A Case Study of a DRAM-NVM Hybrid Memory Allocator for Key-Value Stores

Minjae Kim, Bryan S. Kim, Eunji Lee, and Sungjin Lee

Abstract—As non-volatile memory (NVM) technologies advance, commercial NVDIMM devices have been made readily available for various computing systems. To efficiently utilize the high-density and high-capacity of NVM, the latest Xeon CPUs support a special *Memory Mode* that turns the DRAM into a last-level (L4) cache and uses NVM as the user-addressable system memory. Unfortunately, Memory Mode often provides low performance, even slower than when only NVM is used without any DRAM cache. According to our analysis, this is due to the inefficient management of a DRAM cache by the integrated memory controller, which results in high miss rates. This paper proposes a new hybrid memory allocator, called TARMAC. By employing intelligent yet lightweight memory management policies at the memory allocator level, TARMAC manages two different types of memory devices more efficiently, achieving 37% higher cache hit rate, 67% higher throughput, and 40% shorter memory latency than the hardware-based Memory Mode, on average. TARMAC exposes memory interfaces compatible with traditional memory allocators, enabling existing software to use TARMAC without any manual modification.

Index Terms—Non-volatile memory, memory performance analysis, memory allocator

1 Introduction

The rise of non-volatile memory (NVM) device technologies (e.g., Optane NV-DIMM [1]) has led to various opportunities for optimizing computer systems. Many have attempted to improve system performance by exploiting the persistence of NVM devices – for example, by using them as a non-volatile cache or a logging device [2] for slow storage. However, far less attention has been paid to using NVM devices as a natural extension of volatile DRAM, which benefits many applications (e.g., in-memory KV stores and databases) by enabling them to run larger working sets at a lower cost. A few prior studies exist, but they were mainly conducted under simulated/emulated NVM devices with unrealistic settings [3] or required significant modification of applications, lacking the generality to apply to richer applications [4]. This paper presents a new memory management technique that increases a system memory capacity using real-world NVM devices in a performance-efficient and transparent manner.

Non-volatile memory DIMM (NV-DIMM) modules are populated with DRAM-DIMM modules in a machine. To manage heterogeneous devices, the latest Xeon CPUs offer two types of modes: *App Direct Mode* (AppMode) and *Memory Mode* (MemMode) [1]. When AppMode is chosen, two memory devices are directly exposed to user space. Applications can freely allocate and release memory space in both devices, making use of large and persistent NVM space optimally for their purposes. To take full advantage of AppMode, however, developers must possess a deep understanding of software architectures and modify the source code manually. Conversely, MemMode offers more convenience for developers. In MemMode, DRAM behaves like a last-level cache (L4 cache), while NVM replaces DRAM and is used as system memory. Integrated memory controllers (iMC) in a CPU manage two devices transparently, caching popular data in faster DRAM and keeping other data in slower NVM. Therefore, applications can use huge system memory without any code modification.

Despite this convenience gain, applications running on MemMode (i.e., a DRAM cache plus NVM) often suffer from severe performance drops, showing lower throughput and longer latency even compared to when only NVM is used without a DRAM cache. According to our analysis (see Section 2), this poor performance is due to high DRAM cache miss ratios which result from the limitation of a direct-mapped cache management policy implemented in the memory controller. Even worse, it is technically difficult to put more advanced policies into the controller because of its high design complexity [5].

To address the inherent limitation of MemMode and to enable a broader range of applications to use large NVM as working memory space, we propose a new memory allocator, Tarmac. Tarmac is based on AppMode, thus managing the two devices separately. Similar to MemMode, however, it exposes an NVM space as allocatable memory, hiding DRAM from users. Tarmac uses fast DRAM as a cache for slow NVM, but by using intelligent caching algorithms, it greatly increases the hit ratio of the DRAM cache. In this regard, the proposed Tarmac is like a software-driven MemMode that overcomes the inefficiency of its hardware counterpart. Tarmac is implemented as a user-level C++ library that exposes memory APIs which in turn creates pseudo heap space growing to several terabytes. Tarmac APIs are compatible with traditional heap allocators. Thus, existing C++ codes can be automatically transformed to use Tarmac. This allows providing almost the same level of convenience as existing MemMode.

