings. AM-C, incurs performance slowdown of only 14.91% and offers a TCO savings of 21.26% Also, it can be observed in BFS that AM-*A* and GS-*M* result in around 63% memory TCO savings, but the former performs better by 4.05 percentage points (22.15% vs. 18.10%). This demonstrates that with TierScape, more warm pages can be placed in compressed tiers to achieve better memory TCO savings without hurting performance.

8.3.2 Why multiple compressed tiers?

A particular set of tiers may benefit one application, and for some other application, a completely different tier might make sense. We observe different levels of TCO benefits and performance overheads when using 2 compressed tiers (Figure 7) vs. when using 5 compressed tiers (Figure 13). For example, with the

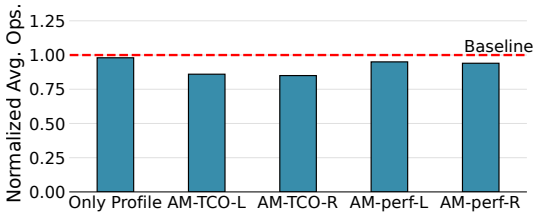


Figure 14. Performance impact due to TS-Daemon (which includes profiling + modeling + migration) in TCO mode and perf mode and with ILP solver on the local machine or on a remote machine. Notation: AM-{TCO/perf}-{**L**ocal/**R**emote}. Baseline is no profiling and no migration.

additional compressed tiers, the total achievable TCO benefits for Memcached and Redis increased to 55% and 65% from 40% and 30%, respectively.

8.4 TierScape Tax

The tax includes telemetry data collection, post-processing, and page migration overhead—including (de)compression for compressed tiers. Pages decompressed due to on-demand faults are excluded from TS-Daemon, as their cost is accounted in the benchmark execution time.

ILP solver tax: Solving an ILP can be compute-intensive depending on the problem and constraints. We use Memcached with memtier to evaluate the performance impact of AM-TCO, performing 1 M read operations across all cores.

1. **Only-profiling**: TierScape uses *perf* to profile access patterns without migration.
2. **Local**: Profiling + AM-TCO and AM-perf with the ILP solver running locally.
3. **Remote**: Profiling + AM-TCO and AM-perf with the ILP solver running remotely.

As shown in Figure 14, profiling introduces minimal overhead. We observe negligible performance difference between local and remote solver execution, as our ILP formulation uses simple constraints—consuming less than 0.3% of a single CPU (via *perf*) and use ≈480 MB of memory.

9 Discussions and future work

This paper proposes a paradigmatic shift toward building server systems with multiple compressed memory tiers. Through experimental evidence, it demonstrates that such systems can enable application- and service-specific memory TCO savings and performance trade-offs. The core idea is that memory tiers with distinct characteristics can be constructed using different compression algorithms, pool allocators, and backing media (e.g., DRAM, NVMM, CXL).

This proposal opens several research directions, including: (i) selecting the optimal set of compressed tiers, (ii) choosing tiers based on data compressibility, (iii) determining the ideal number of tiers, (iv) evaluating the impact of different compression algorithms, and (v) support for co-located applications. Each of these directions warrants in-depth investigation and is noted as future work.

In this paper, we focus on the framework and methodology for harnessing multiple compressed tiers, demonstrating benefits via intelligent page placement using both a basic waterfall model and a more advanced analytical model.

10 Related Work

Several tiered memory systems have been proposed in recent years [18, 19, 21, 26, 28, 29, 36–39, 41, 42, 48, 54, 55], along with data placement and migration policies to optimize performance and memory TCO. Most of the prior works are based on a two-tier system, where the first tier consists of low latency and costly DRAM memory while the second tier consists of high latency but cheaper memory tiers backed by NVMMs [3] or CXL-attached memory [25, 27, 43]. Recently, memory tiering using a compressed memory tier has been explored by a hyper-scale data center provider [38].

Hardware-based memory tiering with NVMMs [18, 26, 36, 54, 55] or CXL-attached memory [19, 21, 28, 41, 42] lacks the flexibility in the software to define memory tiers with distinct access latency, as access latency is determined by the underlying storage media. Prior proposals employ different telemetry techniques to identify hot and cold data [31, 34, 40, 44, 57]. HeMem [48] accumulates access information from PEBS [34] into larger regions to reduce the overheads of tracking pages at 4 KB granularity.

Prior work from Google [38] proposes tiering with DRAM and a single compressed tier to improve memory TCO savings for cold data. It periodically scans the `ACCESSED` bit in page tables to identify cold pages and migrates them to a DRAM-backed compressed tier. An AI/ML-based prefetching technique is used to proactively move pages back to DRAM.

11 Conclusion

We conclude with comprehensive experimental evidence that systematic data placement by harnessing multiple compressed tiers is an optimistic way forward to tame the memory TCO and performance trade-offs in modern data centers. We also point out a few interesting research directions to extract the full potential of multiple compressed memory tiers.

