THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

High Performance Hybrid Memory Systems with 3D-stacked DRAM

EVANGELOS VASILAKIS

Division of Computer Engineering
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

High Performance Hybrid Memory Systems with 3D-stacked DRAM

EVANGELOS VASILAKIS

Advisor: Ioannis Sourdis, Professor at Chalmers University of Technology
Co-Advisor: Vassilios Papaefstathiou, Researcher at FORTH-ICS, Greece
Co-Advisor: Pedro Trancoso, Professor at Chalmers University of Technology

Examiner: Ulf Assarsson, Professor at Chalmers University of Technology

Thesis Opponent: Babak Falsafi, Professor at EPFL, Switzerland

Grading Committee:
– Erik Hagersten, Professor at Uppsala University, Sweden
– Bruce Jacob, Professor at University of Maryland, USA
– André Seznec, Senior research director at IRISA/INRIA, France

ISBN: 978-91-7905-311-6
Doktorsavhandlingar vid Chalmers tekniska högskola.
Series number: 4778
ISSN 0346-718X

Department of Computer Science & Engineering
Division of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

Abstract

The bandwidth of traditional DRAM is pin limited and so does not scale well with the increasing demand of data intensive workloads. 3D-stacked DRAM can alleviate this problem providing substantially higher bandwidth to a processor chip. However, the capacity of 3D-stacked DRAM is not enough to replace the bulk of the memory and therefore it is used together with off-chip DRAM in a hybrid memory system, either as a DRAM cache or as part of a flat address space with support for data migration. The performance of both above alternative designs is limited by their particular overheads. This thesis proposes new designs that improve the performance of hybrid memory systems. It does so first by alleviating the overheads of current approaches and second, by proposing a new design that combines the best attributes of DRAM caching and data migration while addressing their respective weaknesses. The first part of this thesis focuses on improving the performance of DRAM caches. Besides the unavoidable DRAM access to fetch the requested data, tag access is in the critical path adding significant latency and energy costs. Existing approaches are not able to remove these overheads and in some cases limit DRAM cache design options. To alleviate the tag access overheads of DRAM caches this thesis proposes *Decoupled Fused Cache* (DFC), a DRAM cache design that fuses DRAM cache tags with the tags of the on-chip Last Level Cache (LLC) to access the DRAM cache data directly on LLC misses. Compared to current state-of-the-art DRAM caches, DFC improves system performance by 11% on average. Finally, DFC reduces DRAM cache traffic by 25% and DRAM cache energy consumption by 24.5%. The second part of this thesis focuses on improving the performance of data migration. Data migration has significant performance potential, but also entails overheads which may diminish its benefits or even degrade performance. These overheads are mainly due to the high cost of swapping data between memories which also makes selecting which data to migrate critical to performance. To address these challenges of data migration this thesis proposes *LLC guided Data Migration* (LGM). LGM uses the LLC to predict future reuse and select memory segments for migration. Furthermore, LGM reduces the data migration traffic overheads by not migrating the cache lines of memory segments which are present in the LLC. LGM outperforms current state-of-the art data migration, improving system performance by 12.1% and reducing memory system dynamic energy by 13.2%. DRAM caches and data migration offer different tradeoffs for the utilization of 3D-stacked DRAM but also share some similar challenges. The third part of this thesis aims to provide an alternative approach to the utilization of 3D-stacked DRAM combining the strengths of both DRAM caches and data migration while eliminating their weaknesses. To that end, this thesis proposes *Hybrid*^{2}, a hybrid memory system design which uses only a small fraction of the 3D-stacked DRAM as a cache and thus does not deny valuable capacity from the memory system. 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. Depending on the system configuration, *Hybrid*^{2} on average outperforms state-of-the-art migration schemes by 6.4% to 9.1%, compared to DRAM caches *Hybrid*^{2} gives away on average only 0.3%, to 5.3% of performance offering up to 24.6% more main memory capacity.

Keywords:
Hybrid memory systems, 3D-stacked DRAM, DRAM caches, Data migration

Acknowledgments

I always believed that a PhD is a personal endeavour where the student has to show their capabilities and struggle to contribute to their field while also improving themselves through hard work. That said, it would not be possible to walk this path alone without the support of all the people around me.

First of all, a big thanks to my advisor Yiannis for putting up with me for all this time and for guiding me from the beginning of this endeavour. Although we probably disagreed more often than not, we both know that it was always for the best. I will always think of Yiannis not only as a good advisor but also as a good friend. A big thanks also to my co-advisors, Vassilis and Pedro for their valuable help throughout my studies. Without all of you this work would not have been possible.

