Better Memory Tiering, Right from the First Placement

João Póvoas
INESC-ID, Universidade de Lisboa
Lisbon, Portugal
joao.povoas@tecnico.ulisboa.pt

João Barreto
INESC-ID, Universidade de Lisboa
Lisbon, Portugal
joao.barreto@tecnico.ulisboa.pt

Bartosz Chomiński
Huawei Warsaw Research Centre
Warsaw, Poland
bartosz.chominski@huawei.com

André Gonçalves
INESC-ID, Universidade de Lisboa
Lisbon, Portugal
andre.a.goncalves@tecnico.ulisboa.pt

Fedar Karabeinikau
Huawei Warsaw Research Centre
Warsaw, Poland
fedar.karabeinikau@huawei.com

Maciej Maciejewski
Huawei Warsaw Research Centre
Warsaw, Poland
maciej.maciejewski@huawei.com

Jakub Schmiegel
Huawei Warsaw Research Centre
Warsaw, Poland
jakub.schmiegel@huawei.com

Kostiantyn Storozhuk
Huawei Warsaw Research Centre
Warsaw, Poland
kostiantyn.storozhuk1@huawei.com

Abstract

Heterogeneous memory (HMem) architectures have recently emerged and revolutionized the traditional memory hierarchy. This trend is likely to increase with the rise of the Compute Express Link (CXL) standard. A fundamental problem that arises when working with HMem architectures is data placement: at which memory tier should the data objects of an application be placed to optimize its performance? Existing proposals that tackle this problem at the system-level operate transparently to the application, hence without explicit placement hints from it.

Such lack of knowledge is a key challenge to the *first placement* of new objects. Today's state-of-the-art systems for memory tiering solve the *first-placement* problem by blindly guessing that new pages (holding new objects) should be better placed in the fast tier. However, for large working sets, this blind guess fails for a large fraction of pages, which results in important performance shortcomings.

This paper aims to mitigate such shortcomings by replacing blind guess in the first placement with an educated guess, which takes advantage of past object-level access patterns. We propose a novel memory tiering system that addresses the first-placement problem by combining hmalloc, an HMem-aware memory allocation library, with Ambix, a page-based memory tiering system, and exploiting their object and page-level synergies. Our experimental evaluation when running realistic HPC benchmarks on a real HMem system demonstrates that our synergistic approach is effectively able to address the first-placement limitation of previous systems. Our approach achieves up to 2.03x speedup over traditional memory management through intelligent first placement alone. When combined with the state-of-the-art support for tiered page placement provided in the latest versions of Linux, performance further improves, reaching up to 2.28x speedup over modern memory tiering systems in certain HPC workloads.

CCS Concepts

• **Computer systems organization** → *Multicore architectures*; *Processors and memory architectures*; • **Software and its engineering** → **Memory management**; **Virtual memory**; **Main memory**; **Allocation / deallocation strategies**; • **General and reference** → General conference proceedings.

Keywords

Heterogeneous memory architectures, tiered page placement, memory management, memory allocation, operating systems

ACM Reference Format:
João Póvoas, João Barreto, Bartosz Chomiński, André Gonçalves, Fedar Karabeinikau, Maciej Maciejewski, Jakub Schmiegel, and Kostiantyn Storozhuk. 2025. Better Memory Tiering, Right from the First Placement. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3676151.3719378

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
ICPE '25, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1073-5/2025/05
https://doi.org/10.1145/3676151.3719378

1 Introduction

Heterogeneous memory (HMem) architectures have recently emerged and revolutionized the traditional memory hierarchy. HMem architectures may comprise memory nodes of different technologies — not just DRAM, but also die-stacked DRAM, high-bandwidth multi-channel RAM, or persistent memory — possibly organized in complex non-uniform access (NUMA) topologies. By combining different memory technologies, HMem allow today's systems to take advantage of the strengths of each technology — namely, in terms of latency, bandwidth, capacity, or cost per byte. As a result, applications may benefit from improved performance, energy-efficiency, and cost trade-offs. More recently, mainstream servers are quickly embracing the Compute Express Link (CXL) standard [25, 42]. CXL promises to take HMem architectures to an unprecedented level, opening it up to multiple tiers composed by memory devices from

different manufacturers, accessible in a cache-coherent and byte-addressable fashion.

The emerging HMem trend constitutes a notable opportunity for today's data-intensive applications, which are characterized by a high degree of data complexity and parallelism, as well as an ever-increasing demand for memory capacity. The memory tiers provided by HMem let applications place and process much larger working sets than in DRAM-only systems, while minimizing (or even avoiding) accesses to slow block-based storage.

HMem architectures introduce dramatic disruptions to the usual memory hierarchy assumptions that have guided decades of system and software design. Therefore, exploiting the full potential of HMem is notably challenging. A fundamental problem that arises when working with HMem architectures is data placement: *at which memory tier should the data objects of an application be placed to optimize its performance?*

Finding an appropriate data allocation strategy in HMem architectures, taking into account the performance trade-offs between the available tiers, is well-known to be a non-trivial problem with a substantial impact on the effective performance of data-intensive applications. This problem has received increased attention from the research community in recent years. The most prominent proposals tackle it from the operating system (OS) level [15], at virtual page granularity, in a transparent and dynamic fashion. As of today, proposals such as Memtis [23], TMTS [8], TPP [29], Nomad [49], MTM [36] or FlexMem [50] represent the state of the art along this research avenue. Among such proposals, TPP has taken an important step towards a wide adoption, since a big portion of its design has been merged into Linux' automatic NUMA balancing (AutoNUMA).

Since these proposals operate transparently to the application, they lack explicit placement hints from it. This constitutes a major challenge to the *first placement* of new objects. When the application allocates a new object, the pages that hold the object are allocated and, as soon as they are accessed, mapped to a given tier.

Today's state-of-the-art systems for memory tiering solve the *first-placement* problem by blindly guessing that every new page will be among the most accessed ones. Therefore, new pages are always placed in the fast tier, provided it has available space. However, for memory hungry applications whose resident working set is much larger than the fast tier's capacity, this blind guess will fail for a large fraction of pages. Therefore, these systems adopt a reactive strategy to heal any misplaced pages. After a page is first-placed, they track page-level accesses using best-effort low-overhead mechanisms to estimate per-page access frequencies. Then, based on such estimates, they determine what pages should be in the fast tier – typically the hottest ones. Finally, they migrate any page that belongs to that set but is currently standing in the slow tier, possibly demoting any cold pages from the fast tier to free the necessary page frames. This reactive approach tries to fix any wrong guesses at the first-placement instant (as well as adapt to workload changes).

Unfortunately handling the first-placement problem with a blind guess can have important performance shortcomings. First, any page that is initially misplaced will be soon migrated. This easily results in increased migration volumes. This can degrade the overall performance of the tiered memory system [49].

Second, misplaced pages linger in the fast tiers, and quickly consume its available space. As a consequence, when pages become hot – which can either occur when a new (hot) page is first-placed, or when a cold page standing in the slow tier has just become a hot spot –, they cannot be immediately placed to the fast tier. Instead, their promotion must either be aborted or postponed due to lack of free space in the fast tier. If hot pages cannot be timely promoted, the application can suffer important performance costs [50].

Motivated by these two shortcomings, we advocate that memory tiering systems should improve page placement right from the first guess. To achieve this, we take advantage of the prior observation [9] that in many applications, the objects that are allocated from the same place in code, or *allocation context*, tend to share similar access trends. Iterative applications, such as scientific applications for simulating molecular or plasma physics dynamics or machine learning algorithms, constitute notable examples of this trend. In such applications, each iteration typically allocates a variable amount of objects from multiple allocation contexts and frees such objects at the end of the iteration, where each kind of object (originated from a same allocation context) often shares similar access patterns.

