Improving Performance on Tiered Memory with Semantic Data Placement

Allen Aboytes and Pankaj Mehra

***Abstract*—Memory-intensive application working sets continue to grow and demand more memory. Far memory technologies such as CXL potentially solve the memory capacity bottleneck by adding terabytes to system configurations. However, efficient use of far memory requires careful data placement among memory tiers. Recent work uses page-based memory tiering systems to expand the memory available to applications. Unfortunately, most state-of-the-art memory tiering systems largely ignore memory allocation and prioritize placing pages in the fast tier while space remains available. Relying on transparent methods for memory allocation can lead to suboptimal data placement, resulting in more data migration. To address these issues, we propose to place data using application semantics to increase the locality of reference within pages. We present M2T, a system that optimizes the layout of application memory allocations by grouping semantically related memory objects with a custom memory allocator and migrates pages between local and far memory. Our evaluation demonstrates that semantic data placement achieves 3.39–4.69× higher throughput than a key-value store that uses a standard memory allocator on top of various state-of-the-art memory tiering systems.**

***Index Terms*—Memory tiering, memory allocation**

I. Introduction

MEMORY-INTENSIVE applications such as Vector Databases, machine learning inference, and in-memory database management systems continue to demand more memory capacity from data centers. Traditional solutions to the memory capacity bottleneck involve using storage, network devices, or adding more memory devices to servers. However, utilizing I/O is high latency compared to the speed of DRAM, and adding more memory to servers is expensive and results in stranded memory resources [1], [2]. Emerging technologies such as non-volatile memories (NVM) or emerging interconnect protocols such as Compute Express Link (CXL) [3] present alternatives that are low latency and byte-addressable.

Promising technologies such as NVM or CXL memory create a heterogeneous memory hierarchy due to their higher latency and generally lower bandwidth than DRAM. The memory provided by these devices is often referred to as far memory; in this work, we refer to DRAM as local memory. Efficient use of far memory depends on placing application data in the right memory tier. Poor data placement decisions can result in excessive trips to far memory and increased data movement, degrading performance. With heterogeneous memory hierarchies becoming more prevalent, researchers propose tiered memory systems to address such issues.

Memory tiering systems manage the placement of data on heterogeneous memory hierarchies to increase the effective memory capacity. Existing work [1], [2], [4], [5], [6] use a combination of memory access profiling, memory classification, and memory migration to place hot memory pages in the fast tier (promotion) and cold memory pages in far memory (demotion) to improve fast memory capacity utilization. However, state-of-the-art memory allocators and kernel page allocators can have suboptimal placements due to their transparent methods for data placement. Many systems (e.g., Linux) allocate fast tier memory first by default; consequently, hot data may be placed initially in far memory as fast memory capacity diminishes, resulting in more data migration.

We present Mnemonic Memory Tiers (M2T), a system that optimizes memory allocation placement using semantic data placement. M2T groups semantically related data with a custom memory allocator (`mnalloc`) and migrates pages between local and far memory. In summary, the contributions of our study are:

- The design of M2T as a programming system to optimize memory layout, combined with tiered memory.
- An evaluation of a key-value store that uses `mnalloc` to optimize data placement with tiered memory.

II. Related Work

Tiered memory systems aim to keep frequently accessed data in the fast tier, with many state-of-the-art approaches focusing on transparent data placement [1], [2], [4], [5], [6], [7]. For example, Lagar-Cavilla, et al. [7] uses the OS swap path to store compressed DRAM while systems like TMTS [2] dynamically migrate pages informed by memory profiling. Previous work has explored placement of small (i.e., 4 KB) pages, huge (e.g., 2 MB) pages, and memory regions with multiple pages as implemented in TPP [1], Memtis [4], and PET [6] respectively. Transparent page placement techniques, however, can suffer from suboptimal memory object placement due to the default first-touch policy for page allocation.