Also thanks to all the excellent people at Chalmers for making Chalmers a wonderful work environment. A special note for my colleagues and office mates, Alirad, Stavros, Prajith, Ahsen, Albin and Stefano for being there with me through the trenches. Outside Chalmers, my friends, Dimitris, Stella, Giorgos, Anastasia, Giorgos, Anna, Lea, and Despoina who were always there for me when I needed them. Finally, I would like to thank my mother for making me what I am today and my sister for being a close confidant I can always trust.

This work was supported by the European Commission under the Horizon 2020 Program through the ECOSCALE (grant agreement 671632) and SHARCS (grant agreement 644571) projects as well as by the European Research Council (ERC) under the MECCA project (Contract No. 340328).

List of Publications

This thesis is based on the following publications:

[A] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, Ioannis Sourdis "FusionCache: using LLC Tags for DRAM Cache"
2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, pp. 593–596.

[B] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, Ioannis Sourdis "Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache"
ACM Transactions on Architecture and Code Optimization (TACO), vol. 15, no. 4, pp. 65:1–65:23, Jan. 2019.

[C] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, Ioannis Sourdis "LLC-guided Data Migration in Hybrid Memory Systems"
2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019, pp. 932–942.

[D] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, Ioannis Sourdis "Hybrid^2: Combining Caching and Migration in Hybrid Memory Systems"
26th IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, California, USA, February 22–26, 2020.

Other publications

The following publications were also published during my PhD studies. However, they are not included to this thesis because their contents are overlapping or not related to the thesis.

[a] Alirad Malek, Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, Ioannis Sourdis
"Odd-ECC: On-demand DRAM Error Correcting Codes"
Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017, pp. 96-111.

[b] Evangelos Vasilakis, Ioannis Sourdis, Vassilis Papaefstathiou, Antonis Psathakis and Manolis Katevenis
"Modeling Energy-Performance Tradeoffs in ARM big.LITTLE Architectures"
27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017, Thessaloniki, Greece, September 25-27, 2017, pp. 1–8.

Contents

Chapter 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 [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].

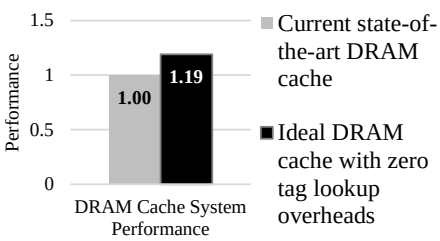
3D-stacking technology can be used to increase memory bandwidth. In particular, 3D-stacked DRAM can be placed near the processor die offering substantially higher bandwidth than off-chip DRAM. 3D-stacked DRAM though has limited capacity and often needs to be complemented with a larger off-chip DRAM that has however lower bandwidth. Currently, there are two approaches to integrate both 3D-stacked DRAM and off-chip DRAM to form a hybrid memory system: the first one is a flat address-space memory system supporting migration between the two types of DRAM [7–11]; the second one is to use 3D-stacked DRAM as a cache [12–26].

DRAM caches have shown excellent potential in capturing the spatial and temporal data locality of applications. By copying the most recently accessed data to the 3D-stacked DRAM, they faithfully follow the working set of applications. Due to the size of DRAM caches, traditional (on-chip) cache architectures are sub-optimal for organizing DRAM caches, making their performance far from ideal.

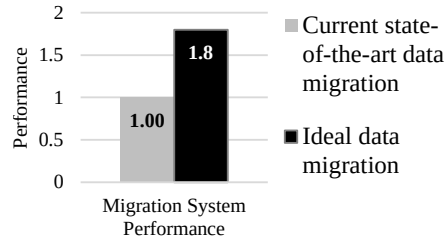
As opposed to DRAM caches, data migration keeps 3D-stacked DRAM capacity available to the system. This means that data cannot be just copied to 3D-stacked DRAM as in caches, but instead has to be swapped which incurs double the overhead of copying. To amortize the overheads of swapping, it is important to select for migration only a subset of the accessed data; preferably the ones with the highest potential for future reuse. Reducing the swapping overheads as well as selecting the most promising data to migrate are critical factors for the performance of systems that support data migration.

This thesis addresses the overheads and limitations of hybrid memory systems by proposing more efficient designs for systems that utilize 3D-stacked DRAM in addition to conventional off-chip DRAM.

The rest of this introductory Chapter is organized as follows: The problem statement is presented in Section 1.1 followed by a discussion of the objectives and contributions of this thesis in Chapter 1.2.



