HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM

Amanda Raybuck
The University of Texas at Austin

Tim Stamler
The University of Texas at Austin

Wei Zhang
Microsoft

Mattan Erez
The University of Texas at Austin

Simon Peter
The University of Texas at Austin

ABSTRACT

High-capacity non-volatile memory (NVM) is a new main memory tier. Tiered DRAM+NVM servers increase total memory capacity by up to 8×, but can diminish memory bandwidth by up to 7× and inflate latency by up to 63% if not managed well. We study existing hardware and software tiered memory management systems on the recently available Intel Optane DC NVM with big data applications and find that no existing system maximizes application performance on real NVM.

Based on our findings, we present HeMem, a tiered main memory management system designed from scratch for commercially available NVM and the big data applications that use it. HeMem manages tiered memory asynchronously, batching and amortizing memory access tracking, migration, and associated TLB synchronization overheads. HeMem monitors application memory use by sampling memory access via CPU events, rather than page tables. This allows HeMem to scale to terabytes of memory, keeping small and ephemeral data structures in fast memory, and allocating scarce, asymmetric NVM bandwidth according to access patterns. Finally, HeMem is flexible by placing per-application memory management policy at user-level. On a system with Intel Optane DC NVM, HeMem outperforms hardware, OS, and PL-based tiered memory management, providing up to 50% runtime reduction for the GAP graph processing benchmark, 13% higher throughput for TPC-C on the Silo in-memory database, 16% lower tail-latency under performance isolation for a key-value store, and up to 10× less NVM wear than the next best solution, without application modification.

SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483550

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; • **Hardware** → **Non-volatile memory**.

KEYWORDS

Operating system, Tiered memory management, Scalability

ACM Reference Format:
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 *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483550

1 INTRODUCTION

Systems with DRAM and non-volatile memory are now commercially available, such as with Intel's Optane DC non-volatile memory (NVM) modules that can share the memory interconnect with DRAM [3]. While NVM offers up to 8× higher capacity per module compared to DRAM, it comes at the cost of up to 7× lower bandwidth and up to twice the latency (cf. Table 1). Tiered main memory management systems using DRAM and NVM need to balance these trade-offs to provide high memory performance to applications.

Lag in OS support for tiered memory (e.g., Linux has no official tiered memory support beyond swapping) has led vendors to provide tiered memory in hardware [20]. This approach has the benefit that it does not require OS support to manage tiered memories. Hardware mechanisms are also nimble. For example, Intel's Optane DC "memory mode" treats DRAM as a direct-mapped cache. In this configuration, cache misses can be served without copying memory in software, tracking of page access bits, or shootdowns of translation lookaside buffers (TLBs) when access bits are reset or pages remapped.

On the other hand, hardware tiered memory also has shortcomings. It has little visibility into the high-level requirements of the applications using the memory or the structure of their data. It also cannot distinguish between different processes accessing main memory simultaneously, and it

Memory	R/W Latency	R/W GB/s	Capacity
DDR4 DRAM	82 ns	107 / 80	1×
Optane DC [3]	175 / 94 ns	32 / 11.2	8×

Table 1: Main memory technology comparison.

must rely on simple memory tracking techniques that can be efficiently implemented in hardware.

Previous research in OS [22, 39] and language-based [17] tiered memory management was evaluated with emulated NVM and, as we show, does not capture the performance characteristics of commercially available NVM. Specifically, existing OS-based systems have overheads that prevent them from scaling to the capacity of commercially available NVM. Existing systems also do not support asymmetric read/write memory performance and have limited flexibility to adapt to the diverse quality of service needs of big data applications.

We present HeMem, an OS memory management system that dynamically manages tiered memory without the CPU overhead of page access bit tracking, associated TLB shootdowns, and memory copies, but with advanced policy support for various memory access and allocation patterns, as well as performance isolation. HeMem is, to our knowledge, the first software-based tiered memory management system designed from scratch for commercially available NVM. Based on a performance study of Intel's Optane DC persistent memory modules (§2.2), we derive new insights and design principles that enable software tiered memory management to scale to the capacity and performance of real NVM. In particular, HeMem leverages these principles:

- **Asynchronous memory access sampling.** Memory access tracking methods that leverage page table access and dirty bits have high CPU overhead for scanning page tables and TLB shootdowns when bits are cleared. They do not scale to large memory capacities (§2.3). Instead, HeMem tracks memory access patterns by sampling accessed virtual memory locations that go to DRAM and NVM via appropriate CPU events. Sampling via CPU events drastically reduces overhead, by reducing the number of events processed and by eliminating the need for frequent data-intensive page table access and associated TLB shootdowns. To further offset CPU overhead, we leverage processor event-based sampling (PEBS) to asynchronously process sampled memory events in batches that the CPU logs into a buffer as applications execute. Our method minimizes CPU overhead at acceptable memory access tracking fidelity.
- **Asynchronous memory management.** OS-based memory management incurs further CPU overhead due to memory migration and associated maintenance of address translation hardware. To minimize the impact of these operations, HeMem not only tracks, but also migrates memory asynchronously, using DMA [28]. Unlike swapping to block devices, tiered memory is always available to processor loads and stores, allowing us to batch memory management tasks as applications continue execution.
- **Data scalability awareness.** Oblivious management of all data in tiered memory can decrease memory access performance due to increased management overhead. Not all data structures scale unbounded in size. In particular, many OS kernel structures and many application-level structures, such as buffers, queues, and stacks are small and often ephemeral. They best remain in DRAM.
- **Focus on asymmetric NVM bandwidth.** Table 1 shows that NVM read/write bandwidth is asymmetric and a fraction of DRAM bandwidth, while NVM access latency differs only slightly from that of DRAM. HeMem tracks write-heavy data structures and places them in DRAM if possible.
- **Flexibility.** HeMem places memory management in a user-level library, allowing it to monitor application memory access and allocation patterns with little overhead. This approach works with unmodified applications. Cloud operators and users can implement per-application memory management policies without modifying the OS.

We make the following contributions.

- We analyze the performance of Intel Optane DC when used as high capacity volatile memory (§2). Our findings inform the design principles in HeMem.
- We present the design (§3) and implementation (§4) of HeMem, a tiered main memory management system.
- We evaluate the performance of HeMem (§5) in a system with Intel Optane DC persistent memory. We compare HeMem's performance to the hardware-based tiered main memory management currently deployed for Optane DC, as well as to several recent proposals for tiered memory management that were previously evaluated only in emulation (cf. §6). In particular, we compare to Linux with nimble tiered memory management extensions [39] and to X-Mem [17], a language-based tiered memory management approach that we emulate.

Our evaluation shows that HeMem outperforms hardware, language, and OS kernel based tiered memory management. A deployment of the Silo [36] in-memory database running TPC-C [1], attains up to 13%, 82%, and 400% higher throughput with HeMem than with Intel Optane DC memory mode (MM), Linux nimble memory management (Nimble), and X-Mem, respectively. A key-value store with 700 GB working set has 14%, 15%, and 17% higher throughput with HeMem than MM, Nimble, and X-Mem, and 75% and 28% better median and 90th percentile (90p) latency than with MM. The GAP graph processing benchmark performs up to 58% and 36% faster than MM and Nimble, respectively. HeMem provides up to 16% better tail-latency for a high-priority task and reduces NVM wear by making up to 10× fewer writes to NVM than MM.

2 BACKGROUND

Modern applications have an increasing demand for memory. We characterize this demand and how high-capacity NVM may fulfill it. In particular, we examine Intel's Optane DC, the first commercially available high-capacity NVM. Finally, we investigate different ways to integrate NVM into a tiered server memory hierarchy and their challenges.

2.1 Application Memory Demands

Modern applications are increasingly data-intensive. For example, web applications access large in-memory key-value stores to build dynamic web pages [19], in-memory databases and graph processing systems scour large in-memory datasets to quickly answer analytical questions [8], and machine learning systems train on huge in-memory datasets to learn how to maximize a reward function [29]. All of these applications have large memory capacity and bandwidth requirements, and often also low memory access tail-latency requirements. These applications benefit from an increase in main memory size, but only if it can be offered without significant performance degradation [35].

The lifetime of memory allocations in these services is often bimodal and correlated with the size of the allocation [17]. Large memory ranges are allocated at application start and pre-filled from disk. These ranges exist throughout the lifetime of the application and their size tends to grow over time, as objects are added. In separate memory ranges, these services also use ephemeral memory objects that are de-allocated after a short time, such as query state.

