Hybrid^2: Combining Caching and Migration in Hybrid Memory Systems

Evangelos Vasilakis
Chalmers University of Technology
evavas@chalmers.se

Vassilis Papaefstathiou
Foundation for Research and Technology Hellas
papaef@ics.forth.gr

Pedro Trancoso
Chalmers University of Technology
ppedro@chalmers.se

Ioannis Sourdis
Chalmers University of Technology
sourdis@chalmers.se

ABSTRACT

This paper considers a hybrid memory system composed of memory technologies with different characteristics; in particular a small, near memory exhibiting high bandwidth, i.e., 3D-stacked DRAM, and a larger, far memory offering capacity at lower bandwidth, i.e., off-chip DRAM. In the past, the near memory of such a system has been used either as a DRAM cache or as part of a flat address space combined with a migration mechanism. Caches and migration offer different tradeoffs (between performance, main memory capacity, data transfer costs, etc.) and share similar challenges related to data-transfer granularity and metadata management. This paper proposes *Hybrid*^{2}, a new hybrid memory system architecture that combines a DRAM cache with a migration scheme. *Hybrid*^{2} does not deny valuable capacity from the memory system because it uses only a small fraction of the near memory as a DRAM cache; 64MB in our experiments. It further leverages the DRAM cache as a staging area to select the data most suitable for migration. Finally, *Hybrid*^{2} alleviates the metadata overheads of both DRAM caches and migration using a common mechanism. Using near to far memory ratios of 1:16, 1:8 and 1:4 in our experiments, *Hybrid*^{2} on average outperforms current state-of-the-art migration schemes by 7.9%, 9.1% and 6.4%, respectively. In the same system configurations, compared to DRAM caches *Hybrid*^{2} gives away on average only 0.3%, 1.2%, and 5.3% of performance offering 5.9%, 12.1%, and 24.6% more main memory capacity, respectively.

1. INTRODUCTION

The performance of computer systems is largely dominated by their memory hierarchy [1]. Besides latency, memory bandwidth can be a limiting factor for many workloads running on Chip Multiprocessors (CMPs) [2–5]. On one hand, data intensive applications as well as the large number of cores and specialized accelerators integrated on a chip increase the demand for higher data rates. On the other hand, memory bandwidth is pin limited [2, 6] and is therefore more difficult to scale [5].

3D-stacking technology can be used to increase memory bandwidth. In particular, 3D-stacked DRAM can be placed near the processor die offering substantially higher bandwidth. This near memory (NM) has limited capacity and often needs to be complemented with a larger far memory (FM) such as an off-chip DRAM that has however lower bandwidth. How to best exploit the NM bandwidth and the FM capacity to maximize system performance is still an open problem. Currently, there are two dominant approaches: the first one uses NM and FM as a hybrid, flat address-space memory system supporting migration between the two [7–12]; the second one uses NM as DRAM cache and FM as the main memory [13–28].

In general, DRAM caches copy data from FM to NM, as opposed to migration schemes that swap data between NM and FM. This results in a number of differences between the two approaches. Firstly, caching takes the NM capacity away from the memory system, as opposed to migration where NM capacity is part of a flat address space; this may have a significant impact on capacity limited workloads [29]. Secondly, swapping incurs double the overheads of copying. To amortize the overheads of swapping, migration schemes try to migrate only data with potential for future reuse. To detect this potential, migration schemes need to observe the data access patterns over time and predict which data are more beneficial to migrate to NM. This makes migration slower to adapt to changes in the working set of applications compared to caches that always bring in requested data.

Despite their differences, caching and migration share some common challenges. One of them is the trade-off between *data granularity* and *metadata overheads*. The smaller the cachelines the larger the tag array and vice versa; similarly, the smaller the size of a migration block the larger the metadata required to keep track of the remapping. In effect, the size of the tag-array and remapping metadata impacts their management overheads and as a consequence performance. Another trade-off, common for caching and migration, is that coarser granularity of cachelines and migration blocks can benefit workloads with high spatial locality, but may hurt workloads with poor spatial locality due to over-fetching.

Finer data granularity has lower over-fetching risk, but may not exploit equally well spatial locality.

In this work we propose *Hybrid*^{2}, a new approach for utilizing a 3D-stacked DRAM that aims to preserve the advantages of both caching and migration as well as at minimizing their overheads. *Hybrid*^{2} employs a small portion of the 3D-stacked DRAM to implement a DRAM cache and offers the rest of its *capacity* to main memory. Besides preserving most of the NM capacity, the small DRAM cache size allows its tag array to fit entirely on-chip, thereby *reducing access latency*. The on-chip tag array is extended to include the remapping metadata required for data migration. This minimizes both the DRAM cache and migration metadata overheads. The DRAM cache quickly adjusts to the working set of the workload by fetching all requested data to the NM. Migrations are decided upon eviction from the DRAM cache which allows observing the access patterns in the DRAM cache in order to make informed migration decisions. Finally, the data of the DRAM cache can be located anywhere in the NM through the use of indirection, therefore, data selected to be kept in the NM after eviction from the cache do not require relocation within NM, *avoiding unnecessary traffic*.

Concisely, *Hybrid*^{2} makes the following contributions:

- proposes a new hybrid memory architecture that combines caching and migration in the 3D stacked DRAM.
- Alleviates the latency and traffic overheads of both DRAM cache tag lookups and data migration address remapping by using the same mechanism.
- outperforms state-of-the-art migration schemes by using a small part of the 3D-stacked DRAM as a cache which quickly adapts to working set changes.
- closely matches the performance of state-of-the-art DRAM caches and in memory-intensive workloads outperforms them, while offering almost all of the 3D-stacked DRAM capacity to the flat memory address space.

The remainder of this paper is organized as follows: Section 2 presents related work and some motivating results for our proposed design. Section 3 describes the *Hybrid*^{2} architecture. Section 4 explains our experimental setup. Section 5 offers our evaluation results and comparison. Finally, Section 6 summarizes our conclusions.

2. RELATED WORK AND MOTIVATION

Various memory technologies exhibiting different tradeoffs in terms of capacity, bandwidth, access latency, and cost [30–35]. A promising direction towards a more efficient memory system design is to combine multiple technologies with complimenting characteristics in a hybrid memory system. One common hybrid approach is to put together a high-bandwidth 3D-stacked DRAM and a high-capacity conventional, off-chip DRAM, the first one denoted as near memory (NM) and the second as far memory (FM). Recent advances in 3D-stacking made it possible to place 3D-stacked DRAM close to the processor and thus provide substantially higher bandwidth than traditional DDR busses [30, 31]. Although the latest 3D-stacked DRAM memories can offer capacities that reach up to 24GB per stack (i.e. HBM2E with 12-High stack) they have very high-cost. Therefore, conventional, off-chip, lower bandwidth DRAM is added to provide the missing capacity. Currently, there are two dominant research directions for best exploiting the high bandwidth of an NM and the capacity of an FM. The first approach uses the NM as a DRAM cache and FM as the main memory; the second one combines NM and FM in a flat address space and supports migration between them.

2.1 Related work on DRAM caches

Most work on DRAM caches focuses on minimizing the tag lookup overheads and on achieving a good balance between cache line size and tag management complexity. Loh and Hill proposed storing tags in DRAM and using compound accesses so the data access is always a row buffer hit, they use 64 Byte cache lines and adjust associativity to fit a set in a single DRAM row [13]. Alloy cache uses a direct mapped design with 64 Byte cache lines and collocates the tag along with the data to access with a single burst [26]. The Tagless DRAM Cache is on the other side of the cache line size spectrum, it uses 4 KByte cache lines and minimizes the tag lookup overheads by using the OS page tables and TLBs to track the DRAM cache contents [22]. These approaches are scalable but limit the design options of DRAM caches [24].

Some less restrictive designs handle the tag management using resources on the processor die. ATCache, uses a small on-chip cache for the DRAM cache tags, which are located in DRAM, to absorb most tag lookups [20]. The Decoupled Fused Cache also keeps the DRAM cache tags in DRAM and re-organizes the tag array of the on-chip LLC to store information about the contents of the DRAM cache [24]. These designs are more generic and allow various cache line sizes, however selecting the right cache line size for a DRAM cache comes with its own tradeoffs.