(a) Performance of a system with current state-of-the-art DRAM cache [19] and with an ideal DRAM cache with zero tag lookup overheads.



(b) Performance of a system with current state-of-the-art data migration scheme [9] and ideal system where all data are in 3D-stacked DRAM without any overheads.

Figure 1.1: Potential performance of DRAM caches and Data migration.

1.1 Problem Statement

Existing hybrid memory systems have significant overheads that limit their performance. DRAM caches suffer mainly from the tag lookup overheads while data migration suffers most performance losses because of the more expensive data movement it requires. Tag management of DRAM caches and data movement overheads in data migration are the first two problems addressed in the thesis as they are the leading causes of inefficiencies in these approaches.

Furthermore, both DRAM caches and data migration offer fixed tradeoffs in the design of hybrid memory systems. DRAM caches waste the 3D-stacked DRAM capacity to allow data replication, which in turn reduces the bandwidth overhead of transferring data. Migration does not waste capacity, but this forces data swapping between the memories wasting bandwidth. The third problem addressed in this thesis is the lack of an alternative design that combines the benefits of DRAM caches and data migration and reduces their overheads.

More details about each problem addressed in this thesis are provided bellow.

1.1.1 Tag Lookups in DRAM Caches

The DRAM cache tag access latency affects performance and depends on the tag organization and management. Different design choices come with different tradeoffs that are tightly related to the DRAM cache line size. Smaller DRAM cache lines offer more flexibility and more efficient use of the cache bandwidth and capacity when the application is characterized by low spatial locality. Larger DRAM cache lines offer better prefetching and overall better performance when the workloads exhibit spatial locality. On the other hand, smaller DRAM cache lines require more tag storage than larger ones for the same cache size making it infeasible to store them on chip. Even for larger DRAM cache lines, the cost of storing the tags on chip is not negligible and it could otherwise be utilized for a larger on-chip Last Level Cache (LLC). Storing the DRAM cache tags in DRAM is more space efficient and also allows for smaller DRAM cache lines but results in substantially higher tag access latency as well as increased 3D-stacked DRAM traffic.

Figure 1.1a shows the performance overhead of tag lookups in ATCache, a current state-of-the-art DRAM cache design [19]. An ideal DRAM cache that performs tag lookups with zero latency would have 19% better performance. This performance gap presents an opportunity for improving existing DRAM cache designs.

1.1.2 Data Migration Overheads and Inefficiencies

Data migration differs from caching in that it does not waste the capacity of the 3D-stacked DRAM from the memory system. To achieve that, data migration schemes need to swap data from off-chip to 3D-stacked DRAM instead of just copying as caches. Swapping however, requires double the memory traffic of copying. Migration traffic competes directly with processor memory requests for bandwidth and increases the queuing latency, especially in off-chip DRAM, which is the memory system bottleneck. To increase the performance of data migration it is important to reduce the migration traffic overheads as well as to select data with good potential for future reuse.

Figure 1.1b shows that an ideal system where all data are always found (with zero overheads) in the high bandwidth 3D-stacked DRAM could achieve 1.8× better performance than a current state-of-the-art data migration scheme, MemPod [9]. This significant gap in performance is due to the migration overheads as well as due to sub-optimal data selection of existing data migration schemes. This thesis aims to bridge this gap by improving these aspects of data migration.

1.1.3 Copying Costs Capacity, Swapping Costs Traffic

On one hand, DRAM caches achieve high performance because they react fast to changes in the working set, but need to waste 3D-stacked DRAM capacity to copy data. On the other hand, migration schemes allow 3D-stacked DRAM space to be part of the main memory, but suffer more expensive data transfers as they need to swap data between the two parts of the hybrid memory system. As a consequence, migration schemes need to be more selective in transferring data and become less reactive giving away performance.

This thesis aims to propose a memory system design that reacts to working set changes like DRAM caches while at the same time does not deny much capacity.

1.2 Thesis Objectives and Contributions

The primary objective of this thesis is to improve the performance of a hybrid memory system that consists of 3D-stacked and off-chip DRAM. We first aim to improve DRAM cache designs, subsequently we try to improve data migration and finally we attempt to combine the advantages of both approaches.

1.2.1 Minimizing the Tag lookup Overheads in DRAM Caches

The first thesis objective is to improve the performance of systems that use 3D-stacked DRAM as a cache by minimizing their tag lookup overheads. The main idea behind this is to:

Store information about the location of DRAM cache lines in the tag array of the on-chip Last Level Cache (LLC) so as to access the data in the DRAM cache directly after LLC misses.

