Trimma: Trimming Metadata Storage and Latency for Hybrid Memory Systems

Yiwei Li
Tsinghua University
China
liyw19@mails.tsinghua.edu.cn

Boyu Tian
Tsinghua University
China
tby20@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
China
Shanghai Qi Zhi Institute
China
gaomy@tsinghua.edu.cn

Abstract

Hybrid main memory systems combine both performance and capacity advantages from heterogeneous memory technologies. With larger capacities, higher associativities, and finer granularities, hybrid memory systems currently exhibit significant metadata storage and lookup overheads for flexibly remapping data blocks between the two memory tiers. To alleviate the inefficiencies of existing designs, we propose Trimma, the combination of a multi-level metadata structure and an efficient metadata cache design. Trimma uses a multi-level metadata table to only track truly necessary address remap entries. The saved memory space is effectively utilized as extra DRAM cache capacity to improve performance. Trimma also uses separate formats to store the entries with non-identity and identity address mappings. This improves the overall remap cache hit rate, further boosting the performance. Trimma is transparent to software and compatible with various types of hybrid memory systems. When evaluated on a representative hybrid memory system with HBM3 and DDR5, Trimma achieves up to 1.68× and on average 1.33× speedup benefits, compared to state-of-the-art hybrid memory designs. These results show that Trimma effectively addresses metadata management overheads, especially for future scalable large-scale hybrid memory architectures.

CCS Concepts

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Hardware → Memory and dense storage**.

Keywords

hybrid memory, DRAM cache, high-bandwidth memory, metadata

ACM Reference Format:
Yiwei Li, Boyu Tian, and Mingyu Gao. 2024. Trimma: Trimming Metadata Storage and Latency for Hybrid Memory Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3689612

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3689612

1 Introduction

Main memory now performs an increasingly critical role in computer systems, especially when executing data-intensive applications with massive amounts of data [51]. Both fast access speed and large capacity are desired for main memory, but these two goals exhibit fundamental conflicts under any existing individual memory technology. Fortunately, hybrid memory systems [7, 9, 24, 29, 30, 50, 52, 60–62, 66, 69, 79], which integrate two or more memory tiers with different characteristics, have the potential to overcome such a tradeoff and achieve both performance and capacity advantages over traditional architectures. Typically, a slow memory tier provides a sufficiently large capacity to hold the entire dataset, while a fast memory tier delivers lower latency and higher bandwidth for the most frequently accessed data.

With new technologies such as non-volatile memories (NVMs) [22, 44] and new interconnection techniques such as Compute Express Link (CXL) [10], the slow memory capacity is able to keep increasing, while the fast memory chooses to maintain its small size for sufficiently fast access speed. Furthermore, to improve performance, hybrid memory systems start to adopt higher associativities for address mapping between the two tiers [46, 60, 73, 74], and also use more fine-grained data blocks for migration [7, 30, 50, 66, 74]. These trends inevitably require highly flexible address remapping schemes covering the entire memory space, so that the most critical data can be effectively identified and moved to the fast memory.

Consequently, in future hybrid memory systems, not only do we need to *store* a large amount of metadata for address remapping, but also these metadata must support sufficiently fast *lookup* to avoid being a performance bottleneck. Unfortunately, existing metadata designs fail to alleviate these storage and lookup overheads when hybrid memory systems scale to large capacities, high associativities, and fine-grained data blocks. For example, cache-style tag matching schemes [29, 50, 61, 66, 79] store address tags only for data blocks in the fast memory, thus exhibiting moderate metadata storage cost. However, associative tag matching cannot support high-associativity designs due to the significant latency to search through the many tags inside a cache set. Alternatively, the address remapping information can be kept using a simple linear table [39, 60, 69, 74], where metadata access only requires a single lookup. Dedicated on-chip caches can further alleviate this access latency [20, 39, 60, 69, 74]. But on the other hand, because the remap table needs to have an entry for *every* data block, it has a huge size that proportionally grows with the *total* memory capacity. Such a huge table needs to be stored in the fast memory, occupying the available high-performance capacity for the actual data. Moreover, the SRAM metadata cache also incurs on-chip area cost and may not scale to effectively buffer the increasingly larger metadata.

To alleviate such metadata overheads, in this work we propose Trimma, the combination of a multi-level metadata structure and an efficient metadata cache design for hybrid memory systems. Trimma is compatible with both cache and flat use modes of the fast memory. It is a hardware-only design that is completely transparent to application and system software. These features make Trimma an easy-to-adopt and widely applicable architecture.

To reduce the metadata *storage* cost, Trimma uses an *indirection-based remap table* (iRT), similar to the multi-level OS page table, to only store the truly necessary metadata. iRT is a scalable structure as its size is proportional to only the fast memory capacity instead of the overall memory size under most cases. To achieve this, iRT eliminates the entries for unallocated *and uncached/unmigrated* data blocks, whose mappings are always the identity function. With a simple yet efficient memory management scheme, iRT can easily allocate/deallocate entries in hardware. Despite being constantly varying and highly fragmented with irregular distribution, the saved fast memory spaces for the unnecessary metadata entries are effectively utilized by Trimma as extra DRAM cache slots to deliver performance gains to the system.

In addition, to reduce the metadata *lookup* latency, especially the even more expensive multi-level traversals in iRT, Trimma uses an *identity-mapping-aware remap cache* (iRC), with separate formats to store the entries with non-identity and identity mappings. The non-identity mappings use the same conventional remap cache design, but with a slightly smaller capacity to save some space for storing identity mappings. The identity-mapping part of iRC instead uses a different and more compact format to store more entries within the same amount of SRAM. This is because we only need to know which addresses exhibit identity mappings, without the need to store the remapped pointers.

With these two key techniques, Trimma is able to offer significant performance improvements. We compare Trimma to state-of-the-art designs under different hybrid memory technology combinations. Trimma achieves up to 1.68× and on average 1.33× speedups on an HBM3 + DDR5 system, and up to 1.80× and on average 1.34× speedups on a DDR5 + NVM system. The reasons for such improvements are twofold. First, there is a large extra DRAM cache area due to metadata savings, from uncached/unmigrated data with identity address mappings. Data accesses missed from the fast memory in Trimma are therefore reduced by 7.9% and migration traffic is reduced by 23%. Second, the identity-mapping-aware remap cache effectively increases the coverage of the remap cache, and reduces metadata access misses. The overall remap cache hit rate has increased from 54% to 67%. Overall, our findings indicate that Trimma effectively improves the scalability of hybrid memory systems by tackling the challenges associated with metadata storage and lookup overheads, particularly suitable for upcoming large-scale hybrid memory architectures.

In summary, we make the following contributions in this paper.

- We identify that the metadata storage and lookup overheads would become potential bottlenecks in hybrid memory systems, given the recent trends towards larger capacities, higher associativities, and more fine-grained block sizes.
- We propose an indirection-based remap table structure, iRT, that only keeps truly necessary remap entries. The memory space for unnecessary remap entries is retrofitted as extra DRAM caching space, hence improving performance.
- We propose an efficient identity-mapping-aware remap cache design, iRC, that uses separate formats and storage for non-identity and identity address mappings. Particularly, the identity mappings use a more compact scheme to improve the remap cache coverage and thus the hit rate.
- We integrate the two techniques in Trimma, a scalable hybrid memory metadata management scheme. Our evaluation shows that Trimma achieves significant performance benefits on various hybrid memory systems over the state-of-the-art designs.

2 Background and Motivations

Hybrid memory systems typically have multiple memory tiers, either by integrating two or more heterogeneous memory technologies, such as 3D-stacked DRAM [21, 27, 28], DDR4/5 DRAM [25, 26], and/or byte-addressable non-volatile memories (NVMs) [22], or extending beyond local memory to remote memory through advanced interconnection technologies like Compute Express Link (CXL) [10]. The *fast memory* tier usually has lower access latency and higher bandwidth, but comes with smaller capacity. In contrast, the *slow memory* tier offers much larger capacity to extend the system main memory at lower cost, while its access speed is inferior. Hybrid memory systems can either use the fast memory as a cache in front of the slow memory [9, 24, 29, 30, 50, 61, 77, 79], or treat the two as a horizontally flat organization both visible to the OS [7, 52, 60, 62, 66, 69]. Data are cached or migrated between the two tiers to let the most critical subset reside in the fast memory. A large volume of recent research has been focusing on specific cache management and flat migration policies [1, 8, 9, 18, 52, 62, 65, 79], in order to exploit the technology heterogeneity and achieve both performance and capacity advantages over traditional architectures.