In general, small cache lines come with higher tag overheads, but use cache space more efficiently. Large cache lines reduce the tag lookup overheads but may lead to over-fetching. Footprint cache tackles the overfetching problem of large cache lines with on-chip tags fetching only the blocks that are predicted to be used [25]. Unison Cache follows the same approach, but stores the tags in DRAM for scalability to larger cache sizes [15]. Finally, Footprint Tagless DRAM Cache combines the Tagless with the footprint design [21]. The above approaches achieve a good balance between tag lookups and over-fetching. However, they may underutilize the DRAM cache space when only small parts of each cache line are fetched.

Another issue of DRAM caches is the unbalanced use of memory bandwidth. Off-chip DRAM bandwidth is only used for serving cache misses and writebacks rather than processor memory accesses. Banshee uses the TLBs to track DRAM cache contents and proposes a bandwidth-aware frequency-based replacement policy [36] to balance bandwidth utilization. BATMAN monitors the number of accesses to both 3D stacked DRAM and conventional DRAM and regulates data movement [37]. Finally, BEAR proposes mechanisms to limit the bandwidth used by secondary operations of DRAM caches [23].

Intel's Knights Landing provides the option to split the MCDRAM (NM) between DRAM cache and flat address space, however, it does not support transparent data migration in HW. Instead it moves the burden to the software to explicitly

allocate data to NM through the hbw_malloc() function [38].

2.2 Related work on data migration

As opposed to DRAM caches, data migration makes 3D-stacked DRAM capacity available to the system. Moreover, it has the potential to utilize the bandwidth of all memories for serving memory requests. As such, data migration can potentially reap the benefits of both higher aggregate bandwidth and capacity by migrating data between the 3D-stacked DRAM and conventional DRAM dynamically. Data migration schemes come in different flavors when it comes to granularity of migrated data, flexibility, and data selection.

Some early work utilizes the OS, with some hardware support, to select the data that would be more beneficial to migrate to 3D-stacked DRAM [39]. On one hand, involving the OS improves the selection of data to migrate and allows the use of page tables for tracking the remapped data. On the other hand, OS-schemes have slow response to working set changes, incur high overheads, and limit the granularity of migrating data to that of an OS page.

As an alternative, hardware mechanisms can respond faster and support more migration granularities, but need to handle the address remapping in hardware in order to remain transparent from the OS. The migration granularity affects the address remapping overheads. A way to alleviate the remapping overheads is to divide memory in congruence groups and allow migration only within a group, like in CAMEO [29]. PoM follows the same group approach with 2 KByte granularity segments. It further uses competing counters in every segment group and a sampling approach to dynamically adjust migration thresholds [7]. Chameleon is based on PoM with the added option to economize on migration bandwidth when the software does not use some memory space [8]. Chameleon requires changes to the operating system and in the Instruction Set Architecture (ISA). Although group-based approaches support fine-granularity at lower cost, they do not perform well for lower ratios of 3D-stacked to off-chip DRAM.

To overcome the limitations of the group-based approaches, some designs choose to offer more flexibility at coarser migration granularity. Mempod opts for all-to-all migration for higher flexibility [9]. It uses the Majority Element Algorithm to identify 2 KByte blocks to migrate to 3D-stacked DRAM in short time intervals [40]. LGM leverages the spatial locality of data in the LLC to select 2 KByte segments for migration, additionally, it economizes migration bandwidth by not migrating cache lines that are present in the LLC, instead they are marked as dirty and written back on eviction. PageSeer proposed using the Memory Management Unit (MMU) to prefetch pages from Non Volatile Memory to conventional DRAM [12]. SILC-FM presents a more flexible group approach [11], it uses set-associative swap-groups and migrates 2 KByte blocks, but allows sub-blocks to interleave data in the 3D-stacked DRAM.

2.3 Motivation

As described above, both DRAM caches and data migration schemes come with their own advantages and limitations. The key difference between DRAM caches and data migration comes from the fact that data migration, contrary to caches, preserves all memory in the address space. To preserve the memory space, data migration must swap data instead of just copying as caches. Swapping however, incurs double the overheads of copying and so migration selection has to be targeted to data with potential for future reuse. To detect this potential, data migration selection mechanisms observe the memory behaviour before making a decision to migrate data. This can make data migration schemes less reactive than caches to working set changes since caches fetch all accessed data to the 3D-stacked DRAM.

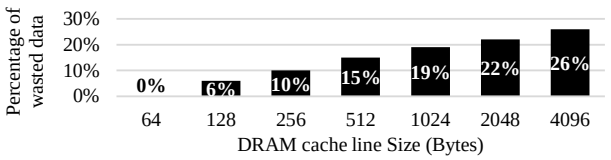


Figure 1: Average percentage of data brought in DRAM cache, but remained unused, with respect to cache line size.

DRAM caches and migration schemes face some similar issues. One issue is the trade-offs associated with data movement granularity. The cache line size for DRAM caches and the migration granularity are critical for performance. Coarse granularity favors spatial locality by effectively pre-fetching data and requires less metadata. However, coarse granularity may consume overly high bandwidth by over-fetching which can be detrimental to workloads with poor spatial locality. Finer granularity on the other hand, utilizes bandwidth more efficiently, however, it requires more metadata and does not reap the benefits of pre-fetching. Figure 1 shows that the amount of data that was fetched by a DRAM cache but not used increases with cache line size and can be as high as 26% on average[1]. For migration schemes similar trade-offs appear with regard to the migration aggressiveness. Overly aggressive migration schemes can generate excessive traffic while less aggressive ones can miss opportunities to migrate data in time. Another common issue for both DRAM caches and data migration is the metadata overhead. Caches require tag lookups while data migration requires address translation mechanisms to locate data in the memory hierarchy. These are always in the critical memory access path of every memory request and can hinder performance.

Figure 2 summarizes our findings from studying both DRAM caches and migration schemes. The graph shows the minimum, maximum and geometric mean speedup with 1 GByte of 3D-stacked DRAM, used as either part of a flat address space with migration or as a DRAM cache, over a baseline without 3D-stacked DRAM. For migration we studied Mempod (MPOD) [9], LGM [10] and Chameleon (CHA) [8] and for caches we show the results for DFC [24], Tagless [22], and an ideal DRAM cache that has no tag lookup overheads (IDEAL). The results show that caches in general can achieve higher average performance than migration designs[2]. The maximum performance shows how small cache line sizes can miss opportunities for higher performance. Large cache lines, on the other hand, can capitalize on spatial locality and have a beneficial pre-fetching effect. The minimum performance on the other hand shows how large cache line sizes can severely

[1]Average results with 1GB DRAM cache for the benchmarks described in Section 4.

[2]We do not account for page faults in this evaluation.

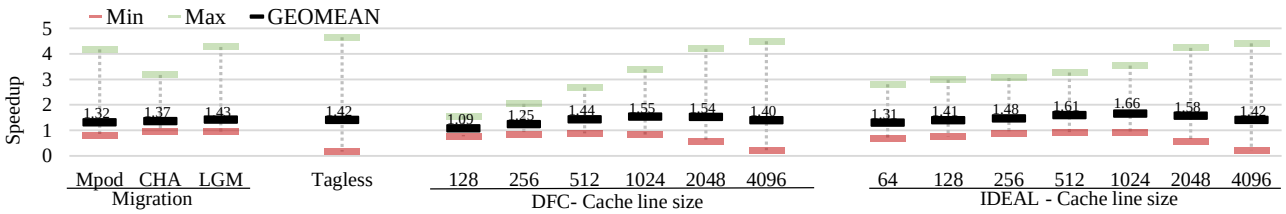


Figure 2: Min, Max, and Geometric mean speedup of migration and DRAM cache designs.

degrade performance due to overfetching. On the contrary, migration schemes do not have that risk as they do not bring in all data eagerly. Moreover, the overheads of tag lookups for caches are apparent when comparing the IDEAL DRAM cache with a realistic one (DFC) at the same cache line size; these overheads are more profound in smaller cache lines.

