

HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System

Kevin Song
University of Toronto, Vector Institute
Toronto, Canada
xinyang.song@utoronto.ca

Jiacheng Yang
University of Toronto, Vector Institute
Toronto, Canada
jiacheng.yang@mail.utoronto.ca

Zixuan Wang
University of California San Diego
San Diego, USA
zxwang@ucsd.edu

Jishen Zhao
University of California San Diego
San Diego, USA
jzhao@ucsd.edu

Sihang Liu
University of Waterloo
Waterloo, Canada
sihangliu@uwaterloo.ca

Gennady Pekhimenko
University of Toronto, Vector Institute
Toronto, Canada
pekhimenko@cs.toronto.edu

Abstract

Modern workloads are demanding increasingly larger memory capacity. Compute Express Link (CXL)-based memory tiering has emerged as a promising solution for addressing this problem by utilizing traditional DRAM alongside slow-tier CXL memory devices. We analyze prior tiering systems and observe two challenges for high-performance memory tiering: adapting to skewed but dynamically varying data hotness distributions while minimizing memory and cache overhead due to tiering.

To address these challenges, we propose HybridTier, an adaptive and lightweight tiering system for CXL memory. HybridTier tracks both long-term data access frequency and short-term access momentum *simultaneously* to accurately capture and adapt to shifting hotness distributions. HybridTier reduces the metadata memory overhead by tracking data accesses *probabilistically*, obtaining higher memory efficiency by trading off a small amount of tracking inaccuracy that has a negligible impact on application performance. To reduce cache overhead, HybridTier uses lightweight data structures that optimize for data locality to track data hotness. Our evaluations show that HybridTier outperforms prior systems by up to 91% (19% geomean), incurring 2.0 − 7.8× less memory overhead and 1.7 − 3.5× less cache misses.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Software and its engineering → Memory management**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1080-3/2025/03
https://doi.org/10.1145/3676642.3736119

Keywords: Compute Express Link, Tiered Memory

ACM Reference Format:
Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. 2025. HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676642.3736119

1 Introduction

Modern applications [5, 6, 84, 88] demand increasingly large memory capacity and high bandwidth. To keep up with this fast growth, one potential solution is to install more and higher-capacity DRAM modules. However, the amount of DRAM within a single server is limited due to space constraints. In addition, main memory is already one of the most expensive components in data center servers. Meta reports that 37% of the total cost of ownership per rack is spent on memory alone [48]. Purchasing more DRAM modules will only exacerbate this trend. Moreover, the growth in DRAM density has been slowing down since the last decade [53, 57], further limiting the capacity scalability of main memory.

Compute Express Link (CXL) based memory tiering is a promising solution to this challenge. CXL is an industry-standard interconnect protocol [60]. Memory (e.g., DDR4/5 DRAM) attached through CXL interface is byte-addressable, directly accessible by the host CPU, and supports standard memory allocation interfaces. Compared to local DRAM, CXL-attached memory has a larger capacity and lower cost-per-GB, as the CXL bus consumes less power and allows DRAM modules to be utilized in a more compact form factor. On the other hand, CXL memory suffers from higher latency and lower bandwidth. As shown in Figure 1, compared to local DRAM, CXL memory devices introduce 50−100 ns of additional access latency [48], while achieving 20−70% of its bandwidth [77]. Therefore, a tiering system should prioritize placing hot data in local DRAM (fast-tier) while keeping cold data in CXL memory (slow-tier) for better performance.

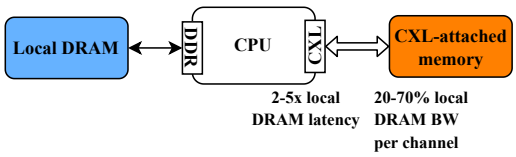


Figure 1. Illustration for CXL-attached memory expansion. Performance numbers obtained from [48, 77].

However, achieving high tiering performance is challenging. We profile large memory workloads and make two observations. 1) Real-world workloads often exhibit skewed yet dynamically varying data hotness distributions [5, 23, 88]. For instance, in-memory caches often exhibit Zipf or power-law access distributions [6, 88], where the majority of accesses are focused on a small fraction of data. At the same time, hot data can become cold in a matter of minutes [6, 88], causing changes in data hotness distribution. 2) Managing data access statistics can incur significant overhead. Large memory servers with terabytes of memory contains billions of pages. Tiering metadata associated with each page combined can decrease the cost-effectiveness of tiering. In addition, frequent access of tiering metadata generates high amounts of CPU cache traffic, degrading application performance due to resource contentions. Therefore, we argue that an ideal tiering system should satisfy three requirements: 1) accurately capture the hot set by placing the hottest data in fast-tier memory 2) quickly adapt to changes in the hotness distribution 3) minimize tiering metadata overhead.

However, prior systems do not meet all three requirements. One class of work adopts frequency-based tiering [41, 69, 86] by storing the access counts of each page to build a hotness histogram. Based on this histogram, the tiering system places the hottest pages in the fast-tier, satisfying requirement 1. However, frequency-based systems do not meet requirements 2 and 3. To maintain histogram freshness, prior works periodically perform "cooling" by reducing the page access counter values for all pages. In Section 2.3.2, we extensively analyze this type of freshness mechanism and demonstrate how it causes tiering systems to be slow at adapting to hotness changes, failing requirement 2. Furthermore, to store access counts for billions of pages, prior frequency-based systems can incur gigabytes of memory overhead per server, translating to lower cost-effectiveness. At the same time, data structures used by prior works to organize a large number of access counters lack data locality. Since access counters need to be frequently updated, tiering systems can generate a large number of cache misses and degrade application performance, failing requirement 3.

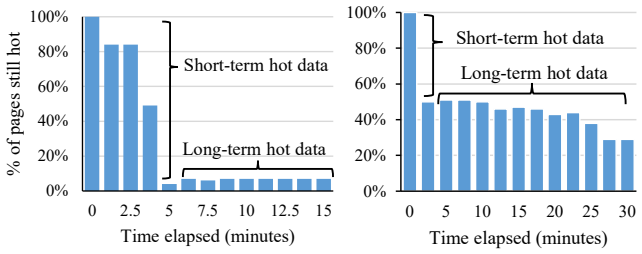
Another class of prior works [46, 48, 52] is recency-based tiering, which uses access recency to approximate data hotness. Such systems use metrics such as time between consecutive page faults to make data placement decisions. In general, recency-based systems meet requirement 3 since recency statistics are less resource-intensive to manage than frequency counterparts [27, 29, 41]. Intuitively, recency systems only need to store the most recent event without the need to track historical events. However, recency-based systems fail to satisfy requirement 1. Since such systems only consider access statics in a short time interval, they are susceptible to misclassifying cold pages as hot [28, 41, 46]. Recency-based systems also do not satisfy requirement 2. While recency metrics are naturally "fresh" and thus do not require cooling, this alone is not sufficient to achieve high adaptiveness. We demonstrate in Section 2.3.2 that satisfying requirement 1 is a prerequisite for requirement 2.

In this work, we propose HybridTier, an application transparent tiering system that addresses the three requirements. To manage workloads with skewed but dynamically varying hotness distributions, HybridTier maintains two metrics for each page to capture both long-term access history and short-term hotness variations. HybridTier considers statistics from both metrics to enable a flexible migration policy. For promotion–moving data from the slow-tier to fast-tier–HybridTier promotes not only pages with high historical hotness but also pages with high access momentum in the short-term to quickly identify pages that have recently become hot. For demotion, HybridTier adopts a second-chance policy to swiftly demote pages that were historically hot but have recently turned cold.

However, maintaining two metrics for every page can exacerbate the problem of high metadata memory overhead. To address this challenge, we make the observation that tiering systems can tolerate a small amount of tracking inaccuracy without noticeably affecting application performance. Based on this observation, HybridTier tracks memory accesses using counting bloom filters (CBF) [21], a *probabilistically* data structure that trades higher memory efficiency for lower tracking accuracy. HybridTier's CBF is also cache efficient, as it is more compact and requires fewer memory accesses compared to data structures used by prior works. To further reduce cache misses, HybridTier optimizes the locality of CBF by adopting blocked CBF [61, 63].

In summary, we make the following contributions:

- We analyze existing tiering systems and reveal three new findings: 1) Adapting to changing hotness is challenging, causing suboptimal performance under real-world workloads 2) Maintaining historical access information can incur high metadata memory overhead 3) Tiering systems can suffer from high number of cache misses due to poor data locality during metadata updates.
- We introduce HybridTier, an application transparent memory tiering system that is adaptive and lightweight. HybridTier adopts a novel access tracking method that captures both long-term hotness distribution and short-term changes in data hotness. At the same time, HybridTier



(a) Graph analytic: Page Rank. **(b)** Machine learning: XGBoost.

Figure 2. Data hotness distribution changes rapidly. The Y-axis is the fraction of pages that were hot at time 0 and remained hot over a certain time (X-axis). In both workloads, most pages are no longer hot after just 5 minutes.

significantly reduces metadata memory consumption and cache misses by adopting probabilistic access tracking and locality-optimized data structures.

- We compare the performance of HybridTier against three state-of-the-art tiering systems over six large memory workloads while varying the fast-to-slow tier memory ratio. HybridTier outperforms prior works by an average of 19% while incurring 2.0 − 7.8× less memory overhead and 1.7 − 3.5× less cache misses due to tiering.