Understanding memory range access and allocation patterns increases the tiered memory subsystem's chances to maximize memory access performance. For example, we can keep hot and short-lived data in fast DRAM while migrating cold and long-lived data to slow NVM. Tight integration with applications, such as via a library, allows memory management to leverage insights from application-level memory organization, such as memory range types, sizes, and lifetimes. Per-application memory management also operates at an adequate granularity, as only a small number of long-lived big data applications typically share a server [21, 32].

2.2 Intel Optane DC Non-volatile Memory

Intel's Optane DC NVM [3] attaches to the memory interconnect alongside DRAM. Optane NVM offers high capacity, allowing as much as 3 TB of NVM per server socket [20]. This high capacity makes Optane DC memory attractive to augment DRAM in a tiered memory configuration. Applications running on servers with Optane NVM can take advantage of a much larger pool of tiered main memory.

However, Optane and DRAM behavior can differ dramatically. Optane has asymmetric read and write bandwidth, media access granularity that is larger than a cache line, and it wears out faster than DRAM. Optane's large capacity also implies that memory translation cost plays a larger role than before. All of these factors contribute to Optane performance being much more sensitive to application memory access patterns than DRAM. To successfully integrate Optane into tiered memory, we have to understand its behavior.

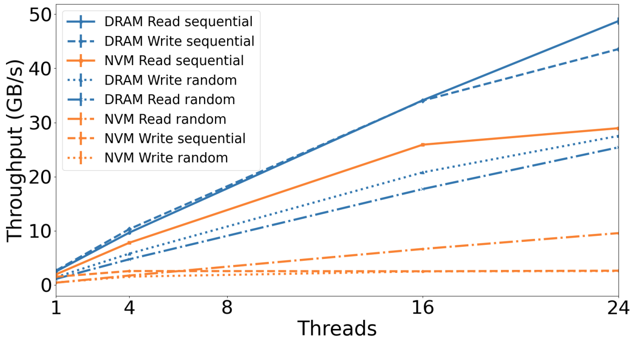
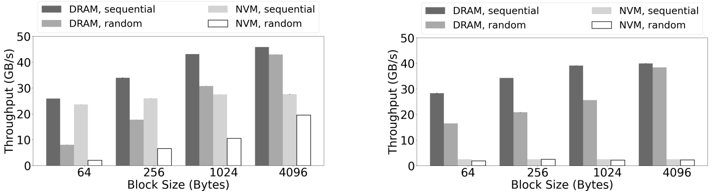


Figure 1: Memory access throughput scalability.

Optane DC performance profile. In order to understand Optane's performance and compare it to that of DRAM, we run a microbenchmark that accesses memory in either Optane or DRAM in a sequential and random pattern (testbed configuration in §5). We configure the benchmark to access memory in 256-byte blocks with regular, cached loads and stores (not SIMD or non-temporal access) that are the majority of memory accesses of our applications [21]. We then vary the number of threads performing these accesses to obtain a picture of memory throughput scalability of these applications. Results are shown in Figure 1.

DRAM bandwidth is plentiful in our test machine. Both read and write throughput scale well with the number of threads. Sequential read and write performance is higher than random due to prefetching and write-combining effects, as well as a more efficient use of DRAM row buffers. On the other hand, Optane's limited write bandwidth limits thread scalability severely. Optane write bandwidth is saturated with four threads, regardless of access pattern. At scale, DRAM random and sequential write throughput are 10.7× and 16.5× higher than Optane, respectively. Random reads from Optane can scale with more threads, but DRAM's random read throughput remains 2.7× above that of Optane. Finally, sequential Optane read throughput is even able to surpass DRAM random access throughput by 14% at scale.

We run the same microbenchmark again, but fix the number of threads at 16 and vary the size at which we access memory. Figure 2 shows the results. Sequential reads from DRAM and Optane achieve the highest throughput due to prefetching. While DRAM throughput improves with increasing access granularity, Optane read bandwidth is almost



(a) Read throughput.

(b) Write throughput.

Figure 2: DRAM and Optane throughput, 16 threads, varying access size.

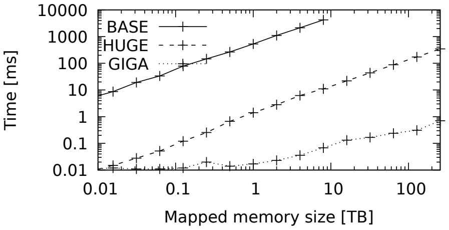


Figure 3: Page table scan time.

immediately saturated and access granularity has no effect. On the other hand, small, random reads from both types of memory suffer from low throughput. This is due to prefetching being less effective with smaller read sizes. In addition, Optane's media access granularity is 256 bytes and accesses smaller than this size suffer further overheads [20]. As the block size increases, the gap closes between sequential and random performance for both memory types. For DRAM, write throughput is similarly affected by access size, while write throughput to NVM is limited by Optane's low write bandwidth, which is saturated at 16 threads.

In summary, our microbenchmark results allow us to make the following observations about Optane performance:

- Writing to Optane is slow when compared to DRAM. Frequent writes to Optane can slow application performance due to the limited bandwidth.
- Accessing small (≤ 4KB) objects randomly on Optane is slow. To mitigate slowdown, small objects should be kept in DRAM when random access is anticipated.
- It is likely that spare memory bandwidth is available to migrate data among tiers in the background. We observe plenty of headroom in our microbenchmark (cf. Table 1). In §5, our application benchmarks confirm that spare bandwidth can indeed be used to migrate memory without affecting application performance.

2.3 NVM Impact on Memory Management

With the terabytes of memory offered to systems by NVM, traditional OS techniques for managing memory can quickly become strained. Traditionally, memory management policy decisions are made by scanning the page table for page accessed and dirty bit information. To assess the overhead of scanning page tables as memory capacity grows, we run an experiment that tracks how much time it takes to check page table accessed bits when the memory is backed by base, huge, and gigabyte-sized pages.

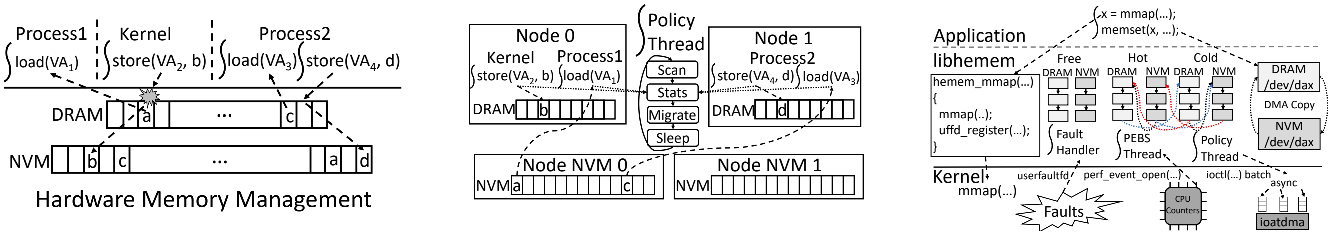
As shown in Figure 3, when memory capacity is small (up to a few 10s of GB), access bits can be checked quickly regardless of page size. However, once we reach terabytes of capacity, the time it takes to scan memory using base pages increases drastically, taking on the order of seconds to scan terabytes of memory. With smaller page sizes, not only do we have to scan more access bits, but the scan must also traverse a deeper page table. Finally, when clearing accessed and dirty bits, we also have to shoot down the TLB, further negatively impacting memory system performance. As memory capacity continues to grow, scanning page tables, even at coarser page granularity, quickly becomes untenable.

2.4 NVM as a Memory Tier

How should NVM be integrated into a tiered memory system? Intel Optane DC is exposed to the OS in one of two ways: *memory mode* (MM) and *app-direct* (AP) mode. Based on these, various memory tiering approaches exist.