Many proposals consider memory object placement through custom memory allocator interfaces [8], [9], [10], data structures [11], or by intercepting memory allocations [6]. Carbink [11] introduces a programming model that swaps objects to and from network-based far memory. Medius [10] is a locality-conscious allocator that places data using programmer

Manuscript received November 4, 2023; revised February 28, 2024. (*Corresponding Author: Allen Aboytes.*)

Allen Aboytes is with the University of California, Santa Cruz, CA 95064 USA. (e-mail: aaboytes@ucsc.edu)
Pankaj Mehra is with The Ohio State University, Columbus, OH 43210 USA. (e-mail: mehra.77@osu.edu)

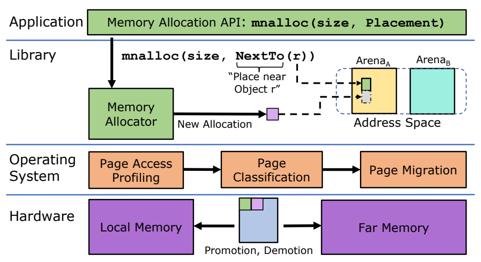

Fig. 1. Semantic Memory Tiering System Architecture. A layered approach combines a specialized memory allocator with a memory tiering system.

hints. Samsung SMDK [12] and SK Hynix HMSDK [13] provide custom allocators that map allocations to memory pools on different devices. Transparent tiering systems, such as PET [6], manage placement at a coarse granularity by intercepting `mmap` calls. Tiering systems such as TMC [9] and X-Mem [8] support fine-grained placement through modified allocators. However, most of these systems do not dynamically migrate pages between directly addressable memory tiers.

Tiered memory systems have primarily focused on page migration and profiling, largely overlooking memory allocation. Our work instead considers memory object placement at fine granularity, an underexplored area in tiered memory systems. We propose a system that leverages application semantics, specified at allocation time, to guide object placement while still supporting runtime page migration. Traditional memory allocators use the size of objects as a semantic for placement on specific pages, but with tiered memory this can lead to suboptimal placement because the system lacks an understanding of application semantics. By incorporating application semantics into placement decisions, we can improve performance through better locality of reference within pages.

III. Semantic Data Placement for Tiered Memory

As shown in Fig. 1, M2T is a semantic tiered memory system that controls memory object placement using application semantics, which are estimates of programmer intent based on program characteristics, observed statically or dynamically.

A. Memory Allocation

M2T provides a custom memory allocator, `mnalloc`, that allows users to influence static placement policies and manage multiple arenas. Memory arenas are the logical locations where related memory allocations are placed, acting as the unit of locality. Each memory arena manages its own independent set of pages and uses a linked list style allocator for memory management. The `Placement` argument associates memory objects with a memory attribute that encodes application semantics, expressing to the system the program behavior from the perspective of its data.

Memory attributes describe functional and non-functional properties that guide the placement of memory objects. Functional properties express rules that must be followed by the system, such as the spatial locality attribute `NextTo`, which places one object near another. Non-functional properties describe preferred placement characteristics, giving the system greater flexibility. For example, a data structure developer can indicate the access frequency of memory objects as hot or cold. In turn, M2T uses this knowledge to place memory objects in their preferred memory tier.

The M2T prototype implementation showcases a memory attribute for spatial locality (`NextTo(r)`) that packs application-level objects into the same memory arena. The `NextTo` placement directive takes a virtual address (`r`) to an existing allocation, and the allocator ensures that the subsequent allocation is logically co-located with that object. M2T achieves this by placing the new memory allocation in the same memory arena as the existing memory object (Fig. 1). Applications integrate with M2T by allocating memory using the `mnalloc` library call.

B. Memory Tiering

M2T's memory tiering component migrates application data at runtime according to page access frequency, implemented using Meta's TPP system [1]. Memory migration is performed in page-sized granules and does not disrupt the placement of memory objects within pages. However, unlike TPP, migrated pages contain related application data as determined by the static placement decisions of `mnalloc`. Although pages allocated to memory objects follow a first-touch policy, `mnalloc` ensures that related objects are placed on the same page when possible, or on neighboring pages within the same memory arena, based on the semantic placement directive. A traditional memory allocator, on the other hand, may scatter related memory objects across the address space—for example, due to allocations occurring at different times.

C. Case Study: Key-Value Store

We implement a multi-threaded key-value store TCP server application to study the impact of semantic data placement on system efficiency and performance. Internally, the server uses a hash table to store key-value pairs and can configurably use a hash table implementation with `NextTo` optimizations or one allocating memory using a standard memory allocator. The key-value store supports insert and lookup requests for keys and values of any size. Multiple threads handle clients concurrently, and each server thread can process multiple client requests on the same connection.