From this observation, we claim that the memory allocation library (e.g. *malloc*) is a previously neglected opportunity to infer valuable application-level hints that can considerably improve the accuracy of first-placements. Concretely, by monitoring the access frequency of objects from a given allocation context in a program, one can predict the access frequency of new objects that are allocated from the same context and use that knowledge to guide the first-placements.

The key insight of our paper is that, by taking advantage of these predictions, the first-placement decisions can now rely on *educated guesses*, instead of a blind guesses. In principle, this can reduce the frequency of misplaced pages and, thus, the associated costs. Still, achieving this vision with low overhead and high accuracy, as well as transparently to the application, is a challenge.

To realize the above vision, this paper proposes the design and implementation of two novel components that cooperate as part of a larger memory tiering system: hmalloc, an HMem-aware memory allocation library, and Ambix, a page-based memory tiering system.

The proposed system is depicted in Figure 1. Both layers operate in synergy. hmalloc is a runtime library that intercepts *malloc* calls and decides at which tier each new object is first-placed ((1) in Figure 1). This non-trivial decision relies on two parameters that are not known *a priori* and that can change as time goes by: i) what is the expected access frequency of objects originated from this allocation context; and ii) how much physical memory is currently used by the objects already allocated from this allocation context. For the first parameter, hmalloc leverages processor event-based sampling (PEBS) to track memory accesses in background and, from such information, hmalloc dynamically estimates the average frequency of accesses to objects associated with each allocation context ((2)).

The second layer, Ambix, is a kernel module that seamlessly relies on the memory management mechanisms of the kernel to monitor memory activity on a per-page granularity and implement a page placement policy. Ambix complements the hmalloc layer

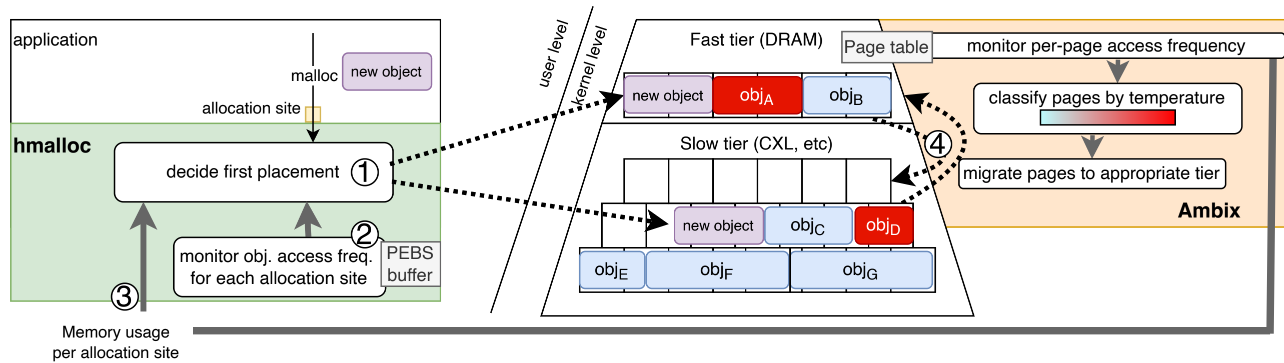

Figure 1: Overview of the proposed architecture.

in two ways. First, Ambix is able to measure how many pages holding objects of a given allocation context are actually mapped memory at some instant. This information is sent upstream, to be used by hmalloc as factor ii), which drives its first-placement decisions ((3)). Second, Ambix is able to determine misplaced pages – e.g. pages in the slow tier that are recently receiving a substantial amount of accesses – and migrate them to the appropriate tier ((4)). This action compensates any wrong first-placement guesses from hmalloc. Such wrong guesses are common in the early stages of an application execution (before hmalloc had the time to accurately infer the access frequency of each allocation context), or in the presence of workload changes.

We have implemented the proposed hmalloc+Ambix system and experimentally evaluated it against the standard memory tiering support from the latest Linux kernel, which is based on TPP [29], a state-of-the-art page placement system designed for CXL-enabled HMem. Our results demonstrate that intelligent first placement achieves up to 2.03x speedup over traditional memory management, effectively addressing first-placement limitations in previous systems. When combined with the state-of-the-art support for tiered page placement provided in the latest versions of Linux, performance further improves, reaching up to 2.28x speedup over modern memory tiering systems in certain HPC workloads.

Overall, this paper makes the following contributions:

- We propose the design of a novel memory tiering system which takes advantage of the synergies between a HMem-aware memory allocation library, hmalloc, and a system-level page placement system, Ambix. In contrast to the state-of-the-art in system-level memory tiering systems, our proposal is able to tackle the first-placement problem more effectively.
- We implement the proposed design as a kernel module that requires no changes to the Linux kernel, combined with a runtime library.
- We evaluate the proposed system against a state-of-the-art memory tiering system, available in the latest Linux kernel, with realistic HPC benchmarks.

The source code of Ambix is publicly available in https://github.com/inesc-id/ambix, and can also be used stand-alone. The hmalloc library is proprietary software.

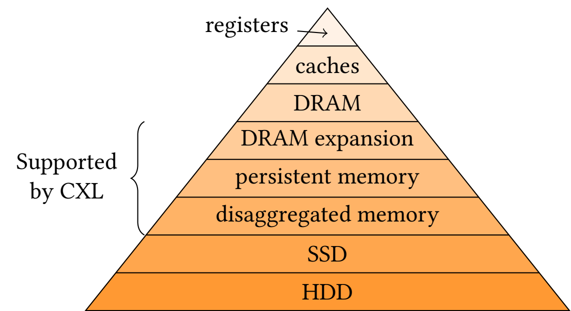

Figure 2: The new heterogeneous memory hierarchy that CXL enables.

The remainder of the paper is organized as follows. Section 2 presents background on memory tiering. Section 3 describes our proposed solution. Section 4 evaluates our proposal. Finally, Section 5 discusses related work, and Section 6 draws conclusions.

2 Background

Historically, data placement in traditional memory hierarchies (such as DRAM, SSD, disk) has relied on inclusive caching policies. In such policies, any block from a lower tier needs to be copied to the top tier to be directly accessed from there. These are well-studied and a good fit for memory hierarchies whose adjacent layers differ in cost, capacity and performance by several orders of magnitude, and where only the fastest layer can be accessed directly by applications [2].

However, the appearance of new memory technologies has introduced new HMem hierarchies that invalidate these fundamental assumptions and call for radically different approaches to data placement. Examples include the NUMA architectures that arise in multi-socket systems [6, 24], high-bandwidth memory+DRAM memory hierarchies [15], CPU-GPU memory hierarchies [10], software-defined far memories [20], or DRAM+persistent memory [3, 32], among others [53].

More recently, CXL-based memory hierarchies have emerged as an additional example of a HMem hierarchy. These will soon extend contemporary systems with HMem hierarchies such as the

one depicted in Figure 2. Compute express link (CXL) defines the interconnect protocol between the CPU and devices connected through the Peripheral Component Interconnect Express (PCIe) serial interface [5]. The memory in a CXL device is byte-addressable and cache-coherent. CXL uses a layered protocol approach with physical, data link, and transaction layers. It defines three transaction layer protocols, CXL.io, CXL.cache, and CXL.mem, multiplexed dynamically on top of the physical layer. CXL.io is used for device discovery, configuration, initialization, and DMA. CXL.cache allows devices to cache host memory coherently, and CXL.mem enables the host to access and manage device memory equivalent to local memory [5]. CXL devices can range from local CXL devices with DRAM expansions or even persistent memory, to disaggregated CXL memory pools.