Related Work: Several designs have been proposed aiming to reduce the DRAM cache tag access latency, however they are not able to nullify it and some of them introduce significant constraints to the system. One such design employs an on-chip SRAM cache of the DRAM cache tags [19]. This reduces the average DRAM cache tag lookup latency however it adds a constant delay to every DRAM cache access for accessing the tag-cache and more on-chip resources are occupied for caching the DRAM cache tags. Another technique places the DRAM cache data addresses directly in the TLB entries [21]. Every TLB entry would then have information about the location of the respective page in the DRAM cache. However, this requires fixing the DRAM cache line size to the Operating System (OS) page size, which can be inefficient for applications with low spatial locality and wasteful in terms of off-chip bandwidth and DRAM cache space. The inefficiencies of this approach would be even more evident in systems that use super-pages/huge-pages [27–29]. Other techniques such as *Alloy Cache* and *Compound Access Scheduling* collocate DRAM cache data and tags in the same DRAM row to allow faster accesses [13,24]. These designs either require a direct mapped cache organization or customizing the cache associativity and cacheline size to the DRAM row size. Such restrictions can impact the hit rate or waste DRAM cache capacity.

In summary, although existing DRAM cache designs reduce the tag lookup latency, they do so by either introducing a constant latency to all accesses, as in the case of tag-caches, or by severely limiting critical DRAM cache parameters such as cache line size and associativity, ultimately limiting the performance of DRAM caches.

Thesis Approach: To minimize the tag lookup overheads for DRAM caches this thesis proposes *Decoupled Fused Cache* (DFC), described in Chapter 2 of this thesis. DFC is a new DRAM cache architecture that mitigates the cost of accessing the DRAM cache tags while enforcing minimal design restrictions. Figure 1.2 provides a conceptual overview of our proposal. DFC takes advantage of the redundancy in the tags within the LLC as well as across the LLC and DRAM cache tag arrays and uses the LLC tag-array to store information about the location of data in the DRAM cache. In the common case, this allows DFC to access the DRAM cache data array without looking up its tags which are stored in 3D-stacked DRAM. DFC decouples the location of LLC tags from the location of the LLC lines in the LLC data array in a way that resembles Decoupled Sector Caches [30]. In a nutshell, an LLC tag is associated with a DRAM cache line, which consists of several LLC lines, while the LLC management (validity, dirty, etc.) is performed (and related information is stored) at LLC line granularity. DFC can support a configurable (at boot time) DRAM cache line size, which is a power-of-two multiple of the LLC line size. In essence the only limitation of DFC is that the DRAM cache lines needs to be at least twice as large as an LLC line.

Contrary to existing work, DFC mitigates the DRAM cache tag access overheads without imposing significant design restrictions. More precisely, DFC does not require any OS support, it does not limit DRAM cache associativity, it does not impose additional overheads in every access, and does not affect LLC performance. Still, DFC offers zero tag access overhead in the common case, and can dynamically (at boot time) support variable DRAM cache line sizes.

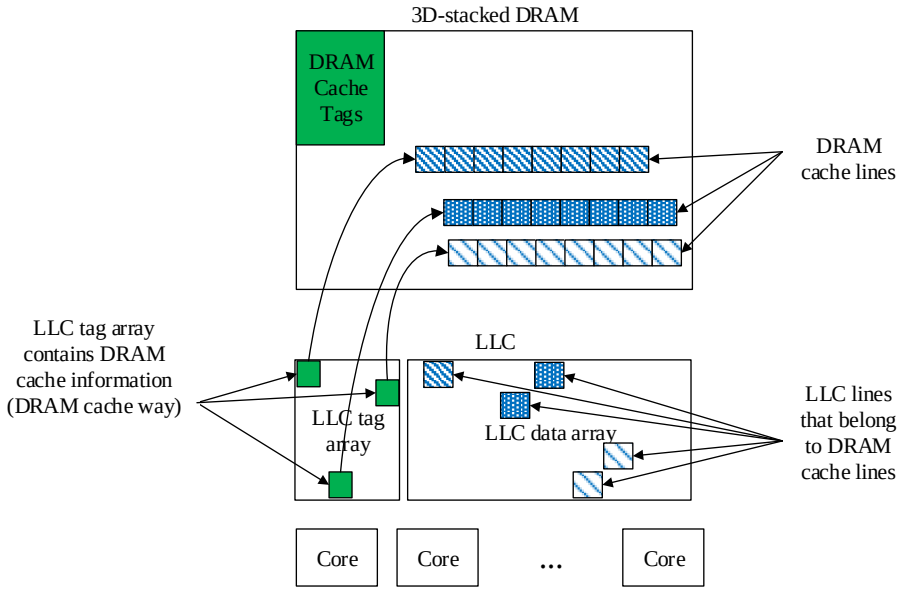


Figure 1.2: Decoupled Fused Cache (DFC) overview.