The design for the hash table index in the key-value store uses chaining for collision resolution. The hash table implementation in Rust is a fixed-sized array that stores the heads of linked lists representing each bucket in the hash table. Each bucket entry contains pointers to dynamically allocated key and value data. We use `NextTo` to optimize the spatial locality of nodes in each bucket by modifying the Rust standard library linked list. Similarly, we use `NextTo` to direct M2T to place key and value data for each bucket together. The optimized hash table implementation uses `mnalloc` to place each bucket in separate arenas and co-locates key-value pairs for each bucket within the same memory arena.

Limitations: The usage of M2T requires developers to analyze memory object characteristics to choose an appropriate `Placement` directive. The current M2T prototype implements support only for `NextTo`, and its layered architecture

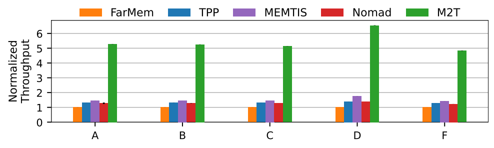

Fig. 2. Throughput of a chained hash table key-value store for YCSB workloads, normalized to using only far memory as baseline performance.

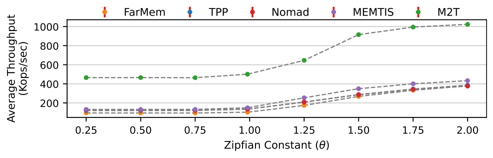

Fig. 3. Average throughput for YCSB-C with varying Zipfian request skew.

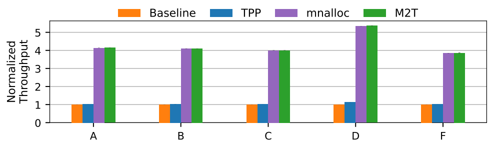

Fig. 4. Performance of each component of M2T, normalized to a baseline with a traditional memory allocator and first touch page placement policy. Results for mnalloc exclude proactive migration from TPP.

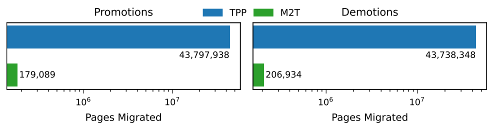

Fig. 5. Promotion and demotion of TPP and M2T for YCSB-C at θ=0.99.

leaves migration policies unchanged. We plan to explore other memory attributes and their interaction with page placement policies in future work.

IV. Evaluation

A. Experimental Setup

Our evaluation uses a CloudLab [14] Intel IceLake machine with two 36 core CPUs, and 128 GB of RAM on each socket. We use a NUMA node emulation of CXL by turning off all cores on one socket, resulting in local/far memory tier latencies and bandwidths of 84.6/156 ns and 176/55 GB/s. The fast memory tier is limited to 64 GB, with all workloads running on a single socket with hyperthreading disabled.

We compare the performance of M2T against TPP [1], Nomad [5], Memtis [4], and a baseline that allocates all memory in far memory (`FarMem`) using a key-value store application. The key-value store that runs on M2T uses `mnalloc` to semantically place data, and other baseline versions are implemented with unmodified standard library components using a traditional memory allocator to place data. M2T, TPP, and Nomad use Linux kernel version v5.15-rc6, and Memtis uses Linux kernel version v5.15.19. The fast tier capacity is constrained for M2T, TPP, and Nomad by offlining memory blocks, and by the use of cgroups for Memtis.

The Yahoo! Cloud Serving Benchmark (YCSB) suite [15] is configured to load 85 million records into the key-value store and execute 170 million operations during the run phase, with keys drawn from a Zipfian distribution (default θ=0.99). The client and server each use 32 threads, and the application memory footprint is approximately 109 GB. We evaluate multiple YCSB workloads and configurations, reporting averages with 95% confidence intervals. We conduct detailed comparisons with TPP to explain the performance characteristics of M2T.

B. Overall Performance

Varying Request Patterns: We ran the YCSB benchmark workloads A, B, C, D, and F. Fig. 2 shows the performance of our M2T prototype compared to representative baselines. The `mnalloc`-augmented key-value store service performs the best (3.39–4.69×) over various read and write access patterns compared to baselines built with standard library components. The performance benefit comes from the increase in spatial locality of both nodes in each bucket and key and value data. Placing data using `NextTo` increases cache efficiency, reduces false sharing at the page level, and reduces page migration activity. The memory tiering baselines are unable to optimize placement smaller than a 4 KB page.