2 Background and Motivation

2.1 CXL-Enabled Memory Tiering Systems

CXL is an open industry standard interconnect running on top of the PCIe physical layer [51, 56, 74, 75]. The key goal of CXL is to better support heterogeneous computing and memory capabilities in future data center architectures. Since its introduction in 2019, the CXL ecosystem has been under rapid development. With support from major vendors across the data center stack, CXL is widely believed to make a significant impact in future data center architectures.

A CXL-enabled memory tiering system utilizes both regular CPU-attached DRAM (fast-tier) and CXL-attached memory (slow-tier) in the same system. Local DRAM offers better latency and bandwidth but has a higher cost-per-GB, while CXL memory provides higher capacity but suffers from lower access performance. Therefore, the goal of a tiered memory system is to accurately identify hot and cold data and place them into the local DRAM and CXL-memory, respectively.

2.2 Dynamic Data Hotness Distribution

Prior studies have shown that many large memory workloads exhibit skewed memory access distribution. Twitter and Meta both report that within a 24-hour window, the popularity of in-memory caching workloads largely follows the Zipfian distribution with a high degree of skewness [6, 9, 88]. For instance, approximately 80% of accesses to Meta's object storage cache focus on the top 10% most popular items.

At the same time, large memory workloads also tend to have dynamically changing hotness distributions. Meta [6] report that production in-memory caches experience rapidly shifting access distributions. At any moment in time, 50% of popular objects are no longer popular after just 10 minutes. Twitter [78, 88] reported that production in-memory caches often use short time-to-live (TTL) in the order of minutes, where objects are removed from the cache after the TTL expires. Figure 2 shows that throughput-oriented workloads such as graph analytic [5] and machine learning training [84] also experience dynamic access distributions. In Page Rank and XGBoost, over 90% and 50% of initially hot pages are no longer hot after just 5 minutes.

2.3 Prior Tiering Systems

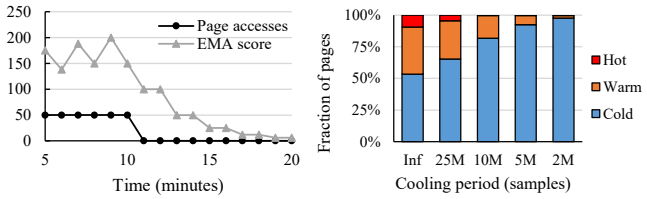
This section analyzes prior works based on three requirements of an effective tiering system: (1) ability to accurately capture hot data (2) adaptability to changing hotness distributions (3) tiering metadata overhead.

2.3.1 Accurately Capture Hot Data. An essential goal of tiering systems is to maximize memory performance by placing hot data in fast-tier memory. Therefore, accurately identifying hot data is critical. Since different applications can exhibit different levels of skewness in their hotness distributions, an effective tiering system must place only the hottest pages in fast-tier memory to maximize performance.

Fortunately, recent works can satisfy this requirement effectively. For instance, Memtis [41], a state-of-the-art tiering system, maintains a histogram to track the overall access frequency distribution of memory pages. By understanding the overall hotness distribution and the fast-tier memory capacity, Memtis can accurately calculate the hotness threshold to ensure only the hottest data are placed in the fast-tier.

2.3.2 Adapting to Varying Hotness Distributions. As discussed in subsection 2.2, real-world workloads often exhibit dynamically changing hotness distributions. As a result, in addition to accurately capturing hot data, tiering systems should also *quickly* identify pages that are turning hot and turning cold. Without this ability, pages that are no longer hot would be left in fast-tier memory, consuming precious resources and leaving performance on the table. To analyze the adaptability of tiering systems, we categorize prior works into frequency-based and recency-based according to the hotness metric used.

Frequency-based Systems. To identify hot pages, one line of work, such as Memtis [41], tracks the overall data access distribution by maintaining dedicated frequency counters for each page. A page is considered hot if its accumulated access frequency exceeds a hotness threshold. To ensure freshness, frequency-based systems typically implement exponential moving average (EMA) [41, 69] with a decay factor of 2, where page access counters are periodically cooled by



(a) Bottom series: a page accessed 50 times per minute for 10 minutes. Top series: EMA score of this page.

(b) Fraction of cold, warm, and hot pages in CacheLib workload identified by EMA score under different cooling period C.

Figure 3. Effect of cooling period on (a) adaptiveness to hotness changes and (b) hotness classification accuracy. Higher cooling periods adapt to changes more slowly. Lower cooling periods capture hot pages less accurately.

dividing all access counters by two[1]. The cooling period C is a pre-determined parameter.

While frequency-based systems can accurately capture the long-term hotness distribution, they are *suboptimal at quickly adapting to changes in hotness*. This is because moving average metrics, including EMA, are lagging indicators. Intuitively, average scores have "inertia" and resist change since they include historical values. Consider a memory page that was historically hot but recently turned cold. Ideally, this page should be demoted quickly to free up space in the fast-tier memory. The bottom series in Figure 3 (a) represents the access count per minute of such a page. The top series represents the EMA score of this page, with cooling performed every 2 minutes, representative of the cooling thresholds used by prior works. After the 10-minute mark, this page is no longer accessed. However, since the EMA score is only reduced by half every 2 minutes, it lags behind the access frequency and only drops below 10 after 19 minutes. This means the tiering system will not be able to identify this page as cold until 9 minutes after it turns cold.

To adapt faster, one potential solution is to reduce C. A lower C indeed has a lower lag, as the EMA scores are refreshed more frequently. However, as demonstrated by prior works [41, 69], lower values of C also capture the overall hotness distribution less accurately, degrading application performance. Intuitively, a lower value of C reduces the number of memory accesses reflected in the overall hotness histogram. To demonstrate, we measure the hotness distribution of a CacheLib workload under different values of C. In Figure 3 (b), C=Inf represents the target distribution that should to be captured. As C decreases, the distribution becomes less accurate because hot and warm pages do not have enough time to accumulate their access counts. Therefore, while reducing C improves adaptability, it undermines the tiering system's ability to identify hot data (requirement 1).

[1]Decay factor 2 is typically used since it can be implemented using bit shift.

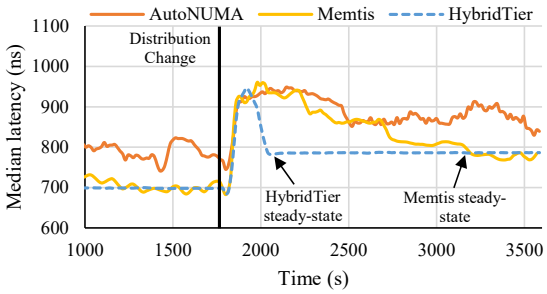


Figure 4. Tiering systems adapting to hotness distribution change for CacheLib workload. Vertical line indicates when the change in distribution occurs. Lower is better.

To evaluate the adaptability of frequency-based tiering systems, we utilize CacheLib, an in-memory cache workload from Meta [6]. At the start of the experiment, 100 million cache items are accessed based on a Zipf distribution[2]. We reproduce the varying distribution reported by Meta [6] by adjusting the access distribution at the 1800-second mark such that 2/3 of previously hot data are no longer hot. Figure 4 shows that Memtis requires roughly 1400 seconds to adapt to the new distribution, 1000 seconds slower than ideal.

Recency-based Systems On the other hand, systems such as AutoNUMA [29] and TPP [48] measure data hotness by its access characteristics over the duration of seconds. For example, AutoNUMA uses the page hint fault time to decide whether a page should be promoted. AutoNUMA periodically scans the application address space and unmaps 256MB of pages. The time elapsed between when an unmapped page is accessed and when it was unmapped is the hint fault latency. If a page has hint fault latency of less than 1 second, it is promoted, regardless of its historical access statistics.

The main drawback of recency-based systems is their inability to accurately identify hot pages. Prior works have shown that migration decisions solely based on recency statistics can be suboptimal [41, 46, 69]. Intuitively, a recently accessed page may or may not be a hot page. For example, a cold page with only a single recent access may be misclassified as a hot page by AutoNUMA. This can be seen by the fact that in Figure 4, AutoNUMA continues to have high latency even when the access distribution is no longer changing. Figure 4 also shows that AutoNUMA adapts the slowest to the change in hotness distribution. This is counter-intuitive at first since recency-based systems do not suffer from the same adaptability drawbacks as frequency-based systems. The reason behind this is that while AutoNUMA can quickly promote new hot pages, it also incorrectly promotes cold pages at the same time. In configurations where the fast-tier capacity is limited, these cold pages consume previous fast-tier space and prevent truly hot pages from

[2]This distribution is provided by Meta as a part of its CacheLib benchmarking framework and reflects Meta's real production environments.

being promoted. From this observation, we argue that accurately capturing hot data (requirement 1) is a prerequisite for high adaptiveness (requirement 2).

> **Observation 1**: Real-world workloads often exhibit skewed but varying hotness distributions. Tiering systems should accurately capture the overall hotness distribution while quickly adapting to changing data hotness.

2.3.3 Tiering Metadata Overhead. Memory tiering systems typically maintain historical access information for each memory page, such as the number of accesses, in order to make future promotion/demotion decisions. We refer to data structures used to store access information as *tiering metadata*. We break down the overhead due to tiering metadata into two types: memory overhead and cache overhead.

Memory Overhead: A common approach used by prior systems is to allocate dedicated metadata for each memory page in the system. However, a modern large memory server can contain up to billions of 4KB pages. Storing additional tiering metadata for each page can easily consume gigabytes of memory. While this overhead might be acceptable on a small scale, it can noticeably impact the cost-effectiveness of tiered memory at the data center scale. Thus, as CXL memory is expected to further increase the amount of memory per server, tiering systems must rigorously optimize the size of metadata associated with each page [12, 13, 26].