Memory mode. In memory mode, DRAM serves as a cache to NVM. Current implementations use a direct-mapped cache with an effective block size that matches the processor cache line size (i.e., 64B). Software sees a single layer of a larger pool of available memory over which it has no explicit control. The tiered memory is instead managed entirely in hardware. Figure 4a shows this approach. We can see that all data is managed in tiered memory. There is no mechanism to detect hot or cold data or support for any complex tiering policy. Any accessed cache line is immediately placed in DRAM and an existing cache line must be evicted. Eviction may be due to capacity or conflict misses. While capacity misses simply occur when the cache is full, conflict misses are unique to this tiering approach, where multiple physical addresses are mapped to the same location in the DRAM cache. Conflict misses increase the number of evictions and reduce cache efficiency. With higher cache occupancy, the chance of conflict misses increases. If a cache line is modified, eviction incurs a write-back, increasing the number of random writes to NVM and reducing tiered memory performance.

Kernel-managed NUMA memory. In AP mode, NVM is exposed to the OS as a separate area of physical memory. The OS can manage this memory explicitly. A natural way to present it to applications is as kernel-managed anonymous memory. Indeed, existing work has proposed to simply integrate NVM into kernel non-uniform memory (NUMA) management [22, 39], by exposing it as a NUMA node that



(a) Intel Optane DC memory mode.

(b) Kernel-managed NUMA memory.

(c) HeMem.

Figure 4: Tiered main memory management system designs.

is at a further distance from all processors. This has the benefit of simplicity—the existing NUMA memory management system can be used and no new code needs to be written.

Figure 4b shows this approach. We can see that NUMA memory management executes asynchronously, on its own policy thread. Data is moved periodically, rather than on every access. This is necessary. Migration granularity is on the order of pages, rather than cache lines. Synchronous movement of entire memory pages would slow down execution (cf. swapping, where synchronous movement into DRAM from disk is required). The memory management policy may be complex and make use of various statistics, such as how often far memory is accessed, before making a migration decision. While it is possible to focus the NUMA policy on a subset of memory ranges, this requires explicit configuration. Finally, the NUMA topology is viewed as exclusive, where data is in one location only. This is in contrast to memory mode, where data may be in multiple locations. This is the approach Nimble [39] takes. All management tasks are executed sequentially on a single kernel thread. Hence, long-running migrations may delay scanning and statistics gathering (cf. §5). Additionally, long-running page table scans over large memory may hinder gathering accurate statistics.

File system. Finally, we can expose NVM as a file system or as a single byte or block addressable device file to applications. Applications can then map the files to memory to access them via processor loads and stores. File system or device file access has primarily been used for persistent data [2]. However, like dedicated huge page file systems (hugetlbfs [18]), this approach can be used to give applications explicit control over their NVM allocations, avoiding any transparent memory management in the OS. It allows programmers and application-specific memory managers full flexibility to decide how to divide available memory in tiers across different memory objects. Specialized systems and libraries such as PMDK [2], libmemkind [11], X-Mem [17], and others [37, 38] follow this approach. HeMem also follows this approach and we describe its design in §3.

3 HEMEM DESIGN

HeMem is a user-space manager for tiered main memory that is dynamically and transparently linked into memory-intensive applications. A high-level view of HeMem is shown in Figure 4c. HeMem intercepts memory management system calls, such as `mmap` and `madvise`, handling calls referring to anonymous memory. To handle page faults, the virtual address range of new memory maps are registered with userfaultfd [6], causing the Linux kernel to forward page faults within these ranges to a dedicated page fault thread within the HeMem process. Once a memory region is mapped, HeMem can manage it purely at user level.

HeMem's policy is informed by memory allocation patterns based on library and system calls (§3.2) and, uniquely, by access patterns gleaned from memory access sampling (§3.1). Memory migration is conducted via DMA [28] or, if not available, on a separate set of migration threads [39]. Migration is triggered in batches, by decisions of the memory management policy thread (§3.3), which runs once every 10 ms. HeMem uses three FIFO queues to track hot, cold, and free pages of each memory type (DRAM and NVM). We now describe each HeMem component in more detail, after which we discuss HeMem's unique design tradeoffs (§3.4).

3.1 Memory Access Measurement

To effectively place data in DRAM and NVM, HeMem needs an efficient mechanism to observe what memory is being accessed and how frequently. As described in §2.3, traditional page table scanning methods struggle with high-capacity main memory. Long scanning times cause the hot set to be over-estimated, as the scan will find most access bits set.

Asynchronous memory access sampling. To identify memory access patterns in a way that scales with memory size, HeMem uses a sampling based approach instead of scanning page tables. In particular, HeMem uses processor event based sampling (PEBS). With PEBS, the processor writes a record to a preallocated memory buffer after a performance counter overflows. We find a set of performance counters that allow us to identify fast and slow memory accesses and to distinguish reads from writes. We configure PEBS to sample from all loads that are served from NVM (MEM_LOAD_RETIRED.LOCAL_PMM) and from DRAM (MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM), as well as all stores (MEM_INST_RETIRED.ALL_STORES), recording the virtual memory address target for the sampled instructions. Sampling frequency is a trade-off between fidelity and overhead. Taking more frequent samples gives a more accurate view of the current application access patterns. However, sampling too frequently requires a larger PEBS buffer to avoid losing samples. We find that a sample period of approximately 5,000 memory accesses is sufficient to distinguish between hot and cold data in the applications we tested. We show a sensitivity study of the PEBS sample period in §5.1.

HeMem samples memory access via a separate *PEBS thread* that continuously reads the PEBS buffer and updates page statistics when appropriate. Using a separate thread minimizes interference from other long-running tasks such as page fault handling and memory migration. This is in contrast to Linux's Nimble tiered memory management [39]. In Nimble, the scanning of page tables and migration of memory occur on the same thread. Not only does scanning take a long time due to the number of page table entries to be scanned, it is further delayed by long-running migration operations, preventing Nimble from accurately distinguishing hot and cold data for large working sets (§5).

A trade-off exists between tracking granularity and fidelity. Finer granularities increase the likelihood that we underestimate the hot set of applications, while coarser granularities may cause us to overestimate. HeMem uses the PEBS recorded addresses to determine which regions of memory are being accessed frequently. When a sample is ready, HeMem examines the virtual address target of the sampled instruction. This allows us to track memory access at any granularity. Convenient granularities are page based and our prototype tracks accesses at huge page granularity. In contrast, the fidelity of page table based approaches is constrained by hardware page sizes and the application's chosen page size to map memory.

Data classification. The PEBS thread classifies data as hot or cold by organizing tracked pages into separate hot and cold lists for each memory type (DRAM and NVM) based on the PEBS samples. If the sample is part of a managed memory region, HeMem increments an access count for the corresponding memory page. HeMem uses separate counters for reads and writes, as identified by the sample. A page is considered hot and placed in HeMem's hot list for its memory type once a threshold number of load or store accesses are recorded to it. In our prototype, this threshold is 8 load accesses or 4 store accesses, which we determined experimentally. We show a sensitivity study of these parameters in §5.1. Lower thresholds overestimate the hot set and keep cold data in DRAM. Higher thresholds result in longer latencies to identify the hot set. A page that exceeds the store access threshold is considered a *write-heavy page.*

To ensure the freshness of HeMem's estimation of the hot set, HeMem regularly cools the pages it is tracking. Once any page accumulates a threshold of sampled accesses (18 in our prototype—determined experimentally), the access counts of each page are halved. If after this operation a page's access count is below the threshold to be considered hot, it is marked as cold and placed in HeMem's cold list for the memory type. To avoid having to traverse the FIFO queues of all tracked memory each time the threshold is reached, we perform cooling using a clock. Once any page meets the cooling threshold, we increment the clock. The next time a page is sampled, if the last time the page was cooled does not match the current clock, then the page is cooled before its access count is incremented.

3.2 HeMem Library Mechanisms

HeMem manages tiered memory in a userspace library. To do so effectively, HeMem needs to handle memory allocation, placement, migration, and page faults.

Allocation. Placing tiered memory management decisions in a library allows HeMem to efficiently obtain runtime-level information about memory use from the application with minimal overhead. For example, HeMem intercepts the mmap call to learn about heap memory range sizes, at the overhead of a function call. The hardware-implemented memory mode cannot monitor this kind of runtime information. Intercepting system calls and C standard library functions allows HeMem to focus tiered memory management on application memory ranges (in particular, heap ranges) that have the highest tendency to grow large and live long.

HeMem allocates both DRAM and NVM via DAX (direct-access) files [7]. HeMem maps per-process DAX files into virtual memory upon process startup for asynchronous migration. Upon mmap, HeMem allocates memory regions according to its policy (§3.3). HeMem then tracks the mapping from virtual address to file offset for each page it manages.