A key technical challenge in designing Tarmac is minimizing software overhead. Read and write latencies of Optane NV-DIMM are longer (e.g., 150∼300 ns for reads and 100 ns for writes) than those of DRAM (e.g., 70∼100 ns) [6], but the gap is not so wide. Although idle write latency is less than read latency thanks to the DDR-T protocol of Optane DCPMM, practical write performance is worse than read performance due to low bandwidth. This implies that inefficient implementation of a software-based memory allocator may nullify the benefit of a high hit ratio over a hardware-based one. Tarmac resolves those issues by (*i*) using lightweight chunk-based indexing; (*ii*) deferring DRAM-NVM migration jobs to background threads; and (*iii*) minimizing synchronization costs between the user and background threads.

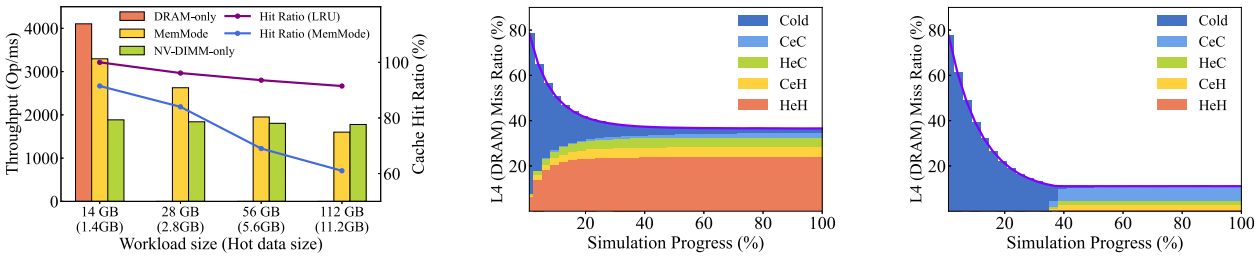
We carry out a preliminary case study using a simple key-value benchmark that runs on a Xeon server with 16 GB DRAM and 85 GB NV-DIMM. Our results using YCSB [7] show that Tarmac achieves a 37% higher cache hit ratio, provides 67% higher throughput, and offers 40% lower latency over MemMode, on average.

2 Performance Analysis of MemMode

In this section, we present our experimental results using real NV-DIMM devices under memory-intensive workloads and explain why MemMode has relatively low performance.

- *Minjae Kim and Sungjin Lee are with the Department of Electronic Engineering & Computer Science, DGIST, Daegu 42998, South Korea. E-mail: {jwya2149, sungjin.lee}@dgist.ac.kr.*
- *Bryan S. Kim is with the Department of Electrical Engineering & Computer Science, Syracuse University, Syracuse, NY 13244 USA. E-mail: bkim01@syr.edu.*
- *Eunji Lee is with the Department of Smart System Software, Soongsil University, Seoul 06978, South Korea. E-mail: eunjicious@gmail.com.*

Manuscript received 2 May 2022; revised 5 July 2022; accepted 27 July 2022. Date of publication 9 August 2022; date of current version 8 September 2022.
This research was supported by the MOTIE (Ministry of Trade, Industry & Energy under Grant 1415181081 and KSRC (Korea Semiconductor Research Consortium) under Grant 20019402 support program for the development of the future semiconductor device, the Samsung Research Funding Incubation Center of Samsung Electronics under Grant SRFCIT1902-03, and the NRF grant funded by the Korea government (Ministry of Science and ICT) under Grants NRF-2018R1A5A1060031 and NRF-2019R1A2C1090337.
(Corresponding author: Sungjin Lee.)
Digital Object Identifier no. 10.1109/LCA.2022.3197654



(a) Hit ratio and throughput (b) Direct-mapped cache simulation (c) LRU cache simulation

Fig. 1. Performance analysis of MemMode. DRAM-DIMM is 16 GB, and NV-DIMM is configured to be 128 GB.