Acknowledgments

We thank our shepherd, Baptiste Lepers, and the anonymous review committee of EuroSys'26 for their insightful feedback, which significantly contributed to improving the quality of this paper. We also thank the artifact evaluation committee for testing the code and sharing their feedback.

References

[1] [n. d.]. Buddy memory allocation - Wikipedia. https://en.wikipedia.org/wiki/Buddy_memory_allocation. (Accessed on 08/10/2023).

[2] [n. d.]. Decadal Plan for Semiconductors. https://www.src.org/about/decadal-plan/decadal-plan-full-report.pdf.

[3] [n. d.]. Intel® Optane™ DC Persistent Memory Product Brief. https://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf. (Accessed on 08/01/2023).

[4] [n. d.]. jshun/ligra: Ligra: A Lightweight Graph Processing Framework for Shared Memory. https://github.com/jshun/ligra. (Accessed on 08/10/2023).

[5] [n. d.]. Memcached: AWS Graviton2 benchmarking - Infrastructure Solutions blog - Arm Community blogs - Arm Community. https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/memcached-benchmarking-aws-graviton2-50-p-p-gains. (Accessed on 08/10/2023).

[6] [n. d.]. New in memtier benchmark: Pseudo-Random Data, Gaussian Access Pattern and Range Manipulation. https://redis.com/blog/new-in-memtier_benchmark-pseudo-random-data-gaussian-access-pattern-and-range-manipulation/.

[7] [n. d.]. PerfMon Events. https://perfmon-events.intel.com/. (Accessed on 08/01/2023).

[8] [n. d.]. Silesia compression corpus. https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia. (Accessed on 08/06/2023).

[9] [n. d.]. ZRAM Will See Greater Performance On Linux 5.1 - It Changed Its Default Compressor - Phoronix. https://www.phoronix.com/news/ZRAM-Linux-5.1-Better-Perform. (Accessed on 08/04/2023).

[10] [n. d.]. zswap — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/mm/zswap.html. (Accessed on 07/22/2023).

[11] [n. d.]. zswap — The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.8/vm/zswap.html?highlight=zbud. (Accessed on 08/04/2023).

[12] 2019. memcached - a distributed memory object caching system. https://memcached.org/. (Accessed on 11/18/2019).

[13] 2019. Redis. https://redis.io/. (Accessed on 11/18/2019).

[14] 2023. Lempel–Ziv–Oberhumer - Wikipedia. https://en.wikipedia.org/wiki/Lempel-Ziv-Oberhumer. (Accessed on 08/04/2023).

[15] 2023. lz4/lz4: Extremely Fast Compression algorithm. https://github.com/lz4/lz4. (Accessed on 08/04/2023).

[16] 2023. LZO [LWN.net]. https://lwn.net/Articles/545878/. (Accessed on 08/04/2023).

[17] 2023. z3fold — The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.8/vm/z3fold.html. (Accessed on 08/04/2023).

[18] 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 631–644. doi:10.1145/3037697.3037706

[19] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. doi:10.1145/3533737.3535090

[20] Shoaib Akram. 2021. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Trans. Archit. Code Optim.* 18, 3, Article 29 (apr 2021), 26 pages. doi:10.1145/3451342

[21] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2023. Exploiting CXL-Based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. doi:10.1145/3545008.3545054

[22] David Barina. 2023. Experimental lossless data compressor. *Microprocessors and Microsystems* 98 (2023), 104803. doi:10.1016/j.micpro.2023.104803

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

[24] Jonathan Corbet. 2012. The zsmalloc allocator [LWN.net]. https://lwn.net/Articles/477067/. (Accessed on 08/04/2023).

[25] CXL. 2023. Compute Express Link. https://www.computeexpresslink.org/.

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

[27] Samsung Electronics. 2022. *Samsung Electronics Introduces Industry's First 512GB CXL Memory Module.*

[28] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57. doi:10.1109/MM.2023.3237491

[29] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. Association for Computing Machinery, New York, NY, USA, 79–92. doi:10.1145/2731186.2731191

[30] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)* (Long Beach, California) *(NeurIPS '17)*. Neural Information Processing Systems Foundation, Long Beach, CA, USA, 1–11. https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

[31] Christian Hansen. 2018. *Linux Idle Page Tracking.* https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt

[32] Danny Harnik, Ety Khaitzin, Dmitry Sotnikov, and Shai Taharlev. 2014. A Fast Implementation of Deflate. In *2014 Data Compression Conference.* 223–232. doi:10.1109/DCC.2014.66

[33] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[34] Intel. 2023. *PEBS (Processor Event-Based Sampling) Manual.* https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-734.html

[35] Seth Jennings. 2013. The zswap compressed swap cache [LWN.net]. https://lwn.net/Articles/537422/. (Accessed on 08/04/2023).