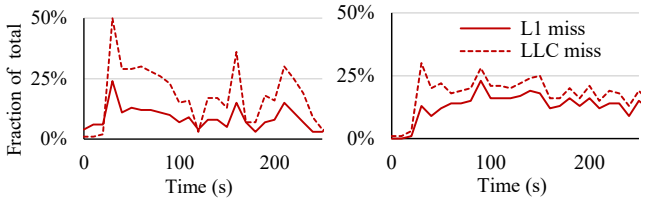
However, the metadata memory overhead of existing works is high. For example, for every 4KB memory page, Memtis adds 16B of metadata for each Linux struct page. For a server with 1TB of memory, a 0.39% memory overhead translates to 3.9GB. In comparison, the Linux kernel (v6.2) only consumes about 400MB of memory upon boot[3]. To illustrate the potential cost of this overhead, we use a small-scale virtual machine serving as an example. The AWS t2.nano instance with 0.5GB memory each at $4.18/month [58]. Assuming a data center with 100,000 such servers, the total memory overhead could have been used to generate $31.8M per year.

> **Observation 2**: As the total main memory capacity increases, tiering metadata can lead to prohibitively high memory overhead, resulting in reduced tiering cost-effectiveness.

Cache Overhead: As discussed in 2.3.3, tiering metadata on a typical system can consume GBs of memory, larger than the capacity of last-level cache on most systems. Frequently updating this metadata can generate cache traffic that interferes with the application.

To efficiently track which memory pages are accessed, prior works have proposed various memory access tracking mechanisms, including utilizing page faults [29, 48], page table scanning [46], and hardware performance counters [41, 69]. In particular, hardware-counter-based access tracking is promising due to its accuracy and scalability [41, 69]. Dedicated event sampling hardware, such as Processor Event-Based Sampling (PEBS) for Intel and Instruction Based Sampling (IBS) for AMD processors, provides a stream of sampled events at a specified frequency. Each sampled event contains the exact virtual address accessed by the application and whether it was in local DRAM or CXL memory.

[3]Measured on 1TB server. Includes kernel code, data, and slabs.



(a) Under 4KB pages. **(b)** Under huge pages.

Figure 5. Cache misses due to Memtis tiering activities as a fraction of the system total when running CacheLib.

Despite dedicated sampling hardware, access tracking can have non-negligible caching overhead, which has been overlooked by prior tiering systems. To measure the cache overhead of Memtis [41], we run the CacheLib workload on a 1:4 configuration. We provide the detailed experiment configuration in Section 6.3.3. As Figure 5 shows, tiering activities in Memtis incur a significant number of cache misses. Memtis on average consumes 9% and 18% of total L1 and LLC cache misses for regular pages, and 13% and 18% for huge pages. Under cache-intensive applications, this large number of cache misses causes cache and memory resource contention, thus degrading performance. In Section 3.3, we analyze the main source of cache misses due to tiering.

> **Observation 3**: Hardware-counter-based memory access tracking can incur non-negligible cache overhead due to frequent metadata updates.

3 HybridTier Key Ideas

In this section, we summarize the key challenges and how HybridTier addresses them.

3.1 Adapting to Varying Hotness Distributions

Frequency-based tiering can effectively capture the overall hotness distribution, but cannot quickly adjust to changes. Recency-based tiering can identify new hot pages quickly, but cannot accurately capture the entire hot set, since they do not consider a page's hotness history. We observe that this tradeoff is the consequence of the fact that *prior systems only tracks one metric for each page*. For Memtis, this metric

Table 1. HybridTier promotion/demotion policies. A page is considered to have high frequency/momentum if its frequency/momentum is above the corresponding thresholds.

Momentum \ Frequency	High	Low
High	Promote / No Action	Promote / No Action
Low	Prompt / 2nd Chance	No Action / Demote

is the exponential moving average score, which is a lagging indicator. For AutoNUMA, this metric is the hint fault latency, which does not capture long-term access information.

Key Idea: Based on this observation, our key idea is to maintain *two* separate metrics for each page instead of a single metric. We refer to the two metrics as access "frequency" and "momentum". Page access frequency tracks historical access frequency in the order of minutes to hours. Access momentum monitors page access intensity in within seconds. To achieve this, HybridTier adopts EMA and sets a high EMA cooling period C for frequency counters and a low C for momentum counters. The key difference between HybridTier and prior EMA-based tiering systems is that two dedicated EMA counters enable HybridTier to accurately capture the long-term hotness distribution while *simultaneously* quickly adapts to hotness changes.

Tracking both frequency and momentum enables a flexible migration policy for HybridTier, shown in Table 1. HybridTier maintains two hotness thresholds: one for frequency and one for momentum. HybridTier automatically adjusts the frequency threshold based on the current hotness distribution and fast-tier capacity size, similar to Memtis [41]. The momentum threshold is determined empirically. HybridTier promotes pages with high frequency *or* high momentum. The heuristic behind this policy is that pages accessed intensely over a short period will likely continue being accessed. This enables HybridTier to quickly promote pages that were cold in the past but recently became hot. We demonstrate the effectiveness of this heuristic in Section 6.4.1.

For demotion, HybridTier immediately demotes pages with low frequency and low momentum. Pages with high momentum but low frequency will not be demoted since they may have been recently promoted. Pages with high frequency but low momentum are given a *second chance* to account for pages that are only cold temporarily. Instead of immediately demoting such pages, HybridTier marks and revisits them for a second chance. By adopting this tiering policy, HybridTier adapts to new access distribution the fastest (Figure 4) using only 250 seconds.

While maintaining two metrics per page appears simple, naively applying this technique would double the amount of memory consumed by metadata, exacerbating the metadata overhead problem discussed in Section 2.3.3. Next, we discuss our key idea to significantly reduce this overhead.

Algorithm 1: Typical access sampling algorithm used by sample-based tiering systems.

```
1 while true do
2     if SampleBuffer is not empty then
3         Sample = SampleBuffer.read()
4         PageAddr = Sample.addr
5         Table[PageAddr]->accesses++
```

3.2 Metadata Memory Overhead

Prior frequency-based tiering systems such as Memtis [41] and HeMem [69] utilize hash table-like data structures to store page access counts. We categorize such data structure as *exact data structures*. An exact data structure guarantees that a lookup will always return the previous latest value inserted. For instance, `hashtable.lookup(key1)` is guaranteed to return `value1` if the last insertion on `key1` was `hashtable.insert(key1, value1)`. The hash table maintains an exactness guarantee by allocating dedicated memory for each item inserted and by resolving hash conflicts. However, as discussed in Section 2.3.3, when a large number of items are inserted, its memory footprint also becomes large.

Key Idea: In the context of tracking memory accesses, we argue that exactness is not a requirement for achieving high tiering performance. Intuitively, even if the access count of a very hot page is off by 1 or 2, it will most likely still be classified as a hot page and no migration decisions will be affected. Exactness only affects migration decisions in rare cases where the access count of a page is near the hotness threshold. We show that such cases are indeed rare and have negligible impact on performance in practice in Section 6.4.2.

Following this observation, one of HybridTier's key ideas is to utilize *probabilistic* data structures to track page accesses. Specifically, HybridTier adopts counting bloom filters (CBFs) for its frequency tracker and momentum tracker. Unlike exact data structures, CBF tracks access counts probabilistically, i.e. "the access count of this page is *probably* x with probability p." Rather than allocating dedicated memory for every page, CBF allocates a fixed-size array of metadata that is shared by all inserted pages. HybridTier utilizes this property to allocate only enough memory to store metadata that is actively used. In addition, HybridTier only allocates 4 bits per access counter, allowing a maximum count of 15. The heuristics behind this approximation is that pages with access count ≥ 15 should all be placed in fast-tier memory, thus there is no need to differentiate between them. We demonstrate that in Section 6.4.2 this approximation is accurate for all workloads we evaluate.

3.3 Tiering Cache Overhead

To understand why prior hardware-counter-based systems incur high cache overhead, we present the algorithm used by Memtis [41] and HeMem [69] for access sampling in

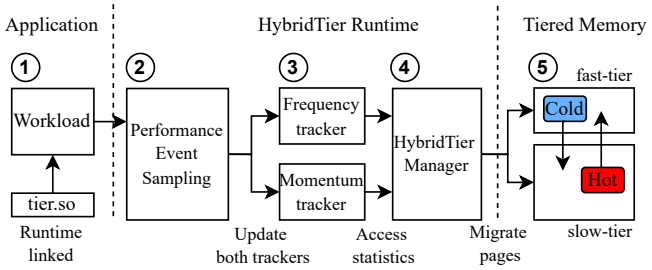


Figure 6. An overview of HybridTier.

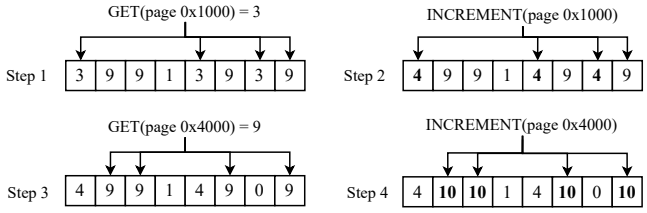


Figure 7. Counting bloom filter illustration.