Experimental Setup. We conduct a set of experiments using a commercially available NV-DIMM module, Intel's Optane DCPMM. Our experimental system is a dual-socket Xeon server platform equipped with 20-core Cascade Lake R CPUs. Each socket has 2 iMCs and 6 memory channels, with a total of 12 DIMM slots. To eliminate the influence of remote memory access and interleaving, we use only one socket with one 256 GB Optane NV-DIMM and one 16 GB DDR4 DRAM-DIMM.

To understand how efficiently the iMC manages DRAM as a last-level (LL) cache for NVM, we use a micro-benchmark that loads different amounts of data – 14 GB, 28 GB, 56 GB, and 112 GB each – and performs 64-byte reads on the loaded data. To mimic localized workloads, the benchmark generates reads with a hotspot of 90:10 locality (i.e., 90% of reads are destined to 10% of the hot data). We measure read throughputs and DRAM cache hit ratios using Intel's VTune Profiler. We compare MemMode with DRAM-only and NV-DIMM-only setups that solely use DRAM and NVM as system memory, respectively.

Experimental Results. Fig. 1a shows results. When the loaded data is small (e.g., 14 GB), the majority of the data is cached in DRAM, so a high hit ratio – 91% – can be achieved. The read throughput of MemMode is also close to that of the DRAM-only mode. As the size of the loaded data increases, however, performance drops are observed. The 28 GB, 56 GB, and 112 GB data sets are larger than the DRAM capacity (i.e., 16 GB), but this DRAM is large enough to accommodate all hot data (i.e., 2.8 GB, 5.6 GB, and 11.2 GB) of the given data sets. While high hit ratios close to 90% are expected, the measured hit ratios are much lower: 84%, 69%, and 61%, respectively. For the 112 GB data set, the read throughput is lower than that of the NV-DIMM-only setting.

The above results tell us that the iMC does not operate efficiently and often fails to cache popular data in DRAM. As the data size gets larger, it is also speculated that the iMC causes a considerable amount of useless data movement between DRAM and NVM, which, in turn, degrades overall system performance. To confirm our hypotheses, we estimate hit ratios using an LRU simulator (explained later) for the same set of memory references. As depicted in Fig. 1a, the LRU exhibits hit ratios above 90% for all workloads.

Cache Miss Analysis. In an attempt to understand in detail why MemMode performs poorly, we develop a functional cache simulator that models the L4 cache (i.e., DRAM) of MemMode. In MemMode, the L4 cache is organized with 64-byte cache blocks managed by a direct-mapped replacement policy in the iMC [6]. For fast evaluation, our simulator models the L4 cache and NV-DIMM in the memory hierarchy and accepts memory references missed on L3 as an input.

Fig. 1b displays L4-cache miss ratios over time. We use a memory trace with 2,048M 64-byte block reads with 90:10 locality. The input data set covers the memory space of 128 GB, but only 10% (i.e., 12.8 GB) of the space is frequently referenced. We categorize cache misses into five types: Cold, CeH, HeC, HeH, and CeC. Cold represents a cold miss that occurs when a cache block to access is empty. The rest are caused by either a conflict or a capacity miss. CeH is a cache miss where a cold block in the cache is evicted by an incoming hot block, while HeC occurs when a hot block is replaced by an incoming cold block. HeH and CeC are cases where a hot (or cold) block replaces a hot (or cold) block in the cache.

The number of Cold misses declines over time. But, the number of HeH misses continues to increase, and its miss ratio converges to 24%. As the 16 GB L4 cache is large enough to accommodate all hot blocks (i.e., 12.8 GB), this high miss ratio might be due to conflict misses. This is also the reason why MemMode in Fig. 1a shows a low hit ratio. In MemMode, hot data competed with each other, resulting in a significantly high DRAM cache miss ratio. Next, we conduct experiments with the LRU simulator where conflict misses never occur. As shown in Fig. 1c, the number of HeH misses becomes almost zero, and the L4 miss ratio is significantly lower than Fig. 1b. This also confirms that a low cache hit ratio of MemMode is attributed to its architectural limit based on the direct-mapped cache.