The above observations motivate our design choice as follows. Using most of the 3D-stacked DRAM for migration keeps most NM capacity for the flat address space of the system and in addition prevents the negative performance effects of caches with large cache lines. Using a small part of the 3D-stacked DRAM as a cache is expected to yield some of the caching performance benefits, responding faster to changes of the working data set. As this cache is small, it can afford to have small cache lines and still require a small tag-array that fits on the processor die offering short access latency. Moreover, our choice to use a sectored cache and migrate data at sector granularity allows us to reduce the metadata overheads and at the same time not waste precious far memory bandwidth by only fetching the requested cachelines on cache misses. In order to put together caching and migration efficiently, our design needs to efficiently address some challenges. Firstly, moving data between the caching and migration space should be done without requiring to relocate data within the 3D-stacked DRAM; this is achieved using indirection as explained in the next section. In addition, a unified mechanism to manage metadata both for caching and migration reduces the associated overheads.

3. HYBRID CACHING AND MIGRATION

This section presents *Hybrid*^{2}, our hybrid memory system architecture which combines a DRAM cache with a flat address space migration scheme. Our approach is to use a small portion of the NM as the data array of a sectored DRAM cache and combine the remaining portion of the NM with the FM to form a flat address space.

3.1 Hybrid^2 System Overview

Hybrid^2 tries to exploit the best-of-both-worlds and proposes architectural support to combine a DRAM Cache with a migration scheme. The idea is to have a relatively small sectored DRAM Cache whose tags can be kept on-chip with reasonable cost, while the data part of the DRAM Cache is kept in NM by utilizing a relatively small portion of NM.

Data is fetched to the DRAM cache at cacheline granularity (e.g. 64 Bytes) while DRAM cache tags are kept at sector granularity (e.g 2 KBytes). Upon memory accesses, the DRAM Cache tags are checked first and in case of a miss, a new entry for the sector is allocated in the DRAM Cache tags. The actual data of the requested sector may reside either in NM or FM and our scheme allocates new space in NM only

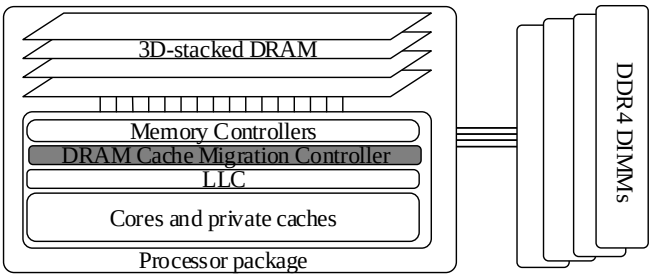


Figure 3: System Overview.

if the requested data are currently located in FM; Section 3.4 describes the details.

The NM is only logically, and not physically, split between DRAM cache and the flat address space by using pointers located in the DRAM cache tags. This permits a sector that is already in NM to be simply linked to the DRAM cache tags but also, more importantly, FM sectors that have been cached can be migrated into NM without moving the cachelines that have already been fetched. When a sector is evicted from the DRAM cache then the migration mechanism decides whether to migrate it into NM or evict it back to FM. The migration decision is based on the cost of migrating in terms of FM traffic as well as the number of accesses to that sector while in the DRAM cache. Moving the migration decision to the time when a sector is evicted from the cache removes the migration related metadata management off the critical path, minimizing their impact on performance. Furthermore, the FM traffic incurred by migrations is dynamically adjusted to the workload behaviour.

A small DRAM cache combined with coarse granularity sectors allows the tags to be kept entirely on-chip. On-chip tags induce only minimal latency to the critical memory access path as all memory requests go through the DRAM cache tag array. The tag array of the DRAM cache also stores the remapped addresses of memory segments, acting as a cache of the full remap tables which are stored in the NM, thus minimizing the address translation and tag lookup overheads. Section 3.2 details the eXtended Tag Array structure which implements the above functionality. Several techniques and optimizations like footprint caching and advanced prefetching, are directly applicable to *Hybrid*^{2}, however such options are mostly orthogonal and we opted not to include them in our base design in order to clearly attribute the performance gains to the proposed techniques.

Figure 3 presents an overview of the system we are considering in this work. The system consists of the processor, 3D-stacked DRAM and conventional DRAM. In this system, the conventional DRAM is the FM and the 3D-stacked DRAM is the NM. The shaded box is the DRAM Cache

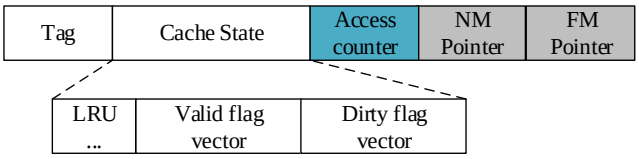


Figure 4: eXtended Tag Array Entry (XTA).

Migration Controller (DCMC). DCMC is a DRAM cache controller which we augment with some additional structures in order to support the migration along with the DRAM cache functionality. The DCMC is responsible for managing the contents of the DRAM cache, translating the addresses of remapped sectors, selecting which sectors to migrate to NM, and orchestrating the migrations. Our design is implemented in the DCMC.

3.2 eXtended Tag Array

The eXtended Tag Array (XTA) is the basic component of the DCMC. The XTA is an on-chip tag array which holds all the tags for the DRAM cache. It is set-associative and each set holds entries for multiple sectors with valid and dirty flags for every cache line of each sector. The individual fields of each entry of the XTA are shown in Figure 4. The white fields are the conventional fields needed for a sectored cache, these are from left to right: the tag for the sector and the state bits for that sector which include valid and dirty bits for every cache line. The shaded fields are additions required for our design, these are a counter and two pointers, one to a NM location and one to a FM location. The counter tracks the number of accesses to the sector, and it is used to decide whether to migrate a sector to NM when it is evicted from the DRAM cache. The pointers facilitate the address translation from the processor physical address of a sector to the actual location of that sector in the memory system. Specifically, the NM pointer points to the NM location which is allocated to this set/way of the DRAM cache. This pointer allows us to decouple the set and way of the DRAM cache from the physical location of the data in the NM. This indirection enables our design to migrate data in the NM when evicted from the DRAM cache without copying data from one NM location to another. The FM pointer points to the physical location of the sector in the FM when that sector is not migrated to NM in order to avoid remap table lookups.

Figure 5 shows an example of the use of XTA entries. The top entry corresponds to a sector that is partially present in the DRAM cache and thus not migrated to the NM, as such, some cache lines of that sector have been fetched to the NM, as denoted by the valid flag vector of the XTA entry. The dirty flag vector marks the cache lines of the sector that have been written while in the DRAM cache. The location of that sector in the NM is shown by the NM pointer while the FM pointer indicates the location of the sector in FM. The bottom entry corresponds to a sector that has been migrated entirely to the NM and the NM pointer indicates its location. In the latter case the FM pointer is not used and as a convention we set all valid and dirty bits.

3.3 Memory space layout and metadata

Figure 6a shows the layout of the NM and FM, the coloured

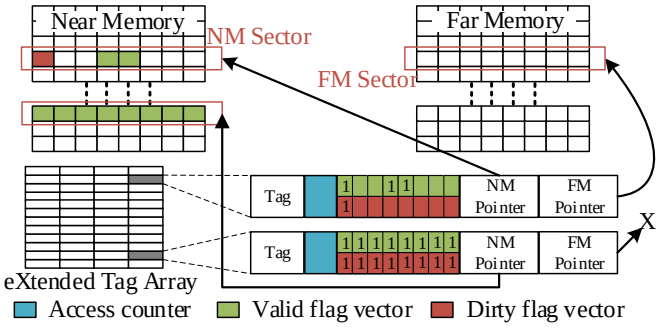


Figure 5: *Hybrid*^{2} XTA example.

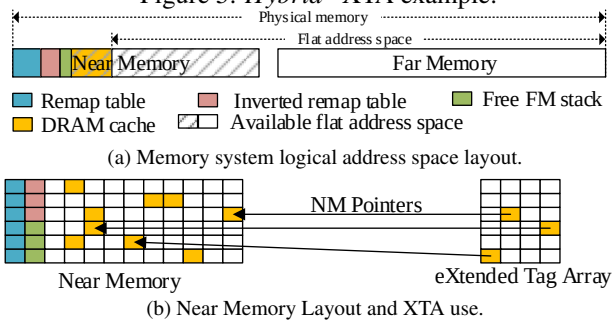


Figure 6: Memory layout and reserved memory.

areas are reserved and the uncoloured area is available as a flat address space. Note that the sectors that correspond to each XTA entry can be located anywhere in the lined area of NM. Figure 6b shows an example of how the DRAM cache sectors can be spread over the NM address space and accessed through the XTA NM pointers.