Algorithm 1. When new access samples are available in the PEBS buffer (line 2), they are collected one by one (line 3). For each sample collected, its page address is extracted (line 4) and the page access count is updated (line 5). For Memtis, this table is the Linux page table[62], while HeMem uses a custom hash table [66]. The main source of cache overhead occurs in line 5. For every sample collected, the tiering thread performs a table lookup (`Table[PageAddr]`). For Memtis, this requires traversing the Linux multi-level page table [62], potentially causing multiple cache misses. For HeMem, since its hash table implements, a lookup may result in multiple pointer dereferences to resolve hash collisions. Since the size of this table can easily exceed the LLC cache size (Section 2.3.3), frequent metadata accesses result in a large number of cache misses, as shown in Figure 5.

Key Idea: The 4-bit CBF introduced in the previous section not only reduces memory overhead but also reduces cache overhead for two reasons. First, 4-bit CBF is more compact. HybridTier assigns at maximum 4×4-bit counters to a memory page, meaning that each 64B cache line can store access counts for at least 32 pages. In contrast, Memtis requires 16B of metadata per page, allowing only 4 metadata per cache line. Second, the CBF is a single-level key-value data structure that intentionally allows hash collisions (details in section 4.2), therefore reducing the number of pointer dereferences per lookup.

However, the standard counting bloom filter can still cause high cache misses. As we will show in Section 4.2, a lookup in the standard CBF performs k accesses to retrieve k counters associated with a page. In the worst case, this incurs k cache misses. To address this, HybridTier adopts blocked CBF [61, 63], an optimization that ensures each CBF lookup will incur exactly one cache access and at most one cache miss. We describe blocked CBF in more detail in Section 4.2.

4 HybridTier

In this section, we describe the detailed design of HybridTier.

4.1 Workflow Overview

Figure 6 illustrates the high-level design of HybridTier. HybridTier is implemented as a single userspace runtime thread that performs tiering for a workload process. ① HybridTier dynamically links the HybridTier shared library into the target application binary using `LD_PRELOAD` dynamic link mechanism. This process is *transparent* to the application and does not require recompiling the target workload. ② HybridTier utilizes Intel Processor Event-Based Sampling (PEBS), where each access sample contains the virtual memory address being accessed. ③ HybridTier stores access statistics using two CBFs, one for each access tracker. For each sample collected, HybridTier updates the access count of the accessed page in both CBFs. ④ The HybridTier manager utilizes access statistics from the two trackers to make migration decisions. ⑤ Finally, HybridTier utilizes system calls to migrate pages between fast and slow-tier memory.

4.2 Counting Bloom Filter

A CBF consists of k hash functions and an array of size M. A CBF supports two operations: `GET` and `INCREMENT`. `GET` calculates k array indices from k hash functions and returns the minimum counters within the k counters. `INCREMENT` calculates k indices from k hash functions and increment the minimum counters. Figure 7 shows an example with $k = 4$ and $M = 8$. At step 1, invoking `GET` on page 0x1000 will return 3. At step 2, `INCREMENT` on page 0x1000 will increase the counters at indices 0, 4, and 6 to the value 4. Similarly, at step 3, `INCREMENT` on page 0x4000 will increase the counters at indices 1, 2, 5, and 7 to the value 10, and at step 4 `GET` on page 0x4000 will return 10.

The example in Figure 7 shows that the access count of one page may be overwritten by other pages, as the CBF does not resolve hash collisions. We refer to this error as a "tracking error". To achieve a balance between tracking error and memory overhead, we compute the size of the CBF m using well-established bloom filter formulas [30]:

$$r = -k/log(1 - exp(log(p)/k)) \quad m = ceil(n * r)$$

where p is the probability of tracking error rate, m is the number of counters in the filter. In HybridTier, we empirically set $k = 4$, $p = 0.001$, and n equal to the number of fast-tier pages. This combination of parameters proved to work well for all our evaluation workloads.

Frequency and Momentum Trackers. Both the frequency and momentum trackers are implemented using CBF. Since the momentum CBF has a lower cooling period, we observe that the number of pages stored at a given moment is significantly less than that of the frequency CBF. This is because the momentum CBF performs cooling frequently,

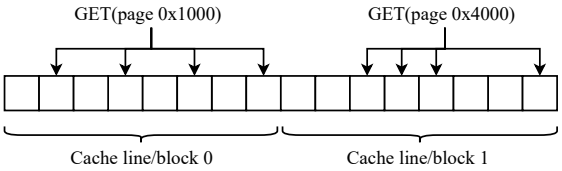


Figure 8. Blocked counting bloom filter illustration.

quickly reducing access counts of most pages to 0. Therefore, we can allocate less memory for the momentum tracker CBF while achieving the desired tracking error. In practice, HybridTier allocates 128× less memory for the momentum CBF than the frequency CBF.

Blocked CBF. A weakness of the standard CBF illustrated is that the k counters associated with a page lack spatial locality since their memory locations are randomly assigned by the hash functions. In the worst case, a lookup results in k cache misses. Blocked CBF [61, 63], illustrated in Figure 8, addresses this by enforcing that all k counters of a page are located in the same 64B cache line. A page can be mapped to any but only one cache line. The k counters can be mapped to any counters within the cache line. For illustration, Figure 8 shows each cache line contains 8 counter slots. In reality, each cache line in a 4-bit CBF contains 128 counter slots. Compared to standard CBF, blocked CBF has a slightly higher false positive rate [61, 63]. However, in practice, we find the performance benefits to be a favorable tradeoff.

4.3 Promotion and Demotion

Promotion. For each access sample collected, HybridTier records this access in both CBFs. Based on the updated page frequency and momentum, HybridTier decides whether to promote this page or not. To reduce system call overhead, HybridTier processes 100,000 samples as a batch and promotes all hot pages in a batch with a single system call.

Demotion. When the amount of free memory in the fast-tier is below `PROMO_WMARK` watermark, HybridTier demotes until the amount of free memory in the fast-tier is greater than `DEMOTE_WMARK`. HybridTier identifies cold pages in fast-tier by linearly scanning the application virtual address space utilizing `/proc/PID/maps` and `/proc/PID/pagemaps`. When a page is marked for second-chance, its current access frequency is saved. HybridTier later revisits previously marked pages and compares their current access frequency count against the previously stored count. If a marked page was not accessed after the revisit time, HybridTier considers this page to be no longer hot and demotes it. We empirically set the revisit time to 1 minute to achieve a balance between demotion accuracy and runtime overhead.

4.4 Huge Page Support

HybridTier supports 2MB huge pages through Linux Transparent Huge Pages (THP). When enabled, HybridTier tracks access frequency/momentum and performs migrations at the huge page granularity. HybridTier increase each CBF counter to 16-bit CBFs to accommodate higher access counts for huge pages. At the same time, the number of elements in each CBF is also reduced as the total number of pages in the system is reduced by 512×. Therefore, HybridTier's metadata memory consumption in huge page mode is 128× lower than in regular page mode.

Table 2. Workloads for evaluation.

Application	Input	Footprint
Content-delivery network Social-graph	CacheLib generator [20]	267GB
Breadth-first search (BFS) Connected components (CC) Page Rank (PR)	Kronecker graph [5] Uniform random graph	335GB
SPEC CPU 2017	603.bwaves [38] 654.roms [24]	150GB
Silo [79]	YCSB-C [10]	208GB
XGBoost [84]	Criteo Click Logs [14]	248GB

4.5 Implementation Details

HybridTier is a userspace runtime thread. We implement HybridTier using 1,577 lines of code in C++. As a userspace runtime, HybridTier requires no workload recompilation and kernel modifications. HybridTier relies on the operating system and hardware support for (1) memory access sampling, (2) memory movement between tiers, and (3) the ability to scan pages in a process address space. These requirements are widely available in modern systems, such as Instruction-Based Sampling (IBS) in AMD processors [59] and NUMA migrations in Windows systems [64].

5 Methodology

5.1 CXL Emulation

While recent studies have revealed performance characteristics of real CXL-memory devices (CXL 1.1 specification) [77], many of these CXL devices are not commercially available on the market. Similar to recent works [41, 42, 48], we use a remote NUMA node on a two-socket system to emulate CXL. The emulated CXL memory has idle latency of 124ns and bandwidth of 34 GB/s, similar to reported in a recent work [77]. Each socket has a 16-core Intel Xeon 4314 processor and 512GB of DDR4 memory. The application runs only on local NUMA node CPUs.

5.2 Baselines and HybridTier Configurations

We compare HybridTier against AutoNUMA [29], TPP [48], Memtis [41], ARC [49], and TwoQ [34]. For AutoNUMA, HybridTier, we use Linux kernel v6.2. For AutoNUMA, we enable its multi-generational LRU (MGLRU) based demotion due to its better performance than regular LRU.

We implement two additional tiering systems based on traditional caching algorithms: ARC [49] and TwoQ [34]. ARC [49] is a self-tuning caching policy that maintains two LRU lists to estimate item recency and frequency. TwoQ [34] is an extension of LRU that utilizes two queues to differentiate between items accessed only once vs. multiple times. Since ARC and TwoQ assume that the cache is initially empty, we initially allocate new memory pages on slow-tier memory for these two baselines. We omit comparisons against HeMem [69] and Tiering-0.8 [65] as Memtis already performs detailed comparisons against it. We do not perform end-to-end evaluations on MTM [70] as its source code is not publicly available at the time of writing.