The most effective way to address the above problems is to adopt advanced replacement policies in the controller. Implementing a set-associative or a fully-associative policy in hardware, however, is unfeasible for a large DRAM cache because they require large SRAM for tag storage or involve extra DRAM references if tags are stored in DRAM [5]. Another option is using AppMode, which allows direct management of NV-DIMM through software libraries like PMDK[8]. However, this option requires manual modification of applications. Those fundamental limitations of the two approaches comprise the basis for our proposal of TARMAC.

3 DESIGN AND IMPLEMENTATION OF TARMAC

Fig. 2 shows the design of TARMAC and its components with the virtual address space. TARMAC is implemented in the form of a user-level library. As opposed to typical memory allocators, TARMAC decides the best memory device to place data in, and if necessary, moves the data from one device to another by accounting for its properties (e.g., locality). These features are transparently implemented behind APIs that are similar to those of existing heap allocators (see Section 3.1).

TARMAC is based on AppMode, so it directly manages two memory devices. Like typical allocators, TARMAC uses standard heap APIs to allocate and free space in DRAM. The portion of DRAM space is

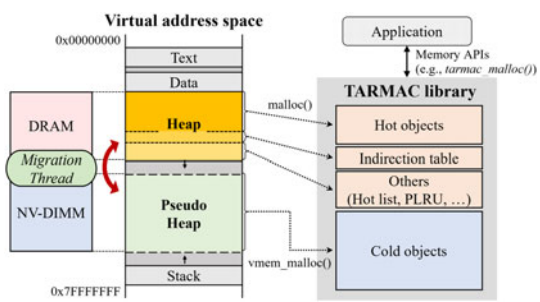


Fig. 2. Overall design of TARMAC.

```
1  char* A = (char*)malloc(8192);
2
3  memcpy(A, src, 8192);
4  for (int i=0; i<8192; i++)
5       src[i] = A[8191−i];
6  free(A);
```

```
7  TARMAC<char> A
8       ((char*)tarmac_malloc(8192));
9  memcpy(A, src, 8192);
10 for (int i=0; i<8192; i++)
11      src[i] = A[8191−i];
12 tarmac_free(A);
```

Fig. 3. A comparison of original and converted codes.

mapped to a heap region on the virtual address space. Conversely, TARMAC uses Intel's PMDK library to manage NV-DIMM. PMDK exports NVM space via Linux's DAX which is then mapped to the virtual address space by `mmap()`. This not only enables us to create a huge pseudo heap space in the application, but also to directly manage NVM without interference from the page cache.

In light of the relatively small latency gaps between DRAM and NVM, key features of TARMAC must be implemented efficiently. TARMAC employs a chunk-based indexing algorithm to index objects over the two devices (see Section 3.2). It is designed to consume a small amount of DRAM to index objects and to minimize extra memory references when accessing objects. To identify hot and cold objects, TARMAC uses a simple pseudo LRU policy, which is combined with a migration policy that moves hot and cold data from one device to another in background at a low synchronization cost (see Section 3.3).

3.1 TARMAC Interface: Class and APIs

We explain how target codes can be converted and combined with the TARMAC library. Fig. 3 compares the original codes and the converted ones. During the preprocessing time of the compilation, typical heap memory APIs are replaced with those from TARMAC. For example, `malloc()` is replaced with `tarmac_malloc()`, which allocates the required memory space, and creates a new TARMAC object (see line 7-8). `tarmac_free()` is invoked to destroy the TARMAC object (see line 12). A TARMAC object is based on a class template to support built-in and user-defined data types (see line 7-8). Many other functions (e.g., `memcpy()`) and operators (e.g., '`[]`', '=', and '+') are automatically substituted during the compile time by leveraging C++ polymorphism and overloading. For now, we manually convert the original code. We will develop an automatic code conversion tool using LLVM [9].