Memory migration. HeMem's memory migration policy executes periodically on a separate background *policy thread*, with a 10ms period. When HeMem decides to migrate a page, it first uses userfaultfd to mark the page as write-protected. This allows reads to proceed to the page while under migration, but any writes to the page must wait until the migration is complete. Once the migration completes, page access rights are restored. In practice, we find write pauses due to migration to be exceedingly rare, occurring on less than 0.00013% of write operations on our most write-heavy benchmark (§5.1). We set a maximum migration rate (10 GB/s) to ensure that the application is not disturbed by migration activity.

If available, HeMem offloads the data migration to an I/OAT DMA engine [28], freeing the CPU of this task. To multiplex DMA resources safely across multiple processes, we extend the Linux kernel `ioatdma` driver to expose a DMA data copy API via `ioctl` calls. A copy request consists of a source and target virtual address in user space, a size, and a set of DMA channel identifiers to use for the copy. DMA channels are (de-)allocated via additional `ioctl` calls. To minimize system call overhead, the copy `ioctl` can accept multiple copy requests in batches of up to 32. To maximize DMA performance, HeMem can use multiple available DMA channels to copy in parallel. Experimentally, we determine that a batch size of 4, using 2 DMA channels concurrently, achieves the highest DMA performance on our system. This is the configuration we use. If a DMA engine is not available, HeMem can use additional threads to copy a page in parallel, akin to Nimble [39]. We find that 4 threads maximize copy performance using this method.

Page fault handling. HeMem handles page faults using userfaultfd [6]. HeMem creates a userfaultfd file descriptor and issues ioctls on it to register managed memory with userfaultfd. When HeMem intercepts memory allocation calls, it registers the virtual address range with userfaultfd, allowing it to receive page and write-protection faults on this range. Fault events are forwarded to HeMem's page fault handling thread from the kernel. This thread reads the userfaultfd file descriptor for page fault events. In the event of a page missing fault, HeMem needs to provide a page to the application. HeMem will map a zero-filled page from either DRAM or NVM depending on its policy at the faulting address and then wake the faulting application thread. If the page fault is due to write-protection and the page is currently undergoing migration (cf. §3.3), HeMem simply waits until the migration is complete before waking the faulting thread.

3.3 HeMem Policies

HeMem's policy determines memory allocation and migration. It leverages data scalability and read/write asymmetry.

Memory allocation and placement. HeMem allocates DRAM if available by removing a page from the free list. This allows ephemeral data to remain in fast memory. Ephemeral data is hot for brief periods of time and is quickly deallocated. When running out of DRAM, HeMem simply allocates from NVM and relies on its PEBS thread to identify when pages in NVM become hot and should be migrated to DRAM. To ensure that most allocations are served from DRAM, HeMem keeps a set amount of DRAM free (we found that 1 GB is enough). When the periodic policy thread notices that this threshold is reached, it forces data migration. If no data in DRAM is cold, HeMem migrates random data to NVM until the threshold amount of DRAM is free.

HeMem keeps smaller memory allocations in DRAM (cf. X-Mem [17]). HeMem determines the size of memory ranges from intercepted memory allocation calls. If the allocation is for a small amount of memory, HeMem forwards the call to the Linux kernel to handle. Otherwise, if the allocation is for a large range of memory, HeMem decides to manage this memory itself. HeMem also tracks the growth of memory regions. If HeMem observes a region growing via small allocations, it will start to manage it once a size threshold is crossed (1 GB in our current implementation). A corollary of this policy is that small memory objects automatically remain in DRAM. NVM is accessed more efficiently at larger granularities, so keeping small ranges in DRAM yields better performance. This is in contrast to systems like X-Mem [17] and HeteroOS [22] that manage all memory by default.

Memory migration. HeMem's migration policy scans the DRAM cold list and the NVM hot list provided by the PEBS thread and migrates pages among them. The policy thread runs periodically (§3.2). Write-heavy pages are given higher priority for migration to DRAM than read-heavy pages, due to NVM having lower write than read performance.

Once a page is classified as a write-heavy page by the PEBS thread, HeMem moves it to the front of the hot list for its current memory type (DRAM or NVM). This ensures that write-heavy pages in NVM are migrated to DRAM before read-heavy pages. During cooling, if a write-heavy page is no longer considered to be write-heavy, it is moved to the hot list for its memory type. This allows it a second chance to remain in DRAM. In the event that the hot set size exceeds the DRAM capacity, HeMem does not migrate any pages, as there are no cold DRAM pages to swap for hot NVM pages.

3.4 Discussion

HeMem is intended for applications that access memory in a pattern where a hot set exists. Applications that access memory uniformly see little benefit from HeMem when their working set exceeds DRAM capacity. HeMem focuses on managing just a subset of main memory as tiered memory. We argue this is the right choice for the use case but the approach forgoes a number of features of a kernel-level memory manager. None of these features are used in the applications we evaluate. Nevertheless, we discuss these features and how we might support them.

Kernel objects in tiered memory. The user-level approach explicitly does not support kernel objects in tiered memory. This support is not necessary. Kernel objects are small and often ephemeral and should be kept in fast memory. The page cache is one possible exception. However, we believe that it warrants a separate mechanism for tiering if desired. For example, big data applications often implement their own page cache.

Program text in tiered memory. While possible, we are not interested in managing program text in tiered memory. Hence, this feature is unsupported in HeMem. Program text is static and does not grow after loading. We also have not encountered a program with a text segment large enough to benefit from tiered memory management.

Swapping. Swapping to a block device can provide an additional, slowest, memory tier. While not useful to the applications that we target, swapping of tiered memory is possible. Both fast and slow memory are backed by files and the file system can be configured with a logical volume manager to swap files in memory to disk.

Dynamic tiered memory allocation. While our prototype allocates a fixed amount of memory for tiered memory management via DAX files at system boot, it is possible to dynamically allocate memory tiers and to share their capacity with kernel anonymous memory management. It is yet unclear how Linux will integrate NVM into anonymous memory management. One possibility is via the NUMA memory subsystem [39]. HeMem can then use the appropriate NUMA calls to dynamically allocate from DRAM and NVM anonymous memory, instead of using preallocated DAX files.

Shared tiered memory. Shared tiered memory can be supported. The user links HeMem into only one process to manage the tiered memory. Any other processes that share the tiered memory automatically see the same mappings, even if they are changed by HeMem. For OSes that do not reflect updated mappings across all processes sharing memory, another library can be linked to synchronize the updated mappings cooperatively. Since processes sharing memory already cooperate, this procedure has no effect on security.

Global tiered memory management. HeMem manages per-process pools of tiered memory. This is adequate for big data applications, where few processes operate on large, private working sets. If global dynamic memory management is desired, a userspace HeMem daemon can coordinate per-process HeMem instances. Processes would request memory from the HeMem daemon, which manages the global pool, attaches to each processes' userfaultfd and PEBS buffers, and migrates memory on behalf of these processes.

4 IMPLEMENTATION

We implement HeMem as a user-level library on top of Linux version 5.1.0, with two userfaultfd patches applied. Firstly, we make use of an existing kernel patch to userfaultfd [6] that implements support for write-protection faults in addition to page missing faults [4]. Write-protection faults are already part of the userfaultfd system call interface, but support is not in the mainline kernel as of yet. Furthermore, we develop and apply our own patch to userfaultfd to support user-space page fault handling on memory backed by DAX files. This involves modifications to the DAX device page fault handling path to forward page fault and write protection events for userfaultfd-registered address ranges via userfaultfd rather than handling them in the kernel. Overall, we add 1,337 lines of code to the Linux kernel userfaultfd module, memory management code, and DAX handling code.

We use NVM in App-Direct mode and expose it via a DAX file to user-space. Our prototype also manages DRAM via a DAX file, which we reserve via the kernel `memmap` command-line argument. Dynamically sharing DRAM with the existing kernel memory manager is possible (§3.4), for example via pinned anonymous mappings, but requires additional userfaultfd support for these mappings. Userfaultfd is under active development in the Linux kernel to support an increasing number of memory mapping types.

We implement libHeMem in 4,177 lines of C code. It is linked into applications via the `LD_PRELOAD` dynamic link mechanism to transparently enable tiered memory management. Memory allocation calls from the application are intercepted by libHeMem using libsyscall_intercept [5].