5.3 Workloads

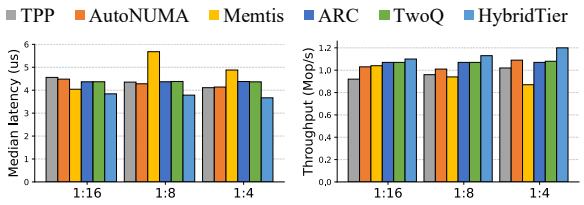
We evaluate HybridTier on workloads in Table 2. All experiments use 16 threads mapped to 16 physical cores. CacheLib is an in-memory cache used by Meta [6, 8]. We evaluate two workloads: content delivery networks (CDN) and social graphs. Each workload is characterized by a custom popularity distribution, size distribution, and operation composition that are representative of production workloads. GAP is a collection of standard graph processing kernel implementations [5]. We generate two graphs: Kronecker graph and uniform random graph [5], each with 2 billion nodes and 8 billion edges. The uniform random graph represents the worst case in terms of locality, where every vertex has an equal probability of being a neighbor of every other vertex. We evaluate three kernels: breath-first search (BFS), connected components (CC), and page rank (PR). SPEC CPU 2017 is an industry-standard CPU intensive benchmark suite [7]. We select 603.bwaves [38] and 654.roms [24] as they have the largest memory footprints. We follow the official SPEC CPU 2017 guidelines [24, 38] to scale up their input sizes to achieve 150GB resident set size. Silo [79] is an in-memory database engine. Similar to Memtis [41], we use YCSB-C input workload to stress the database engine. XGBoost is a widely used gradient-boosting library implemented using C++ commonly executed on CPU systems [2, 32]. We evaluate XGBoost training using the Criteo Click Logs dataset [14].

6 Evaluation

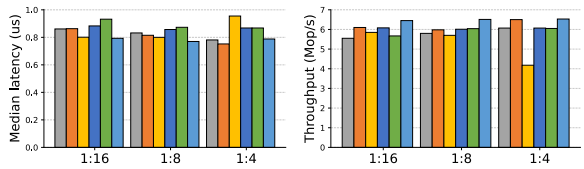
In this section, we evaluate HybridTier by performing end-to-end performance comparisons on regular 4KB pages (Section 6.1) and huge pages (6.2). Then, we perform detailed comparisons against Memtis (6.3) and conduct experiments to understand HybridTier's performance (6.4).

6.1 Regular Page Performance

Figure 9 and Figure 10 show the performance comparison of HybridTier. The x-axis indicates the ratio between fast and slow-tier memory capacity, where the slow-tier capacity is



(a) CacheLib CDN workload.



(b) CacheLib Social-graph workload.

Figure 9. Performance evaluation for CacheLib workloads. Lower is better for latency. Higher is better for throughput.

fixed at 512GB. On average (geomean), HybridTier outperforms TPP, AutoNUMA, Memtis, ARC, and TwoQ by 32%, 11%, 29%, 50%, and 40% respectively.

CacheLib. Figure 9 shows the cache access median latency and throughput of HybridTier and prior works. We make two observations. 1) Under the same fast:slow memory ratios, HybridTier performs the best in all but two experiments. On average, HybridTier outperforms TPP, AutoNUMA, Memtis, ARC, and TwoQ by 10%, 9%, 18%, 14%, and 15% respectively in terms of median latency. HybridTier also improves throughput by 15%, 7%, 23%, 7%, and 8% respectively. 2) HybridTier often requires 2× less fast-tier memory to achieve the same level of performance as the second best performing system. On the CDN workload, HybridTier with 1:16 configuration outperforms all other systems with 1:8 configuration.

Compared to the frequency-based Memtis, HybridTier's speedups mainly come from its adaptability and low cache overhead. Compared to recency-based AutoNUMA and TPP, HybridTier can identify hot pages more accurately. Surprisingly, Memtis often performs worse with a higher fast:slow ratio. We profile Memtis and observe that under larger fast-tier memory, Memtis performs additional background activities that result in higher runtime overhead.

GAP. Figure 10 (a) to (f) show the relative performance of HybridTier compared to baselines. On average, HybridTier outperforms TPP, AutoNUMA, Memtis, ARC, and TwoQ by 51%, 16%, 29%, 88%, and 88% respectively. Out of the GAP workloads, HybridTier shows the largest speedup for BFS, outperforming the second best system by 33% on average for both input graphs. The reason behind this is that BFS is a "single-source" kernel, where a different source vertex is selected for each iteration [5]. In contrast, CC and PR are "whole-graph" kernels, where the entire graph is processed the same way every trial. As a result, the BFS kernel experiences different hotness distributions for different source

vertices. HybridTier's adaptive tiering policy can quickly adjust to this change in hotness distribution.

In terms of absolute runtime, all tiering systems perform worse under uniform random input graph than under Kronecker graph. This is expected since uniform random graph represents the worst case in terms of locality. HybridTier's speedup over the second-best system grows from 15% on the Kronecker graph to 53% on the uniform random graph for BFS. This occurs since uniform random graph amplifies variations in node hotness. A more uniform graph is more likely to produce diverse hot sets, whereas a more concentrated graph, such as the Kronecker graph, tends to maintain a more consistent hot set of nodes.

In general, ARC and TwoQ show similar performances. This is expected, since both ARC and TwoQ use multiple LRU queues to estimate item recency and frequency. While in theory, TwoQ can perform worse since it has two parameters that need to be tuned (Kin and Kout), we found that the default values provided by the original paper [34] worked well: $K_{in} = maxSize/4$ and $K_{out} = maxSize/2$. Compared to other tiering systems, ARC and TwoQ generally perform worse. We profile ARC and TwoQ's page migrations and observe that this is mainly because of their lenient promotion policies. Upon a cold miss (the first time a page is sampled), both systems directly promote the missed page. We observe that this promotion policy is often too aggressive and can mistakenly promote cold pages. Except for Memtis, as fast-tier capacity increases, the performance gap between HybridTier and other baselines reduces, since the penalty for mispromotion is low with abundant fast-tier memory. Similar to CacheLib, we observe Memtis performance drops at higher fast:slow configurations.

SPEC CPU, Silo, and XGBoost. On average, HybridTier outperforms the second best system by 3%, 20%, and 8% for SPEC CPU, Silo, and XGBoost respectively. While AutoNUMA has the second best performance on SPEC CPU and XGBoost, it performs worse than Memtis on Silo. Silo uses YCSB, which uses a workload generator that assumes no changes in hotness, as each key remains equally hot throughout the benchmark. This is advantageous for the hotness histogram adopted by Memtis, as discussed in Section 2.3.1.

Comparison against all fast-tier. Figure 11 shows the performance of HybridTier normalized against baseline where only the fast-tier memory is used. This represents the performance upper bound of memory tiering systems. Under 1:16, 1:8, and 1:4 memory configurations, HybridTier is on average 14%, 9%, and 6% slower than all fast-tier.

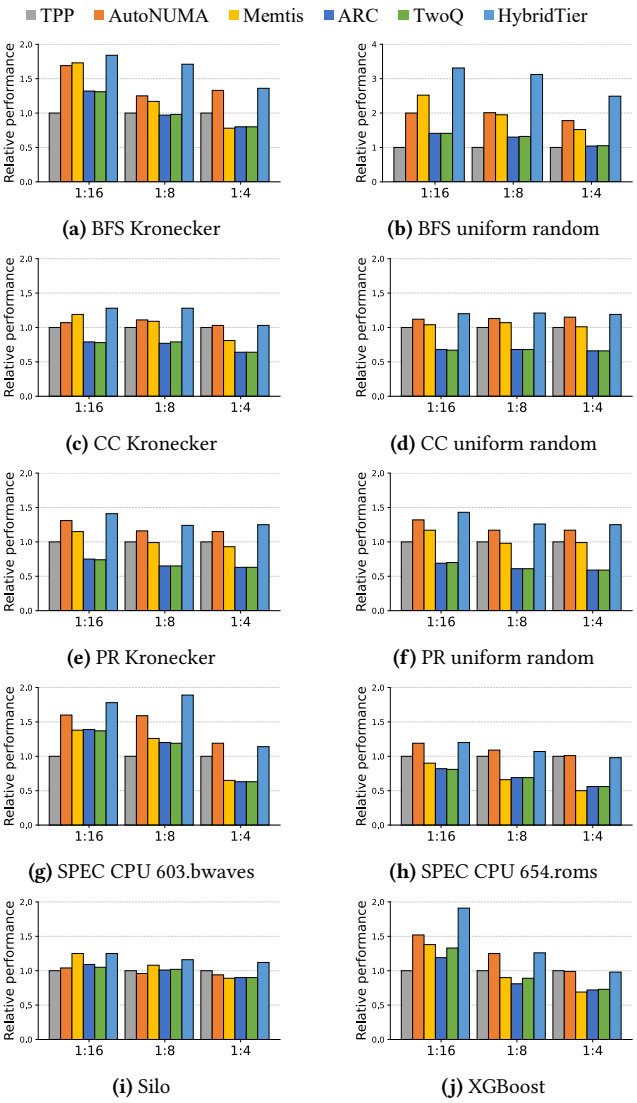


Figure 10. Performance comparison of HybridTier. All performance normalized against TPP. Higher is better.

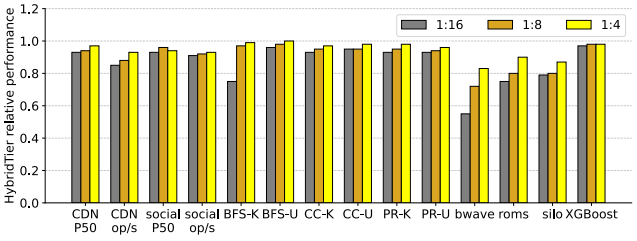


Figure 11. HybridTier performance normalized against baseline using all fast-tier memory.