3.2 Management of Heterogeneous Memory Space

Behind the interface, TARMAC manages the memory space of the two devices as illustrated in Fig. 4. All TARMAC objects are initially considered cold and thus stored in NVM. When `tarmac_malloc()` is invoked to create a new object, TARMAC internally calls `vmem_malloc()` from PMDK to allocate continuous memory space in NVM. Therefore, an 8-byte *direct pointer* is enough to manage memory space for an object. Every memory allocation is aligned at a 64-byte cache-line boundary (see ① in Fig. 4).

A hot object kept in NVM should be cached in faster DRAM. For large objects, TARMAC splits the allocated NVM space into smaller chunks and promotes only a hot chunk(s) to DRAM (②). This approach, however, requires maintaining an indirection table as chunks are not continuously allocated anymore and scattered across the two devices. Moreover, for performance, an indirection table must be always held in DRAM. To selectively cache only hot data in DRAM but to reduce a table size, the largest chunk size is decided to be 256 bytes. Given a 704-byte object, it is split into three chunks, 256, 256, and 192 bytes. The indirection table has 3 entries, each of which is an 8-byte *indirect pointer* to chunk data stored in either DRAM or NVM. A direct pointer is then updated to locate the indirection table. The original address is saved in an 8-byte *backup pointer* that is used when the object moves back to NVM.

For caching hot chunks, TARMAC must allocate space in DRAM. To avoid fragmentation and extra costs caused by frequent invocations

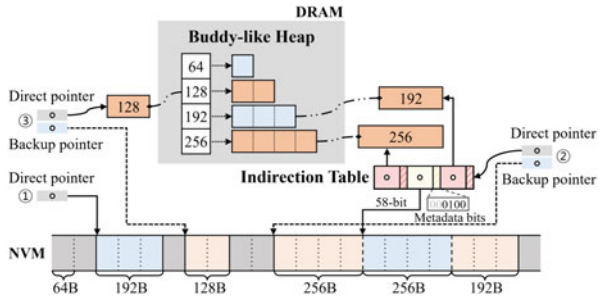


Fig. 4. Memory management of TARMAC.

of `malloc()` and `free()` in the DRAM heap area, TARMAC reserves large heap space (that grows and shrinks later) and manages it using a buddy-like allocator. The allocator maintains memory blocks of four different sizes – 64, 128, 192, and 256 bytes – and chooses the best-fit one. Each block is aligned at a 64-byte boundary.

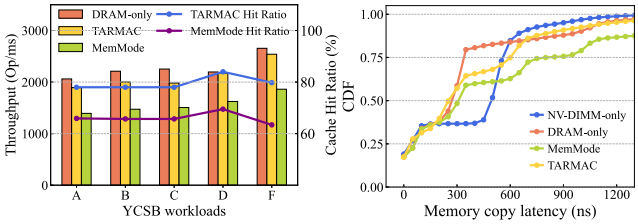
For a hot object smaller than 256 bytes (i.e., the largest chunk size), TARMAC migrates the entire object data to DRAM. The direct pointer points to its new location in DRAM directly, and the backup pointer keeps an original address (③). No indirection tables are needed. A direct pointer points to one of three different areas (i.e., a DRAM heap, pseudo NVM heap, or table area in Fig. 2). Since they are allocated in different ranges of virtual memory space, TARMAC can identify which area a direct pointer points to by checking its address number.

TARMAC can directly access chunk data without extra memory operations if a direct pointer locates a DRAM heap or pseudo NVM heap area. However, if a direct pointer indicates an indirection table, TARMAC has to perform address translation using a relative object address to access. TARMAC uses the higher 58-bit of the object address as an index for a table to get a memory address where a corresponding chunk is stored. The lower 8-bit is used as an offset within the chunk. The use of indirection tables causes space and time overheads, but these are not significant. TARMAC requires an indirection table only for a large object with hot chunks in DRAM, and the space for indirection tables can be reduced when no chunk of an object is in a DRAM heap. The size of indirection tables in Section 4 is less than 1 GB, and looking up indirection tables degrade performance at most 4%. Moreover, since a large object is likely to be referenced sequentially, the overhead of looking up the table can be diminished through CPU caching and prefetching.