5 EVALUATION

We first break down HeMem's performance with a number of evaluations based on the GUPS [9] microbenchmark. We then evaluate how HeMem affects application-level performance using a number of big data applications, such as the Silo in-memory database, a key-value store, and a graph processing system. We compare HeMem's behavior and performance to that of Intel Optane DC memory mode (hardware tiered memory management), as well as the Linux Nimble tiered memory management system [39] (Nimble). Where appropriate, we also compare to X-Mem [17], which we emulate. Nimble and HeMem use background threads for memory access measurement and policy. In addition, Nimble uses threads for memory migration and we configure 4 threads, which is most efficient.

Our evaluation answers the following questions:

- How do each of HeMem's design principles (§1) contribute to HeMem's performance and how does performance compare to the other tiered memory management implementations (§2.4)? We evaluate various hot set sizes for the GUPS microbenchmark, and various dataset sizes for GUPS with a uniformly random access pattern. We also evaluate how well hot sets are tracked, and break down tracking overheads and parameters. Finally, we break down the benefits of each of HeMem's policy decisions (§5.1).
- How can various big data applications benefit from HeMem compared to other approaches, in terms of runtime, operation and transaction throughput, and latency (§5.2)? How well is application performance isolated (§5.2.2)?

- What is the impact of each tiered memory management system on NVM device wear (§5.2.3)?

Evaluation platform. We run our evaluation on a single socket of a dual-socket Intel Cascade Lake-SP system running at 2.2GHz with 24 cores per socket and a 100 GbE ConnectX-5 Mellanox NIC. Each socket has 192 GB of DDR4 DRAM and 768 GB of Intel Optane DC NVM. To leverage all 6 memory channels, there are 6 DIMMs of DRAM and NVM per socket. The machine runs Debian 10.9 with Linux kernel version 5.1.0rc4. NUMA effects of tiered memory are beyond the scope of this paper. Hence, we run each benchmark pinned to a single NUMA node. All evaluated tiered memory management systems support this mode of operation.

Overhead of userfaultfd. We evaluate the overhead of using userfaultfd for tiered memory management. We found that, while userfaultfd can slow down applications with frequent page faults, this is not a factor in our benchmarks. Memory-intensive applications opt for large-capacity memories precisely to avoid page faults during runtime due to their high overhead. HeMem's system and library call interception incurs negligible overhead.

5.1 Microbenchmarks

We use GUPS [9] as a microbenchmark to evaluate the behavior of the different tiered memory management systems. GUPS executes parallel read-modify-write operations to fixed size objects in a uniform or skewed random pattern in its working set and measures the giga update operations per second (GUPS) it performs. After a warm-up round, we run each benchmark 3 times and report the average GUPS. GUPS uses a configurable number of threads, working set and object sizes, and memory access skew. Each thread has its own exclusive working set partition that it accesses without synchronization. Unless otherwise noted, we run GUPS with 16 threads, each performing 1 billion updates (16 billion updates in aggregate) to 8-byte objects. Using 16 threads allows us to compare systems without overloading the 24-core socket. Nimble, in particular, uses several memory migration threads and suffers with more GUPS threads. We compare HeMem and MM explicitly with more threads.

Uniform random access (system overhead). We first investigate the overhead of the tiered memory management systems when no memory migration is required. To do so, we run GUPS using uniform random memory access over different working set sizes, up to 256 GB. In this case, tiered memory management should provide performance close to DRAM when the working set fits in DRAM, and close to NVM otherwise. To run GUPS in NVM, we modify `mmap` to map memory from the NVM DAX file. This configuration emulates X-Mem [17], which places large heap data structures with random memory access into NVM.

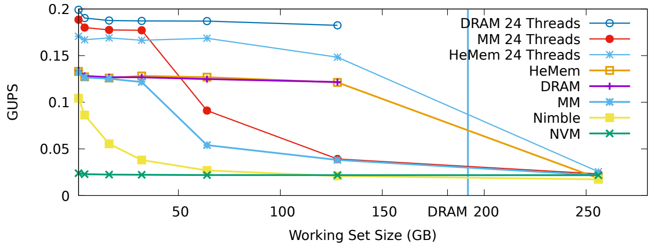


Figure 5: Uniform GUPS (higher is better).

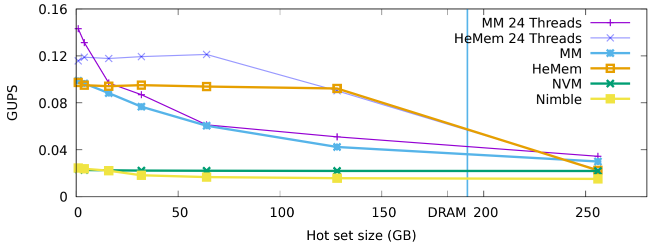


Figure 6: GUPS with different hot set sizes (512 GB working set; higher is better).

Figure 5 shows that tiered memory system performance varies. When the working set is small (≤ 32 GB), HeMem and memory mode (MM) perform nearly identically to DRAM. In this case, both systems are able to keep the dataset in DRAM and incur no NVM access. As the working set size approaches the size of DRAM (vertical line), MM performance degrades due to increased conflict misses to the DRAM cache, which do not exist for HeMem. We verified this behavior by observing NVM access performance counters for both systems. With a working set size of 128 GB, HeMem provides 3.2× higher GUPS than MM. Nimble provides a maximum of 78% of the GUPS of MM and HeMem and degrades as the working set size becomes larger. This shows Nimble's scanning and migration mechanism overhead. Once the working set exceeds DRAM capacity, all systems provide performance equivalent to GUPS in NVM. Finally, running with 24 threads improves performance for MM and HeMem, but the additional GUPS threads now contend with HeMem for limited cores. For small working sets (≤ 32 GB), HeMem sees 10% lower throughput than MM. On larger working set sizes, we see a similar trend as with the 16 thread case. As the working set size approaches the size of DRAM, HeMem can provide up to 3.7× the performance of MM.

Hot set. We now make a random set of each thread's objects *hot* (frequently accessed): 90% of the operations of each thread uniformly access its hot objects, while the remaining 10% of operations uniformly access the thread's entire working set partition. This benchmark measures how well each memory management system identifies the hot set to keep it

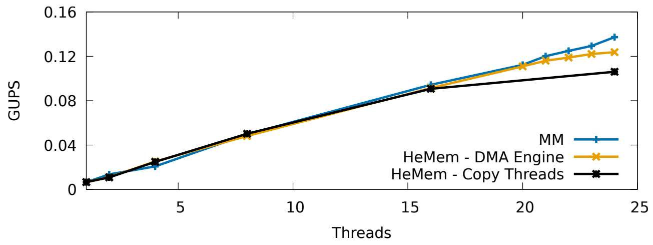


Figure 7: GUPS scalability (512 GB working set, 16 GB hot; higher is better).

in DRAM. We fix the aggregate GUPS working set to 512 GB and vary the size of the aggregate hot set from 1 to 256 GB. In this scenario, approximately 10% of accesses must go to slow NVM and therefore overall performance is 27% lower than uniform random GUPS (Figure 5), when the working set fits in DRAM.

Beyond this, our results in Figure 6 show a similar pattern. As long as the hot set fits into DRAM, HeMem identifies it and ensures that it remains in DRAM. HeMem occasionally migrates cold data between DRAM and NVM, incurring minimal overhead. MM performance suffers as the GUPS hot set size approaches the capacity of DRAM. As the hot set grows, MM's direct-mapped caching approach exhibits more misses and more of the hot data is being pushed to NVM. HeMem performs up to 2× better with increasing hot set size. As before, running with 24 threads improves GUPS performance, but contends with HeMem for limited CPU resources. In this case, for hot set sizes of less than 8GB, MM performs up to 24% better than HeMem. However, the same pattern emerges as the hot set size grows, and HeMem performs up to 2× better than MM for hot sets larger than 8GB. Nimble suffers from high overhead due to sequential scan and migration. Even when the hot set fits in DRAM, Nimble achieves only 25% of the GUPS of MM. When the hot set does not fit in DRAM, the performance of all configurations converges. HeMem identifies this case and stops migration.

Dynamic hot set. How quickly can memory management react to changes in the hot set? We configure GUPS to run with 16 threads, a 512 GB working set, and a 16 GB non-consecutive hot set. After warm-up, we allow each thread to run for 150 seconds. We then change the hot set so that 4 GB of the original hot set becomes cold and 4 GB of the original cold set becomes hot.