Our design allows for all-to-all address remapping for pages in NM and FM. For this purpose we keep a remap table and an inverted remap table stored in the NM. The remap table stores the mappings from processor physical address to the actual location in NM or FM where each sector is located. The inverted remap table holds the processor physical address for all locations in NM; this is used upon migration of blocks out of the NM and more details are provided in Section 3.5. The XTA also acts as a cache for the remap table entries of FM sectors that are currently (partly or fully) in the DRAM cache through the FM pointers shown in Figure 6b.

In addition to the remap and inverted remap table, we keep a stack of all FM locations that currently hold no valid data (*Free-FM-Stack*), this means that these sectors have been migrated to NM but they have not been overwritten by other data yet. The size of this stack is bound to the number of sectors that can fit in the DRAM cache. The stack pointer as well as a number of top entries of the *Free-FM-Stack* are kept on-chip in the DCMC to avoid accessing NM. Overall, the space required for the remapping data structures is 3.5% of the NM capacity.

3.4 Memory access path

In *Hybrid*^{2} all memory requests go through the DCMC which communicates with the memory controllers to access the NM and FM. Where each request is served from depends on the current location of the data. Since our design supports all-to-all address remapping, the data can be located anywhere in NM or FM. Sectors that are located in the FM

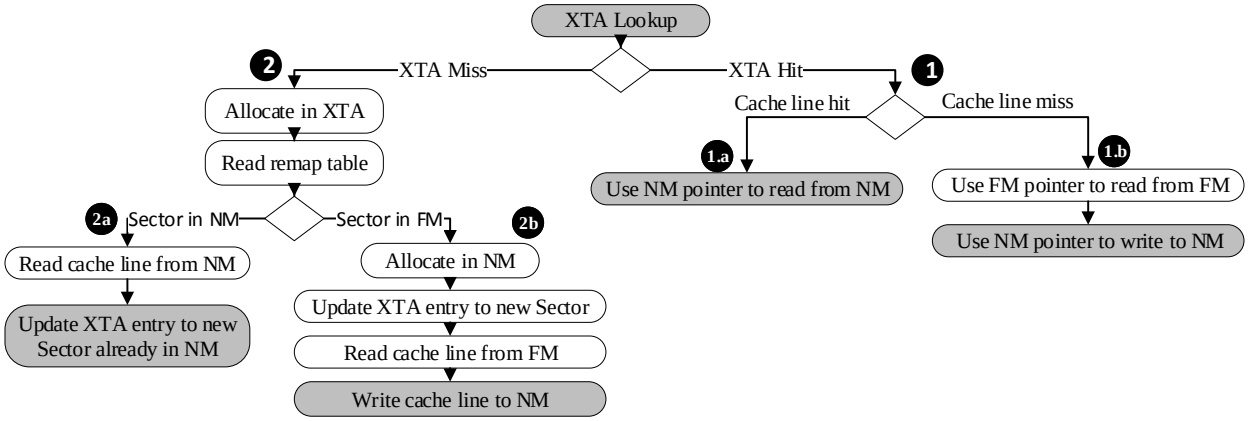


Figure 7: Memory access path.

can be (partially or fully) present in the DRAM cache with a corresponding entry in the XTA. Sectors that are located in the NM can be either fully in the DRAM cache (there exists an entry in the XTA for them) or not. When a request arrives in the DCMC, the address is used to index the XTA to determine if the sector and specific cache line is available in the DRAM cache. There are four possible outcomes as shown in Figure 7, these are:

❶ XTA Hit: In this case, the XTA contains an entry that matches with the requested sector. Even though there is an entry for the sector in the XTA, the cache line requested might be in NM 1a or not 1b.

1a XTA hit/Cache line hit: In this case the requested cache line is located in the NM. The sector can be located either in the NM or in the FM, either way, the requested cache line is available in NM through the NM pointer of the XTA entry.

1b XTA hit/Cache line miss: In this case there is an entry for the sector in the XTA but the specific cache line is not valid; this means the sector is located in the FM and only some cache lines have been fetched to the DRAM cache. Then, the FM pointer is used to read the cache line from the FM and the NM pointer is used to write it to the appropriate location in NM.

❷ XTA Miss: In this case, the XTA does not contain an entry that matches with the requested sector. The requested sector can be located either in the NM 2a or in the FM 2b. To find the location of the sector in the memory system, the remap-table is accessed using the processor physical sector address as an index. Regardless of whether the sector is located in NM or FM, an entry is allocated in the XTA for that sector.

2a XTA Miss/ Sector in NM: If the sector is located in NM then all cachelines of that sector are already in NM and the XTA entry is updated accordingly; the NM pointer is set to the NM location of the sector and all cachelines are marked as valid and dirty. The FM pointer of the XTA entry is set to zero to indicate that this sector is in NM.

2b XTA Miss/ Sector in FM: If the sector is located in FM then we need to allocate space in the NM for the sector and fetch the requested cache line from FM to the newly allocated location in NM (details about the allocation process in the NM follow in Section 3.5). Subsequently, the XTA is updated with the new sector; the NM pointer is set to the newly allocated NM location; the FM pointer is set to the FM location of the sector; the valid flag is set only for the fetched cache line and the dirty flag depending on the request type. Furthermore, the inverted remap table is updated with the sector processor physical address even though this sector is not migrated to NM yet. We do this to ensure correctness when allocating in NM as explained in detail below. This case requires several metadata operations, however, on average only 9.3% of accesses to the memory system require such handling and our evaluation shows that metadata management has minimal impact on performance (Figure 14 – No Remap).

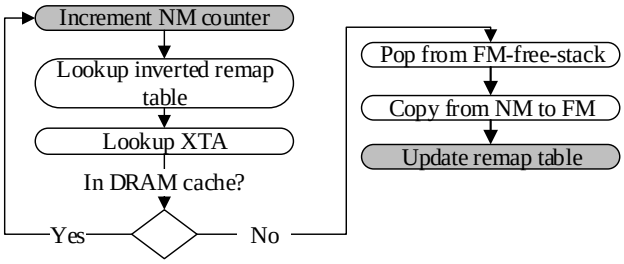


Figure 8: Allocating a sector in NM.

3.5 Allocating NM

In case of an XTA miss where the requested sector is in FM (2b in Figure 7) a new sector must be allocated in the NM. In order to make space for this new sector another sector must be migrated to FM[3]. For this purpose we need to first identify the victim sector in the NM, second, find a free sector in FM, third, copy the data from the NM sector to the FM sector, and finally, update the remapping structures with the new location. The process is illustrated in Figure 8.

To find a victim sector in NM we use a FIFO policy similar to LGM and Mempod, to do this we need a counter (*NM-counter*) which wraps around all available NM locations (lined part of NM in Figure 6a) and is incremented every time we require a new location in NM. However, the victim NM sector might be currently assigned to the DRAM cache

[3]This is the common case. At boot the cache is empty so we use a simple counter for the initially allocated NM space to the cache (shown in Figure 6a)

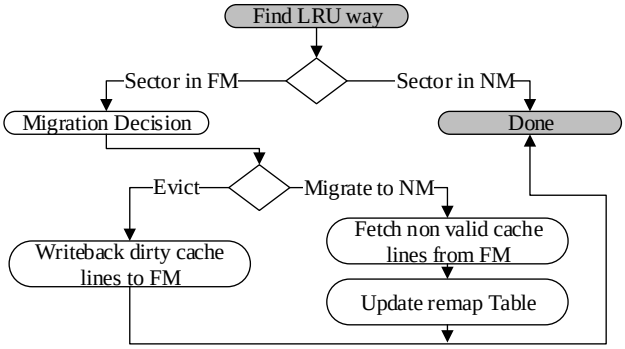


Figure 9: Eviction from DRAM cache.

(an XTA entry NM pointer points to it). For this reason we need to lookup the XTA for that sector's processor physical address. To get the processor physical address we index the inverted remap table with the sector's location. In case the sector is in the XTA, we proceed with the next one until we find one that is available. This ensures correctness as a sector that is in the DRAM cache must not be migrated to FM. Additionally, this provides a better replacement decision than just FIFO, sectors that are accessed often will most probably reside in the DRAM cache and thus not migrated to FM.