6.2 Huge Page Performance

To evaluate HybridTier's huge pages performance, we compare against Memtis [41] on all workloads. Figure 12 shows that on average, HybridTier outperforms Memtis by 9% and 11% for 1:8 and 1:4 configurations while performing on par for 1:16. HybridTier shows the most performance improvements over Memtis on CacheLib social-graph, BFS, and PR.

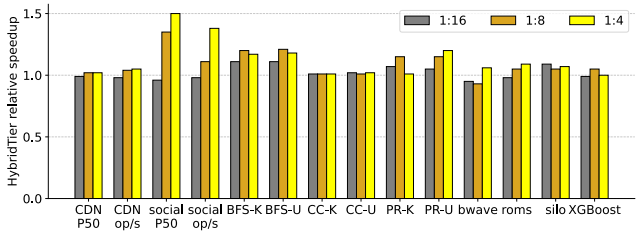


Figure 12. HybridTier huge page performance normalized against Memtis. Higher is better for HybridTier.

Table 3. Minutes required to adapt to new access distribution (reach within 1% of the steady-state median latency).

	CDN			Social-graph		
	1:16	1:8	1:4	1:16	1:8	1:4
Memtis	>60	42.6	>60	34.2	>60	29.6
HybridTier	25.6	25.2	23.4	9.6	10.1	8.9
Relative Reduction	2.3×	1.7×	2.6×	3.6×	5.9×	3.3×

Table 4. Size of metadata relative to total memory capacity.

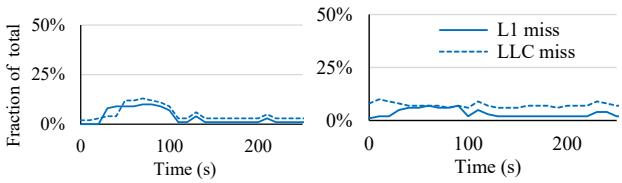
	1:16	1:8	1:4
Memtis	0.39%	0.39%	0.39%
HybridTier	0.050%	0.097%	0.192%
Relative Reduction	7.8×	4.0×	2.0×

6.3 Detailed Comparison Against Memtis

In this section, we compare HybridTier against Memtis, the state-of-the-art frequency-based tiering system that also uses PEBS hardware sampling. We compare in terms of adaptiveness to dynamic distributions (Section 6.3.1), metadata memory overhead (6.3.2), and tiering cache overhead (6.3.3).

6.3.1 Adaptiveness to Dynamic Distributions. Table 3 shows the amount of time required for HybridTier and Memtis to adapt to a new hotness distribution on two CacheLib workloads. We measure how long it takes for each tiering system to reach within 1% of the steady-state median latency. On average, HybridTier requires 3.2× less time to adapt. HybridTier's tiering policy considers both long-term access frequency and short-term access momentum, which enables it to quickly capture pages that recently turned from cold to hot and vice versa. On the other hand, Memtis must wait for its cooling mechanism to reduce the page's access count, resulting in a long delay before the page can be demoted.

6.3.2 Metadata Memory Overhead. Table 4 shows the relative amount of metadata incurred by HybridTier compared to Memtis. On average, HybridTier incurs 4.6× less metadata overhead than Memtis. Since HybridTier's metadata size scales with the size of fast-tier memory, it achieves larger memory savings at lower fast-tier sizes. On the other hand, Memtis metadata overhead scales with the total memory capacity and thus remains constant in our setup.



(a) Under 4KB page. **(b)** Under huge page.

Figure 13. Cache misses due to HybridTier tiering activities as a fraction of the system total.

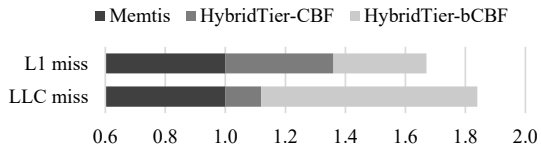


Figure 14. HybridTier cache miss reduction breakdown.

6.3.3 Cache Overhead. To evaluate tiering cache overhead, we use `perf` to separately record the number of cache accesses issued by the application and by tiering threads. Figure 13 shows that on average, HybridTier generates 5% and 4% of total cache misses under regular and huge pages respectively. Overall, HybridTier reduces the total number of L1 and LLC cache misses by 1.7× and 1.8× when using regular pages, and 3.2× and 3.5× under huge pages. Figure 14 breaks down the impact of adopting 4-bit CBF and blocked-CBF under regular pages. Using standard CBF (HybridTier-CBF) moderately reduces cache misses by 12 – 36% and applying blocked CBF (HybridTier-bCBF) reduces misses by another 31 – 72%. We conclude that both optimizations are effective while blocked CBF provides a larger reduction.

6.4 Understanding HybridTier Performance

In this section, we analyze the impact of HybridTier's key ideas. Specifically, we discuss 1) how frequency-momentum tracking affects performance (6.4.1) 2) how counting bloom filter impacts tiering accuracy (6.4.2).

6.4.1 Frequency Momentum Tracking. Figure 15 shows that tracking both frequency and recency is most effective for CacheLib and XGBoost, improving their performance by 8.5% on average. For BFS, CC, and PR, the performance is similar. This can be attributed to the fact that these three workloads have small hot sets that can easily fit in fast-tier memory. Figure 16 shows the cumulative access distribution for all workloads we evaluate. For GAP workloads under Kronecker graph, 94% of allocated pages have 0 access frequency, meaning that only 6%, or 20GB of pages are considered warm or hot. Since the fast-tier capacity is 64GB, HybridTier can achieve good performance without the momentum tracker.

6.4.2 Counting Bloom Filter. We first justify HybridTier's design decision to cap the size of each access counter to 4

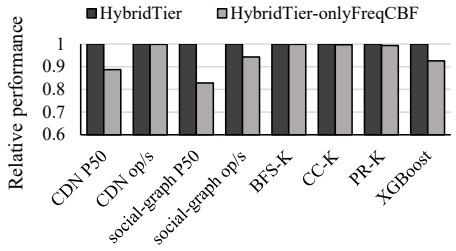
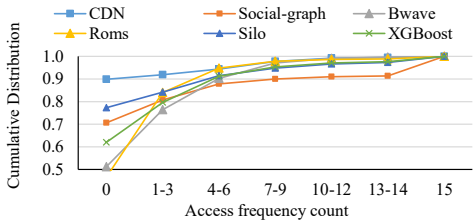
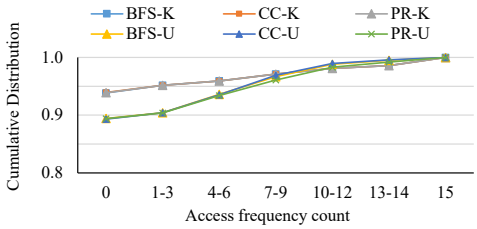


Figure 15. Performance of HybridTier if only the frequency tracker is used (1:8 configuration).



(a) CacheLib, SPEC CPU, Silo, and XGBoost.



(b) GAP. K: Kronecker graph, U: uniform random.

Figure 16. Access hotness distributions of 12 workloads evaluated. Social-graph has the largest fraction of pages with access count >= 15, equal to 25GB of memory.

bits. From Figure 16, we observe that for all workloads except for social-graph, the fraction of pages with frequency ≥ 15 is less than 3%. If the ratio between fast and slow-tier memory is greater than 3% (or roughly 1:32), which is common in practical settings, such pages should all be placed in fast-tier memory. Thus, the tiering system can treat such pages equally and does not need to differentiate between them. In cases where more than 4 bits are required, users can choose to either increase the counter width to 8 or 16 at the expense of higher memory and cache overhead.

Next, we wish to understand the accuracy of CBF. To measure CBF accuracy, we modify HybridTier to maintain a hash table in addition to the CBF. Since the hash table guarantees exactness (Section 3.2), we use access statistics in the hash table as the ground truth. Table 5 shows that a 64MB CBF agrees with the hash table for more than 99.6% of migration decisions. In the 1:16 configuration, HybridTier uses 128MB CBF to ensure high tracking accuracy. This confirms that despite being probabilistic, CBF can introduce minimal inaccuracy in the context of memory tiering.

Table 5. Accuracy of migration decisions made by counting bloom filter running CacheLib under 1:16 configuration.

CBF size (MB)	256	128	64	32	8
Accuracy	99.72%	99.65%	99.62%	99.42%	96.92%

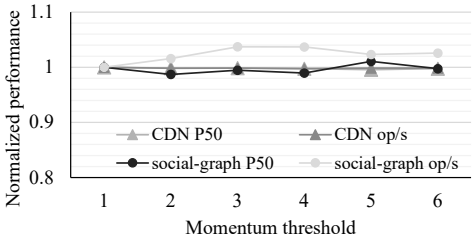


Figure 17. Momentum threshold sensitivity.

6.4.3 Momentum Threshold. We conduct a sensitivity study to understand the effect of momentum thresholds on HybridTier performance, shown in Figure 17. The social-graph workload is more sensitive to momentum threshold than CDN since it has a larger hot set (Figure 16), making fast-tier memory more scarce. Reducing the momentum threshold below 3 negatively impacts HybridTier's performance, since a cold page that is accessed only a few times can be mistakenly promoted. Increasing the momentum threshold beyond 3 does not significantly improve performance for workloads we evaluated. If the default momentum threshold is suboptimal for a specific application, HybridTier allows users to adjust its value to improve performance.

7 Discussions

Userspace vs. Kernel Tiering. We implement HybridTier in the userspace due to its flexibility. This advantage is recognized by both prior memory tiering systems [69, 73] and software systems such as userspace networking systems [15, 45] and file systems [4]. HybridTier's core design principles (frequency-momentum metrics, probabilistic access tracking) are general and can be implemented in kernel space.