[36] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21).* USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[37] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN*

International Symposium on Memory Management (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 66–79. doi:10.1145/3459898.3463907

[38] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. http://doi.acm.org/10.1145/3297858.3304053

[39] 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*. 17–34.

[40] Michael Lespinasse. 2023. *V2: idle page tracking / working set estimation*. https://lwn.net/Articles/460762/

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

[42] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[43] Inc. Micron Technology. 2022. *Micron Launches Memory Expansion Module Portfolio to Accelerate CXL 2.0 Adoption*.

[44] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3366626.3368125

[45] Bo Peng, Yaozu Dong, Jianguo Yao, Fengguang Wu, and Haibing Guan. 2022. FlexHM: A Practical System for Heterogeneous Memory with Flexible and Efficient Performance Optimizations. *ACM Trans. Archit. Code Optim.* 20, 1, Article 13 (dec 2022), 26 pages. doi:10.1145/3565885

[46] Laurent Perron and Vincent Furnon. 2023. *OR-Tools*. Google. https://developers.google.com/optimization/

[47] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John F. J. Mellor, Irina Higgins, Antonia Creswell, Nathan McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, L. Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, N. K. Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Tobias Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew G. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem W. Ayoub, Jeff Stanway, L. L. Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. 2021. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *ArXiv* abs/2112.11446 (2021).

[48] 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

[49] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 803–817. doi:10.1145/3627703.3650075

[50] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/2442516.2442530

[51] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Anand Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *ArXiv* abs/2201.11990 (2022).

[52] John Tramm, Andrew Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The development and verification of a performance abstraction for Monte Carlo reactor analysis.

[53] VMware. 2024. Memory Tiering: Power Your Server Infrastructure with Memory Innovations. https://www.vmware.com/explore/video-library/video/6360757998112. Accessed: 2025-05-14.

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

[55] 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

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.

[57] Yu Zhao. 2022. Multigenerational LRU Framework. https://lwn.net/Articles/880393/.

A Artifact Appendix

A.1 Abstract

The artifact evaluates the capabilities, models, and different configurations of those models presented in the paper. Although presented for Intel Optane, the artifact can be evaluated with any memory tier: HBM, CXL-attached memories, etc., with appropriate changes in the *config* files.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at: https://github.com/IntelLabs/tierscape/ and https://zenodo.org/records/17222213

A.2.2 Hardware dependencies. The system should contain

1. Intel Optane memory tier
2. Root permission to install a patched Linux kernel

A.2.3 Software dependencies. To run the analytical model, we need Google OR-Tools.

A.2.4 Benchmarks. The current version of artifact supports:

1. Masim: A microbenchmark to test the setup process
2. Memcahed: To reproduce the results in the paper.

A.3 Set-up

Please see https://github.com/IntelLabs/tierscape/#configuration for a detailed guide in setting up the artifact. Change the following variables as per your system:

```
vim <root_dir>/skd_daemon/skd_config.sh
# Change these============
FAST_NODE="0"
SLOW_NODE="1"
# path to perf binary as per the system
PERF_BIN="/usr/bin/perf"
# ====================
```

A.3.1 Running Tiering with Memcached – No Kernel Patch. TierScape setup and Installation

```
cd <root dir of repo>

make setup # Sets up the environment and dependencies
# This will also run make python_setup automatically.
     If this fails due to network issues (or
    something else), please run the following after
    fixing the issues.
make python_setup # Required for plotting

# Install memcached server
make install_memcached

# Install memtier_benchmark (if not already installed
    )
make install_memtier_benchmark
```

Tiering with Memcached

```
make start_memcached # Starts memcached server
make load_memcached # Loads 40GB dataset with 4K
    objects

# Baseline (no tiering)
make tier_memcached_memtier_baseline

# HeMem tiering strategy
make tier_memcached_memtier_hemem agg_mode=0 # 0
    conservating 1 moderate 2 aggressive

# ILP-based tiering strategy
make tier_memcached_memtier_ilp agg_mode=0 # 0
    conservating 1 moderate 2 aggressive

# Waterfall tiering strategy
make tier_memcached_memtier_waterfall agg_mode=0 # 0
    conservating 1 moderate 2 aggressive
```

A.3.2 Linux kernel patching. Apply patches:

```
$ cd <root dir of repo>
$ git clone https://github.com/torvalds/linux.git --
    branch v5.17 --depth 1
$ cd linux
$ git am ../linux_patch/0001-tierscape-eurosys26.
    patch

$ git log

commit 8d955619e152eabd14acefa19c4c819c053cf96a (HEAD
     -> tierscape)
Author: Sandeep Kumar <sandeep4.kumar@intel.com>
Date: Tue Sep 2 04:48:30 2025 +0530

    tierscape eurosys26

commit f443e374ae131c168a065ea1748feac6b2e76613 (
    grafted, tag: v5.17)
Author: Linus Torvalds <torvalds@linux-foundation.org
    >
Date: Sun Mar 20 13:14:17 2022 -0700

    Linux 5.17

    Signed-off-by: Linus Torvalds <torvalds@linux-
        foundation.org>
```