We plot the instantaneous GUPS over time in Figure 9. For all systems but HeMem-PT-Async, which is HeMem configured to do asynchronous page table scans rather than PEBS scans, GUPS are initially high as the hot set is entirely in DRAM after warm-up. Due to scanning overhead, HeMem-PT-Async is unable to track the hot set. Once the hot set shifts, there is a noticeable drop in GUPS for all systems, as newly hot data is initially accessed from NVM. All systems now migrate data among tiers. Both HeMem and MM identify and move the newly hot data into DRAM and recover GUPS within 20 seconds. MM's cache-line sized migrations are lighter-weight than HeMem's and maintain higher performance during migration. HeMem-PT-Async takes longer to recognize the new hot set and does not recover GUPS, remaining at 54% the throughput of HeMem and MM. This demonstrates that page table scanning indeed has high overhead for large memory sizes.

System	GUPS	×
Nimble	0.020	0.36
MM	0.048	0.86
HeMem	0.056	1

Table 2: GUPS write skew.

Scalability. HeMem uses threads to handle page faults, read PEBS buffers, and make migration policy decisions. To understand how HeMem affects application performance at high thread counts, we run the dynamic hot set experiment with different thread counts and report the average GUPS.

Results are shown in Figure 7. At low thread counts, both HeMem and MM scale well. At 21 threads, however, performance of HeMem and MM diverge. MM is a pure hardware approach and consumes no cores for background work. HeMem's background threads contend with GUPS threads, lowering GUPS by 10% compared to MM. To show HeMem performance without a DMA engine, we configure HeMem to run with 4 migration threads to copy pages. This configuration lowers GUPS throughput by 23% and 14% versus MM and HeMem with DMA copies, respectively.

Asymmetric access pattern. To measure how well each tiered memory management system performs with an asymmetric read/write access pattern, we configure GUPS with a skewed read/write pattern—of a 256 GB hot set out of a 512 GB working set, 128 GB are write-only, while the remainder of the working set is read-only. As before, 90% of accesses go to the hot set. Table 2 shows the results (× column shows performance normalized to HeMem). HeMem is able to recognize the write-only portion of the memory and ensure it remains in DRAM, avoiding the limited write bandwidth of NVM. MM and Nimble, which are blind to the read/write skew, perform 14% and 64% worse than HeMem, respectively.

HeMem overheads. To understand the overheads of our design, we run GUPS with 16 threads, a 512 GB working set and a 16 GB hot set and selectively enable design contributions. Results are shown in Figure 8. *Opt* manually places the hot set into DRAM and keeps it there during the entire GUPS duration without scanning or migrations. *PEBS* enables the PEBS scanning thread, but keeps migrations disabled, showing negligible overhead versus *Opt*. *PT Scan* uses page table

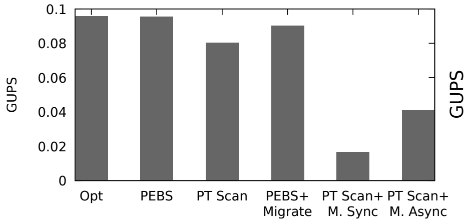


Figure 8: HeMem overheads.

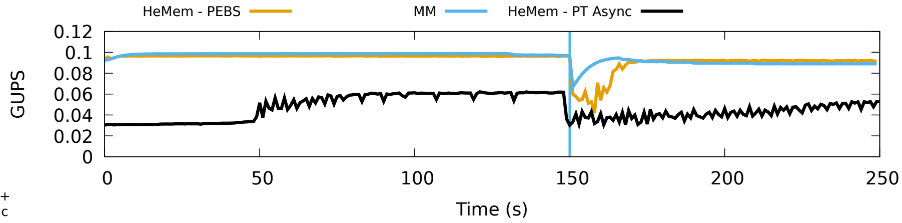


Figure 9: Instantaneous GUPS (higher is better).

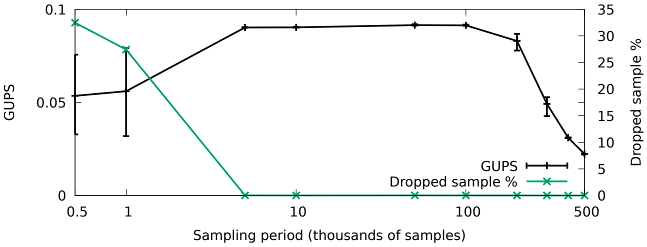


Figure 10: PEBS sampling period sensitivity (512 GB working set, 16 GB hot; higher is better; log x axis).

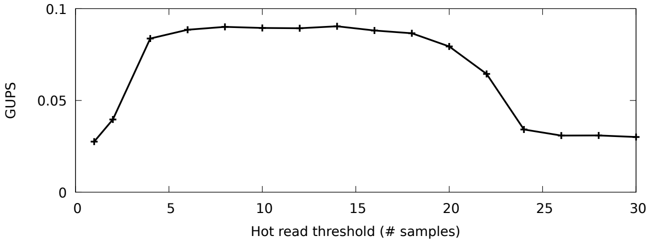


Figure 11: Hot memory read threshold sensitivity (512 GB working set, 16 GB hot; higher is better).

accessed/dirty bit scanning/clearing instead of PEBS. Due to TLB shootdowns, *PT Scan* reduces throughput by 18% versus *PEBS*. *PEBS + Migrate* enables the policy and cooling threads and does not manually place the hot set into DRAM, so migrations do occur. The GUPS throughput under this configuration is within 5.9% of *Opt*, showing that HeMem is able to identify and migrate the hot set with low overhead.

To show the fidelity difference between PEBS scanning and page table based scanning, we again replace HeMem's PEBS scanning with page table based scanning. We investigate both a synchronous (*PT Scan + M. Sync*) and asynchronous scanning method (*PT Scan + M. Async*). In *M. Sync*, a single migration policy thread performs the page table scan and migrations sequentially (cf. Figure 4b). In *M. Async*, we introduce a separate page table scanning thread. *M. Sync* achieves only 18% of the throughput of *Opt*, as page table scanning is delayed by long-running migrations, increasing the chance of hot set overestimation. Indeed, *M. Sync* considers nearly all of the 512 GB working set to be hot. *M. Async* increases GUPS by over 2× compared to *M. Sync*, but still only achieves 43% of *Opt*. The overheads of scanning the page tables for 512 GB of memory and the necessary TLB synchronization mean that even the asynchronous configuration overestimates the hot set, considering up to 300 GB hot, causing extraneous NVM accesses and migrations.

HeMem parameters. To understand how sensitive HeMem is to its various parameters, we perform a sensitivity study. We run the GUPS microbenchmark with 16 threads, a 512 GB working set, and a 16 GB hot set and vary the PEBS sampling period. Figure 10 shows these results. The error bars show the maximum and minimum measured GUPS over three runs. At low sample periods, variance in performance is high. This is due to the PEBS thread being unable to keep up with the samples generated. Up to 30% of samples are dropped because they are generated faster than the PEBS thread can read them. As the sampling period increases, the performance variance decreases and fewer samples are dropped. Sampling periods between 5k and 100k provide good performance while dropping fewer than 0.02% of samples. Sampling periods above 100k show low performance because samples are not collected frequently enough.

Next, we fix the sampling period at 5k and vary the hot memory read threshold. Figure 11 shows these results. The hot write threshold is kept at half the hot read threshold. At low thresholds, the hot set is overestimated and performance suffers. Thresholds between 6 and 20 accesses result in high GUPS as these values are sensitive enough to distinguish between hot and cold pages. Thresholds higher than 20 result in lower GUPS as the hot set is underestimated due to hot pages requiring more accesses to be considered hot.

Finally, we study the sensitivity of HeMem's memory cooling threshold. To do so, we measure instantaneous GUPS after a warm-up period. After 150 seconds, we shift the hot set so that 4 GB of the original hot set become cold and 4 GB of the original cold set become hot. We vary the memory cooling threshold and observe the results in Figure 12. When the cooling threshold is equal to the hot threshold (8), HeMem underestimates the hot set because it is cooling too aggressively, resulting in low GUPS. Higher cooling thresholds result in less aggressive cooling and quicker adaptation

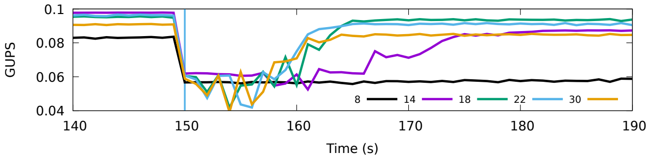


Figure 12: Memory cooling threshold sensitivity.