Global Tiering. To support global memory tiering (e.g., multi-tenant VM, co-located applications), one could use a central HybridTier controller that coordinates with individual HybridTier instances. Each HybridTier instance would report local hot/cold items to the central controller, which makes global promotion/demotion decisions.

One-time-only Access Patterns. To handle applications with one-time-only access patterns such as scanning and pointer chasing, the user may tune HybridTier's momentum hotness threshold parameter. A higher momentum threshold makes HybridTier more resistant to fast-tier memory pollution due to one-time-only accesses, but at the same time reduces HybridTier's ability to adapt to hotness changes.

Selecting Configurable Parameters. HybridTier automatically adjusts the frequency hotness threshold based on the hotness distribution, similar to the approach taken by Memtis. We empirically the momentum hotness threshold to 3, which works well across workloads we evaluated. We perform a sensitivity study in section 6.4.3.

8 Related Works

Memory Tiering Systems. Tiered memory systems can utilize various types of memory technologies as the slower tier. Examples include solid state drives [9, 25, 35] or persistent memory [1, 18, 36, 37, 46, 69]. The slow-tier memory can be placed on the same server as the fast-tier memory (near memory) [29, 48, 69], or on remote servers (far memory) [73, 87, 90], where memory accesses are served from the network. In terms of programmability, tiering systems range from application-transparent [39, 48, 69, 72, 82] to requiring manual modifications to the application [67, 73].

AutoNUMA [29] and TPP [48] are recency-based CXL tiering systems. Memtis [41] is a frequency-based tiering system designed for both persistent memory and CXL memory. HeMem [69] is a persistent memory tiering system that performs frequency-based tiering at the page granularity. Application-guided tiering systems such as memkind [67], SMDK [55], Unimem [83], Xmem [17], and 2PP [81] offer a potential for tiering systems to understand application semantics through profiling or application modifications. Prior works [33, 50, 68] have accelerated tiering using specialized hardware. In contrast, HybridTier does not require any changes to the hardware or application. Memstrata [89] is a multi-VM memory allocator that utilizes Intel Flat Memory Mode, which requires hardware support. HybridTier is a runtime solution that is hardware agnostic. NOMAD [85] focuses on removing page migration from the critical path of program execution. Colloid [80] focuses on balancing access latencies. Both NOMAD and Colloid can be integrated with existing memory tiering systems such as HybridTier.

General Caching Algorithms. LRU tracks item access recency. CLOCK [11] is an approximation of LRU and thus suffers from the same drawback [49]. HybridTier adopts a hybrid frequency-recency caching algorithm. Among prior hybrid caching algorithms, we found the following to be the most relevant: LRFU [40] tracks the weighted average for each item. By adjusting the decay factor, LRFU provides a spectrum of policies between LRU and LFU. Memtis [41] adopts a variation of LRFU policy with decay factor of 1/2. However, LRFU combines the access recency and frequency of an item into a single weighted average, LRFU cannot track both frequency and recency accurately. For example, a lower decay factor captures frequency more accurately but sacrifices recency. HybridTier addresses this challenge by allocating two counters to track frequency and recency independently. ARC [49] maintains two LRU lists, one for items seen only once, and the other for items seen at least twice. The first list estimates "recency" and the second "frequency". This approach has been adopted by several memory tiering works, such as TPP [48] and Multi-CLOCK [46]. However, since ARC does not maintain frequency counts [49], it cannot distinguish between hot and warm items, which is critical for memory tiering in resource-constrained scenarios [41]

Approximate Data Structures. Caching policies using approximate data structures have been proposed in various domains [16, 22, 31, 76]. More specifically, prior works study counting bloom filters from the theory side [54, 71] or apply counting bloom filters in domains such as database, in-memory caching, and network communication [3, 19, 43, 44]. While we drew inspiration from these works, to our knowledge, HybridTier is the first work to practically apply the unique advantages of CBF to memory tiering. HybridTier introduces efficient promotion/demotion mechanisms based on CBF's unique properties, alongside flexible migration policies to make fine-grained frequency-based tiering practical.

Disaggregated Memory Systems. Memory disaggregation [47, 73, 87, 90] expands the main memory capacity by placing additional memory modules in remote servers. Similar to memory tiering on a single machine, disaggregated memory systems also aim to place the hottest data on the local fast-tier memory. Our work is orthogonal to works in this domain. In addition to locally-attached CXL memory managed by HybridTier, the target server can further expand its memory capacity through disaggregated tiering systems.

9 Conclusion

We propose HybridTier, an adaptive and lightweight tiered memory system. HybridTier quickly adapts to changing access distributions by tracking both long-term data access frequency and short-term access momentum simultaneously. At the same time, HybridTier achieves low metadata memory and cache overhead by adopting probabilistic access frequency tracking. HybridTier outperforms prior works on a wide range of workloads and memory configurations while reducing tiering memory and cache overhead.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Sergey Blagodurov, for their valuable feedback. We thank members of the EcoSystem Research Group at the University of Toronto for their suggestions. Special thanks to Kan Wu, Kim Keeton, Nandita Vijaykumar, Natalie Enright Jerger, and Samira Khan for their feedback. This research was supported in part by the Ontario Graduate Scholarship, Innovation JELF Grant, NSERC Discovery Grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

References

[1] Neha Agarwal and Thomas F. Wenisch. 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 (ASPLOS)*, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Amazon. XGBoost algorithm. https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html#Instance-XGBoost-training-cpu. Accessed: 2024.

[3] Apache Software Foundation. Class countingbloomfilter. https://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/util/bloom/CountingBloomFilter.html. Accessed: 2024.

[4] The Linux Kernel Archives. Fuse. https://www.kernel.org/doc/html/next/filesystems/fuse.html. Accessed: 2024.

[5] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite, 2015.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 753–768, November 2020.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[8] CacheLib. Cachebench overview. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_Overview/. Accessed: 2024.

[9] CacheLib. Hybrid cache. https://cachelib.org/docs/Cache_Library_Architecture_Guide/hybrid_cache. Accessed: 2024.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[11] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A Paging Experiment with the Multics System*. Project MAC. Massachusetts Institute of Technology, 1968.

[12] Jonathan Corbet. Cramming more into struct page. https://lwn.net/Articles/565097/, 2013.

[13] Jonathan Corbet. Persistent memory support progress. https://lwn.net/Articles/640113/, 2015.

[14] Criteo. Criteo 1TB click logs dataset. https://ailab.criteo.com/criteo-1tb-click-logs-dataset/. Accessed: 2024.

[15] DPDK. Data plane development kit. https://github.com/DPDK/dpdk. Accessed: 2024.

[16] Amit Dua, Megha Shishodia, Nikhil Kumar, Gagangeet Singh Aujla, and Neeraj Kumar. Bloom filter based efficient caching scheme for content distribution in vehicular networks. In *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2019.

[17] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4), nov 2017.

[20] Facebook. Cachelib. https://github.com/facebook/CacheLib. Accessed: 2024.

[21] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *SIGCOMM Comput. Commun. Rev.*, 28(4):254–265, oct 1998.

[22] Jie Gao, Shan Zhang, Lian Zhao, and Xuemin Shen. The design of dynamic probabilistic caching with time-varying content popularity. *IEEE Transactions on Mobile Computing*, 20(4):1672–1684, 2021.

[23] Christina Giannoula, Kailong Huang, Jonathan Tang, Nectarios Koziris, Georgios Goumas, Zeshan Chishti, and Nandita Vijaykumar. DaeMon: Architectural Support for Efficient Data Movement in Fully Disaggregated Systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(1), March 2023.

[24] The ROMS/TOMS Group. 654.roms spec cpu 2017 benchmark description. https://www.spec.org/cpu2017/Docs/benchmarks/654.roms_s.html.

[25] Herodotos Herodotou and Elena Kakoulli. Automating distributed tiered storage management in cluster computing. *Proc. VLDB Endow.*, 13(1):43–56, sep 2019.

[26] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An evolutionary study of linux memory management for fun and profit. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 465–478, Denver, CO, June 2016.

[27] Ying Huang. memory tiering: hot page selection with hint page fault latency. https://patchwork.kernel.org/project/linux-mm/patch/20210722031819.3446711-5-ying.huang@intel.com/. Accessed: 2024.

[28] Ying Huang. memory tiering: hot page selection with hint page fault latency. https://lore.kernel.org/linux-mm/bf23f05830db51bab3b06bac6e54d4743d37e955.camel@intel.com/. Accessed: 2024.

[29] Ying Huang. [patch -v4 0/3] memory tiering: hot page selection. https://lwn.net/ml/linux-kernel/20220622083519.708236-1-ying.huang@intel.com/. Accessed: 2024.

[30] Thomas Hurst. Bloom filter calculator. https://hur.st/bloomfilter/. Accessed: 2024.

[31] Hideo Inagaki, Ryota Kawashima, and Hiroshi Matsuo. Improving apache spark's cache mechanism with lrc-based method using bloom filter. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 496–500, 2018.

[32] Intel. Maximize your CPU resources for XGBoost training and inference. https://www.intel.com/content/www/us/en/developer/videos/maximize-cpu-resources-xgboost-training-inference.html#gs.47qye6. Accessed: 2024.

[33] Intel. Why is the Intel Optane Persistent Memory in Memory Mode not persistent? https://www.intel.com/content/www/us/en/support/articles/000055895/memory-and-storage/intel-optane-persistent-memory.html. Accessed: 2024.