Build and install the kernel

```
$ cp tierscape_config .config
$ make -j $(nproc)
## Select default values for any new options
$ sudo make modules_install -j $(nproc)
$ sudo make install -j $(nproc)

$ sudo reboot
```

```
$ uname -r
5.17.0-ntier-noiaa-v1+
```

Enabling Compressible Tiers

```
$ cd <root dir of repo>
$ make ntier_setup
Using ZRAM
Removing zram
Setting up zram
FAST_NODE: 0
SLOW_NODE: 1
Disabling the prefetching
kernel.zswap_print_stat = 1
[ 3904.686103] zswap: Looking for a zpool zsmalloc
    zstd 0
[ 3904.686104] zswap: It looks like we already have a
     pool. zsmalloc zstd 0
[ 3904.686104] zswap: zswap: Adding zpool Type
    zsmalloc Compressor zstd BS 0
[ 3904.686105] zswap: Total pools now 4
[ 3904.686117] zswap: Looking for a zpool zsmalloc
    lzo 0
[ 3904.686118] zswap: using existing pool zsmalloc
    lzo 0
[ 3904.686125] zswap: ..
               Request for a new pool: pool and
                    compressor is zsmalloc lzo
                    backing store value is 0
[ 3904.686125] zswap: Looking for a zpool zsmalloc
    lzo 0
[ 3904.686126] zswap: It looks like we already have a
     pool. zsmalloc lzo 0
[ 3904.686126] zswap: zswap: Adding zpool Type
    zsmalloc Compressor lzo BS 0
[ 3904.686126] zswap: Total pools now 4
[ 3904.686745]
             ------------
             Total zswap pools 4
[ 3904.686747] zswap: Tier CData pool compressor
    backing Pages isCPUComp Faults
[ 3904.686749] zswap: 0 0 zsmalloc lzo 0 0 true 0
[ 3904.686751] zswap: 1 0 zsmalloc zstd 0 0 true 0
[ 3904.686752] zswap: 2 0 zsmalloc zstd 1 0 true 0
[ 3904.686753] zswap: 3 0 zbud zstd 0 0 true 0
```

Executing Experiments with Kernel Patches: Rebuild TierScape with kernel patches enabled. Ensure the configuration is done as in Configuration section.

```
$ cd <root dir of repo>
$ make setup ENABLE_NTIER=1
$ make tier_masim_ilp agg_mode=2
```

Run experiments with memcached as described earlier. The results will be saved in the dir with suffix _EN1 indicating kernel patches are enabled.

A.4 Evaluation workflow

A.4.1 Major Claims. Major claims made in the paper:

- *(C1)*: Enabling multiple compressed tiers with different configuration allows for an aggressive memory tiering of warm pages. This is proved by Figure 7, Figure 8, and Figure 9 of the paper.
- *(C2)*: Analytical model allows for a configurable memory tiering at different cost-performance points. This is proven by Figure 10 of the paper.

A.4.2 Experiments. *Experiment (E1): [Waterfall + Analytical Model] [30 human-minutes + 1 compute-hour]: [How to Run: Preparation and Execution]*
Please see Section A.3 on instructions for how to run experiments.

[Results] The results from the experiments are stored inside the `evaluation` directory which is in the root directory. If you run either of the following commands:

```
make tier_memcached_memtier_all agg_mode=0 # 0
    conservating 1 moderate 2 aggressive
# or
make tier_memcached_memtier_all agg_mode=0 # 0
    conservating 1 moderate 2 aggressive
```

The experiments generate data comparing different tiering strategies:

1. Baseline: No tiering, all data in single tier
2. HeMem [48]: HeMem-based tiering algorithm
3. ILP: Analytical model (Integer Linear Programming) - based optimal tiering
4. Waterfall: Waterfall-based tiering strategy

The naming convention for directories inside the evaluation directory will follow a structure: `perflog-ILP-F10000-HT.9-R0-PT2-W5-20250909-200453`. Breakdown of the dir name:

1. perflog: Prefix indicating performance logs
2. ILP: Tiering strategy used (Baseline, HeMem, ILP, Waterfall)
3. F10000: PEBS frequency (10000)
4. HT.9: Hotness threshold (0.9)
5. R0: Remote mode (0 disabled 1 enabled)
6. PT2: Number of push threads to move data around
7. W5: Profile window in seconds
8. 20250909-200453: Timestamp of the experiment run