2.1 Trends of Hybrid Memory Systems

With technology advances and system development, hybrid memory systems are currently observing several architectural trends. First, the typical memory capacities of the fast and slow memories keep diverging, resulting in **increasingly higher slow-to-fast capacity ratios**. Memory-intensive big-data applications like neural networks [32, 40, 42, 43], genome alignment [3, 19, 34], and graph analytics [14, 56, 59, 68, 70] have been continuously driving the growth of main memory capacity. The emerging NVM technologies, such as phase-change memories [44] and Intel 3D XPoint [22], promise to handily extend the byte-addressable memory in a system to TB scales. Similarly, advanced interconnection technologies like CXL 3.0 switches could also extend the system's remote memory capacity to several TBs [11, 54]. In contrast, fast memory technologies, primarily the in-package 3D-stacked HBM modules, are limited in their capacity due to physical area cost, vertical integration difficulty, and thermal dissipation requirements. Currently even the most recent HBM3 only offers a maximum of 24 GB per stack [27], two orders of magnitude smaller than an NVM-based slow memory. As a concrete example, the Intel Sapphire Rapids processors [23] integrate four HBM2E stacks and eight DDR5 channels, theoretically capable to scale to 64 GB HBM and 4 TB DRAM, resulting in a slow-to-fast capacity ratio as high as 64:1.

Second, the increasing slow-to-fast capacity ratio necessitates hybrid memory systems to adopt **higher associativities for mapping between fast and slow memories**. Early hybrid memory systems use very low associativities, usually partitioning the two memories into many small sets with only one fast block per set which several slow blocks can map to [7, 39, 61, 69]. With increasing capacity divergence, many slow blocks will compete for the single slot in the fast memory, incurring significant conflict misses and frequent replacements. To relieve the rigid mapping bottleneck, systems with higher associativities are recently proposed [46, 60, 73, 74], and result in higher hit rates in the fast memory. As shown in Figure 1, we run PageRank in a 16-core system with hybrid memory of DDR5 and HBM3, under a capacity ratio of 32:1 (detailed configurations are in Section 4). If we do not consider the metadata overheads (discussed in Section 2.2), increasing the associativity from 1 to 1024 could bring a significant speedup of 1.5× in the ideal case.

Third, hybrid memory systems would prefer **fine-grained blocks**. Previous designs have demonstrated the tradeoff regarding data block granularities [15, 60, 65, 69, 74]. In general, coarse-grained blocks exploit spatial locality, but also result in over-fetching that wastes significant memory bandwidth. A larger granularity also leads to fewer blocks under a fixed capacity, and incurs more capacity and conflict misses. On the other hand, fine-grained blocks increase both capacity and bandwidth utilization, but the large number of blocks may result in substantial management overheads such as metadata storage. Several designs have used sub-blocking techniques to balance bandwidth utilization and metadata overheads [30, 63, 66, 74], i.e., use a coarse-grained block size but only fetch the demanded fine-grained sub-blocks. In summary, most hardware hybrid memory systems prefer block granularities smaller than the OS page size, from a few hundreds of bytes (64 B or 256 B) [7, 30, 50, 66, 74] to the DRAM page size (2 kB) [69, 74].

2.2 Challenges of Metadata

In this work, we focus on hardware-only hybrid memory management that is completely transparent to software. Because any data block could potentially be moved across memory tiers, *every* data access request to a *physical address* must first lookup a hardware-based *metadata* structure to determine the actual location of this data block, i.e., the *device address* on the fast or slow memory tiers. As a result, the metadata design is particularly vital in terms of both performance and system cost. We discuss several existing metadata schemes below, and show how they would become a new potential bottleneck, especially in future hybrid memory systems that are scaling to large slow-to-fast capacity ratios, high associativities, and fine block granularities, following the trends in Section 2.1.

One popular metadata structure is the cache-style *tag matching* scheme, used by many prior systems, especially DRAM cache designs [8, 17, 29, 50, 61, 66, 79]. Within a specific cache set in the fast memory (DRAM cache), all tags must be compared against the access request. Upon finding a matched tag, the corresponding fast memory address is the remapped location, while no match indicates the data block is in the original slow memory address, i.e.,

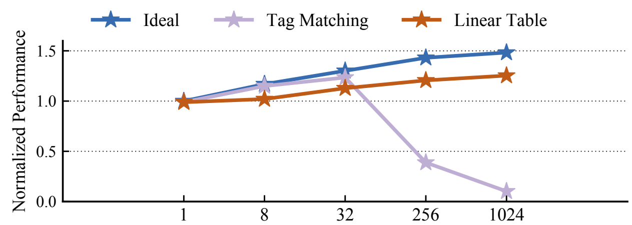


Figure 1: Performance comparison among various metadata management schemes for the PageRank workload. Simulation details are in Section 4. "Ideal" represents the theoretical scenario without metadata storage and lookup overheads. Normalized to the ideal case at an associativity of 1.

an identity mapping between the physical and device addresses. This metadata structure only needs to track the blocks in the fast memory, so its storage cost is moderate.

However, tag matching is viable only at low associativities. This is not only constrained by the hardware cost of associative search, but also due to the extra latency to access the tags from the memory. Usually the tag storage is too large to be kept on-chip, but they are accessed from off-chip memory. Assuming each tag is 4 B, each 64-byte access can only retrieve 16 tags. Thus, for designs with associativities higher than 16, multiple metadata lookups are needed, incurring significant extra overheads. Figure 1 shows that although tag matching approaches the ideal efficiency at low associativities, its performance quickly drops at high-associativity cases.

An alternative approach involves a *linear table* metadata format [39, 60, 69, 74], a.k.a., a *remap table* that tracks every data block in both the fast and slow memory tiers. Each access request uses the physical address to look up the remap table to find out the actual device address. Even at high associativities, the remap table lookup only needs a single memory access, more efficient than tag matching. Previous designs have proposed to use a simple on-chip *remap cache* to further filter these off-chip remap table accesses [20, 39, 60, 69, 74]. While a remap cache could achieve over 90% hit rates, its scalability is limited considering the limited on-chip area vs. the increasing fast and slow memory capacities.

Furthermore, a more severe issue with linear remap tables is the storage cost. The remap table is stored in the fast memory, but it needs to cover all data blocks across both memory tiers and thus has a size growing with the slow memory capacity. At large slow-to-fast capacity ratios, the remap table would consume a substantial fraction of the fast memory, constraining the available space for actual data and hence degrading the performance. For instance, a 16 GB HBM paired with a 512 GB DRAM at the 256 B granularity mandates over two billion entries. Assuming each entry is 4 B, half of the fast memory would be spent on metadata storage. Finer data block granularities, as proposed in certain studies [7, 50, 61], further enlarge the remap table size and exacerbate the storage overhead.

In summary, the metadata *lookup* and *storage* overheads are becoming increasingly challenging and must be effectively addressed for future hybrid memory systems that have large slow-to-fast capacity ratios, high associativities, and finer block granularities. This is the focus of our work.

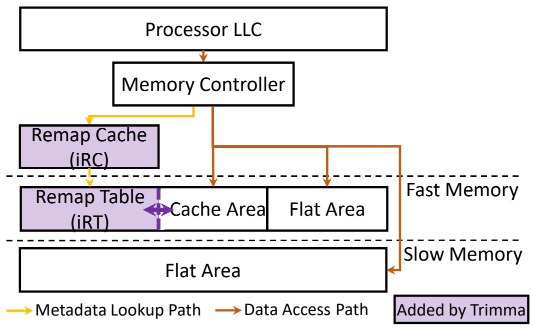


Figure 2: The overview architecture of Trimma. Designs different from the baseline hybrid memory system are highlighted. The saved capacity from iRT can be flexibly used as extra cache space.

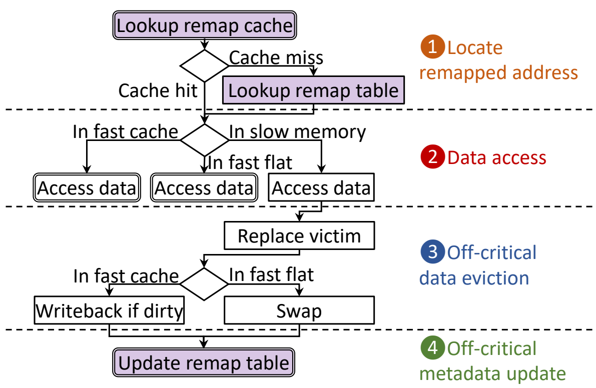


Figure 3: The overall access flow of Trimma. Changes beyond the baseline hybrid memory system are highlighted.

3 Design

We propose Trimma, a multi-level metadata structure for hybrid memory systems with efficient *storage* and *caching* approaches. Trimma is compatible with both cache and flat use modes of the fast memory, or even a hybrid of the two [2, 22, 39, 74]. It is also completely transparent to application and system software. Trimma adopts an *indirection-based remap table* (iRT) to effectively eliminate the need to store unnecessary metadata entries. It uses the saved fast memory space from the significantly reduced remap table size (up to 93%) as an extra DRAM cache area to improve the performance. In addition, to overcome the extra latency overhead of looking up the multi-level table, Trimma also incorporates an efficient *identity-mapping-aware remap cache* (iRC). By storing identity and non-identity address mappings with different cache entry formats, the overall hit rate of the remap cache increases, which further enhances the system performance.