To find a free sector in FM we use the *Free-FM-stack* which is stored in NM, and partially in the DCMC. Every time a sector is migrated from FM to NM, the original FM location is pushed on that stack. That FM location is then available to be overwritten. After identifying the NM victim sector location and the FM free sector location, the DCMC copies all cache lines from the victim sector in NM to the free sector in FM and updates the remap table accordingly.

3.6 DRAM cache evictions

The DRAM cache eviction logic is illustrated in Figure 9. It uses a standard LRU algorithm to decide which sector to evict. The DRAM cache can contain (1) sectors that have already been migrated to the NM or (2) sectors that are located in FM where some (or all) of their associated cache lines have been fetched to NM.

When an already migrated sector has to be evicted from the DRAM cache (case 1), all cache lines of that sector are located in NM, therefore no data movement is required within the NM or between NM and FM. The remap table is already updated with the location of the evicted sector when it was migrated to NM. The inverted remap table was updated with the processor physical address of the evicted sector when it was first fetched in the DRAM cache. So, we can simply re-assign the XTA entry of the evicted sector to the newly allocated sector.

When a sector that resides in the FM has to be evicted from the DRAM cache (case 2), the DCMC decides whether to migrate the sector to the NM or to evict it back to FM. Migrating a sector requires fetching the cache lines not already present in NM and updating the remap table and inverted remap table accordingly. Evicting a sector to FM requires all dirty cache lines to be written back to FM while no remapping data structures need to be altered. We present the algorithm for deciding between evicting and migrating in detail below.

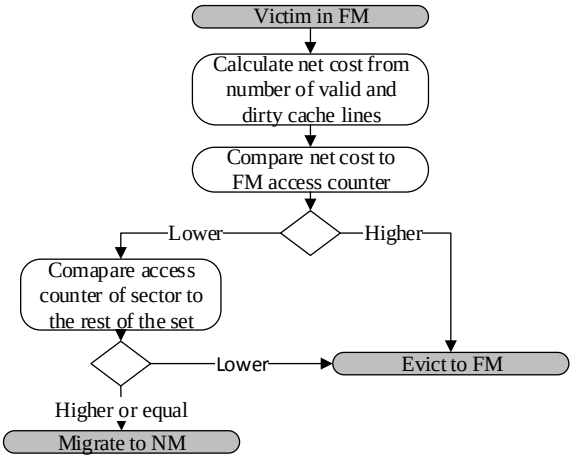


Figure 10: Migration decision.

3.7 Migration Decision

Figure 10 illustrates the algorithm used to decide whether to migrate a sector into NM when evicted from the DRAM cache. It is based on the following three factors: (i) An access counter, (ii) A cost function, (iii) The available migration bandwidth. When a FM sector is evicted from the DRAM cache these three factors are taken into account. The details of each factor are explained below.

3.7.1 Access counter

For every sector in the DRAM cache we keep a counter in the XTA which is incremented on every access to that sector (Figure 4). On evictions, the value of the counter of the victim sector is compared against the counters of the rest of the sectors of the cache set. This comparison concerns counters of few bits (9 bits for our design were enough) and happens on evictions from the DRAM cache so it is not in the critical path of memory accesses. In case the value of the counter is greater or equal to all other sectors in the set, then the sector is considered for migration. In case there is another sector in the set with a counter value greater than the victim sector counter then the victim sector is evicted and not migrated. The counter is only incremented for sectors that have not been migrated to ensure there is no starvation in a set from NM sectors that have many accesses and therefore not evicted from the XTA. Furthemore, to prevent starvation from FM sectors that remain in the cache for very long periods, we ignore the sectors whose counters have reached the maximum value.

3.7.2 Cost Function

The cost function calculates the cost of migrating a sector to NM. The inputs to the cost function are the number of valid and dirty cache lines of a sector. When evicting a sector which is not migrated to NM from the DRAM cache there are two options; either migrate the sector to NM by fetching the rest cache lines of the sector to NM, or evict it back to FM which requires all dirty cachelines be written-back to FM. The number of FM accesses for each case depends on the number of valid and dirty cachelines. Specifically, the eviction cost (E_{cost}) is equal to the number of dirty cachelines which have to be written back to FM (N_{dirty}). The migration cost (M_{cost})

is equal to the total number of cache lines per sector (N_{all}) minus the number of valid cachelines already in the DRAM cache (N_{valid}). Furthermore, the (M_{cost}) includes the cost of swapping out a sector from NM to make room for the new one, this cost is equal to N_{all} as all cache lines of the swapped-out sector have to be written back to FM. Additionally we add a fixed factor of one access to all migration costs to account for the cost of updating the remap tables. Concisely, the cost function facilitates the decision from migrating a sector to NM by calculating the net cost (Net_{cost}) of migration by subtracting the eviction cost from the migration cost:

$$M_{cost} = N_{all} - N_{valid} + N_{all} + 1 = 2 * N_{all} - N_{valid} + 1$$

$$E_{cost} = N_{dirty}$$

$$Net_{cost} = M_{cost} - E_{cost} = 2 * N_{all} - N_{valid} - N_{dirty} + 1$$

The Net_{cost} can vary from 1 when all cache lines of a sector are valid and dirty, to $2 * N_{all}$ when only one cacheline of a sector is valid and clean when evicted from the DRAM cache.

3.7.3 Migration bandwidth

We keep a single counter (*FM access counter*) for all accesses to FM that are a result of processor memory requests. This counter is incremented for every DRAM cache miss which must be fetched from FM. The cost of migrating a sector to NM is compared to this counter, if the migration cost (Net_{cost}) is smaller than the counter value then the sector is considered for migration and the Net_{cost} is subtracted from the counter. In essence, it sets the upper limit to the number of FM accesses that are allowed for migration and is reset periodically (every 100K Cycles) to adjust to workload phase changes.

3.8 Using more free space

Hybrid^2 uses only a small part of the NM as a DRAM cache, this is enough to reap the benefits of caches while keeping most of the memory capacity available to the software so as not to affect capacity limited workloads negatively. However, Chameleon [8] has shown that not all memory is always used by the OS. This unused memory can be utilized by a migration mechanism to avoid unnecessary swaps and has shown to be quite effective for Chameleon.

Although we do not consider it in this paper, *Hybrid*^{2} could support using more free space with the help of the OS. Using the same mechanisms as proposed by Chameleon (ISA-Alloc and ISA-free instructions), *Hybrid*^{2} could utilize that space and avoid copying unused sectors from NM to FM when allocating NM (Section 3.5). To support this functionality, we need to add more information in our remap table and inverted remap table to indicate unused sectors. Furthermore, the dirty state of sectors in the DRAM cache must be saved to the respective remap tables when a sector is migrated in NM so that, if/when the sector is eventually migrated back to FM, it is only written back if dirty. Finally, since "valid" copies of a sector could exist in both FM or NM, the remap table, or some other data structure, must be able to locate both.

4. EXPERIMENTAL SETUP

In this Section we provide the details of the experimental setup and the benchmarks used for our evaluation.

System configuration: As shown in Table 1, our system configuration considers an eight core processor with private L1 and L2 caches and a shared last level cache (LLC). We evaluate memory systems that consist of 16GB DDR4 FM and NM of 1GB, 2GB, and 4GB; that is NM to FM ratios of 1:16, 2:16 and 4:16.

Simulator: Our evaluation is performed using an in-house simulator based on *Pin* [42] following the interval-based simulation methodology [43] for the processor and cycle-accurate modelling of the memory system using DRAMSim2 [44]. We use *Cacti* to determine the access times for the caches [45]. Through all of our experiments the memory pages are allocated randomly in the HBM or DDR4 proportionally to their capacity. The migration-based schemes offer larger system memory capacity, compared to cache-based systems, and can accommodate applications with larger memory footprints. When executing applications with large memory footprints the cache-based systems would suffer more from page-faults and disk swaps compared to migration-based schemes. In our simulations we do not model page-faults which favors cache-based schemes.