In such memory HMem hierarchies, the traditional caching principle that each requested block needs to be brought into and served from the fastest memory becomes obsolete. In fact, Zhang et al. [53] have shown that optimal data placement policies resort to so-called *cache bypassing* strategies [30] that intentionally place some blocks in a slower memory to be accessed directly by applications.

An HMem hierarchy is exposed to the OS as distinct physical memory devices. Starting with Linux 4.0, the different memory types can be abstracted as distinct non-uniform memory access (NUMA) nodes. The (virtual) pages of an application can be placed at either node and are directly accessed through load and store operations. Therefore, applications have access to a larger aggregate main memory capacity, since the DRAM capacity is no longer hidden as a cache. In this setting, the default NUMA placement policy of Linux dictates that once a page is *first-touched* it is placed on the fastest node (DRAM) as long as it has free space; otherwise, the slowest node is selected.

2.1 Tiered page placement

The scope of our paper is on dynamic approaches to data placement for HMem. Dynamic approaches typically manage data placement on a per-page granularity, therefore we hereafter designate them as *tiered page placement* approaches. In contrast to static compile-time approaches, dynamic page placement typically incurs no programming burden, easily supports legacy applications, can dynamically adapt to workload changes by migrating pages across tiers, and naturally manages multiple concurrent applications from a system-wide perspective. The next section describes this approach in more detail.

This topic witnessed a sudden spike of attention following the emergence of new persistent memory technologies. A vast number of papers have studied this problem even before any persistent memory technology was commercially available [1, 13, 15, 17, 21, 22, 27, 37, 39, 43, 44, 52]. As soon as DCPMM and, later, CXL were released and became commercially available, many other proposals have appeared, this time on real HMem systems. This wave of proposals includes systems such as Autotiering [16], Hemem [34], MULTI-CLOCK [28] bkmalloc+MAT daemon [14], TMTS [8], vtMM [41], TPP [29], Memtis [23], Nomad [49], MTM [36], FlexMem [50] and Colloid [45].

Although these systems have important differences, they all share a common design backbone. Each time that a page is accessed

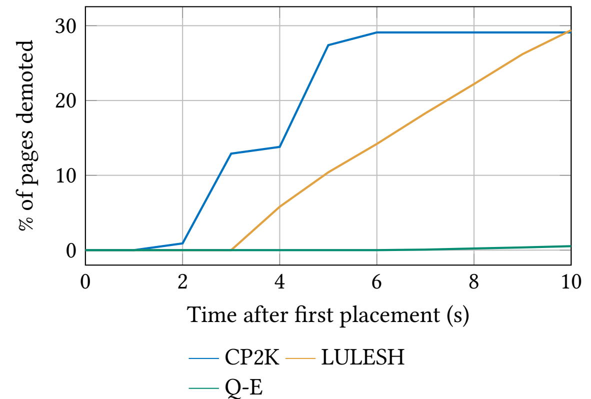

Figure 3: Relative volume of early demotions, using AutoNUMA to manage different HPC applications.

for the first time, a page fault is triggered, and the page is placed in the fast tier (typically DRAM) as long as free page frames exist in that tier. Otherwise, the page is placed in a lower tier. This constitutes the standard first-placement policy.

The fast tier is gradually filled up by new pages as they are first-accessed. Once a given DRAM occupancy threshold is reached, a set of cold pages is selected to be evicted to the slower tier. In the opposite direction, any page that has been mapped to the slow tier but is found to exhibit high enough access frequency is promoted to DRAM, as long as there is space available in that tier. In case there is not enough space in DRAM, the promotion is either aborted or halted, and a batch of cold pages in the fast tier is selected to be demoted.

The reliance on a blind guess for the first placement is a crucial limitation of this common design. First, any page that is misplaced at first is likely to be soon migrated. This results in increased migration volumes, which penalize the performance of the memory system. To illustrate this shortcoming, we run the CP2K, Quantum-Espresso and Lulesh HPC applications on a real HMem system based on DRAM with Intel Optane DC Persistent Memory, in which we enable AutoNUMA as the memory tiering system (for detailed specifications, see Section 4).

For every observed page demotion, we measure how much time has passed since the instant where that page was first-placed in memory. Figure 3 presents the relative distribution of page demotions, as a function of such time. This allows us to distinguish the early page demotions. As we can see, for CP2K and LULESH, a substantial fraction of pages is demoted a few seconds after their first-placement in every application. The root cause of this significant share of early promotions is incorrect first-placement of pages. In contrast, the Q-E benchmark appears to be more immune to this phenomenon.

Second, misplaced pages linger in the fast tiers, and quickly consume its available space. As a consequence, when the memory tiering system determines that a set of pages currently placed in the slow tier have become hot and, therefore, should be immediately

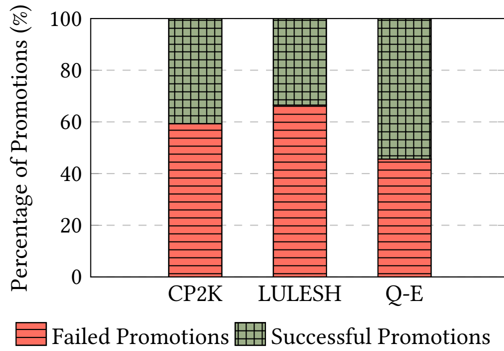

Figure 4: Successful promotions vs. failed promotions in AutoNUMA.

promoted to the fast tier, such promotion may be aborted (or postponed) due to lack of free space in the fast tier. In practice, these scenarios can severely constrain the action of the placement policy.

To understand the severity of this shortcoming, we also measured the amount of page promotions that are successful vs. those that fail due to insufficient free space in the fast tier. As we can see in Figure 4, AutoNUMA's behavior is strongly affected by a high rate of promotion failures, across the different applications.

3 Architecture

This paper proposes a novel memory tiering system that stacks hmalloc, an HMem-aware memory allocation library, on top of Ambix, a page-based memory tiering system. Both layers operate in synergy, as depicted in Figure 1.

The top layer, hmalloc, is a runtime library that intercepts *malloc* calls. It monitors the access frequency to a subset of the objects that it has allocated in the past, as well as the typical size that such objects take up in physical memory. For the first parameter, hmalloc leverages event-based sampling, as provided by the processor's Performance Monitoring Unit (PMU). The access profiling performed by hmalloc is performed at the granularity of allocation contexts, which we define precisely in Section 3.1.

Ambix serves as the second layer of our system architecture, functioning as a kernel module that utilizes the kernel's memory management mechanisms to monitor memory activity at a per-page granularity and implement an effective page placement policy. Ambix complements the hmalloc layer in two ways. First, Ambix evaluates the number of pages containing objects from a specific allocation context that are currently mapped into physical memory. This information is relayed upstream to hmalloc, providing a secondary parameter that informs its initial placement decisions. Second, Ambix identifies pages that are improperly located, such as those in the slower memory tier that are experiencing a high rate of recent accesses, and proactively migrates them to a more suitable tier. This corrective action mitigates any suboptimal initial placement choices made by hmalloc, which are common during the early stages of application execution (before hmalloc has adequately learned the access frequency of each allocation context), or when workload patterns change.