3.1 Design Overview

Figure 2 illustrates the overall architecture of Trimma. Trimma is built on top of a basic hybrid memory design that uses the fast memory as either a DRAM cache or a part of the flat memory space,

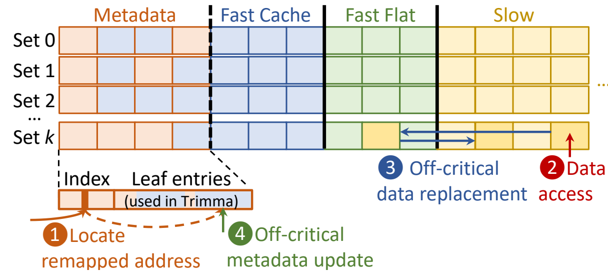


Figure 4: The set-associative memory layout in Trimma, with the access flow actions in Figure 3 performed upon. iRT enables some unused metadata blocks to be used as extra cache space (shown in blue in the metadata area).

or even a combination of the two modes, such as Chameleon [39] and Hybrid2 [74]. Both the fast and slow memory spaces are divided into *blocks*, as the granularity of caching/migration between the two tiers. To translate physical addresses to device addresses during data accesses, Trimma uses a forward-direction remap table as its metadata, similar to traditional linear remap tables but using a different data structure described later. This remap table is stored in the fast memory, and consumes its precious capacity which could have been used for the actual data. All blocks in both memories are partitioned into disjoint *sets* as a set-associative organization in Figure 4. Blocks are cached/migrated between the two tiers only within each set, so each set tracks the metadata separately. Trimma is particularly efficient for high-associative configurations.

Figure 3 illustrates the overall access flow of accessing a physical address in the OS-visible flat area in either the fast or slow memory, and Figure 4 shows the corresponding actions on the memory layout. As data could be cached in or migrated to another place, we first determine the actual device address, by looking up the remap entry in the on-chip remap cache or from the remap table in the fast memory (❶). If the remap entry indicates that the data are in the fast memory (either the cache or flat area), we directly access it. Otherwise we fetch the data from the slow memory (❷), return it to the processor, and handle replacement off the critical path (❸❹). The exact replacement policy choice is orthogonal to our design and discussed in Section 3.3.

Figure 3 also highlights the changes in Trimma that adopt novel remap table and remap cache designs, which are merely in the metadata lookup and update phases (❶❹) without affecting the rest data access/eviction. Hence innovations including replacement policies [1, 18, 52, 62, 65], selective migration [8, 9, 79], and cache-flat dual modes [2, 13, 39] can be orthogonally integrated with the remap table and remap cache designs in Trimma.

Indirection-based remap table. Rather than the conventional linear remap table, Trimma uses an *indirection-based remap table* (iRT, Figure 5), inspired by the multi-level OS page tables in the x86-64 architecture. Essentially iRT resembles a generic radix tree structure, but is completely managed by hardware. Unnecessary remap entries in iRT are not allocated, in order to save the fast memory space used by metadata. We identify and exploit two saving opportunities: (1) *unallocated data blocks* are never accessed and do not need metadata [39]; (2) *data blocks that stay at their original* *locations without being cached/migrated* do not need to be translated, since they have identity address mappings (physical address == device address). While the first one is trivial and already leveraged by OS page tables, the second source is particularly effective and novel for hybrid memory systems. Specifically, with a high slow-to-fast capacity ratio, only a small subset of the slow memory blocks can be cached or migrated to the fast memory; the rest majority must stay in their original places due to the slow swap policy [66, 69]. As a result, the total number of entries in iRT is proportional to only the *fast memory capacity*, instead of the *overall capacity*.

iRT is a much more scalable metadata design, as the slow memory capacity will be rapidly growing in the foreseeable future, while the fast memory size changes much slower due to the physical difficulties. The saved fast memory spaces from the smaller iRT can be effectively utilized as extra caching slots that extend the existing ones, increasing the DRAM cache hit rate for higher performance.

Nevertheless, as a more complex structure than linear tables, iRT faces several design challenges, calling for unique optimizations that set it apart from OS page tables. First, as iRT is managed by hardware, its entry allocations/deallocations and updates must remain sufficiently simple and efficient while keeping the saving opportunities. Second, with frequent data block movements between the two memory tiers, the saved remap entries may scatter into a highly fragmented layout, and continuously come and go. It is hence quite difficult to effectively utilize these irregular spaces, and quickly reclaim them when needed by newly allocated metadata. Finally, we need to identify the optimized detailed configurations, including the number of levels and the tag bit width for each level. We address these issues in Sections 3.2 and 3.3.

Identity-mapping-aware remap cache. One other major concern of using a multi-level table is the increased lookup latency. An L-level remap table may introduce up to $L + 1$ additional off-chip accesses in the worst case. Just like TLBs for page tables, a better caching approach is desired.

We observe that, we not only could skip storing an identity mapping in the remap table, we also *do not need to look it up if we could know it is an identity mapping*, i.e., we simply use the original physical address as the device address to access the data. This motivates us to design our remap cache in an identity-mapping-aware manner, saving cache spaces by not storing redundant (identity) remapped addresses, and improving cache coverage and utilization.

However, a naive design of completely skipping identity mappings in the remap cache will not work. The key difficulty is that if we do not find an entry in the remap cache, we have *no way to distinguish* between whether it is due to a miss to an actually valid remap entry, or because the entry is an identity mapping and thus skipped by the remap cache. Therefore, if we think of a remap entry as a key (physical address) value (device address) pair, we still need to store the keys though the values can be saved. As a result, we split the original single remap cache into two components in our *identity-mapping-aware remap cache* (iRC): a *NonIdCache* for valid remap entries as before, and an *IdCache* to filter the skipped remap entries with better SRAM space utilization. Specifically, the IdCache uses a similar design as a sector cache [63], and groups many entries into one cacheline to save space. Each entry only uses a single bit to indicate whether it is an identity mapping or not. We discuss the detailed design in Section 3.4.

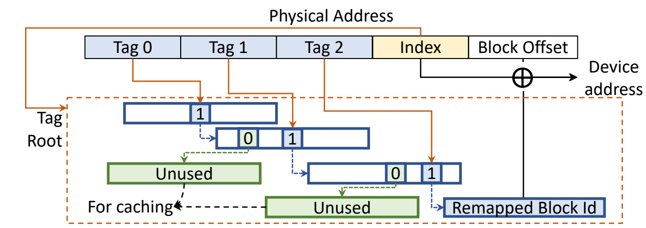


Figure 5: The indirection-based remap table structure and its lookup flow.

3.2 Indirection-Based Remap Table

Figure 5 illustrates the structure of iRT. To support set-associative hybrid memory systems, iRT uses separate trees for different sets. iRT is compatible with any associativity, and is particularly effective for high associativities when each set has more entries and thus more saving opportunities. Given a physical address, the index bits first select the *tag root* for its set, which points to the root of the corresponding table. The tag bits in the physical address are divided into multiple parts, which are used to traverse the multi-level table, until locating a leaf entry that stores the remapped block ID to concatenate with the block offset to get the device address.

Different from OS page tables that are managed by kernel software, iRT is searched and updated purely in hardware. To simplify the management, we reserve a continuous space in the fast memory. The table of each set, which is a complete radix tree assuming all intermediate and leaf entries are allocated, is linearized to a standard linear layout in its breadth-first order. Because the tables of all sets are contiguously stored and have the equal allocated size, the tag root of a set can be directly located without using a lookup table. Furthermore, the address of any entry in a tree can also be inferred from the tag bits without being explicitly stored. Essentially, each entry, when present, always resides at its own reserved and fixed location. Thus the intermediate entries of iRT just use bit vectors to indicate whether the next-level entries are allocated or not, and only the leaf entries store the remapped block IDs.

iRT does not populate all entries in the table, and eliminates as many unnecessary entries as possible to reduce the table size in the fast memory. If the lookup misses in any level of the iRT, we assume an identity mapping with the device address equal to the physical address, as if the block has not been moved. Such a default also covers unallocated data blocks, which should be never accessed. However, the unused entries in iRT could be highly fragmented and scattered randomly in the fast memory. We describe how we effectively utilize them as cache spaces in Section 3.3.