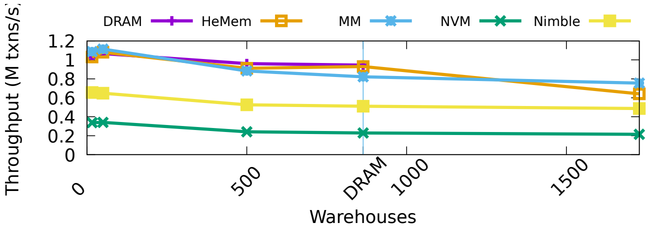


Figure 13: Silo TPC-C warehouse scalability (higher is better).

to hot set changes, as newly hot pages can meet the hot threshold before being cooled. Cooling thresholds that are too high (30) result in too many pages being considered hot. These pages compete for DRAM space, reducing GUPS.

5.2 Application Benchmarks

We now evaluate how various big data applications perform using each tiered memory system. We evaluate transactional throughput and latency of the TPC-C benchmark in the Silo in-memory database (§5.2.1) with various dataset sizes. We also evaluate operational throughput and latency of an in-memory key-value store with a skewed access distribution and with priorities (§5.2.2). Finally, we evaluate the runtime of the GAP graph processing benchmark (§5.2.3).

5.2.1 Silo. We measure the performance of Silo [36], an in-memory transactional database, under the various tiered memory systems. We run TPC-C [1], which models a retail outfit, simulating customers making orders from a number of warehouses. Most orders are fulfilled from local warehouses, but a small number of transactions involve remote warehouses. The resulting access pattern is random with little read and write reuse [16].

To scale working set size, we run Silo with 16 threads and vary the number of warehouses from 16–1,728 and measure the average throughput; 864 is the maximum number of warehouses that fit in DRAM. Results are shown in Figure 13. When the working set fits in DRAM, HeMem is able to achieve up to 13% and 82% higher throughput than MM and Nimble, respectively. Conversely, when the number of warehouses no longer fits in DRAM, MM outperforms HeMem by 17%. When placing Silo's working set in NVM (cf. X-Mem), performance degrades to 32% that of HeMem and MM. This result matches the GUPS microbenchmark analysis with a uniform random memory access pattern. We see similar behavior when investigating transaction latency.

	16GB	128GB	700GB	50p	90p	99p	99.9p
MM	1.09	1.03	0.93	35	44	53	63
HeMem	1.14	1.11	1.06	20	26	34	49
Nimble	1.07	1.05	0.92	-	-	-	-
NVM	0.91	0.91	0.90	-	-	-	-

Table 3: FlexKVS throughput (Mops/s) & latency (μs).

	Priority			Regular		
μs	50p	99p	99.9p	50p	99p	99.9p
HeMem	86	239	341	146	318	409
MM	127	278	342	156	310	380
%	47	16	0	6	-2	-8

Table 4: FlexKVS latency with priority.

5.2.2 Key-value store. We evaluate the FlexKVS [24] scalable key-value store with tiered memory. FlexKVS is Memcached [10] compatible, but uses a segmented log [33] for items to reduce synchronization overheads, as well as a block chain hash table [30] to minimize cache coherence overhead on item lookup. We evaluate the throughput of FlexKVS with a server running 8 threads and 1 client machine running 48 threads. We use 4 KB value sizes and a varying number of key-value pairs to achieve different working set sizes. Each client performs 90% GET requests and 10% SET requests [13], while 20% of the keys are in a hot set. The hot keys are accessed 90% of the time.

The results can be seen in Table 3. All systems perform similarly when the working set fits in DRAM (≤ 128 GB). HeMem and Nimble provide 7% and 2% higher throughput than MM, respectively. When the working set size exceeds DRAM capacity (700 GB), HeMem outperforms MM and Nimble by 14% and 15%, respectively. In this case, the hot set (140 GB) still fits in DRAM. Placing key-value pairs into NVM (X-Mem) results in 18% lower throughput than HeMem. We also evaluated a uniform random workload, which performs similar to GUPS uniform random (§5.1).

Latency. We evaluate FlexKVS latency in the 700 GB case, by running the server at 30% load using the TAS [25] low-latency network stack on server and clients. HeMem provides up to 16% lower median and up to 69% lower tail latency than MM. Nimble crashes TAS, so we omit this result.

Priority. Using the Linux TCP stack on the clients and the server, we run two FlexKVS instances, one with a higher priority. The non-prioritized instance serves 2 clients with 48 threads each, with a working set of 500GB, accessed uniformly at random. The prioritized instance serves a single client, also with 48 threads, with a working set of 16GB (the priority set). For HeMem, we configure the priority instance to keep all key-value pairs in DRAM, while the non-prioritized instance leverages both memory tiers. Table 4

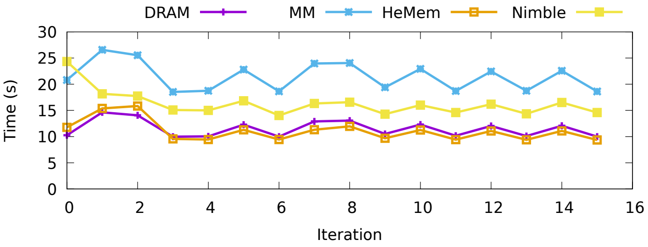


Figure 14: BC on 2^{28} vertices (lower is better).

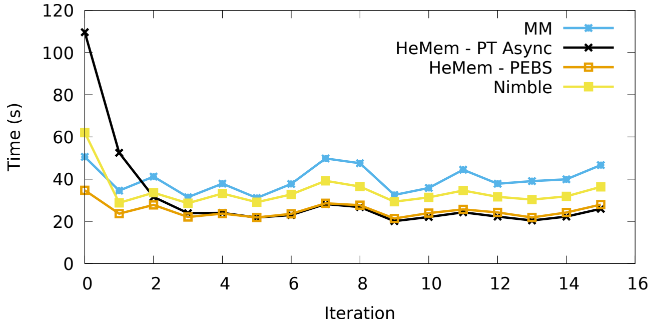


Figure 15: BC on 2^{29} vertices (lower is better).

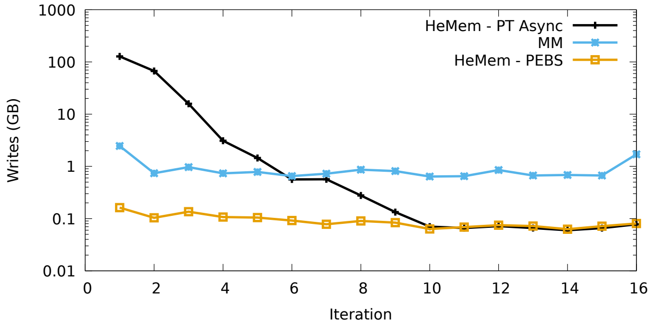


Figure 16: NVM writes, BC on 2^{29} vertices. Log scale.

shows that HeMem provides up to 47% better latency than MM from the priority instance, without tangible negative impact on the non-priority instance. This is expected, as MM does not support prioritization.

5.2.3 Graph processing. We measure the graph processing performance of HeMem with an application of the GAP benchmark suite [15]. GAP generates a Kronecker power-law graph (average degree 16). Power-law graphs have locality, as vertex traversal frequency increases with degree [14]. We run 15 iterations of the betweenness centrality (BC) algorithm on graphs with 2^{28} (fits in DRAM) and 2^{29} vertices (exceeds DRAM). BC determines the number of shortest paths between any pair of vertices through a given vertex, which we choose randomly on each iteration.

When the graph fits in DRAM (Figure 14), HeMem outperforms MM by an average of 93%. HeMem keeps all data structures in DRAM, while MM suffers conflict misses, incurring expensive NVM access. The BC algorithm accesses the graph using small accesses, which incur extra overhead on NVM due to NVM's larger media access granularity and the BC data structures are write intensive, so NVM access is very costly. When the entire graph is stored in NVM, performance is 16× worse than any system and we omit this result from the graph. HeMem has very low overhead when compared to DRAM-only performance due to its sampling mechanism. Nimble's high scanning and migration overhead causes up to 47% higher runtime than HeMem, but it is able to outperform MM by 32%. Neighbors to vertices are likely located on the same memory page. Page-based migration approaches, such as HeMem and Nimble are able to exploit this locality, outperforming MM.