TARMAC should keep track of hot/cold chunks and support efficient data migration between the devices. TARMAC uses 4-bit metadata, *hot*, *reference*, *busy*, and *modify* bits, to identify hot/cold chunks and control data migration. DRAM and NVM space is allocated to align with 64 bytes, so the lower 6-bit of a direct pointer or an indirect pointer in a table is never used. The 4-bit metadata can be stored within a pointer, which saves memory for metadata. It also reduces extra memory accesses when updating metadata since a pointer is already referenced to access data and thus is likely to be cached in a CPU.

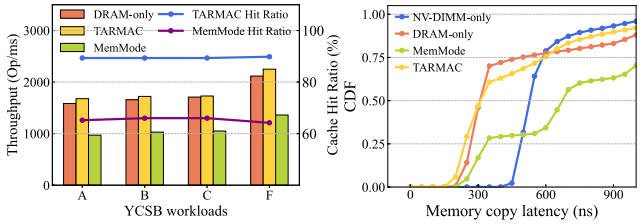
3.3 Data Migration Between DRAM & NVDIMM

Whenever a memory API (e.g., `memcpy()`) is called, TARMAC updates object's metadata to promote/demote chunks to/from DRAM. If a memory chunk referenced is located in slow NVM, TARMAC checks its *hot bit* in the direct or indirect pointer of that chunk (maybe in a CPU cache). If the hot bit is unset, TARMAC sets it to '1' and adds the chunk to a *hot list* so that it is promoted to DRAM later. The hot list is implemented as a lock-free queue and requires one atomic store operation when a new item is being inserted. If the chunk is in DRAM, TARMAC sets a *reference bit* that is used to detect cold chunks in DRAM. To hide user-perceived delay, the movement of chunks is deferred and performed by a separate *migration thread*.



(a) Throughput and miss ratio (b) Latency (YCSB-C)

Fig. 5. Experimental results in YCSB workloads (vanilla).



(a) Throughput and miss ratio (b) Latency (YCSB-C)

Fig. 6. Experimental results in YCSB workloads (hotspot).

The migration thread regularly scans the hot list in background and copies a candidate chunk to DRAM. After the migration, the chunk is added to a *Pseudo-LRU* (*PLRU*) *list* that manages DRAM-cached chunks. The hot bit of the chunk is then unset. When a given DRAM space is exhausted, TARMAC evicts relatively cold chunks to make a room in DRAM. To find a cold chunk, a variant of CLOCK algorithms is used that moves a clock hand, changing a reference bit of a chunk to '0' from '1'. If it finds a chunk with a bit of '0', that chunk is evicted.

Two race conditions may occur while moving a chunk from one device to another: (*i*) the first is when the migration thread attempts to move a chunk that is being modified by a user thread; (*ii*) the second is when a user thread attempts to modify a chunk while the migration thread is moving it. The first case is handled by using a *busy* bit. A user thread sets a busy bit before updating a chunk and resets it before returning to the user code. If the busy bit is set, the migration thread spins and waits for the user thread to finish. The second is resolved by using a *modify* bit. The migration thread resets a modify bit to '0' before moving data. Conversely, a user thread sets the modify bit to '1' before updating a chunk. After copying data, the migration thread sees if the modify bit is converted to '1'. If so, the user thread modified the chunk while it was being copied. In that case, the migration thread cancels the migration and repeats copying the data until it succeeds. This policy reduces migration throughput but have no effect on user-perceived latency, as a user thread only needs to update two bits (probably in CPU caches) without waiting or retrying.