Contributions: For DRAM caches this thesis proposes Decoupled Fused Cache [31], a new cache hierarchy which:

- Stores information about the contents of the DRAM cache in the LLC to avoid DRAM cache tag lookups for most LLC misses.
- Supports any DRAM cache line size power-of-two multiple of a LLC cache line (up to 4KB in our experiments), which is configurable at boot time.
- Improves performance by an average of 11% compared to a state-of-the-art DRAM cache [19].
- Reduces DRAM cache traffic by 25% and DRAM cache energy by 24.5% versus the current state-of-the-art.

1.2.2 Minimizing Migration Overheads and Improving Data Selection

The second objective of this thesis is to improve the performance of systems that use 3D-stacked DRAM as part of a flat address space with data migration. It does so by reducing the migration traffic overheads and improving the selection of data that are migrated. The ideas behind this objective are to:

> *Use the on-chip LLC to guide data selection for migration based on the observed spatial locality.*
>
> *Reduce migration traffic by not migrating cache lines already present in the LLC, as they can be written to their new location upon eviction.*

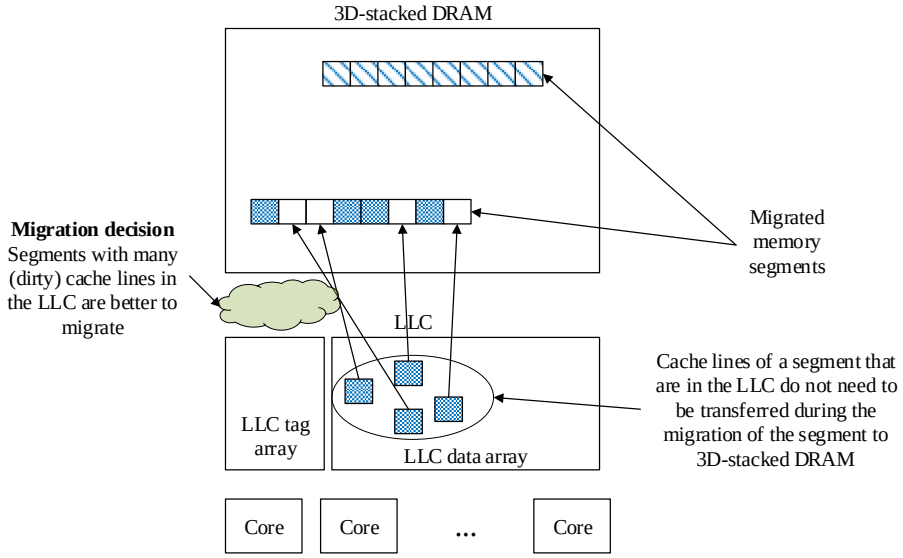


Figure 1.3: LLC-guided data migration (LGM) overview.

Related Work: There exists a large body of prior work on data migration for hybrid memory systems. The core component of every data migration strategy, is the way a memory segment is selected for migration. Most approaches use counters to keep track of accesses to memory segments [32] or counters for every segment within a group [7, 10]. So far, the most promising approach has been the activity tracking mechanism proposed by Mempod [9], which uses the *Majority Element Algorithm* (MEA) [33]. MEA has been shown to predict the hottest pages within an interval with high accuracy and at minimal hardware cost.

Different approaches trigger migrations in different ways. Many of them do it on time intervals [9, 32], while others do it on an event, e.g. CAMEO migrates at every memory access that is not in the 3D-stacked DRAM [34]. Some approaches trigger migrations when the values of selection counters go beyond some threshold [7, 10].

Another aspect that characterizes the different approaches is whether the migration mechanism is based on software or hardware, or a combination of the two. Some migration mechanisms rely on the Operating System (OS) with some hardware support to identify the working set and orchestrate the migration [32], others only involve the hardware and are transparent to the OS [7, 9, 10, 34].

Overall, current approaches to data migration are far from their ideal performance. Their performance is limited partly due to the increased overheads of data migration and also because of the difficulty to select data to migrate with good potential for future reuse.