Workloads: We evaluate our design with both multiprogrammed (MP) and multithreaded (MT) workloads. For the multi-programmed workloads we use the SPEC2017 benchmark suite [46]. For the multi-threaded workloads we use the OpenMP version of the NAS parallel benchmarks [47] [48]. For each of the NAS benchmarks we used the biggest *class* that we could run in our simulator. In both cases we use all benchmarks from each suite with memory footprint, for the simulated portion, higher than the LLC capacity (8MB). For the multi-programmed workloads we run eight instances of the same benchmark at the same time ensuring they do not share the same address space. Overall we run 21 SPEC and 9 NAS benchmarks for a total of 30 workloads. For the SPEC benchmarks we use *simpoints* to select a representative slice of one billion instructions [49] while for the NAS benchmarks we simulate one billion instructions for each thread after the initialization phase. Table 2 shows the average Last Level Cache (LLC) Misses per Kilo Instructions (MPKI), the memory footprint, and the total memory traffic for the simulated portion of each benchmark. For our evaluation in Section 5 we group our 30 benchmarks in three categories of 10 workloads each based on MPKI (high, medium, and low). While the low MPKI benchmarks do not stress the memory system much, we choose to include all benchmarks from both suites for completeness.

5. EVALUATION

Table 1: System configuration.

Cores	8 cores, out-of-order, 4-way issue/commit, 3.2 GHz
L1 Cache	Private, 64 KB, 4-way, 1 cycle access latency
L2 Cache	Private, 256 KB, 8-way, 9 cycles access latency
L3 Cache	Shared 8MB, 16-way, 14 cycles access latency, non-inclusive, non-exclusive
Near Memory	HBM2 2GHz, 1,2,4 GB, 8 128-bit channels, 8 banks, tCAS-tRCD-tRP: 7-7-7, RD/WR+I/O energy: 6.4pJ/bit, ACT/PRE energy: 15nJ
Far Memory [41]	DDR4-3200, 16 GB, 2 64-bit channels, 8 banks, tCAS-tRCD-tRP: 22-22-22, RD/WR+I/O energy: 33pJ/bit, ACT/PRE energy: 15nJ

Table 2: Benchmark characteristics.

High MPKI			
Bechmark	**MPKI**	**Footprint(GB)**	**Traffic(GB)**
cg.D (MT)	90.6	7.8	43.3
sp.D (MT)	30.1	11.2	21.6
bt.D (MT)	30.1	10.7	21.3
fotonik3d (MP)	28.1	6.4	19.9
lbm (MP)	27.4	3.1	21.7
bwaves (MP)	26.8	3.3	13.8
lu.D (MT)	25.8	2.9	19.1
mcf (MP)	25.8	0.1	12.6
gcc (MP)	21.2	1.6	13.0
roms (MP)	15.5	2.3	9.7
Medium MPKI			
Bechmark	**MPKI**	**Footprint(GB)**	**Traffic(GB)**
mg.C (MT)	14.2	2.8	8.9
omnetpp (MP)	9.8	1.5	6.9
is.C (MT)	9.0	1.0	5.4
dc.B (MT)	8.4	4.0	8.0
ua.D (MT)	7.8	3.1	4.9
xz (MP)	5.6	0.7	4.3
parest (MP)	4.3	0.2	2.2
cactus (MP)	3.4	0.8	2.0
ft.C (MT)	3.1	0.9	2.6
cam4 (MP)	2.2	0.3	1.6
Low MPKI			
Bechmark	**MPKI**	**Footprint(GB)**	**Traffic(GB)**
wrf (MP)	1.4	0.4	1.1
xalanc (MP)	1.1	0.1	1.0
imagick (MP)	1.1	0.4	0.9
x264 (MP)	0.9	0.3	0.6
perlbench (MP)	0.7	0.2	0.4
blender (MP)	0.7	0.2	0.3
deepsjeng (MP)	0.3	3.4	0.2
nab (MP)	0.2	0.2	0.1
leela (MP)	0.1	0.1	0.1
namd (MP)	0.13	0.1	0.1

In this Section we present the evaluation of *Hybrid*^{2}. For our evaluation we compare against three state-of-the art migration designs and two cache designs. These are:

- Mempod (MPOD) [9]. For Mempod we performed a design space exploration on the number of MEA counters and found the best value for our system to be 64 MEA counters with 50μs intervals.

- Chameleon (CHA) [8]. For Chameleon the *K* parameter value for our memory system characteristics is 14. Additionally, we allow the same NM capacity our design uses as a DRAM cache to be used in Chameleon's *cache mode*.

- LLC-guided data migration (LGM) [10]. For LGM we performed a design space exploration on the migration *high Watermark* and found that the best performance is achieved at 256 with 50μs intervals.

- Tagless DRAM cache (TAGLESS) [22]. For the Tagless DRAM cache, we optimistically do not model any operating system overheads.

- Decoupled Fused Cache (DFC) [24]. For DFC we found the best performance is achieved at a cacheline size of 1 KByte and compare against this configuration.

For Mempod, LGM, and Chameleon we adjust the size of their respective remap cache to be equal to that of the XTA in *Hybrid*^{2} for a fair comparison. All our results are normalized to a Baseline system without 3D-stacked DRAM.

5.1 Design space exploration.

Hybrid^2 can be configured with any size of DRAM cache, sector size, and cache line size. These design choices affect performance as well as the size of the XTA. To have a design proportional to the evaluated system, we limit the XTA size to 512 KBytes and explore all possible configurations within this limit. A bigger XTA or a bigger remap cache for the migration designs would incur higher access latency. Furtermore, a 512 KByte remap cache has been shown to avert most remap table accesses for Mempod and LGM [10]. We examine DRAM cache sizes of 64 MBytes and 128 MBytes, sectors of 2 Kbytes and 4KBytes, and cache lines of 64,128, 256, and 512 Bytes all with 16-way associativity.

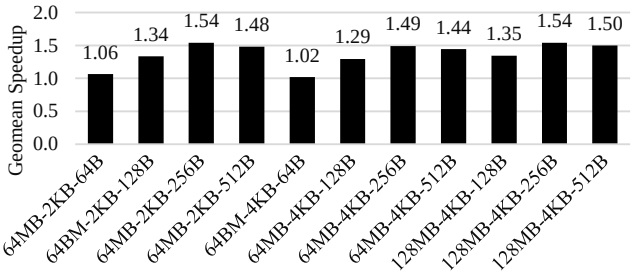


Figure 11: Design space exploration, Geometric mean speedup over baseline for different *Hybrid*^{2} configurations.

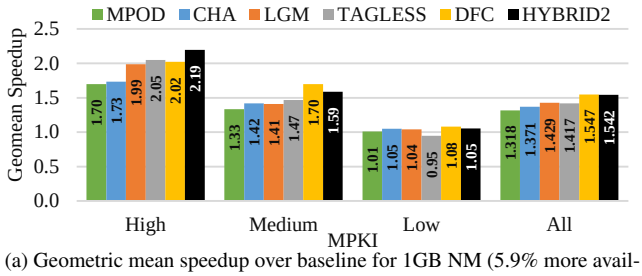
Figure 11 shows the results of our design space exploration for all combinations of the above mentioned parameters. Through this design space exploration we find that the best performance is achieved with 256 Byte cache lines. Smaller cachelines miss the opportunity to exploit spatial locality and pre-fetching. Larger cache lines over-fetch and decrease performance. So, a cache line of 256 Bytes is a good compromise between spatial locality and bandwidth waste as 90% of the data fetched are used on average (Figure 1). For the same DRAM cache size and cache line size, 2 KByte sectors perform better than 4 KByte sectors. Bigger sectors decrease address translation overheads while smaller ones use NM space better. Our design achieves its best performance at 64 MBytes DRAM cache with 2 KByte sectors and 256 Byte cache lines. For the rest of this evaluation we present our results for 64 MByte cache with 2 KByte sectors and 256 Byte cache lines.

5.2 Performance

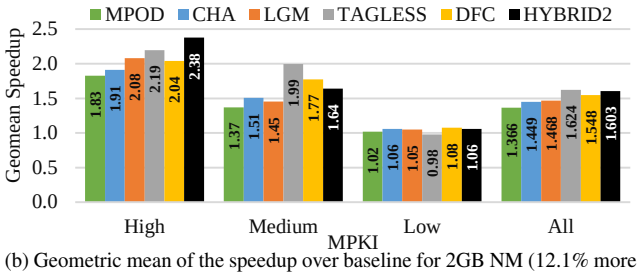
Figures 12a, 12b, and 12c show the geometric mean speedup for all MPKI classes as well as the geometric mean of all benchmarks for three different NM to FM ratios (1:16 , 2:16 , 4:16) over a baseline without NM. From the results we see that all designs benefit from larger NM:FM ratios.