Example. Figure 4 illustrates an example access to a 2-level iRT. We perform one access to the intermediate level and the other to the leaf entry (❶). These accesses can happen in parallel as the entries are always in their fixed locations. If the intermediate-level bit is 0 meaning that the leaf entry is unallocated (as in this example), the block is in the original location (❷). After the accessed block is cached/migrated into the fast memory (❸), we also need to update the remap table (❹), by allocating the previously missing leaf entry and setting the corresponding upper-level bit (described below).

This metadata block may have cached another data block, which must be evicted as described in Section 3.3.

Table update. When a block is moved due to either caching, migration, eviction, or swap-out, its iRT entry should be accordingly updated. If the block is moved from its original location to another place, we allocate entries down to the leaf level; if it is restored back to its original slot, we clear the existing entry, and also free the upper levels if the entire block becomes empty. Such allocation/deallocation is simple because the iRT's linearized layout requires the entries to be at fixed locations. However, updates may involve changes in multiple levels. We reduce the cost by always buffering the intermediate-level entries in the on-chip controller when conducting the iRT lookup, so no backtracing is needed when these entries are later updated. In Trimma, a block will never be moved between two non-original locations; if it is evicted, it must go back to its initial place. This follows the slow swap policy [66, 69].

Detailed configuration selection. In iRT, we use 4-byte leaf entries to store the remapped block IDs. With a typical block size of 256 B [74], iRT can support up to $2^{32} \times 256\,\text{B} = 1\,\text{TB}$ memory *per set*. We can freely use multiple sets to support even higher capacity, e.g., using 1024 sets to cover 1 PB. This is more than enough for hybrid main memories even considering future scaling.

Regarding the level division in iRT, more levels provide more metadata space saving opportunities, but also increase the overall lookup latency. In order to conveniently utilize the saved metadata spaces without internal fragmentation, we require all iRT entries to be allocated/deallocated in a unit no smaller than the block size. This follows the same idea as the x86-64 page table which makes its intermediate levels always 4 kB aligned. With 256-byte blocks, each leaf metadata block in iRT would store 64 individual entries. Because the intermediate index levels store a single bit for each child instead of the address, each index block can hold 2048 children, corresponding to 11 bits for a tag chunk. Larger granularities than a single block bring limited benefits (see Section 5.3), while requiring more complex multi-block eviction. So Trimma uses 11-bit tag chunks for all iRT levels. This effectively realizes a 2048-ary radix tree. Because of the huge fanout, in this setting a simple 2-level iRT would be sufficient and more levels do not enable much additional space savings (Section 5.3). However, for even finer granularities such as 64 B data blocks, deeper iRTs would be useful.

Metadata storage savings. iRT saves significant metadata storage to be used as extra cache spaces. Assuming the above configurations of 4 B remap entries, 256 B blocks, and a 32:1 slow-to-fast capacity ratio (recall from Section 2.1 that real-world systems may have a ratio up to 64:1 [23]), a linear remap table occupies $(32 + 1) \times 4/256 = 52\%$ of the fast memory capacity, which further grows when the slow-to-fast ratio increases. If we apply a 2-level iRT design, the extra intermediate level has negligible storage overheads (worst-case $1/2048 = 0.05\%$) due to the use of valid bits rather than full addresses. The number of valid leaf blocks depends on the specific workload. In the best case, all remapped entries (equal to the number of fast data blocks) are densely packed into a contiguous set of leaf metadata blocks, and we only consume $4/256 = 1.6\%$ of the fast memory, plus the intermediate level storage. In the average cases, the leaf metadata blocks may be only partially occupied, and result in larger total occupation of metadata. On average, iRT reduces the metadata size to 11.0% of the fast memory.

Novelty beyond OS page tables. While iRT shares a similar structure as OS page tables, it has several key unique novel features. First, iRT is specially optimized for hardware, with predetermined and fixed addresses assigned to all the entries at different levels. This enables fast and parallel lookups, as well as efficient allocations/deallocations and updates, as described above. Second, iRT exhibits a unique opportunity that avoids storing the metadata for uncached/unmigrated data blocks, which OS page tables are not able to do so. Finally, iRT also could effectively utilize the saved fast memory spaces as extra cache spaces to hold more data and improve performance. We describe it next.

3.3 Using Saved Spaces for Caching

While iRT has a high potential to significantly reduce the metadata storage overhead and free up considerable fast memory capacity, it is not easy to effectively utilize the saved spaces. Available metadata blocks rapidly come and go when data blocks are cached/migrated/evicted, and they are highly scattered and fragmented in the fast memory with irregular distribution. Some prior hybrid memory designs such as Chameleon [39] rely on the OS to manage the varying memory usage when it allocates/deallocates data. This approach will not work well when we additionally consider data blocks with identity mappings, whose states change with caching and migration, much more frequently than memory allocation. The software overheads would then become unacceptable.

Instead, in Trimma we keep the saved metadata spaces invisible to software, and use them as extra DRAM cache spaces managed completely in hardware. This allows us to adapt much faster to the quickly varying metadata size and timely exploit the short-term performance opportunities. Notice that in order to cache a slow memory block into such an unused metadata slot, we use the same iRT to store the bidirectional mappings. The forward mapping (from the slow memory block to the DRAM cache block) is used for the look up process, which is illustrated in Figure 5. The inverted mapping (from the DRAM cache block to the slow memory block) is used at eviction. In other words, to utilize one 256-byte unused block, we need to insert two 4-byte entries into the same iRT.

To track the availability of each metadata block and determine whether it can be used for caching, we reuse the iRT intermediate levels. For example, in a 2-level iRT, each leaf metadata block has a corresponding index bit, where "1" means it is used as metadata and "0" means it is unused. For a multi-level iRT, the availability of an intermediate-level block is recorded in its upper level. We describe how to use these bits for cache replacement below.

Cache replacement of metadata and data. In order to simplify metadata allocation in hardware, the rigid iRT memory layout enforces each entry, if allocated, must reside in a certain fast memory block (Section 3.2). As a result, the metadata have higher priorities to use the block than the data. When an iRT update needs an entry, we directly evict the current data block cached in that location if there is one, regardless of its hotness. This data block could be refetched if needed and replace another less critical data block.

On the other hand, the replacement policy among data blocks should take into account the extra metadata slots if they are free to use. We distribute all reserved metadata blocks in the fast memory across all sets of the hybrid memory system, so each metadata block

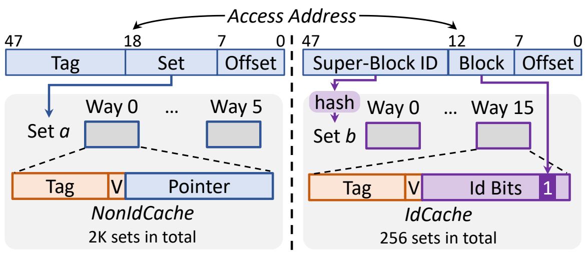


Figure 6: The identity-mapping-aware remap cache design.

is dedicated to a certain set. If the corresponding index bit of this block is "0", it contributes to the set capacity. The difficulty is that the effective set associativity keeps varying over time, and also differs across different sets. We need to accordingly adapt the scope of data replacement policies.

Because Trimma is especially beneficial for highly associative systems, we extend popular replacement policies under high associativities to the extra cache space. Due to the high cost of tracking LRU information under high associativities, recent hybrid memory systems choose simple FIFO or random replacements [17, 74, 79], or use area-efficient access counting [60], in order to quickly locate an acceptable victim. A FIFO policy simply skips blocks with index bits equal to "1". For a random policy, we check the index bit after randomly selecting a candidate, and resample if needed. Note that we can always evict a non-metadata block in the original cache area after a few times of retries.

More complex policies, such as LRU [29, 66], CLOCK [45], and MEA in MemPod [60], can also be applied. We provision enough replacement information storage for the maximum associativity including all metadata blocks. This is a small overhead because the metadata area is usually smaller than or comparable to the basic cache area, and replacement information is a small portion compared to other data storage. The hardware tracks the access behaviors for all blocks in the set as usual. When an eviction is demanded, we use the index bits to skip actual metadata blocks, and find the best candidate in the remaining slots.

Trimma chooses to use the FIFO policy, which can further adopt a simple optimization to reduce the cost of off-chip index bit accesses when selecting victims. Specifically, due to the FIFO ordering, we can prefetch the next chunk of index bits to a small on-chip buffer (a few bytes would hide most latency). We tried other more complex policies, e.g., an ideal LRU, but found only less than 1% hit rate improvements (possibly because of the high associativity), while incurring huge off-chip update traffic [78].

3.4 Identity-Mapping-Aware Remap Cache

The iRT in Trimma allows us to save the metadata storage, but exacerbates the metadata lookup latency and consumes more fast memory bandwidth by introducing multiple metadata accesses for each data block. Therefore we further propose an *identity-mapping-aware remap cache* (iRC) to alleviate these issues.