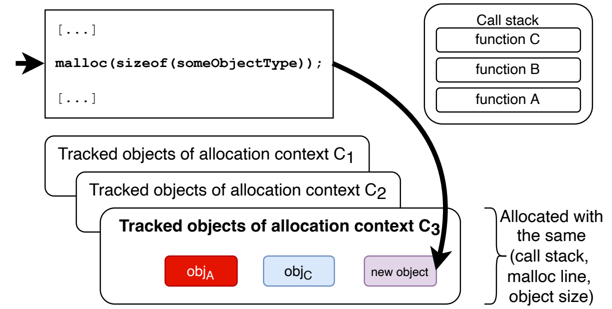

Figure 5: Allocation contexts and tracked objects in hmalloc.

3.1 Object allocation with hmalloc

Hmalloc library provides both direct and autonomous placement. Direct placement enables application developers to specify which object should target which memory, whereas autonomous placement is a transparent mode that automatically selects the target memory for each allocated object. In the remainder of the paper, we focus on autonomous mode.

The design of hmalloc is motivated by the findings of a previous study [9] that compared the effectiveness of using different program data features to predict memory usage and guide memory management, and concluded that the *allocation context* was the most effective feature to cluster objects with similar access trends. Therefore, we start by defining the notion of allocation context in hmalloc. We say that two objects belong to the same allocation context if the following 3 conditions hold: (i) both objects were allocated from the same place in code, (ii) when invoked from the same call stack context, and (iii) both objects have the same size. Therefore, when handling a new allocation invocation, hmalloc decides the target tier by taking an educated guess based on the past access patterns of previous objects coming from the same *allocation context*.

3.1.1 Estimating the hotness of allocation contexts. hmalloc intercepts functions that allocate memory and release memory in an application (*malloc()*, *calloc()*, *free()*, etc.). To accomplish this interception without changing the application code or recompiling it, we use the LD_PRELOAD mechanism when running an executable file. This allows hmalloc to be loaded before other libraries at runtime and selectively override functions in other shared libraries.

For a certain fraction of intercepted allocations, hmalloc identifies the corresponding allocation context and stores the address range of the newly allocated object to a list of *tracked objects* of that allocation context. This is illustrated in Figure 5.

As we describe next, accesses to tracked objects are monitored and used to guide future first-placement decisions. Monitoring only a fraction of allocations allows hmalloc to bound its memory and CPU overheads.

hmalloc classifies each allocation context according to its *hotness*, which estimates the per-byte access frequency of the objects originated from that allocation context. hmalloc uses event-based sampling as provided by the processor's PMU to monitor memory accesses to tracked objects. On x86, we use *MEM_TRANS_RETIRED.LOAD_LATENCY_GT_16*, which is accessible via *perf*. This

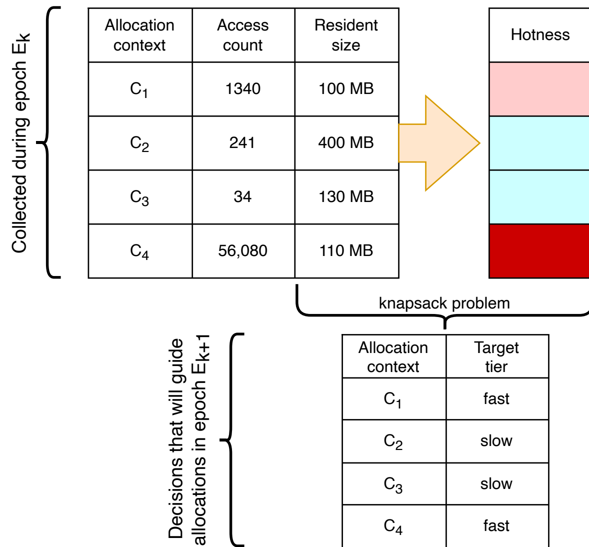

Figure 6: At a given epoch, hmalloc monitors access count and resident size for each allocation context and, from the calculated hotness, determines the target tier that will guide allocations in the next epoch.

allows hmalloc to get the virtual address of sampled memory accesses and the memory hierarchy level in which the memory was touched. The virtual address of the sample is looked up among the address ranges of tracked objects of every allocation context. If a match is found, the access count of the corresponding allocation context is incremented.

The hotness of an allocation context is estimated by dividing its access count by the total size of tracked objects from the same allocation context. However, the total allocated size may not precisely tell us how much physical memory is actually occupied by the tracked objects. In fact, we observe that, when big objects is allocated (e.g., >1GB), only a portion of their size is actually resident in physical memory. Therefore, to more accurately estimate per-allocation context hotness, hmalloc should ideally use the total *resident* size. However, although OSs such as Linux provide overall *resident set size* statistics, their standard interfaces do not provide such information in a per-object basis. To work around this limitation, hmalloc takes advantage of its synergy with Ambix, and requests per-allocation context physical memory usage from it. We describe this in more detail when we present Ambix, in Section 3.2.

3.1.2 Placement policy. We now describe the placement policy of hmalloc, which is illustrated in Figure 6. hmalloc divides the execution of the application in epochs (T_1, T_2, etc.). When a new epoch starts, each allocation context is assigned to a memory tier. A placement algorithm determines where the new objects allocated at that site will be placed during the new epoch.

To decide the target tier of each allocation context, the placement algorithm assumes that, in the new epoch, the ratio between the number of new allocations and the physical memory space they will occupy, as well as the relative hotness of their allocation contexts, will be similar to trends observed in the past.

Algorithm 1 Update Placement Policy

```
1: procedure UpdatePolicy(allocationContexts, orderedMemoryTiers)
2:     for each context in allocationContexts do
3:         context.hotness ← context.accessCount/context.totalSize
4:     sortedAllocContexts ← sortByHotnessDescending(allocationContexts)
5:     for each context in sortedAllocContexts do
6:         for each tier in orderedMemoryTiers do
7:             if context.realMemoryUsageFromAmbix < tier.freeMemory then
8:                 tier.policyAdd(context)
9:                 tier.decrease(context.realMemoryUsageFromAmbix)
10:                break
```

Based on such assumption, the placement algorithm tries to find an efficient way of distribute the to-be-allocated objects from each allocation context between the fast and slow memory tiers, while ensuring that a given memory usage ratio (betwen tiers) is preserved. This ratio is called the *capacity ratio*, and is a parameter of hmalloc.

The placement treats this problem as an instance of the knapsack problem. The knapsack represents the fast tier, and its capacity is given by multiplying the current physical memory usage of the process by the target capacity ratio. Each item to fit into the knapsack corresponds to the set of new objects that are expected to be allocated from each allocation context. Hence, the value of each item is the current hotness of the allocation context, whereas the size of each item is given by the object size of the allocation context multiplied by the number of new objects that are expected to be allocated from it. Every allocation context that is placed into the knapsack is assigned the fast tier during the next epoch. This typically contains the hottest allocation contexts. The remaining – typically colder – allocation contexts, are assigned the slow tier.

Each epoch lasts for a number of events: memory accesses sampled by PEBS or memory allocations/de-allocations, where the duration of consecutive epochs grows exponentially (N^1 events in T_1, N^2 events in T_2, and so forth). (Starting from a base of 1,000 events, scaling up to a maximum of 500,000. Beyond this threshold, the number of events is capped). The rationale behind the increasing duration of epochs is to allow faster adaptation to changing trends in memory usage patterns and object allocation in the early stages after the application is launched. As time goes by, we expect such trends to stabilize, so subsequent revisions of the tiers assigned to each allocation context may be based on longer observations. Ideally epoch size in iterative workloads should capture at least one iteration's memory access patterns, with subsequent epochs progressively refining placement decisions by leveraging increasingly comprehensive data.

3.1.3 Enforcing the first-placement decisions. After deciding on the initial placement of an object, it is placed in the appropriate memory kind. Memory kinds are custom arenas created via a non-standard API provided by jemalloc. They contains memory pages that are bound to specific NUMA node via *mbind()*. Each NUMA node represents separate memory types in the system. In this way, there is a separation of objects between hot and cold pages.