[34] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[35] Elena Kakoulli and Herodotos Herodotou. OctopusFS: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, page 65–78, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS - OS design for heterogeneous memory management in datacenter. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, 2017.

[37] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *USENIX Annual Technical Conference (ATC)*, pages 821–837, July 2021.

[38] Mark Kremenetsky. 603.bwaves SPEC CPU 2017 benchmark description. https://www.spec.org/cpu2017/Docs/benchmarks/603.bwaves_s.html. Accessed: 2024.

[39] 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. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[41] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Wenjing Liu, Zhiwei Xu, Jie Tian, and Yujun Zhang. Towards in-network compact representation: Mergeable counting bloom filter vis cuckoo scheduling. *IEEE Access*, PP:1–1, 04 2021.

[44] Ben Manes. Caffeine. https://github.com/ben-manes/caffeine. Accessed: 2024.

[45] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP)*, 2019.

[46] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic tiering for hybrid memory systems. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, 2022.

[47] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *USENIX Annual Technical Conference (ATC)*, pages 843–857, July 2020.

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

[49] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003.

[50] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, 2015.

[51] Micron. CZ120 memory expansion module. https://www.micron.com/solutions/server/cxl#:~:text=CXL%20memory%20expansion%20serves%20as,workloads%20for%20CXL%2Denabled%20servers. Accesed: 2024.

[52] Diego Moura, Daniel Mosse, and Vinicius Petrucci. Performance characterization of AutoNUMA memory tiering on graph analytics. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, nov 2022.

[53] Onur Mutlu. Memory scaling: A systems architecture perspective. https://users.ece.cmu.edu/~omutlu/pub/mutlu_memory-scaling_memcon13_talk.pdf, 2013.

[54] Sabuzima Nayak and Ripon Patgiri. countBF: A general-purpose high accuracy and space efficient counting bloom filter. In *17th International Conference on Network and Service Management (CNSM)*, 2021.

[55] OpenMPDK. Scalable memory development kit. https://github.com/OpenMPDK/SMDK. Accessed: 2024.

[56] Panmnesia. Panmnesia technologies. https://panmnesia.com/#technology. Accessed: 2024.

[57] J. Thomas Pawlowski. Prospects for memory. https://passlab.github.io/mchpc/mchpc2019/presentations/MCHPC_Pawlowski_keynote.pdf. Accessed: 2024.

[58] Amazon EC2 T2 instances. https://aws.amazon.com/ec2/instance-types/t2/. Accessed: 2024.

[59] AMD research instruction based sampling toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit. Accessed: 2024.

[60] Computer Express Link. https://computeexpresslink.org/. Accessed: 2024.

[61] A high performance caching library for java. https://github.com/ben-manes/caffeine/blob/3f4c1599992accac7d596e3047fcb0866cabe048/caffeine/src/main/java/com/github/benmanes/caffeine/cache/FrequencySketch.java#L42. Accessed: 2024.

[62] Memtis: Efficient memory tiering with dynamic page classification and page size determination. https://github.com/cosmos-sjtu/memtis/blob/838a802680a8a760d3dea50754d6ea8a8530f6aa/linux/mm/htmm_core.c#L1030. Accesed: 2024.

[63] Modern bloom filters: 22x faster! https://save-buffer.github.io/bloom_filter.html#org7b03738. Accessed: 2024.

[64] Performance counters tools. https://learn.microsoft.com/en-us/windows/win32/perfctrs/performance-counters-tools. Accessed: 2024.

[65] Release tiering-0.8. https://kernel.googlesource.com/pub/scm/linux/kernel/git/vishal/tiering/+/refs/tags/tiering-0.8. Accessed: 2024.

[66] UT hash. https://troydhanson.github.io/uthash/. Accessed: 2024.

[67] pmem.io. memkind. https://pmem.io/memkind/. Accessed: 2024.

[68] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, page 85–95, New York, NY, USA, 2011. Association for Computing Machinery.

[69] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[70] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA,

2024. Association for Computing Machinery.
[71] Pedro Reviriego and Ori Rottenstreich. The tandem counting bloom filter - it takes two counters to tango. *IEEE/ACM Transactions on Networking*, 27(6):2252–2265, 2019.
[72] Vinod Chegu Rik van Riel. Automatic numa balancing. https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf, 2014. Accesed: 2024.
[73] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, November 2020.
[74] Samsung. Samsung electronics introduces industry's first 512gb CXL memory module. https://news.samsung.com/us/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module/. Accesed: 2024.
[75] Debendra Das Sharma. Introduction to compute express link. https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf. Accessed: 2024.
[76] David Starobinski and David Tse. Probabilistic methods for web caching. *Perform. Eval.*, 46(2–3):125–137, October 2001.
[77] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung H. Ahn, Tianyin Xu, and Kim Nam S. Demystifying cxl memory with genuine cxl-ready systems and devices. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.
[78] Sysomos. Inside twitter: An in-depth look inside the Twitter world. https://www.key4biz.it/files/000270/00027033.pdf, 2014. Accesed: 2024.
[79] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.
[80] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 79–94, New York, NY, USA, 2024. Association for Computing Machinery.
[81] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting program semantics to place data in hybrid memory. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 163–173, 2015.
[82] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.
[83] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2017. Association for Computing Machinery.
[84] XGBoost. XGBoost: eXtreme gradient boosting. https://github.com/dmlc/xgboost. Accessed: 2024.
[85] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: non-exclusive memory tiering via transactional page migration. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024.
[86] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, Santa Clara, CA, July 2024.
[87] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.
[88] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 191–208, November 2020.
[89] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024.
[90] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 55–71, Carlsbad, CA, July 2022.

A Artifact Appendix

A.1 Abstract

We provide the source code of HybridTier and scripts to reproduce the results presented in Figures 9, 10, and 12. The artifact includes the HybridTier runtime and various open-source large memory workloads. To facilitate the AE process, we provide access to remote access to the authors' machine with the pre-installed software.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Memory tiering
- **Compilation:** g++
- **Hardware:** Multi-NUMA node systems
- **Metrics:** End-to-end wall clock time
- **Experiments:** End-to-end evaluation using regular page (4KB) and huge page (2MB)
- **Experiments:** Reproduce part of the results in Figures 9, 10, and 12
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes
- **How much time is needed to complete experiments (approximately)?:** 45 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT license

A.3 Description

A.3.1 How to access. The artifact is available for download on GitHub: https://github.com/kevins981/hybridtier-asplos25-artifact. To facilitate the AE process, we provide reviewers with remote server access. In such a case, please skip directly to section A.5. Otherwise, we provide detailed instructions to reproduce in the GitHub repository.

A.3.2 Hardware dependencies. Requires a single x86_64 Linux host. The processor must support Processor Event-Based Sampling (PEBS). The system must have multiple tiers of memory. The paper uses a system with two NUMA nodes to emulate tiered CXL memory systems. The provided source code will not work out of the box on a system with e.g., real CXL memory or persistent memory. To do so, the processor performance counters must be modified to capture the appropriate memory access events. We provide more details in the GitHub repository.

A.3.3 Software dependencies. This artifact depends on the following environment.

- Ubuntu 20.04.4 LTS
- g++ 9.4.0

While the HybridTier runtime does not require a specific kernel, emulating CXL memory using remote NUMA node requires a few minor changes to the Linux kernel. We provide details on the kernel modifications required and the source code of the kernel we used to evaluate HybridTier in the GitHub repository.

A.4 Installation

```
git clone https://github.com/kevins981/hybridtier-
    asplos25-artifact.git
```

This artifact has the following structure:

- `repro.sh`: Reproduce major results
- `run_{workload}.sh`: Run experiments for a particular workload.
- `tiering_runtime/`: HybridTier implementation
- `hook/`: source code for launching HybridTier runtime
- `tools/`: auxiliary tools used for experiments

A.5 Experiment workflow

We provide a script, `repro.sh`, to reproduce the major results. Experiment progress can be found in `./exp_log`. After all experiments complete, `repro_postprocess.sh` extracts results and create plots.

```
cd hybridtier-asplos25-artifact
sudo ./repro.sh
# wait for all experiments (~45 hours)
./repro_postprocess.sh
```

To evaluate different tiering systems and fast:slow memory ratios, `repro.sh` will automatically load the appropriate kernel and reboot the server. Thus, it is normal for the server to be temporarily offline during reboots. Upon reboot, `repro.sh` will automatically perform the next experiments. No manual interventions are needed during this process.

Each experiment requires approximately 1 hour to complete. As a result, running all experiments in Figures 9, 10, and 12 would consume more than 200 hours. To reduce time required for artifact evaluation, we made the following changes to the experiments in Figures 9, 10, and 12:

- Out of the 6 tiering systems, we reproduce HybridTier (ours) and Memtis, the best-performing prior tiering system.
- Out of the 3 memory configurations, we reproduce 1:4 and 1:16 fast:slow memory ratios. 1:4 represents systems with an abundance of fast-tier memory, and 1:16 represents systems with limited fast-tier memory.
- Out of the two input graphs used for graph analytic workload (GAP), we reproduce results using the kron graph, which is closer to real-world graphs than uniform random graph.
- We leave out SPEC CPU experiments as the workload is not open source.
- We leave out XGBoost experiments as each run can take close to 2 hours to complete.

A.6 Evaluation and expected results

After all experiments are completed, `repro_results.csv` will be created that summarizes the result of all experiments performed. This file contains both absolute and relative performance results. Each table is also plotted and saved under `figs/`.