Under roughly the same SRAM capacity budget, we split the conventional remap cache into two components: a *NonIdCache* that stores the valid (i.e., allocated) remap entries with non-identity mappings, and a special *IdCache* that is organized similar to a sector cache [63] and stores bit vectors to indicate whether the entries associated with the data blocks in a larger *super-block* use identity mappings or not. Assuming the same SRAM capacity, the IdCache is able to store more entries due to the compressed format, thus improving the overall iRC coverage and runtime performance.

Figure 6 shows the cacheline formats of the NonIdCache and IdCache respectively. The NonIdCache operates in the same way as the conventional remap cache. In the IdCache, each cacheline uses the original space of a 4 B remap pointer to store 32 bits that indicate whether the corresponding super-block of 32 contiguous data blocks (8 kB) exhibits identity mappings or not. Each bit associates with one 256 B block in this super-block. Due to the different granularities (block vs. super-block) in the two parts, they use different index and tag schemes. For the IdCache, the lower address bits for the offset within the super-block are used to select one bit. Moreover, we utilize a hash-based index scheme [33] and a higher associativity in the IdCache to reduce conflict misses from the large number of identity mappings. We empirically select the proper associativities to balance the access latencies of both caches. To summarize, the NonIdCache has 2048 sets of 6 ways each, and the IdCache has 256 sets of 16 ways each (Table 1). The total capacity is equal to a 64 kB conventional remap cache.

Each metadata data lookup would access the two caches in parallel, and either the IdCache or the NonIdCache would hit, or both miss. If hit in the IdCache, which means that a matched bit vector is found and the corresponding bit is 1, we directly use the original physical address as the device address. If hit in the NonIdCache, we use the pointer stored in the cache. If both are misses, we go to the off-chip remap table (i.e., iRT) to retrieve the entry. If the entry is valid, it is inserted into the NonIdCache. If the entry is not allocated, it is inserted into the IdCache.

Note that when iRT gets updated because of data block caching and migration, the corresponding iRC entries should also be updated for consistency. We simply invalidate the entries from iRC.

Bloom filters as an alternative? At the first glance, the iRC could use Bloom filters [6] as compressed storage. The physical block addresses with identity or non-identity mappings essentially form two sets, and the purpose of the iRC is to test whether a request address is in each set or not. However, due to the false positives in Bloom filters, we cannot use them to store the identity-mapping set; doing so may incorrectly classify an address with non-identity mapping into the identity-mapping set. On the other hand, storing the non-identity-mapping set in Bloom filters has limited benefits; we still need to store their original remap entries in the cache.

3.5 Discussion

More saving opportunities. Trimma saves remap table entries for unallocated and uncached/unmigrated data blocks with identical physical-to-device address mappings, using a simple hardware-only solution. If we further leverage software support, there could be more opportunities. For example, when a data block is deallocated by the workload, it would never be accessed again and its remap entry can be recycled. Hardware alone cannot know this information, unless told by the software through a well-defined interface [39]. We leave a detailed design as future work.

Table 1: System configurations.

Cores	x86-64, 3.2 GHz, 16 cores
L1I	32 kB per core, 4-way, 64 B cachelines, LRU
L1D	64 kB per core, 8-way, 64 B cachelines, LRU
L2	1 MB per core, 8-way, 14-cycle latency, LRU
LLC	32 MB shared, 16-way, 60-cycle latency, LRU
SRAM	Conventional remap cache: 2048-set, 8-way, 3-cycle iRC: 2048-set, 6-way (NonIdCache) + 256-set, 16-way (IdCache)
HBM3 + DDR5	HBM 3.0, 1600 MHz, 16 channels; RCD-CAS-RP: 48-48-48 DDR5-4800, 1 channel, 2 ranks, 16 banks; RCD-CAS-RP: 40-40-40
DDR5 + NVM	DDR5-4800, 2 channels, 2 ranks, 16 banks; RCD-CAS-RP: 40-40-40 NVM, 1333 MHz, 2 channels, 1 rank, 8 banks; RD 77 ns, WR 231 ns

Another source of saving is to apply huge *physical* pages, so that only one remap entry can be used for many data blocks. This is similar to huge OS pages that reduce virtual-to-physical mapping overheads [16, 41, 55, 58, 81]. While the current iRT structure naturally supports this, we find that without a co-designed OS to organize the physical page layout, the chances of contiguous data blocks together migrating between the fast and slow memories are quite low. We plan to explore this opportunity in the future.

Fast swap policy. Trimma assumes the slow swap policy [69], which provides more opportunities of identical mapping. Alternatively, a fast swap policy does not require an evicted block to go back to its initial location, so it avoids cascaded migration and potentially offers higher performance. In order to apply iRT to fast swap, we can adopt proactive migration in the background that exchanges slow memory blocks and restores their original locations, similar to Chameleon-Opt [39]. However, this complicates the overall system and we do not further discuss it.

4 Experimental Setup

System configurations. We use zsim [67], a Pin-based simulator, to evaluate our designs. Table 1 summarizes our detailed system configurations. We model two types of hybrid memory systems, HBM3 + DDR5, and DDR5 + NVM, both with a 32:1 slow-to-fast capacity ratio according to Section 2.1. The default block size is 256 B. Other memory configurations are also considered in Section 5.3. The specifications of HBM3, DDR5, and NVM are extracted from recent literature and open-source implementations [26, 27, 36, 75, 80]. We use CACTI [4] to model the SRAM-based iRC. We select the overall system capacity to be larger than the workload memory footprints (details below), so no application suffers from page faults.

Workloads. We use the memory-intensive subset of SPEC-CPU 2017 [57] for multi-program workloads, similar to previous work [39, 66, 69]. The others are insensitive to memory performance and Trimma has negligible impact on them. For multi-threaded applications, we use the GAP benchmarks [5], the in-memory database silo [71] with the TPC-C workload, and the memcached key-value store [13] with two workloads YCSB-A and YCSB-B [12]. We run each SPEC benchmark in the rate mode with 16 copies, and use 16 threads for the multi-threaded workloads. For SPEC, we fast-forward over the initialization phase and simulate 5 billion instructions. For each GAP workload, we manually mark and only simulate the 2nd to 5th iterations. For database workloads, we skip data loading and only simulate query execution. We use weighted speedup as the performance metric for multi-program workloads.

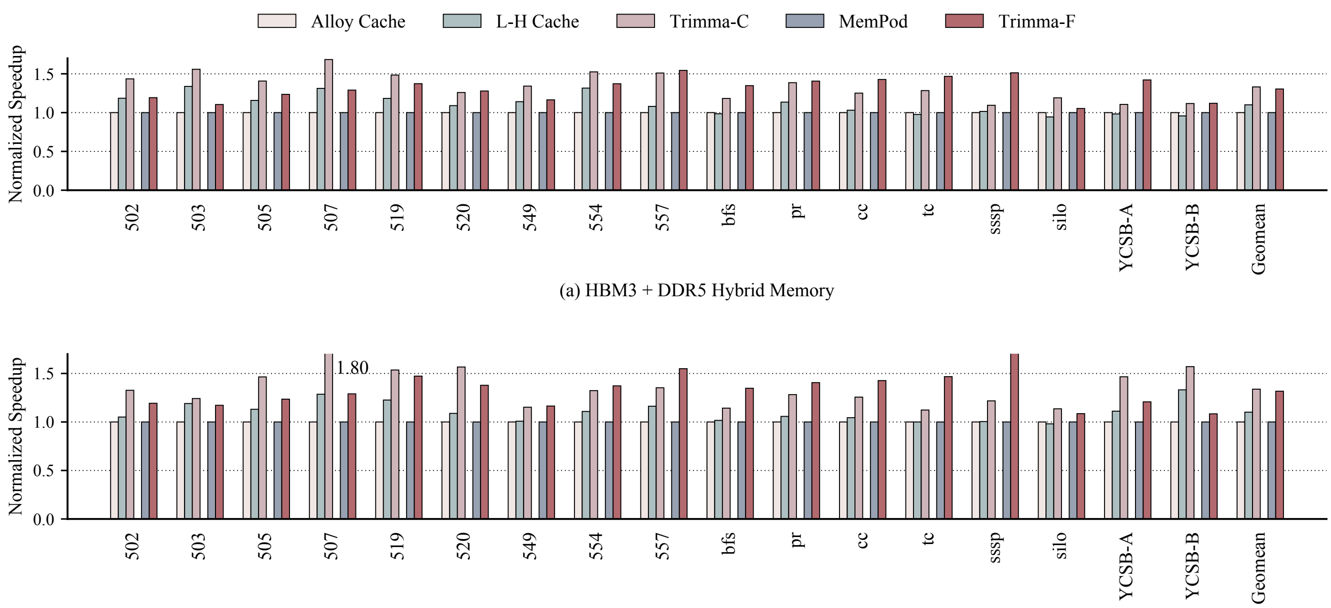
The original benchmarks have diverse memory footprints, ranging from 6 GB (16 processes of `519.lbm_r`) to 18 GB (`twitter` in GAP). Thus we set the slow memory capacity to 20 GB to match the maximum footprint, and the fast memory to be 1/32 of it. Then, in the simulation we scale up each workload's footprint to reach the full memory capacity, so roughly each workload can put 1/32 of its data in the fast memory. This ensures that our workloads touch large footprints and stress memory. Note that although the total memory footprint exceeds the fast memory capacity, many applications still exhibit locality and an optimized high-associativity design is necessary to efficiently capture the active working set in the fast memory.