To further reduce CPU overhead, we take advantage of the prior observation [34] that most of the small allocations are short-lived and frequently accessed. Therefore, for allocations smaller than 64 bytes, we always allocate them in DRAM via faster, default jemalloc API, without counting and monitoring them.

3.2 Page-level tiering with Ambix

Ambix serves as the dynamic memory tiering component in our solution's architecture, designed to complement the hmalloc allocator. Its design philosophy is to reduce interference with the allocator's initial decisions, operating on the premise that hmalloc generally makes correct initial page placements. Unlike traditional tiering systems that aggressively migrate pages based on short-term access patterns in order to compensate for the blind first-placement guesses, Ambix adopts a less reactive approach. This design balances the benefits of dynamic tiering with the advantages of informed initial placements taken by hmalloc.

At its core, Ambix uses a page table scanning approach to evaluate memory usage, leveraging the access and dirty bits of each page table entry (PTE) to adjust a cumulative *page hotness* score. This score reflects ongoing real-time access patterns in read and write behavior, and influences whether pages are promoted or demoted between memory tiers.

Ambix employs a *delayed migration policy* that aligns with the allocator's long-term predictions. Instead of making immediate adjustments, Ambix carefully monitors and verifies the allocator's initial page placement decisions over time. It intervenes only when sustained access patterns indicate that the original placement was suboptimal, ensuring that changes are based on meaningful, long-term trends. This approach aims at preventing premature page migrations, in order to mitigate unnecessary data movement.

When Ambix migrates pages, it aims to maintain free space in the fast memory tier, ensuring the allocator can place new, potentially performance-critical pages in the fast tier. To achieve this, Ambix selectively promotes only a restrictive number of pages it deems most essential from the slow tier to the fast tier, while proactively demoting the least accessed (coldest) pages from the fast tier to the slow tier. This balanced approach efficiently utilizes high-performance memory, ensuring it remains available for future needs while avoiding unnecessary data movement and supporting the allocator's overall strategy.

3.2.1 *Runtime Metrics and Decision Making.*

Ambix collects runtime page metrics through incremental page table scans, where a fixed number of pages are periodically scanned. Each page has a score that reflects its activity, and this score is updated in each scan by examining PTE bits (e.g., accessed and dirty bits). If a page is accessed, its score increases by 2; if it is also written, the score increases by an additional 1 as writes can be particularly impactfull to performance in persistent memory devices [12]. If neither condition is met, the score decreases by 1.

The placement policy of Ambix is guided by a dynamic *partition threshold*, which borrows inspiration from Memtis[23]. Conceptually, the partition threshold defines what are *hot* and *cold* pages: pages above this threshold are considered *hot* and are suitable for fast memory, while those below are deemed *cold* and should remain in or eventually move to slow memory.

At the end of each full memory scan, Ambix recalculates this threshold based on the updated hotness scores of each page. Concretely, Ambix ranks all pages by their hotness scores and cuts off the ranking where the number of pages equals the fast memory's capacity. The updated threshold is then used in the next scan to manage page migrations.

3.2.2 *Handling just-placed Pages.*

When a page is first-placed, its initial score is set to the value of the optimal partition threshold, categorizing the page as "warm"—neither hot nor cold—allowing the system to adjust the score based on actual usage patterns before making migration decisions. This initial state ensures that future scans will gradually adjust the page's score based on its actual usage patterns. This approach ensures that newly placed pages are given time to show their true behavior before any migration decisions are made.

Additionally, Ambix tracks how many times each page has been scanned since its first placement. Pages must be scanned at least 10 times before they are eligible for migration. By requiring multiple scans before a page is considered for migration, Ambix avoids prematurely overriding the object allocator's initial decisions by delaying migration, allowing it to trust and verify placements based on sustained, consistent access patterns rather than reacting to inital short-term access patterns. As a result, Ambix can complement the object allocator's long-term strategy while still leveraging dynamic tiering to optimize memory performance.

3.2.3 *Implementation Considerations.*

Ambix is implemented as a kernel module that seamlessly integrates into the operating system. It uses kernel functions such as `walk_page_range` for page table scanning and `migrate_pages` for migrating pages between memory tiers. These functions enable Ambix to efficiently monitor and manipulate page placements without requiring significant changes to the kernel. Both the score and age are stored in the existing Linux `page_struct` without requiring changes to the kernel source code, minimizing memory overheads and improving portability. This design choice ensures compatibility across different kernel versions and hardware configurations.

3.2.4 *Interface With Hmalloc.*

Ambix provides an interface for hmalloc to receive memory usage statistics on a per-allocation context basis. The API includes functions to retrieve memory information and enable and disable monitoring of allocation contexts, which are specified as ranges of pages where each allocation context allocates its objects. For example, the `get_object_mem_info` and `get_program_mem_info` functions allow hmalloc to query the memory usage of specific objects or the entire program, respectively. This interface helps hmalloc make informed allocation decisions by leveraging the detailed memory usage data collected by Ambix (recall Section 3.1).

3.3 Limitations

hmalloc is a system that builds optimal placement logic based on past behavior. It has the best results in workloads having an iterative nature or which are doing a lot of malloc/free calls during the application's lifetime. For applications that allocate all objects at the start before they are used, hmalloc is not able to bring about improvements.

4 Evaluation

In this section, we experimentally evaluate the proposed hmalloc+Ambix system against the alternatives for page placement that are available in the latest Linux kernel.

Our evaluation aims at answering the following main questions:

Workload	Data set size	`malloc` calls	Allocation contexts	Execution time (Linux default policy)
CP2K	140 GB	9,586,283	5,283	22,453 s
LULESH	220 GB	2,270	410	5,500 s
Q-E	160GB	1,601,701	622	33,557 s

Table 1: Summary of evaluated workloads

- How effective is hmalloc in taking first placement guesses? (Section 4.2)
- What gains can hmalloc provide when combined with different tiered placement systems? (Section 4.3)
- From an end-to-end perspective, how does the proposed hmalloc+Ambix system perform when compared to relevant alternatives? (Section 4.4)

Before addressing each question above, Section 4.1 starts by detailing the experimental setup.

4.1 Experimental methodology

Workloads. We study three computational benchmarks that emphasize diverse and memory-intensive HPC applications. They are summarized in Table 1.

- The **H2O-1024 benchmark** from the CP2K suite [19] performs ab-initio molecular dynamics on liquid water using the Born-Oppenheimer approach, with a resident working set size of 140 GB.
- **LULESH** (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) models a basic shockwave propagation, representing how materials respond to extreme forces, with a 220 GB working set and a problem size of 630^3.
- The **GRIR443 benchmark** from the Quantum ESPRESSO (Q-E) suite [11] simulates complex carbon–iridium system, with a memory usage of 160 GB.

Evaluated solutions. We evaluate a full-fledged implementation of our proposed system.

We compared our system against the following set of alternative solutions:

(1) AutoNUMA in *tiering mode*.
(2) Linux default allocation policy.
(3) Each component of our proposal running in isolation: hmalloc stand-alone and Ambix stand-alone.

For hmalloc, while using an application's expected maximum memory usage to set hmalloc's memory ratio parameter can yield satisfactory results, optimal performance often requires a more fine-tuned approach. For CP2K and Q-E, which report their expected maximum memory usage at initialization, this value provides a good baseline. However, we found that analyzing the average memory consumption during execution led to more efficient ratios. In Q-E, this analysis resulted in a 2:3 DRAM to slow memory ratio, which outperformed the ratio based solely on maximum usage.