Varying Request Skew: We measure the performance of the key-value store with the read-only YCSB-C workload with keys sampled from a Zipfian distribution whose constant θ we vary. Fig. 3 shows the throughput for workload C as a function of θ ranging from 0.25 to 2.0. The results show that semantic data placement improves performance even as the request skew varies. At low skew, we see that M2T is 3.52–3.87× faster than other systems, and at high request skew, the performance difference is 2.36–2.65×. As the request skew increases, meaning that the number of frequently accessed keys decreases, the overall performance improvement of using `NextTo` decreases. Despite that, our evaluation shows that placing data with `mnalloc` provides more than a 2× speedup.

C. Understanding M2T Performance

Performance Breakdown: Fig. 4 shows the performance of each component of M2T normalized to a baseline that uses a standard memory allocator with no migration. Enabling memory tiering through TPP improves performance from 2.3% to 14%. Optimizing the memory layout with `mnalloc` and `NextTo` yields 3.86–4.14× speedups over the baseline. When we combine semantic data placement with page migration (M2T), we observe modest improvements ranging from 0.05%–0.41% compared to using only `mnalloc` (no TPP). Each technique contributes to the overall performance, but the benefit comes primarily from `mnalloc`.

Page Placement Efficiency: We analyze memory tiering event counters for M2T and TPP under YCSB workload C for the default configuration to explain the observed performance differences. M2T incurs only thousands of page migration events (Fig. 5), reducing promotion and demotion activity by 244× and 211×, respectively. In contrast, TPP exhibits a high migration volume, indicating frequent page promotion and demotion. Fig. 6 shows the migration rate, measured

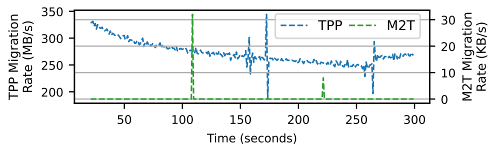

Fig. 6. Migration rate of TPP and M2T for YCSB-C at θ=0.99.

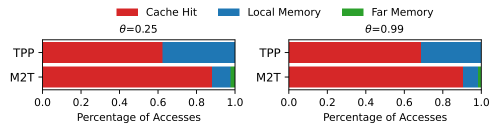

Fig. 7. Memory access breakdown for YCSB-C for θ=0.25 and θ=0.99.

TABLE I
CACHE MISS RATES AND CACHE MISSES FOR YCSB WORKLOAD C

Cache	θ	TPP	M2T
L1 % (#)	0.25	37.950 (5.065e+11)	16.110 (2.286e+11)
	0.99	38.238 (5.070e+11)	16.776 (2.341e+11)
L2 % (#)	0.25	98.904 (5.013e+11)	70.720 (1.656e+11)
	0.99	98.451 (4.986e+11)	71.131 (1.626e+11)
L3 % (#)	0.25	99.531 (4.969e+11)	99.546 (1.641e+11)
	0.99	84.405 (4.178e+11)	82.309 (1.327e+11)
TLB % (#)	0.25	25.573 (4.007e+08)	2.690 (9.521e+07)
	0.99	25.403 (4.081e+11)	2.743 (4.410e+10)

as bytes migrated per second. TPP incurs 180–310 MB/s of migration overhead, while M2T sustains only 4–32 KB/s. This disparity arises from the effectiveness of TPP's page table access-bit scanning and NUMA hint fault telemetry, which are triggered by a larger number of L3 cache and TLB-load misses. However, frequent migrations also amplify TLB misses through address remapping and consequent TLB invalidations [2]. Conversely, placing data with `NextTo` reduces cache and TLB misses, giving M2T fewer opportunities to detect memory accesses and reduce far-memory usage. As a result, M2T performs relatively few migrations.

Cache Performance: Table I shows the cache miss rates and cache miss events for YCSB workload C. We note that the reported metrics only contain events for memory loads. The key-value store that uses `mnalloc` has lower cache miss rates overall. M2T at low skew (θ=0.25) shows 2.16× and 3× fewer L1 and L2 misses, respectively. Although the L3 miss rates are comparable, the unoptimized version using standard library components experiences 3x more L3 cache misses. The optimized version with `NextTo` maintains better TLB performance with 9.25× fewer TLB misses. At higher skew (θ=0.99), M2T achieves 2.28×, 3.06×, 3.14×, and 9.51× fewer L1, L2, L3, and TLB misses than TPP.