Baselines. We evaluate our design under both the cache mode (**Trimma-C**) and the flat mode (**Trimma-F**) to show its general benefits. We compare Trimma-C with **Alloy Cache** [61] and **Loh-Hill Cache** [50] as the cache mode baselines. Although Alloy Cache and Loh-Hill Cache utilize different formats to embed tags with the data or within the same DRAM row to optimize metadata accesses, they essentially adopt the tag matching strategy illustrated in Section 2 and are limited to low associativities. In Alloy Cache, metadata and data are accessed in a single burst, so we do not simulate extra metadata access cost. However, it is limited to the direct-mapped organization. For Loh-Hill Cache, we set the associativity to 30, i.e., 30 256 B blocks together with their metadata in one 8 kB DRAM row. We access metadata as DDR accesses with DRAM row buffer hits. Furthermore, we apply the RRIP replacement policy to Loh-Hill Cache, which offers a 2.1% speedup over LRU. In contrast, Trimma-C can scale to fully associative. We assume a perfect Memory Access Predictor in Alloy Cache and a perfect MissMap structure in Loh-Hill Cache, ignoring some of the metadata overheads to optimistically estimate their performance. For flat mode designs, we compare Trimma-F with **MemPod** [60]. For both designs we use 4 sets with high associativities, as in MemPod. Both systems adopt the first-touch policy as in current NUMA systems [37, 48], i.e., greedily allocating the workload data in the fast memory first, until its capacity is exhausted.

5 Evaluation

5.1 Overall Performance Comparison

Figure 7 presents the overall performance of different designs under different memory technology combinations. All workloads we evaluate can benefit from the Trimma design. For the HBM3 + DDR5 system, on average, Trimma-C achieves 1.33× speedup over Alloy Cache while Trimma-F obtains 1.30× speedup over MemPod. Because Trimma can provide more available fast memory spaces and scale to high associativities, it provides higher improvements for workloads with high memory footprint or high associativity requirements. Take `557.xz` as an example. The low-associativity cache-mode baseline designs suffer from conflict misses in the fast memory and high migration traffic. In contrast, Trimma-C is 1.51× faster than Alloy Cache, by providing a higher associativity and more fast memory spaces. Trimma also excels if the workload exhibits more metadata savings, providing more extra caching spaces from the iRT design. This is the case for `507.cactuBSSN_r`, where iRT reduces the metadata size by over 75% and Trimma-C is 1.68× faster than Alloy Cache.



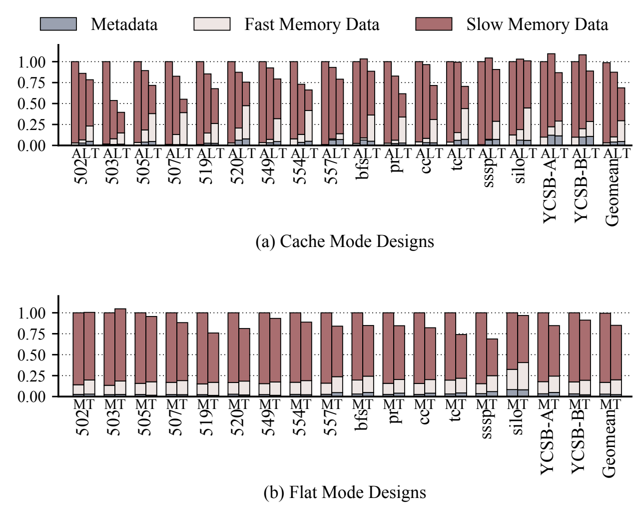
(a) HBM3 + DDR5 Hybrid Memory

(b) DDR5 + NVM Hybrid Memory

Figure 7: Performance comparison between Trimma and the baselines on (a) HBM3 + DDR5 and (b) DDR5 + NVM. Performance of cache mode designs (Alloy Cache, L-H Cache, Trimma-C) is normalized to Alloy Cache. Performance of flat mode designs (MemPod, Trimma-F) is normalized to MemPod.

A similar trend can also be observed with DDR5 + NVM. Overall Trimma-C achieves 1.34× speedup over Alloy Cache while Trimma-F obtains 1.32× speedup over MemPod. For workloads with higher memory footprints, e.g., `sssp` with up to 16 GB, Trimma shows higher speedup on DDR5 + NVM, because the saved metadata space reduces the writeback traffic as in Figure 10(b), which significantly saves the limited slow memory bandwidth.

To further analyze the speedup sources, we break down the average memory access time into three parts: metadata lookup, fast memory data access, and slow memory data access, as shown in Figure 8. Generally, Trimma-C effectively reduces the slow memory access time by over 50% compared to Alloy Cache, where these accesses become hits in the fast memory, and increase the fast memory access time by 22%. The metadata lookup cost is generally small over all designs, but Trimma-C increases this overhead by 4.6%, due to more remap table accesses in iRT. Metadata lookups are insignificant because they are handled by the fast memory with low latency and high bandwidth, and many of them are filtered by the on-chip remap cache.



(a) Cache Mode Designs

(b) Flat Mode Designs

Figure 8: Memory access latency breakdown on HBM3 + DDR5, for (a) the cache mode, including Alloy Cache (A), L-H Cache (L), and Trimma-C (T), and (b) the flat mode, including MemPod (M) and Trimma-F (T).

5.2 Effectiveness Analysis of iRC and iRT

Next we use the flat mode designs (Trimma-F and MemPod) under the HBM3 + DDR5 configuration as a representative example, to explain the performance improvements from iRT and iRC.

iRT. Figure 9 compares the metadata sizes of iRT and the baseline linear table. We capture the metadata sizes at the end of our simulation as it represents the most fragmented state of the memory. Trimma iRT could effectively eliminate unnecessary identity mapping entries to save the metadata space by 43% on average and up to 85%. Higher spatial locality in the workload leads to higher savings. The saved spaces are used as extra caching spaces to improve performance. Similar to previous work [7, 66], we use the *fast memory serve rate*, i.e., the percentage of accesses handled by the fast memory, and the *bandwidth bloat factor* [8], i.e. the ratio

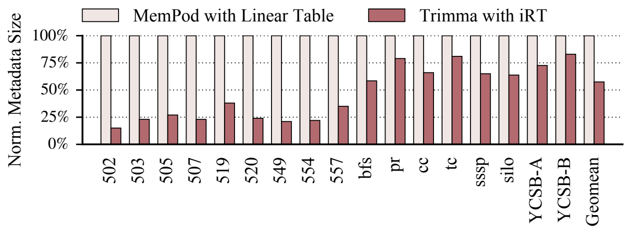


Figure 9: Metadata size comparison between Trimma using iRT and MemPod using a linear table. The metadata size is obtained at the end of simulation.

Figure 10: Detailed performance analysis between Trimma and MemPod, with (a) the percentage of memory accesses served by the fast memory (higher is better); (b) the ratio between total fast memory traffic and useful data traffic to the processor [8] (lower is better).

between total fast memory traffic (including migration/swapping with the slow memory) and useful data traffic to the processor, to quantify the caching benefits and the migration overheads, respectively, as shown in Figure 10. On average Trimma improves the fast memory serve rate by 7.9%. Generally speaking, higher fast memory serve rate increases lead to more performance gains, e.g., `519.lbm_r`, `557.xz_r`, and `tc`. Particularly, workloads with relatively low serve rates initially, e.g., `557.xz_r`, have higher demands for fast memory spaces, which are exactly what Trimma provides by reducing the metadata space. On the other hand, Trimma also reduces 23% memory migration traffic because of reduced conflict misses from larger available fast memory spaces. This is especially critical to the bandwidth-limited NVM-based slow memory.