LULESH presents a more complex case. It doesn't explicitly report expected usage, but its memory consumption increases predictably with problem size. By examining its runtime behavior, we observed a maximum usage of 220 GB, but an average of only 110 GB. Given approximately 45 GB of free DRAM on our test system, this insight allowed us to fine-tune the ratio. We determined that a range of 1:1 to 1:2 (DRAM to slow memory) was appropriate, ultimately settling on 1:2 for optimal performance.

Similarly, for CP2K, deeper analysis of its memory usage patterns led us to a 3:2 ratio, which proved more effective than a ratio based on maximum usage alone.

These examples underscore the importance of considering not just peak memory requirements, but also average usage and runtime behavior when configuring hmalloc's memory ratio. This more granular approach can significantly enhance performance across various applications and problem sizes.

Ambix's setup involves several key parameters that control its memory management decisions. Ambix was configured to scan 40 GB of memory per scanning and migration round, with these rounds occurring at 1-second intervals. To prevent excessive data movement, a maximum migration limit of 500 MB per round is imposed. Additionally, Ambix was setup to maintain 5% of the faster tier memory free, providing space for high-priority allocations.

Machine. We use a dual-socket machine with Intel® Xeon® 8358 CPU, running at 2.60 GHz. Since CXL memory devices were not yet commercially available, our machine has an HMem hierarchy comprising DRAM as the fast tier and Intel Optane DC Persistent Memory as the slow tier. More precisely, each socket is populated with 4 DRAM modules (4x32 GB DDR4 3200 MT/s) and 4 DCPMM modules (4x256 GB, DDR-T 3200 MT/s Series 200), resulting in a total of 128 GB of DRAM and 1024 GB of DCPMM. DRAM size is limited to 45 GB via the *memmap* grub parameter.

We have installed a Rocky Linux distribution on our machine with kernel v5.10. The only exception are the AutoNUMA experiments, in which we use kernel v6.1 in order to take advantage of critical features made to AutoNUMA, including its tiered memory specific update. We set the *swappiness* value to 0, which effectively disables swapping in our experiments. We restrict every experiment to a single socket by using *numactl* to bind each application to exclusively execute on the cores of the second socket and place its pages in the local DRAM and DCPMM nodes. Finally, we disable transparent huge pages (THP), since both AutoNUMA's and Ambix's iplementtions do not target THPs.

4.2 How effective is hmalloc in taking first placement guesses?

As a first phase of our evaluation, we wish to study how effective the informed first-placement guesses provided by hmalloc are, when compared to the standard approach of blind guesses based on the default placement policy of Linux. This default policy determines

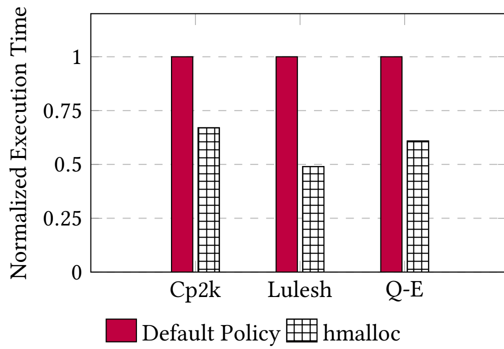

Figure 7: Performance comparison of hmalloc against default allocator, without dynamic tiering systems. Lower values indicate better performance.

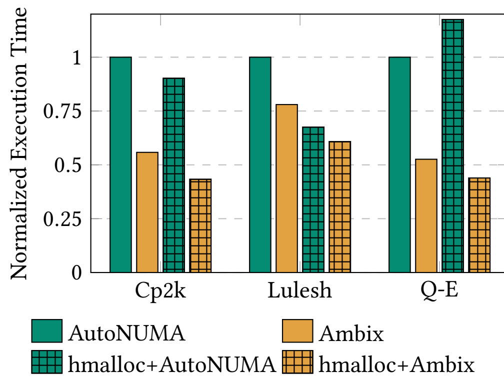

Figure 8: Performance impact of integrating hmalloc with existing dynamic tiering systems. Lower values indicate better performance.

that, once a page that is absent from physical memory is *first-touched*, it is placed on the fast tier as long as there are free page frames there; otherwise, the placement spills to the slow tier.

In these experiments, placement is merely static. Once a page is first-placed, it stays in the same memory tier until the end of the execution, even if the access patterns of the application change over time.

The performance comparison between hmalloc and the default Linux policy reveals significant advantages of informed first-placement strategies in heterogeneous memory architectures. Figure 7 presents hmalloc's performance improvements across all benchmarks, from 1.49x to 2.04x, even in a static placement scenario. hmalloc's ability to make informed placement decisions without runtime adjustments suggests its potential for even greater optimizations when combined with dynamic tiering systems.

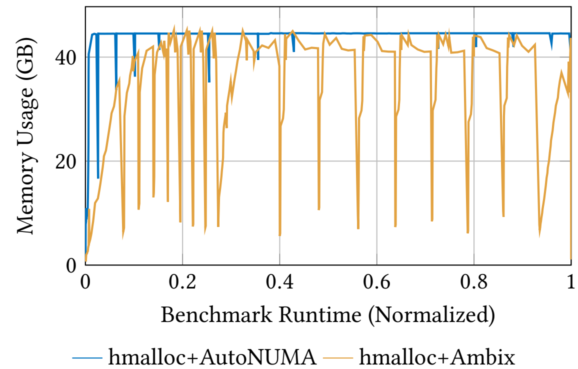

Figure 9: DRAM utilization comparison between Ambix and AutoNUMA combined with hmalloc for Q-E benchmark.

4.3 What gains can hmalloc provide when composed on top of different tiered page placement systems?

hmalloc provides substantial gains to tiered page placement systems, as shown in Figure 8. The graph compares AutoNUMA and Ambix, both standalone and combined with hmalloc.

hmalloc, when combined with Ambix, achieves the best overall performance, with a speedup up to 2.3x against standalone AutoNUMA. This highlights the synergistic effect of combining hmalloc's intelligent first placement with a tiering system that takes into account the allocator's strategies. Conversely, the combination of hmalloc and AutoNUMA shows mixed results when compared to standalone AutoNUMA. It improves performance for LULESH and Q-E, but actually performs worse for CP2K.

To understand these performance differences, we analyzed the memory usage patterns of these combinations. Figure 9 compares the DRAM usage differences that occur when using hmalloc with either Ambix or AutoNUMA. Ambix's delayed and constrained migration strategies result in significantly lower DRAM memory usage throughout the benchmark's runtime. Additionally, Ambix avoids utilizing all available DRAM, maintaining a substantial percentage of free DRAM, which further aligns with hmalloc's design.

The integration of hmalloc with dynamic tiering systems also reveals complex interactions in memory management when considering migration volumes, as can be seen in Figures 11 and 12. When combined with AutoNUMA, for CP2K and Q-E, an increase in promotion volumes is observed, likely due to the additional DRAM made available by hmalloc's placement strategy. Conversely, LULESH shows a reduction in promotion volume, which suggests that hmalloc's informed first placement strategy is particularly effective in identifying and placing hot pages in DRAM. As a result, AutoNUMA may find fewer pages that need to be promoted from the slower memory tier during runtime. A consistent trend across all benchmarks is the decrease in demotion volumes when hmalloc is used in conjunction with AutoNUMA.