4 Experimental Results

Experimental Setup. We conduct a preliminary case study using a simple key-value store with the YCSB benchmark. Our key-value store allocates a set of objects in a load phase and accesses objects in a run phase. We configure YCSB to generate two memory reference patterns with different localities: (*i*) a vanilla configuration that follows a zipfian distribution for YCSB-A/B/C/F and a latest distribution for YCSB-D; (*ii*) a hotspot configuration that uses a hotspot distribution of 90:10 locality for YCSB-A/B/C/F. The object sizes follow a generalized Pareto distribution between 32-byte and 2048-byte to describe real-world workloads [10].

Experimental Results. Figs. 5 and 6 display results. Fig. 5 shows the throughput, hit ratios and latencies of YCSB with the vanilla configuration, and Fig. 6 shows the results with the hotspot configuration. DRAM – only is a system with 85 GB DRAM without NV-DIMM. MemMode and TARMAC use a system with 16 GB DRAM and 85 GB NV-DIMM. MemMode manages the two devices using Memory Mode, while TARMAC employs the proposed techniques to manage them.

TARMAC exhibits 35% higher throughput with 20% higher hit ratios than MemMode (see Fig. 5a). As depicted in Fig. 6a, TARMAC performs better under the hotspot distribution, showing 67% higher throughput with 37% higher hit ratios than MemMode. Oddly, TARMAC exhibits even higher performance than DRAM – only. This anomaly is due to the high TLB miss ratio of DRAM – only which is caused by repetitive accesses to hot objects scattered in a wide memory space. Because each hot object is much smaller than a 4 KB page, access to the scattered hot objects results in frequent TLB misses. In contrast, TARMAC collects hot objects in DRAM, so TLB can efficiently keep translation entries for hot pages clustered in the DRAM heap. This phenomenon applies equally to the vanilla distribution (Fig. 5a), but is more evident in the hotspot distribution (Fig. 6a) where all objects are clearly divided into hot and cold.

To analyze the difference in behavior between TARMAC and MemMode, we measure the latency of memory copies during the experiments (see Figs. 5b and 6b). DRAM – only exhibits the best performance, while NV – DIMM – only shows the worst. MemMode has fairly short latency when data is served by DRAM, but it cannot avoid a high penalty on latency when DRAM-cache misses happen. In TARMAC, more data than MemMode are served by DRAM thanks to higher hit ratios. As a result, TARMAC offers shorter average latency than MemMode.

5 Conclusion

This paper proposed the new memory allocator, TARMAC, for DRAM-NV-DIMM hybrid systems. TARMAC employed advanced DRAM-cache management policies that achieved a higher cache hit ratio than MemMode. TARMAC was designed to provide transparent memory interfaces so that existing applications took advantage of such a high hit ratio without any code modification. Our results showed that TARMAC exhibited a 37% higher hit ratio, providing a 67% higher throughput and 40% shorter memory latency than MemMode.

References

[1] Intel, "The challenge of keeping up with data," 2019. [Online]. Available: https://intel.ly/3ABQiuV
[2] T. Yao et al., "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2020, pp. 17–31.
[3] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Trans. Comput.*, vol. 69, no. 9, pp. 1401–1413, Sep. 2020.
[4] Intel, "PMEM-Redis: A version of Redis that uses persistent memory," 2018. [Online]. Available: https://github.com/pmem/pmem-redis
[5] V. Young, Z. A. Chishti, and M. K. Qureshi, "TicToc: Enabling bandwidth-efficient DRAM caching for both hits and misses in hybrid memory systems," in *Proc. IEEE 37th Int. Conf. Comput. Des.*, 2019, pp. 341–349.
[6] J. Izraelevitz et al., "Basic performance measurements of the Intel Optane DC persistent memory module," 2019, *arXiv: 1903.05714*.
[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[8] Intel, "Persistent memory development kit," 2019. [Online]. Available: https://pmem.io/
[9] LLVM Developer Group, "The LLVM compiler infrastructure," 2003. [Online]. Available: https://llvm.org/
[10] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–224.