Thesis Approach: For data migration between off-chip and 3D-stacked DRAM, this thesis proposes LLC-guided Migration (LGM), a novel scheme for data migration in hybrid memory systems aiming both at improving the selection of migrated data as well as at reducing their traffic overheads. Figure 1.3 provides a conceptual overview of our proposed design, described in Chapter 3 of this thesis. Improving the selection of migrated data is achieved by using the LLC to guide the selection of memory segments to be migrated by detecting high spatial and temporal locality. More precisely, the

LLC is used to identify memory segments that have a large number of cachelines on-chip. This is an indication for potential future reuse, which gets stronger when these cachelines are dirty. Employing the LLC to select segments for migration ensures that these segments are at that moment –at least partly– present in LLC. This can be used for reducing migration traffic. This is because when a fraction of a memory segment is located in the LLC, it can be omitted from the migration to reduce the migration traffic, as long as the LLC writes it back to memory when evicted.

The main novelty of our approach is the following: Firstly, the migration overheads are reduced by avoiding traffic for cache lines already present in the LLC. Secondly, the quality of selecting of data for migration is improved. Even more important is that segments are selected for migrations when a large fraction of them resides in the LLC, this timing further reduces the migration traffic.

Contributions: For data migration this thesis proposes LLC-guided Data Migration (LGM) [35], a data migration scheme which:

- Employs the LLC to detect locality and leverages it for selecting data with higher potential for reuse to migrate.
- Reduces the migration traffic overhead by avoiding to migrate data that already reside in the LLC.
- Increases the benefits of the above migration traffic reduction because the selected data are more likely to be in the LLC when migrated.
- Reduces migration traffic to almost half and enables more data to be migrated therefore increasing the ratio of memory requests serviced by the 3D-stacked DRAM.
- Improves performance by 12.1% and reduces memory system dynamic energy by 13.2% compared to the current state-of-the-art [9].

1.2.3 Combining Caching and Data Migration

The third objective of this thesis is to define a new memory system design that combines the strengths of DRAM caches and data migration while reducing their respective weaknesses. The main idea behind this objective is to:

> *Use a small part of the 3D-stacked DRAM as a cache to retain the reactiveness of caches with minimal capacity waste and use the rest of the 3D-stacked DRAM for data migration.*
>
> *Design a unified mechanism that seamlessly manages both DRAM cache and data migration metadata and reduces their respective overheads.*

Related Work: DRAM caches, as described above, present one solution to the utilization of 3D-stacked DRAM which trades capacity for less wasted bandwidth [13, 19, 21, 24]. Data migration designs present a different tradeoff, preserving capacity for more expensive swapping operations between the conventional and 3D-stacked DRAM [7, 9, 10, 32, 34]. Furthermore, there are two designs that strike a different balance. Chameleon is based on PoM with the added option to economize on migration bandwidth when the software does not use some memory space [7] [8]. Chameleon

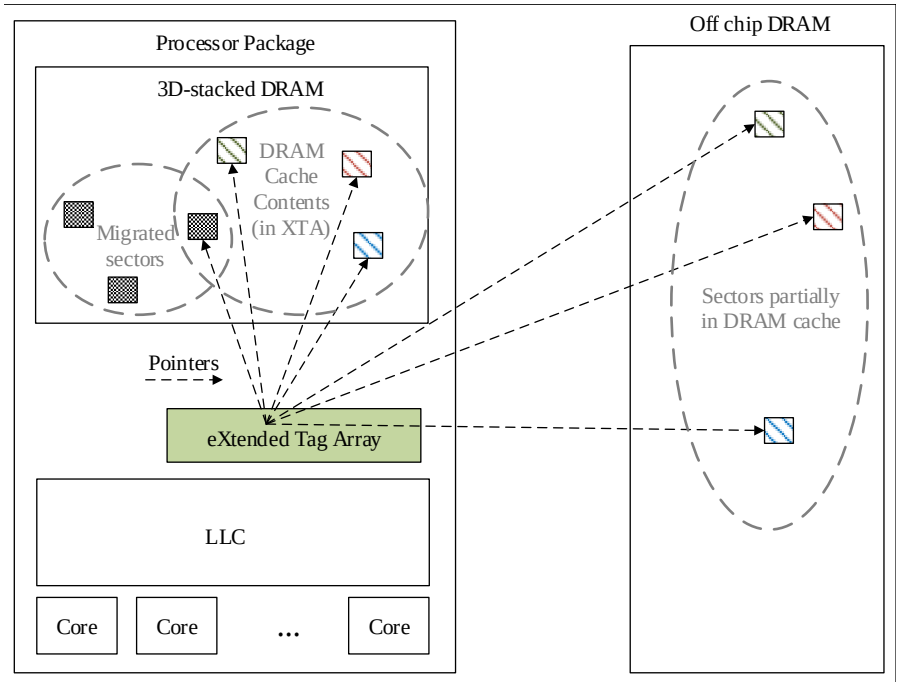


Figure 1.4: $Hybrid^2$ Overview.