For High MPKI benchmarks *Hybrid*^{2} outperforms all other designs by at least 6.8% on average for the 1GB NM case (1:16 ratio), outperforms all other designs by at least 8.6% on average for the 2GB NM case (2:16 ratio), matches the best performing cache scheme (TAGLESS) for the 4GB NM case (4:16 ratio). For all NM:FM ratios, *Hybrid*^{2} outperforms migration schemes by at least 8.4% on average.

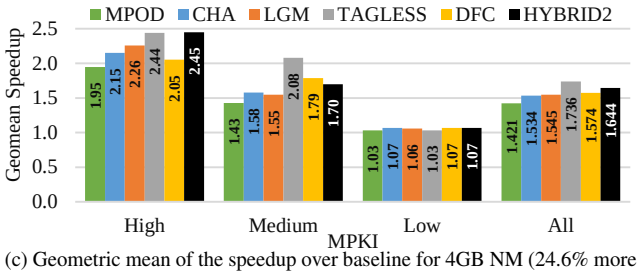
For Medium MPKI benchmarks, *Hybrid*^{2} again clearly outperforms all migration schemes, however, caches gain a performance advantage as a larger portion of the memory footprint of the benchmarks fits in the cache space. This effect is even more pronounced in bigger NM sizes.



(a) Geometric mean speedup over baseline for 1GB NM (5.9% more available memory than caches).



(b) Geometric mean of the speedup over baseline for 2GB NM (12.1% more available memory than caches).



(c) Geometric mean of the speedup over baseline for 4GB NM (24.6% more available memory than caches).

Figure 12: Geometric mean of the speedup over baseline for high, medium, low MPKI, and all benchmarks for NM sizes of 1GB, 2GB and 4GB.

For Low MPKI benchmarks, all designs perform similarly except for the TAGLESS cache which suffers at benchmarks with low spatial locality like *deepsjeng*.

-Overall, for all benchmarks classes and NM:FM ratios, *Hybrid*^{2} outperforms the competing migration schemes and performs similarly to caches even though our comparison is conservative since we do not take page-faults into account that would degrade the performance of caches more severely than migration schemes.

For the rest of this Section we present detailed results for the 1:16 NM:FM ratio as it stresses all designs more due to the smaller NM size which is smaller than the memory footprint of most benchmarks.

Figure 13 shows the speedup achieved over a baseline without NM for *Hybrid*^{2} and all migration and cache designs for the 1:16 NM to FM ratio. The benchmarks are sorted by MPKI. *Hybrid*^{2} performs consistently well for benchmakrs with high MPKI and big memory footprints like *cg.D*, *sp.D*, *cg.D* and *fotonik3d*. For Medium MPKI benchmarks *Hybrid*^{2} achieves 6.5% lower speedup than the best DRAM cache while outperforming all other competing designs. For low MPKI workloads all designs achieve similarly low speedups as there is not enough room for improvement. Notice how large cacheline sizes can severely degrade performance for benchmarks with limited spatial locality. For example, the Tagless DRAM cache degrades the performance of *omntepp* and *deepsjeng* to 1/5 of the baseline. *Hybrid*^{2} only shows minimal performance degradation for *dc.B* and *deepsjeng*. For *dc.B* all designs show little difference from the Baseline performance because of the streaming nature of its memory accesses which provide little potential for data reuse. For *deepsjeng* none of the evaluated designs surpassed the Baseline as it is characterized by low memory intensity with a wide memory footprint and very limited spatial locality, still *Hybrid*^{2} does not degrade performance significantly.

5.2.1 *Performance breakdown*

The performance of *Hybrid*^{2} can be attributed to both the DRAM cache and the migration components as well as the elimination of address translation overheads. To show the effects of each factor above we conducted a series of experiments. Figure 14 shows the geometric mean speedup achieved for a number of different alternatives for *Hybrid*^{2}. From left to right are: *Cache-only* shows the performance of a 64 MByte sectored DRAM cache alone, without any data migration or address translation overheads. *Migr-All* and *Mirg-None* show the performance of *Hybrid*^{2} if we choose to migrate *All* data when evicted from the DRAM cache, or *None*, respectively. *No-Remap* shows the effects of removing all address translation overheads from our design, that is we assume all accesses to the remap table, inverted remap table, and Free-FM-Stack complete instantly. The DRAM cache alone (*Cache-Only*), achieves a significant speedup overall, equal to the best migration design in our evaluation (LGM). This shows that even a small DRAM cache can be very beneficial to performance. *Hybrid*^{2} however performs better than *Cache-Only* and both *Migr-None* and *Mirg-All*. This quantifies the contribution of our migration selection criteria to performance improvement. Furthermore, *Hybrid*^{2} performs only marginally lower (2.5%) than *No-Remap*, this shows that our design effectively tackles the address translation overheads of data migration. Overall the address remaping structures in NM account for only 4.1% of the high-bandwidth NM traffic and 3.5% of NM space. This point is also shown by the small difference in performance between *Cache-Only* and *Migr-None*, the difference between these two points is solely the overheads imposed by address translation.

5.3 NM Utilization

Figure15 shows the geometric mean of the percentage of processor memory requests that were served by the NM for high, medium, and low MPKI benchmark groups. A higher percentage does not necessarily correlate with higher performance, for example we see that the Tagless DRAM cache shows the highest percentage of all designs with 90% of all requests served from NM while its performance is considerably lower. *Hybrid*^{2} achieves an average of 84% of processor requests served from NM with higher percentages for High and Medium MPKI workloads. DFC achieves a slightly higher ratio, with 85% of requests served from NM on average for all benchmarks. *Hybrid*^{2} achieves higher rates than other migration designs in almost all benchmarks. Mempod achieves the worst ratio with 40% on average, LGM comes next with

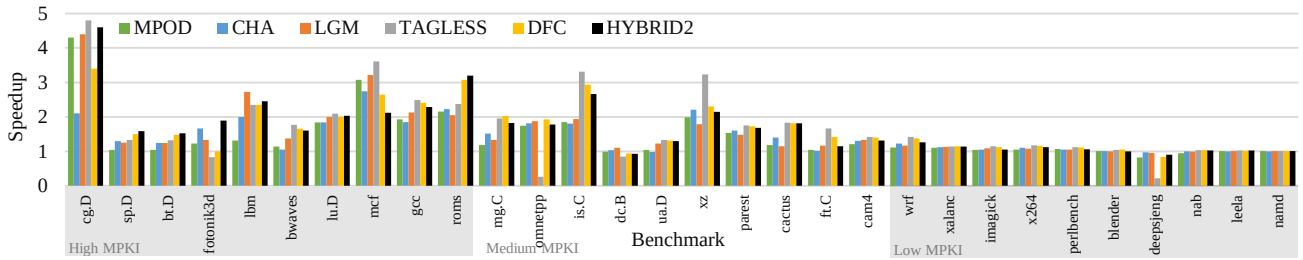


Figure 13: Speedup over baseline.

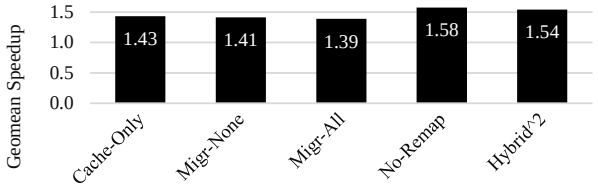


Figure 14: *Hybrid*^{2} Performance factors breakdown.

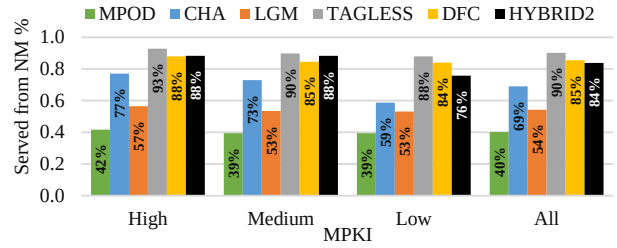


Figure 15: Geometric mean of normalized processor requests served from NM for benchmarks with high, medium, and low MPKI.

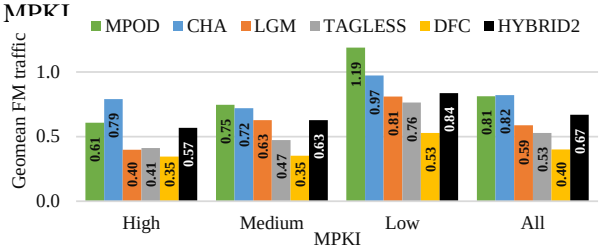


Figure 16: Geometric mean of normalized FM traffic for benchmarks with high, medium, and low MPKI.