On the other hand, iRT also incurs some overheads, as it requires multiple metadata accesses, rather than one access in a conventional linear table. However, these accesses in iRT are parallelized rather than serial (Section 3.2). Our experiments show that iRT only introduces at most 8.6% extra latency cost (without a remap cache)

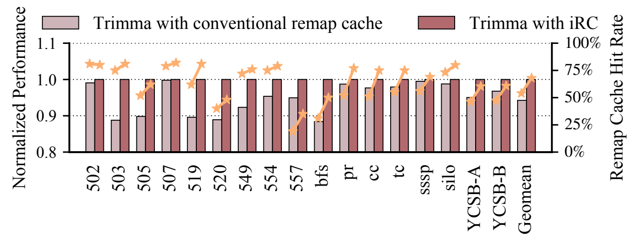


Figure 11: Performance (bars) and remap cache hit rate (lines) comparison between the conventional remap cache and iRC.

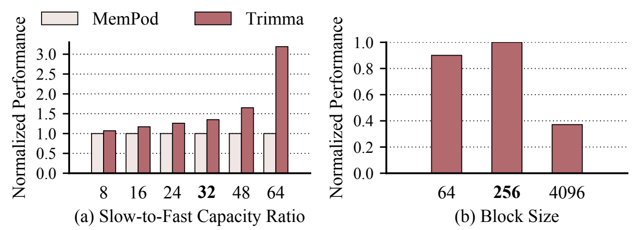


Figure 12: Performance comparison between (a) different slow-to-fast capacity ratios (default is 32); and (b) different block sizes (default is 256 B). Performance is the geomean across all workloads.

and an average of 1% (with iRC) compared to the single-probe case. For updates, as in Section 3.2, all updates are buffered on-chip and written back to iRT together. This writeback occurs off the critical path and has negligible performance impact.

iRC. Figure 11 shows the benefits of our iRC design that separates the identity and non-identity mappings in the remap cache, compared to a conventional non-split remap cache. The overall remap cache hit rate increases from 54% to 67% on average, and the performance improves by 6.4%. We find that the conventional remap cache achieves good hit rates for non-identity mappings, while the hit rates are as low as 6% for identity mappings. This is perhaps due to the relatively low hotness of identical mappings (otherwise these data blocks would be cached/migrated and become non-identity mappings). Using a dedicated iRC IdCache significantly increases the identity mapping hit rate to 32%. Meanwhile, the non-identity part is not sensitive to the capacity loss because of the good locality of these hot entries [69, 73].

5.3 Sensitivity Studies

Slow-to-fast memory capacity ratios. Figure 12(a) evaluates the speedups offered by Trimma under different capacity ratios between the slow and fast memories, from 8:1 to 64:1. Note that when the ratio reaches 64:1, the baseline linear remap table would occupy the entire fast memory capacity and all data are stored in the slow memory, rendering substantial performance degradation. On average, the speedup of Trimma increases roughly proportionally to the capacity ratio, varying from 1.07× at 8:1 to 3.19× at 64:1. Higher capacity ratios result in larger linear remap tables in the baseline, while the iRT size does not change much. Therefore Trimma enables

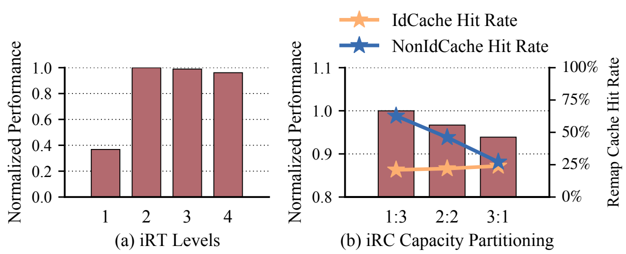


Figure 13: Performance comparison between (a) different iRT schemes (default is 2-level iRT); and (b) different iRC capacity ratios between NonIdCache and IdCache (default is 1:3). Performance is the geomean across all workloads.

more metadata space savings and thus allows more fast memory blocks to be used for application data.

Block sizes. In Trimma we use 256 B blocks as previous hybrid memory designs [74]. We also test other granularities and Figure 12(b) shows the relative performance over 256 B. The reason for low performance at 64 B is that it fails to effectively exploit the spatial locality in the evaluated workloads and thus achieves lower hit rates. This result is validated by previous work [74]. When using the 4 kB granularity as the OS page size, it generally has similar or even higher hit rates compared to 256 B, but the data over-fetching bandwidth consumption offsets the hit rate improvements, decreasing the overall performance by over 60%.

iRT configurations. Figure 13(a) shows the performance with more multi-level schemes of iRT normalized to the default 2-level design in Trimma, including one level of bit vector and one level of the original remap table (organized in blocks). The single-level design simply falls back to the basic linear remap table, without the bit vector. The four-level design is similar to Tag Tables [15], which slices the address tag part into four 6-bit chunks under 256 B blocks. While Tag Tables did not use the saved space for caching, we exploit this opportunity in this comparison. Even so, its performance is worse than Trimma. Although more levels provide more opportunities to use the intermediate level blocks for caching, we find that the increased metadata lookup cost and the additional complexity do not compensate for the metadata saving benefits, so the performance does not improve.

iRC capacity partition. We evaluate several iRC capacity partitioning schemes and the results are shown in Figure 13(b). Although storing some identity mapping entries helps the overall hit rate and system performance, borrowing too much space from the non-identity-mapping cache would hurt. As mentioned in Section 5.2, identity mapping entries tend to be cold, so we do not need a large space to cache too many of them. Eventually we adopt 25% space for identity mappings because it shows a high overall hit rate.

6 Related Work

Hybrid memory use modes. Flat-mode hybrid memory systems have larger OS-visible physical memory capacities to reduce page faults [7, 52, 60, 62, 66, 69, 73], while cache-mode systems provide cheaper data migration for better performance [9, 17, 20, 24, 29, 30, 50, 61, 72, 79]. This tradeoff inspires researchers to consider combining both modes. Chameleon [39] was the first design with flexible dual-mode support. It reused the unused space for caching, but required OS hints about data allocation. Trimma is instead software transparent and also exploits additional space saving opportunities. Hybrid2 [74] used a fixed caching space as a staging area for better migration. Baryon [47] supported data compression and sub-blocking for better capacity and bandwidth utilization. Stealth-persist [2] proposed user-defined cache/flat area, but it focused on persistence rather than performance.

Remap table design. Most hybrid memory systems managed by hardware use simple linear remap tables [39, 60, 69, 74]. Several DRAM cache designs [17, 29, 50, 53, 61] make the metadata *inlined* together with the data, e.g., in the same cacheline or DRAM row, so that a single access can fetch both. Such inlined metadata techniques are only applicable to direct-mapped or low-associativity systems, as prediction is required to decide a location before knowing the accurate metadata. Also, inlined metadata only hide the metadata lookup overhead but do not reduce their size.

Another line of work eliminates the metadata cost by integrating the physical-to-device address translation with the OS virtual-to-physical address mapping [1, 31, 35, 38, 49, 52, 62, 64, 76]. These designs require OS-level co-design and are less portable. Their performance is also limited in several aspects. First, the block size is restricted to coarse-grained 4 kB OS pages. Second, they only support epoch-based migration which cannot quickly adapt to program phase changes. Third, migration requires software involvement with significant interrupt and TLB shootdown overheads.

Tag Tables [15] also adopted a multi-level remap table. Trimma is different from it. First, Tag Tables worked only for DRAM caches while Trimma supports both modes. Second, Tag Tables did not make use of the saved metadata space as Trimma does to improve performance. Third, Trimma simplifies metadata management by fixing their locations, while Tag Tables followed the complicated OS allocation. Nevertheless, many strategies in Tag Tables like lazy expansion and compressed entries can also be applied to Trimma.

7 Conclusions

We propose Trimma, an indirection-based metadata structure and an efficient metadata cache design for hybrid main memory systems. Trimma integrates two techniques to address the metadata storage cost and lookup latency challenges. A multi-level remap table eliminates identical address mappings to save the metadata size in the fast memory. Performance is also improved when we utilize the saved space for extra data caching. We also propose a remap cache design that uses separate formats for non-identity and identity mapping entries, in order to increase the coverage and the hit rate. Overall, Trimma realizes scalable metadata management for future large-scale hybrid memory architectures.

Acknowledgments

The authors thank the anonymous reviewers and shepherd for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262). Mingyu Gao is the corresponding author.

References

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-tiered Main Memory. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 631–644. https://doi.org/10.1145/3037697.3037706

[2] Mazen Al-Wadi, Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2021. Stealth-Persist: Architectural Support for Persistent Applications in Hybrid Memory Systems. In *27th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 139–152. https://doi.org/10.1109/HPCA51647.2021.00022

[3] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic Local Alignment Search Tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410.

[4] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 14:1–14:25. https://doi.org/10.1145/3085572

[5] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619* (Aug. 2015).

[6] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 1–12. https://doi.org/10.1109/MICRO.2014.63

[8] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In *42nd International Symposium on Computer Architecture (ISCA)*. ACM, 198–210. https://doi.org/10.1145/2749469.2750387

[9] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *3rd Internatinoal Symposium on Memory Systems (MEMSYS)*. ACM, 268–280. https://doi.org/10.1145/3132402.3132404

[10] CXL Consortium. 2020. Compute Express Link Specification. https://www.computeexpresslink.org.