Combining hmalloc with Ambix yields different results. Both CP2K and LULESH show a decrease in total migration volumes. Interestingly, Q-E exhibits a slight increase in promotion volume, but this is accompanied by a significant performance improvement

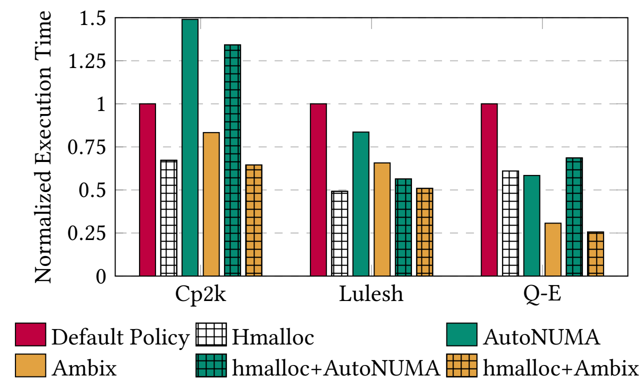

Figure 10: Performance comparison of various memory management configurations. Lower values indicate better performance.

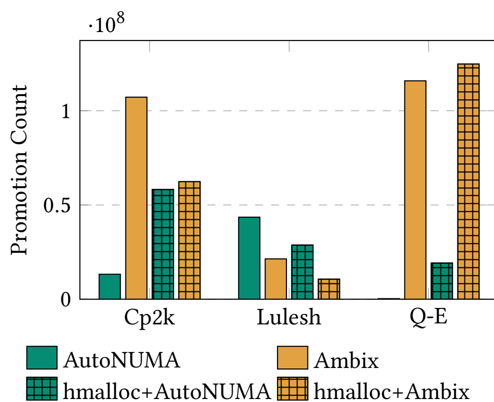

Figure 11: Promotion Volume. Lower is better.

among the tested applications. This suggests a strong collaboration between hmalloc and Ambix for Q-E, despite the slight increase in page promotions. As with the AutoNUMA integration, the combination of hmalloc and Ambix consistently reduces demotion volumes across all benchmarks.

4.4 End-to-end results

Figure 10 presents the end-to-end results for the three workloads. The combination of hmalloc and Ambix reveals nuanced performance dynamics across different workloads. In LULESH, Ambix seems to negatively impact hmalloc's performance by a slim margin. In contrast, in CP2K, it offers only a marginal improvement of approximately 4.2%. However, these results highlight a key strength of the combined solution: unlike AutoNUMA, which consistently degrades hmalloc's standalone performance, Ambix at worst has minimal negative impact. This aligns with Ambix's core principle of minimal interference when hmalloc's initial placement decisions are effective. Notably, in scenarios where Ambix does enhance hmalloc's performance, such as in Q-E with a substantial 2.4x speed-up over hmalloc standalone, the benefits are significant. This variability underscores that while smart allocators like hmalloc can often

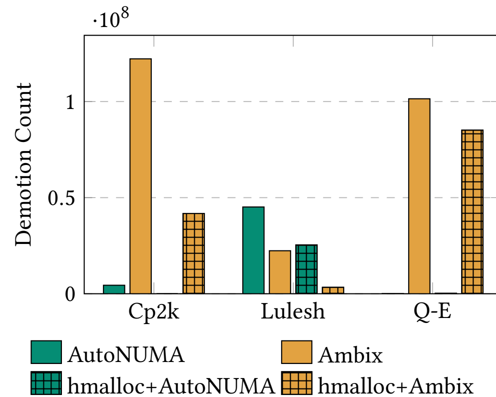

Figure 12: Demotion Volume. Lower is better.

optimize memory placement effectively, certain workloads may still benefit considerably from the assistance of dynamic tiering systems. Importantly, across all benchmarks, either the combined hmalloc+Ambix solution or hmalloc standalone consistently delivers the best performance, demonstrating the robustness of our approach in diverse memory-intensive applications. Our approach delivers up to 2.03x speedup over traditional memory management through intelligent first placement. Leveraging Ambix it further achieves up to 2.28x speedup compared to AutoNUMA.

5 Related work

The backbone of any system for tiered page placement, as introduced in Section 2, relies on some mechanism to track page accesses, whose output lets the placement policy classify which pages are hot, hence should be in the fast tier; and cold, which can be demoted to free up page frames in the fast tier. Different techniques have been proposed regarding tracking page accesses in tiered page placement systems, which range from monitoring page table access bits as in traditional page replacement algorithms (e.g. [15, 28, 36, 51]), possibly complemented with TLB miss interception (e.g. [1, 27]); to page poisoning (e.g. [16, 29]); or event-based sampling (e.g. [23, 34, 50]).

Another underlying component is the page migration mechanism. Although Linux has native support for page migration across memory nodes, some works have proposed optimized migration implementations. These improvements are achieved by reducing the page management costs associated with migration [18, 49, 51], exploiting optimized alternatives to copy page contents across memory nodes [27, 34, 51], among others.

Most of the proposals for tiered page placement in literature operate exclusively at the system level. Hence, they are restricted to monitoring and managing pages, with no knowledge of how the application uses the objects that are held in such pages. To our knowledge, there are only aware of three exceptions of systems that, similarly to our proposal, use some object-level information to guide their page placement policy.

First, Hemem [34] intercepts *malloc* calls to inspect the size of the allocation. Below a given size, the object is directly placed in a page in DRAM that is not managed by the Hemem tiering system. This decision is based on the observation that small allocations

are frequently accessed and short-lived [34]. This is an effective heuristic to handle the first-placement of small objects. However, for larger objects, Hemem must take an uninformed decision, just like the remaining systems.

A second exception is the system proposed by Kammerdiener et al. [14], which also relies on a memory allocator (bkmalloc), which provides a system-level page tiering system (MAT daemon) with allocation-specific attributes (such as allocation time or size). These attributes allow the system-level module to implement richer placement policies that can also take such object-level inputs into account. Still, these attributes are very hardly useful for the purpose of improving first-placement. For instance, all policies that the paper considers always map a newly-allocated object/page to DRAM, which boils down to the same first-placement strategy that is adopted by most of the remaining page tiering systems in literature.

A thirs exception is Unimem [48]. By default, it first-places objects in the slow tier. However, it selectively first-places those data objects with the largest amount of memory references in the fast tier. However, this method has limited applicability and is inaccurate since it ignores caching effects [48].

Finally, we remark that, in literature, there are other approaches to the problem of data placement in HMem systems. A first approach is handcrafted applications which, using programming interfaces such as libvmmalloc library from the Persistent Memory Development Kit (PMDK) [38], explicitly allocate each object on the most appropriate tier [26, 35, 46]. To try to mitigate the burden on the programmer, some proposals rely on profiling runs to gather an accurate and detailed knowledge on per-object access frequencies and, based on that, instrument the memory allocation invocations to place each object at the most appropriate memory tier [4, 7, 31, 33, 40, 47, 48]. Although this approach has proved its effectiveness in specific application domains, it has crucial shortcomings. In many use cases, it is not practical, or not even viable, to have a profile run before the application is actually up and running. Second, this approach places objects statically (on allocation time) on average access frequencies, which is ineffective in applications characterized by distinct phases with distinct access patterns, or with unpredictable workloads.

6 Conclusion

This paper proposes a novel memory tiering system that addresses the first-placement problem by combining hmalloc, an HMem-aware memory allocation library, with Ambix, a page-based memory tiering system, and exploiting their object and page-level synergies.

Our experimental evaluation when running realistic HPC workloads on a real HMem system based on DRAM with Intel Optane DC Persistent Memory demonstrates that our novel synergistic approach is effectively able to address the first-placement limitation of previous systems. Our approach delivers up to 2.03x speedup over traditional memory management through intelligent first placement. Leveraging Ambix, our custom page-level dynamic memory tiering system, it further achieves up to 2.28x speedup.