does so by involving the Operating System and adding new instructions to the instruction set, this enables the underlying PoM migration mechanism to safely overwrite memory that is not in use, therefore alleviating some of the migration traffic overheads. Intel's Knights Landing provides the option to split the *MCDRAM* between DRAM cache and flat address space, however, it does not support transparent data migration in hardware. Instead it moves the burden to the software to explicitly allocate data to the 3D-stacked DRAM through the *hbw_malloc()* function [36, 37].

Thesis Approach: In order to combine caching and migration in the same memory system this thesis proposes $Hybrid^2$. $Hybrid^2$ aims to preserve the advantages of both caching and migration as well as to minimize their overheads. A conceptual overview of this design is presented in Figure 1.4. $Hybrid^2$ employs a small portion of the 3D-stacked DRAM to implement a DRAM cache and offers the rest of its capacity to main memory. The DRAM cache quickly adjusts to the working set of the workload by fetching to the 3D-stacked DRAM all the data that is requested by the processor. Besides preserving most of the 3D-stacked DRAM 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 additionally extended to act as a cache for the remapping metadata required for data migration. Migrations are decided upon eviction from the DRAM cache which allows observing the access patterns in the DRAM cache and make informed migration decisions. Finally, the data of the DRAM cache can be located anywhere in the 3D-stacked DRAM through the use of indirection, therefore, data selected to be kept after eviction from the cache do not require relocation within the 3D-stacked DRAM, this avoids unnecessary traffic.

Contributions: To combine the benefits of DRAM caches and data migration this thesis proposes [38], a memory system design which:

- Uses a small part of the 3D-stacked DRAM as a cache and utilizes the rest for data migration.
- Employs a small on-chip tag array which eliminates DRAM cache tag lookup overheads and also to alleviate the data migration remapping metadata overheads.
- Utilizes the DRAM cache as a staging area to select the data with most potential for future reuse and selectively migrates data when evicted from the DRAM cache.
- Avoids unnecessary data movement in the 3D-stacked DRAM by employing indirection in the DRAM cache tag array.
- Improves performance on average by 6.4-9.1% compared to migration mechanisms.
- Offers 5.9-24.6% more main memory capacity than caches while giving away only 0.3-5.1% of performance on average.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the design and evaluation of *Decoupled Fused Cache*, A DRAM cache design that uses the LLC tag array to store information about the contents of the DRAM cache. Chapter 3 presents the design and evaluation of *LLC-guided Data Migration in Hybrid Memory Systems*, a data migration scheme that uses the LLC to select memory segments to migrate based on their observed spatial locality and to reduce the migration traffic overheads. Chapter 4 presents the design and evaluation of *Hybrid^2: Combining Caching and Migration in Hybrid Memory Systems*, a memory system design that combines the benefits of caching and migration in the same memory system while tackling their respective overheads. Finally, The conclusions of this thesis are presented in Chapter 5.

Chapter 2

Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache

As 3D-stacking technology becomes more widely used, DRAM and processor can be integrated in the same package sustaining higher memory bandwidth and substantially lower energy per transferred bit [39]. Compared to the narrower-channels of traditional off-chip DRAM, 3D-stacked DRAM achieves 2-4 times higher bandwidth [40]. This in turn alleviates the queuing latency and contention effects that traditional DRAM channels suffer from leading to lower average memory access time. Currently, 3D-stacked DRAM falls short in meeting the main memory capacity requirements of high-end systems, but it is orders of magnitude larger than on-chip SRAM caches. Their capacity and high bandwidth makes 3D-stacked DRAMs a suitable building block for an off-chip (off the processor die) cache. DRAM caches (DC) exploit the coarse grain spatial locality of applications and reduce the number of accesses to main memory, however they are still far from their ideal access latency [16]. This, in turn, may negatively impact system performance as it adds delay to all memory requests that miss in the on-chip caches.

A significant factor of DC latency, as in any cache, is the tag lookup time, which is added to the memory access time for both hits and misses and can be as costly as the data access. As shown in Figure 2.1, a simple DRAM cache (DC) gives away an average 37% of system performance compared to using an ideal DC with zero tag access latency. Designs that try to mitigate the tag access overhead, such as *Fusioncache* (FC) [41], which uses the LLC tags to access the DRAM cache, and *ATCache*, which is a DRAM Cache with an on-chip Tag-Cache (DCTC) [19] bridge that performance gap significantly, however there is still ample room for improvement, 16% for the DCTC and 11% for the FC on average.