The difference in cache performance arises from `NextTo` object placement and reduced page migration activity. The increase in cache hits is due to `mnalloc` co-locating items within each hash bucket, enhancing locality during key-value insertions and lookups. The reduction in TLB misses for M2T stems from both `mnalloc` and fewer page migrations. Since `NextTo` increases memory object locality within pages, it reduces TLB misses and page table walks. By comparison, TPP can only optimize locality at the page granularity within the same memory tier. Frequent page migrations in TPP also degrade TLB efficiency through expensive TLB invalidations to update virtual-to-physical mappings.

Fig. 7 compares the memory accesses to cache, local memory, and far memory for YCSB workload C at different Zipfian request skews. We observe that `mnalloc` increases the number of accesses that hit the cache by 41.24% (θ=0.25) and 31.85% (θ=0.99) over TPP. The results at low request skew indicate that fewer cache hits are observed and more accesses to memory occur (compared to θ=0.99), in particular TPP has 6% less accesses to cache, which mostly hit local memory, and M2T has 2.22% less cache accesses of which 1.72% go to local memory and 0.5% got to far memory. At low and default request skew, M2T incurs more far memory accesses (2.14% and 1.64%) than TPP (less than 1%), as fewer TLB misses reduce the tiering layer's ability to detect memory accesses. Even though M2T experiences more far memory accesses, improved cache hits have a greater impact on performance because of the large latency gap between cache and memory.

V. CONCLUSION

M2T groups semantically related memory objects onto the same pages—via a parameter in the memory allocator—and migrates them together, improving locality and reducing migration overhead. M2T achieves 3.39–4.69× higher performance than a key-value store with a standard allocator on state-of-the-art memory tiering systems due to better cache efficiency and reduced data movement.

REFERENCES

[1] H. A. Maruf *et al.*, "TPP: Transparent page placement for cxl-enabled tiered-memory," in *Proc. Int. Conf. Arch. Support Program. Lang. Operating Syst.*, 2023, pp. 742–755.
[2] P. Duraisamy *et al.*, "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proc. Int. Conf. Arch. Support Program. Lang. Operating Syst.*, 2023, pp. 727–741.
[3] "CXL specification," [Online]. Available: https://computeexpresslink.org/cxl-specification/.
[4] T. Lee *et al.*, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proc. Symp. Operating Syst. Princ.*, 2023, pp. 17–34.
[5] L. Xiang *et al.*, "Nomad: Non-Exclusive memory tiering via transactional page migration," in *USENIX Symp. Operating Syst. Des. and Impl.*, 2024, pp. 19–35.
[6] W. Doh *et al.*, "PET: proactive demotion for efficient tiered memory management," in *Proc. European Conf. Comput. Syst.*, 2025, p. 854–869.
[7] A. Lagar-Cavilla *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proc. Int. Conf. Arch. Support Program. Lang. Operating Syst.*, 2019, pp. 317–330.
[8] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proc. European Conf. Comput. Syst.*, 2016, pp. 1–16.
[9] Y. Ni *et al.*, "Tmc: Near-optimal resource allocation for tiered-memory systems," in *Proc. Symp. Cloud Comput.*, 2023, pp. 376–393.
[10] A. Jula and L. Rauchwerger, "Two memory allocators that use hints to improve locality," in *Proc. Int. Symp. Memory Manage.*, 2009, pp. 109–118.
[11] Y. Zhou *et al.*, "Carbink: Fault-Tolerant far memory," in *USENIX Symp. Operating Syst. Des. and Impl.*, 2022, pp. 55–71.
[12] Samsung, "Scalable memory development kit," [Online]. Available: https://github.com/OpenMPDK/SMDK.
[13] S. Hynix, "Heterogeneous memory software development kit," [Online]. Available: https://github.com/skhynix/hmsdk.
[14] "The cloudlab manual - hardware," [Online]. Available: https://docs.cloudlab.us/hardware.html.
[15] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with ycsb," in *Proc. Symp. Cloud Comput.*, 2010, pp. 143–154.