Acknowledgements

We thank the valuable feedback provided by the anonymous reviewers. We also acknowledge the support of Vasco Correia with the initial integration of the Ambix and hmalloc modules, which preceded the contributions described in this paper.

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020); the Portuguese Recovery and Resilience Plan through project C645008882-00000055 (Center for Responsible AI); and European Commission through the Horizon Europe Programme, with the Grant Agreement GAP-101189689.

References

[1] 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. https://doi.org/10.1145/3037697.3037706

[2] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Syst. J.* 5, 2 (June 1966), 78–101. https://doi.org/10.1147/sj.52.0078

[3] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. 2017. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Trans. Des. Autom. Electron. Syst.* 23, 2, Article 14 (Nov. 2017), 32 pages. https://doi.org/10.1145/3131848

[4] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/3368826.3377922

[5] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *Comput. Surveys* 56, 11 (2024), 1–37.

[6] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. *ACM SIGPLAN Notices* 48, 4 (2013), 381–394.

[7] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[8] 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. https://doi.org/10.1145/3582016.3582031

[9] T. Chad Effler, Brandon Kammerdiener, Michael R. Jantz, Saikat Sengupta, Prasad A. Kulkarni, Kshitij A. Doshi, and Terry Jones. 2019. Evaluating the effectiveness of program data features for guiding memory management. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 383–395. https://doi.org/10.1145/3357526.3357537

[10] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/3307650.3322224

[11] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. 2009. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter* 21, 39 (sep 2009), 395502. https://doi.org/10.1088/0953-8984/21/39/395502

[12] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 626–639. https://doi.org/10.14778/3436905.3436921
[13] Ying Huang. 2020. autonuma: Optimize memory placement in memory tiering system. https://lwn.net/Articles/803663/. [Online; accessed 19-November-2020].
[14] Brandon Kammerdiener, J. Zach McMichael, Michael R. Jantz, Kshitij A. Doshi, and Terry Jones. 2023. Flexible and Effective Object Tiering for Heterogeneous Memory Systems. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA, 163–175. https://doi.org/10.1145/3591195.3595277
[15] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245
[16] 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, Berkeley, CA, USA, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon
[17] Sungho Kim, Sang-Ho Hwang, and Jong Wook Kwak. 2018. Adaptive-Classification CLOCK: Page replacement policy based on read/write access pattern for hybrid DRAM and PCM main memory. *Microprocessors and Microsystems* 57 (2018), 65–75.
[18] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. *SIGPLAN Not.* 53, 2 (March 2018), 651–664. https://doi.org/10.1145/3296957.3173198
[19] Thomas D. Kühne, Marcella Iannuzzi, Mauro Del Ben, Vladimir V. Rybkin, Patrick Seewald, Frederick Stein, Teodoro Laino, Rustam Z. Khaliullin, Ole Schütt, Florian Schiffmann, Dorothea Golze, Jan Wilhelm, Sergey Chulkov, Mohammad Hossein Bani-Hashemian, Valéry Weber, Urban Borštnik, Mathieu Taillefumier, Alice Shoshana Jakobovits, Alfio Lazzaro, Hans Pabst, Tiziano Müller, Robert Schade, Manuel Guidon, Samuel Andermatt, Nico Holmberg, Gregory K. Schenter, Anna Hehn, Augustin Bussy, Fabian Belleflamme, Gloria Tabacchi, Andreas Glöß, Michael Lass, Iain Bethune, Christopher J. Mundy, Christian Plessl, Matt Watkins, Joost VandeVondele, Matthias Krack, and Jürg Hutter. 2020. CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations. *The Journal of Chemical Physics* 152, 19 (05 2020), 194103. https://doi.org/10.1063/5.0007045 arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0007045/16718133/194103_1_online.pdf
[20] 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 *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, 317–330. https://doi.org/10.1145/3297858.3304053
[21] Minho Lee, Dong Hyun Kang, Junghoon Kim, and Young Ik Eom. 2015. M-CLOCK: Migration-optimized Page Replacement Algorithm for Hybrid DRAM and PCM Memory Architecture. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. ACM, New York, NY, USA, 2001–2006. https://doi.org/10.1145/2695664.2695675
[22] S. Lee, H. Bahn, and S. H. Noh. 2014. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. *IEEE Trans. Comput.* 63, 9 (Sep. 2014), 2187–2200. https://doi.org/10.1109/TC.2013.98
[23] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. https://doi.org/10.1145/3600006.3613167
[24] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and memory placement on NUMA systems: asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 277–289.
[25] 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. https://doi.org/10.1145/3575693.3578835
[26] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021. Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 190–202. https://doi.org/10.1145/3447818.3460355
[27] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical hybrid memory management in OS for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2223–2236.
[28] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 925–937. https://doi.org/10.1109/HPCA53966.2022.00072
[29] 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. https://doi.org/10.1145/3582016.3582063
[30] Sparsh Mittal. 2016. A Survey of Cache Bypassing Techniques. *Journal of Low Power Electronics and Applications* 6, 2 (2016). https://doi.org/10.3390/jlpea6020005
[31] M. Ben Olson, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, M. Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/NAS.2018.8515694
[32] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/3357526.3357568
[33] Ivy B. Peng and Jeffrey S. Vetter. 2018. Siena: exploring the design space of heterogeneous memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 33, 14 pages.
[34] 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. https://doi.org/10.1145/3477132.3483550
[35] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing Large-Scale Plasma Simulations on Persistent Memory-Based Heterogeneous Memory with Effective Data Placement across Memory Hierarchy. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/3447818.3460356
[36] 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. https://doi.org/10.1145/3627703.3650075
[37] Reza Salkhordeh and Hossein Asadi. 2016. An operating system level data migration scheme in hybrid DRAM-NVM memory architecture. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe* (Dresden, Germany) *(DATE '16)*. EDA Consortium, San Jose, CA, USA, 936–941.
[38] Steve Scargall. 2020. *Programming Persistent Memory - A Comprehensive Guide for Developers* (1st ed.). Apress.
[39] Hyunchul Seok, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. 2011. Efficient Page Caching Algorithm with Prediction and Migration for a Hybrid Main Memory. *SIGAPP Appl. Comput. Rev.* 11, 4 (Dec. 2011), 38–48. https://doi.org/10.1145/2107756.2107760
[40] Harald Servat, Antonio J. Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 126–136. https://doi.org/10.1109/CLUSTER.2017.50
[41] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 283–297. https://doi.org/10.1145/3552326.3587449
[42] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[43] Zhiwen Sun, Zhiping Jia, Xiaojun Cai, Zhiyong Zhang, and Lei Ju. 2015. AIMR: An adaptive page management policy for hybrid memory architecture with NVM and DRAM. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, IEEE, 284–289.

[44] Yujuan Tan, Baiping Wang, Zhichao Yan, Qiuwei Deng, Xianzhang Chen, and Duo Liu. 2019. Uimigrate: Adaptive data migration for hybrid non-volatile memory systems. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, IEEE, 860–865.

[45] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3694715.3695968

[46] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 76, 19 pages. https://doi.org/10.1145/3295500.3356159

[47] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. 2018. ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 263–273. https://doi.org/10.1145/3205289.3205320

[48] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 58, 14 pages. https://doi.org/10.1145/3126908.3126923

[49] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 19–35. https://www.usenix.org/conference/osdi24/presentation/xiang

[50] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/atc24/presentation/xu-dong

[51] 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. https://doi.org/10.1145/3297858.3304024

[52] Seongdae Yu, Seongbeom Park, and Woongki Baek. 2017. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/3079079.3079092

[53] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (2020), 1–27.