When the graph exceeds DRAM (Figure 15), HeMem is able to identify hot and written data and migrate it to DRAM. Using lightweight PEBS sampling, HeMem quickly identifies the hottest portions of the graph compared to HeMem with page table scanning (HeMem - PT Async). Page table scanning overestimates the hot parts of the graph and, as a result, takes more time to migrate them to DRAM. Extra migrations from the page table scanning approach slow down the first few iterations of the BC algorithm by up to 3× compared to PEBS-based scanning. Once the page table scanning approach has migrated the hot parts of the graph to DRAM, it has identical performance to the PEBS-based scanning configuration. Nimble's sequential scanning and migration approach degrades performance by an average of 36% versus that of HeMem. Both HeMem and Nimble outperform MM by 58% and 16%, respectively, because they exploit locality. When the entire graph structure is stored in NVM, performance is 17× worse than any system and we omit this result from the graph.

NVM device wear. To evaluate the effect of tiered memory management on NVM device wear, we measure the number of writes to NVM while running BC on a graph of 2^{29} vertices. Results can be seen in Figure 16. MM makes a constant number of writes to NVM throughout each iteration of the BC algorithm. Only a few vertices are write-hot and HeMem-PEBS identifies them quickly and migrates them to DRAM, making 10× fewer NVM writes than MM on each iteration. HeMem in page table scanning configuration makes three orders of magnitude more NVM writes than with PEBS during early iterations, while it is in the process of identifying and migrating the write-hot set of the graph data structures to DRAM. Once this has occurred, HeMem-PT Async and PEBS make the same number of NVM writes.

6 RELATED WORK

Tiered NVM systems. HeteroOS [22] is an OS/VMM-level heterogeneous memory manager. HeteroOS coordinates memory placement and migration with guest OSes. Nimble [39] focuses on fast kernel huge page migration, extending Linux' NUMA migration mechanisms. Thermostat [12] identifies hot memory by page table sampling. These systems evaluate emulated NVM. Unfortunately, emulation is unable to capture the real characteristics of NVM performance [20]. In contrast, HeMem is an application-level memory manager that does not rely on virtualization techniques, programmer hints, or page table sampling. HeMem is built for real NVM.

Libraries, like libmemkind [11] and PMDK [2], expose NVM directly to applications. Programmers must determine for themselves which data structures are best served from DRAM or NVM and use specialized memory allocation interfaces from these DRAM and NVM pools. Unimem [38], X-Mem [17], and 2PP [37] are runtime systems that can automatically determine the best data placement using a special memory allocation API to identify important memory objects and an extra profiling step to collect various statistics, such as memory access frequencies, access patterns, and read/write ratios. In contrast, HeMem uses traditional memory allocation interfaces, like `mmap`, to automatically place application data in DRAM and NVM without an extra profiling step and requires no code changes to work with existing applications.

KLOCs [23] allow kernel objects to be tiered along with application objects. HeMem focuses on application memory only and does not tier kernel objects. Kernel objects are often small and ephemeral, so HeMem leaves them in DRAM. OSim [31] discusses the problems that arise with memory management algorithms that rely on periodic scanning of the entire address space, such as access bit scanning for page replacement or huge page compaction as memory capacity increases. Ingens [26] improves upon Linux's transparent huge page management and attempts to preserve memory contiguity in a fair and efficient manner. To reduce memory access latency, Ingens operates in the background. HeMem builds on this work and also operates asynchronously in the background.

Remote memory systems. Disaggregated memory can be used to provide additional main memory capacity. Similar to NVM, disaggregated memory systems require that the most frequently accessed application data remains in local memory. Software-defined far memory [27] considers warehouse-scale computers, where remote memory constitutes a "far memory" tier. Memory pages that have not been accessed within a period of time are considered cold and are compressed and swapped to remote memory. A learning-based auto-tuner adapts the system parameters to more accurately classify the cold memory pages based on application behavior across the entire datacenter. HeMem also classifies hot and cold memory pages based on access frequencies, but considers the slow memory tier to be NVM rather than remote memory. Additionally, the authors target servers that run a large variety of smaller workloads whereas HeMem targets systems where only a few applications that all require a large amount of memory are running at a time.

AIFM [34] allows swapping of application-level memory objects to far memory at object granularity, using a specialized API and runtime. HeMem migrates memory to and from NVM at page rather than at object granularity.

7 CONCLUSION

HeMem is, to our knowledge, the first software-based tiered memory management system designed from scratch for commercially available NVM. HeMem dynamically manages tiered memory without the CPU overhead of page access bit tracking, associated TLB shootdowns, and memory copies, but with advanced policy support for various memory access and allocation patterns and priorities. On a system with Intel Optane DC, HeMem outperforms hardware, Linux Nimble, and X-Mem tiered memory management, providing up to 13% higher throughput, 16% lower latency, 16% lower tail-latency under performance isolation, and up to 10× less NVM wear when processing large datasets using a key-value store, the Silo database, and the GAP graph processing benchmark.

Acknowledgments. For their insights and valuable comments, we thank the anonymous reviewers and our shepherd, Steven Hand. We acknowledge funding from NSF grants CNS-2008884 and CNS-1719061, as well as from Huawei.

REFERENCES

[1] 2010. TPC-C benchmark (Revision 5.11). http://www.tpc.org/tpcc/.

[2] 2017. Persistent Memory Programming. http://pmem.io/.

[3] 2019. Intel Optane DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory.

[4] 2019. userfaultfd: write protection support. https://patchwork.kernel.org/cover/11005675/.

[5] 2020. syscall_intercept. https://github.com/pmem/syscall_intercept.

[6] 2020. userfaultfd(2). http://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[7] 2021. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[8] 2021. The Graph500 Benchmark. http://www.graph500.org/.

[9] 2021. GUPS (Giga Updates Per Second). http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[10] 2021. Memcached. http://www.memcached.org/.

[11] 2021. Memkind. https://memkind.github.io/memkind/.

[12] 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 (ASPLOS '17)*. Association for Computing Machinery. https://doi.org/10.1145/3037697.3037706

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. Association for Computing Machinery. http://doi.acm.org/10.1145/2254756.2254766

[14] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC '15)*. IEEE Computer Society. https://doi.org/10.1109/IISWC.2015.12

[15] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[16] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Rec.* 39, 3 (Feb. 2011), 5–10. https://doi.org/10.1145/1942776.1942778

[17] 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 (EuroSys '16)*. Association for Computing Machinery. https://doi.org/10.1145/2901318.2901344

[18] Mel Gorman. 2010. Huge pages part 2: Interfaces. https://lwn.net/Articles/375096/.

[19] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522722

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://arxiv.org/abs/1903.05714v2.

[21] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery. https://doi.org/10.1145/2749469.2750392

[22] 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 (ISCA '17)*. Association for Computing Machinery. https://doi.org/10.1145/3079856.3080245

[23] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery. https://doi.org/10.1145/3445814.3446745

[24] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery. https://doi.org/10.1145/2872362.2872367

[25] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery. https://doi.org/10.1145/3302424.3303985

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association. http://dl.acm.org/citation.cfm?id=3026877.3026931

[27] 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 (ASPLOS '19)*. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304053

[28] Thai Le, Jonathan Stern, and Stephen Briscoe. 2017. Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[29] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[30] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[31] Mark Mansi and Michael M. Swift. 2020. OSim: Preparing System Software for a World with Terabyte-Scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378451

[32] Jason Mars and Lingjia Tang. 2013. Whare-Map: Heterogeneity in "Homogeneous" Warehouse-Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery. https://doi.org/10.1145/2485922.2485975

[33] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. Association for Computing Machinery. https://doi.org/10.1145/121132.121137

[34] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20) (OSDI '20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/ruan

[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association. https://www.usenix.org/conference/osdi18/presentation/shan

[36] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522713

[37] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society. https://doi.org/10.1109/PACT.2015.10

[38] 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 (SC '17)*. Association for Computing Machinery. https://doi.org/10.1145/3126908.3126923

[39] 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 (ASPLOS '19)*. Association for Computing Machinery. https://doi.org/10.1145/3297858.3304024