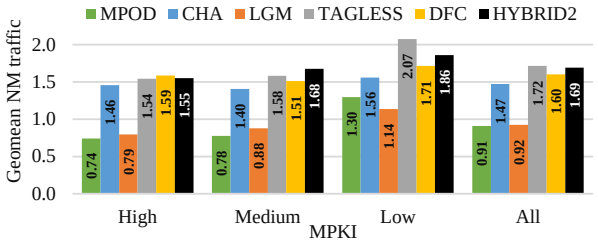


Figure 17: Geometric mean of normalized NM traffic for benchmarks with high, medium, and low MPKI.

54% because its bandwidth saving mechanism allows it to migrate more aggressively, and finally Chameleon achieves the best of all migration designs at 69% on average.

5.4 Traffic

Figure 16 shows the FM traffic normalized to the baseline for each benchmark group. The advantage of caches over migration is visible from the overall lower traffic in FM. This comes from the intrinsically lower cost of copying against swapping. *Hybrid*^{2} incurs lower FM traffic compared to Mempod and Chameleon but higher compared to LGM. LGM however, is optimized to economize bandwidth as its migration decisions are based on the observed spatial locality of memory segments. For high MPKI workloads LGM produces FM traffic similar to the caches. Overall *Hybrid*^{2} produces 67% of the FM traffic compared to the baseline.

Figure 17 shows the geometric mean of NM traffic, normalized to the memory traffic of the baseline system for our benchmark groups. *Hybrid*^{2} produces slightly higher NM traffic than the caches although the percentage of requests served from NM is lower. This is because the NM traffic includes the accesses to the address translation data structures. Even though these accesses have minimal impact on performance in *Hybrid*^{2}, they still incur some traffic to the NM. The low values of NM traffic for Mempod and LGM are explained by the few processor requests that are served from NM (Figure 15).

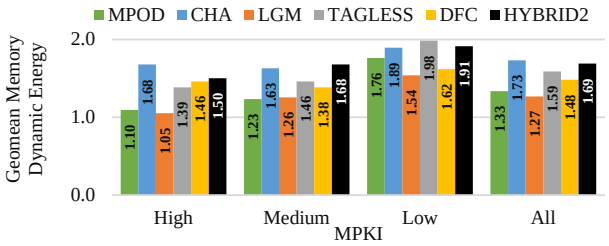


Figure 18: Geometric mean of normalized dynamic memory energy consumption for benchmarks with high, medium, and low MPKI.

5.5 Energy consumption

Figure 18 shows the normalized geometric mean of the *dynamic* memory system energy consumption. *Hybrid*^{2} consumes 2.3% less dynamic energy than Chameleon and about 30% higher than the other migration schemes, mostly due to higher NM traffic, which is however capitalized in better performance. Compared to DRAM caches, *Hybrid*^{2} consumes about 6.3-14.2% more dynamic memory energy mostly due to higher FM traffic, which is a reasonable price to pay for larger memory capacity. We do not report processor energy consumption or memory static energy consumption (refresh energy) as these are mostly proportional to the runtime, which is in general better for *Hybrid*^{2}.

6. CONCLUSIONS

This paper presented *Hybrid*^{2}, a hybrid memory system that combines caching and migration. *Hybrid*^{2} considers a

high bandwidth near memory complemented with a larger, lower bandwidth, far memory. A small fraction of the near memory is reserved to host a sectored DRAM cache. The remaining near memory capacity is available to the flat address space of memory system and implements transparent data migration in HW. The small DRAM cache is used to select candidate data for migration in NM and permits efficient migration via indirection that avoids copying data between the cache and the flat address space. The metadata required for caching and migration is supported by a common mechanism which alleviates the corresponding overheads. Compared to migration schemes, *Hybrid*^{2} performs 6.4-9.1% better and, compared to DRAM caches, it offers 5.9-24.6% more main memory capacity giving away only 0.3-5.1% of performance without taking into account the impact of page faults.

7. REFERENCES

[1] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH C.A. News*, 23(1):20–24, 1995.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA-42*, pages 336–348, 2015.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA-42*, pages 105–117, 2015.

[4] Milan Pavlovic, Yoav Etsion, and Alex Ramirez. On the memory system requirements of future scientific applications: Four case-studies. In *IISWC*, pages 159–170, 2011.

[5] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *ISCA-36*, pages 371–382, 2009.

[6] Y. Zhou and D. Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *ISCA-43*, pages 532–544, 2016.

[7] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, Dec 2014.

[8] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir. Chameleon: A dynamically reconfigurable heterogeneous memory system. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 533–545, Oct 2018.

[9] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen. Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 433–444, Feb 2017.

[10] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. LLC-guided data migration in hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[11] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John. Silc-fm: Subblocked interleaved cache-like flat memory organization. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 349–360, Feb 2017.

[12] A. Kokolis, D. Skarlatos, and J. Torrellas. Pageseer: Using page walks to trigger page swaps in hybrid memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–608, Feb 2019.

[13] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–464, Dec 2011.

[14] G. Loh and M. D. Hill. Supporting very large dram caches with compound-access scheduling and missmap. *IEEE Micro*, 32(3):70–78, May 2012.

[15] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37, Dec 2014.

[16] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring dram cache architectures for cmp server platforms. In *2007 25th International Conference on Computer Design*, pages 55–62, Oct 2007.

[17] S. Mittal and J. S. Vetter. A survey of techniques for architecting dram caches. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1852–1863, June 2016.

[18] Cheng-Chieh Huang, Rakesh Kumar, Marco Elver, Boris Grot, and Vijay Nagarajan. C3d: Mitigating the numa bottleneck via coherent dram caches. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 36:1–36:12, Piscataway, NJ, USA, 2016. IEEE Press.

[19] C. Chou, A. Jaleel, and M. K. Qureshi. Candy: Enabling coherent dram caches for multi-node systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[20] C. Huang and V. Nagarajan. Atcache: Reducing dram cache latency via a small sram tag cache. In *2014 23rd International Conference on*

Parallel Architecture and Compilation Techniques (PACT), pages 51–60, Aug 2014.

[21] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee. Efficient footprint caching for tagless dram caches. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 237–248, March 2016.

[22] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A fully associative, tagless dram cache. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 211–222, June 2015.

[23] C. Chou, A. Jaleel, and M. K. Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 198–210, June 2015.

[24] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. Decoupled fused cache: Fusing a decoupled llc with a dram cache. *ACM Trans. Archit. Code Optim.*, 15(4):65:1–65:23, January 2019.

[25] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 404–415, New York, NY, USA, 2013. ACM.

[26] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, Dec 2012.

[27] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. Bi-modal dram cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 38–50, Washington, DC, USA, 2014. IEEE Computer Society.

[28] S. Franey and M. Lipasti. Tag tables. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 514–525, Feb 2015.

[29] C. C. Chou, A. Jaleel, and M. K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, Dec 2014.

[30] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1. `http://hybridmemorycube.org/`. [Online].

[31] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2017.

[32] JEDEC. Wide I/O Single Data Rate (Wide I/O SDR). `https://www.jedec.org/standards-documents/docs/jesd229`. [Online].

[33] Micron. NVDIMM. `https://www.micron.com/products/dram-modules/nvdimm`. [Online].

[34] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[35] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[36] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 1–14, New York, NY, USA, 2017. ACM.

[37] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 268–280, New York, NY, USA, 2017. ACM.

[38] Intel. Allocate Memory Efficiently on an Intel Xeon Phi Processor. `https://software.intel.com/sites/default/files/managed/5f/5e/MCDRAM_Tutorial.pdf`.

[39] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, Feb 2015.

[40] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, March 2003.

[41] Micron. DDR4 SDRAM datasheet MT40A1G8SA-062E. `https://www.micron.com/products/dram/ddr4-sdram/part-catalog/mt40a1g8sa-062e`.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[43] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[44] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[45] S. J. E. Wilton and N. P. Jouppi. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[46] R. Panda, S. Song, J. Dean, and L. K. John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, Feb 2018.

[47] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[48] SNU. SNU NPB Suite. `http://aces.snu.ac.kr/software/snu-npb/`.

[49] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.