[11] CXL Consortium. 2022. Compute Express Link 3.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC)*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[13] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004), 5.

[14] Alex Fornito, Andrew Zalesky, and Michael Breakspear. 2013. Graph Analysis of the Human Connectome: Promise, Progress, and Pitfalls. *NeuroImage* 80 (2013), 426–444. https://doi.org/10.1016/J.NEUROIMAGE.2013.04.087

[15] Sean Franey and Mikko H. Lipasti. 2015. Tag Tables. In *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 514–525.

[16] Mel Gorman and Patrick Healy. 2008. Supporting Superpage Allocation without Additional Hardware Support. In *7th International Symposium on Memory Management (ISMM)*. ACM, 41–50. https://doi.org/10.1145/1375634.1375641

[17] Nagendra Dwarakanath Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 38–50. https://doi.org/10.1109/MICRO.2014.36

[18] David Gureya, João Neto, Reza Karimi, João Barreto, Pramod Bhatotia, Vivien Quéma, Rodrigo Rodrigues, Paolo Romano, and Vladimir Vlassov. 2020. Bandwidth-Aware Page Placement in NUMA. In *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 546–556. https://doi.org/10.1109/IPDPS47924.2020.00063

[19] Robert S Harris. 2007. *Improved Pairwise Alignment of Genomic DNA*. The Pennsylvania State University, USA.

[20] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 51–60. https://doi.org/10.1145/2628071.2628089

[21] Hybrid Memory Cube Consortium. 2013. Hybrid Memory Cube Specification 1.0.

[22] Intel. 2020. Intel Optane DC Persistent Memory. https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-telecom-use-case-workloads.pdf.

[23] Intel. 2023. 4th Gen Intel Xeon Processor Scalable Family, Sapphire Rapids. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[24] Hakbeom Jang, Yongjun Lee, Jongwon Kim, Youngsok Kim, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2016. Efficient Footprint Caching for Tagless DRAM Caches. In *22nd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 237–248. https://doi.org/10.1109/HPCA.2016.7446068

[25] JEDEC. 2021. DDR4 SDRAM Standard. https://www.jedec.org/standards-documents/docs/jesd-79-4a.

[26] JEDEC. 2022. DDR5 SDRAM Standard. https://www.jedec.org/standards-documents/docs/jesd-79-5b.

[27] JEDEC. 2023. High Bandwidth Memory (HBM3) DRAM. https://www.jedec.org/standards-documents/docs/jesd238a.

[28] JEDEC Solid State Technology Association. 2013. High Bandwidth Memory (HBM) DRAM. JESD235.

[29] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 25–37. https://doi.org/10.1109/MICRO.2014.51

[30] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *40th International Symposium on Computer Architecture (ISCA)*. ACM, 404–415. https://doi.org/10.1145/2485922.2485957

[31] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *44th International Symposium on Computer Architecture (ISCA)*. ACM, 521–534.

[32] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-Scale Video Classification with Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 1725–1732.

[33] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. 2004. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *10th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 288–299. https://doi.org/10.1109/HPCA.2004.10015

[34] Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. 2019. Graph-Based Genome Alignment and Genotyping with HISAT2 and HISAT-Genotype. *Nature Biotechnology* 37, 8 (2019), 907–915.

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

[36] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letter* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[37] Andi Kleen. 2005. A NUMA API for Linux. *Novel Inc* (2005).

[38] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–608. https://doi.org/10.1109/HPCA.2019.00012

[39] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *49th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 533–545. https://doi.org/10.1109/MICRO.2018.00050

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *26th Annual Conference on Neural Information Processing Systems (NIPS)*. 1106–1114.

[41] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 705–721.

[42] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.

[43] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[44] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *36th International Symposium on Computer Architecture (ISCA)*. ACM, 2–13. https://doi.org/10.1145/1555754.1555758

[45] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. 2014. CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures. *IEEE Trans. Comput.* 63, 9 (2014), 2187–2200. https://doi.org/10.1109/TC.2013.98

[46] Yongjun Lee, Jongwoo Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2015. A Fully Associative, Tagless DRAM Cache. In *42nd International Symposium on Computer Architecture (ISCA)*. ACM, 211–222. https://doi.org/10.1145/2749469.2750383

[47] Yiwei Li and Mingyu Gao. 2023. Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking. In *29th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 137–151. https://doi.org/10.1109/HPCA56546.2023.10071115

[48] Linux Kernel Development Team. [n. d.]. NUMA-aware Allocation. https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt.

[49] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *31st International Conference on Supercomputing (ICS)*. ACM, 26:1–26:10.

[50] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *44th International Symposium on Microarchitecture (MICRO)*. ACM, 454–464. https://doi.org/10.1145/2155620.2155673

[51] Sally A. McKee. 2004. Reflections on the Memory Wall. In *1st Conference on Computing Frontiers (CF)*. ACM, 162.

[52] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[53] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. 2012. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Computer Architecture Letter* 11, 2 (2012), 61–64. https://doi.org/10.1109/L-CA.2012.2

[54] Micron. 2023. Micron Launches Memory Expansion Module Portfolio to Accelerate CXL 2.0 Adoption. https://investors.micron.com/news-releases/news-release-details/micron-launches-memory-expansion-module-portfolio-accelerate-cxl.

[55] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association. http://www.usenix.org/events/osdi02/tech/navarro.html

[56] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 456–471.

[57] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. 2018. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?. In *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 271–282. https://doi.org/10.1109/HPCA.2018.00032

[58] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 347–360. https://doi.org/10.1145/3297858.3304064

[59] Yonathan Perez, Rok Sosic, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. 2015. Ringo: Interactive Graph Analytics on Big-Memory Machines. In *2015 ACM International Conference on Management of Data (SIGMOD)*. ACM, 1105–1110. https://doi.org/10.1145/2723372.2735369

[60] Andreas Prodromou, Mitesh R. Meswani, Nuwan Jayasena, Gabriel H. Loh, and Dean M. Tullsen. 2017. MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-Level Memories. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 433–444. https://doi.org/10.1109/HPCA.2017.39

[61] Moinuddin K. Qureshi and Gabriel H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *45th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 235–246. https://doi.org/10.1109/MICRO.2012.30

[62] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *25th International Conference on Supercomputing (ICS)*. ACM, 85–95. https://doi.org/10.1145/1995896.1995911

[63] Jeffrey B. Rothman and Alan Jay Smith. 2000. Sector Cache Design and Performance. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, 124–133. https://doi.org/10.1109/MASCOT.2000.876437

[64] Marta Rybczyńska. 2021. Top-Tier Memory Management. https://lwn.net/Articles/857133/.

[65] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *32nd International Conference on Supercomputing (ICS)*. ACM, 352–362.

[66] Jee Ho Ryoo, Mitesh R. Meswani, Andreas Prodromou, and Lizy K. John. 2017. SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 349–360. https://doi.org/10.1109/HPCA.2017.20

[67] Daniel Sánchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *40th International Symposium on Computer Architecture (ISCA)*. ACM, 475–486. https://doi.org/10.1145/2485922.2485963

[68] Julian Shun and Guy E. Blelloch. 2013. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 135–146.

[69] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 13–24. https://doi.org/10.1109/MICRO.2014.56

[70] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[71] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[72] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2019), 65:1–65:23. https://doi.org/10.1145/3293447

[73] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. LLC-Guided Data Migration in Hybrid Memory Systems. In *33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 932–942. https://doi.org/10.1109/IPDPS.2019.00101

[74] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2020. Hybrid2: Combining Caching and Migration in Hybrid Memory Systems. In *26th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 649–662. https://doi.org/10.1109/HPCA47549.2020.00059

[75] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *53rd International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[76] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 331–345. https://doi.org/10.1145/3297858.3304024

[77] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2018. ACCORD: Enabling Associativity for Gigascale DRAM Caches by Coordinating Way-Install and Way-Prediction. In *45th International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 328–339. https://doi.org/10.1109/ISCA.2018.00036

[78] Vinson Young and Moinuddin K. Qureshi. 2019. To Update or Not To Update?: Bandwidth-Efficient Intelligent Replacement Policies for DRAM Caches. In *37th IEEE International Conference on Computer Design (ICCD)*. IEEE, 119–128.

[79] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. 2017. Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation. In *50th International Symposium on Microarchitecture (MICRO)*. ACM, 1–14. https://doi.org/10.1145/3123939.3124555

[80] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *36th International Symposium on Computer Architecture (ISCA)*. ACM, 14–23. https://doi.org/10.1145/1555754.1555759

[81] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference (ATC)*. USENIX Association, 829–842.