The DC tag access latency depends on the tag management. Each design choice comes with tradeoffs that are tightly related to the DC-cacheline size. Since DCs are in the order of hundreds of MB in size, the options for the DC-cacheline size range from the conventional cacheline size of on-chip caches (often 64 Bytes) to a full Operating System (OS) page (4KB). Various designs have been proposed advocating particular DC-cacheline sizes and as also shown in our evaluation there is no single size that

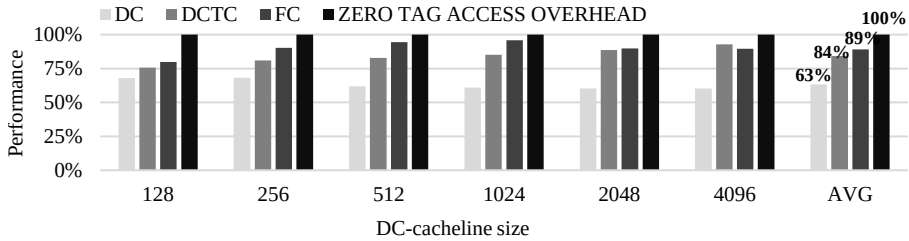


Figure 2.1: Performance normalized to DRAM cache without tag access overhead. The depicted results are the average performance results retrieved for the SPEC and NAS benchmarks used in our evaluation section.

fits all applications [12, 21, 24]. Smaller DC-cachelines offer more flexibility and more efficient use of the cache bandwidth when the application is characterized by low spatial locality. Larger DC-cacheline offer better prefetching and overall better performance when the workloads exhibit spatial locality [20, 23]. On the other hand, smaller DC-cachelines require more tag storage than larger ones for the same cache size making it infeasible to store them on chip. Even for larger cachelines, the cost of storing the tags on chip is not negligible and it could otherwise be utilized for a larger on-chip Last Level Cache (LLC). Storing the DRAM cache tags in DRAM is more space efficient and as such allows for smaller DC-cachelines but it results in substantially higher tag access latency as well as increased DRAM cache traffic [22].

Several designs have been proposed aiming to reduce the DC tag access latency, however they are not able to nullify it and some of them introduce significant constraints to the system. One such design employs an on-chip SRAM cache of the DC tags [19]. This reduces the average DC tag lookup latency however it adds a constant delay to every DC access for accessing the tag-cache and more on-chip resources are occupied for caching the DC-tags. Another technique places the DC addresses directly in the TLB entries [21]. Every TLB entry would then have information about the location of the respective page in the DRAM cache. However, this requires fixing the DC-cacheline size to the OS page size, which can be inefficient for applications with low spatial locality and wasteful in terms of off-chip bandwidth and DC space. The inefficiencies of this approach would be even more evident in systems that use super-pages/huge-pages [27–29]. Other techniques such as *Alloy Cache* and *Compound Access Scheduling* collocate DC data and tags in the same DRAM row to allow faster accesses [13, 24]. These designs either require a direct mapped cache organization or customizing the cache associativity and cacheline size to the DRAM row size. Such restrictions can impact the hit rate or waste DC capacity. In summary, although existing DC designs reduce the DC tag lookup latency, they do so by either introducing a constant latency to all accesses, as in the case of Tag-cache, or by severely limiting critical DRAM cache parameters such as DC-cacheline size and associativity. As a consequence, minimizing the tag lookup latency remains an open challenge in the design of a DC.

In this thesis we propose *Decoupled Fused Cache* (DFC). DFC is a new DRAM cache architecture that mitigates the cost of accessing the DRAM cache tags while enforcing minimal design restrictions. Our design achieves zero tag access latency in the common case by storing information about the location of DC cachelines in the tag array of the on-chip LLC. DFC can support a configurable (at boot time) DC cacheline

size, which is a power-of-two multiple of the LLC-cachelines. In essence the only limitation of our proposal is that the DC-cachelines needs to be at least twice as large as an LLC-cacheline. Then, considering an inclusive cache model, each cacheline stored in the LLC is always part of a DC-cacheline stored in the DC. Our work builds upon our initial work on DRAM caches, *Fusioncache*, which used the LLC tags to access the DC reducing its latency [41]. Our previous design required LLC cachelines that belong to the same DC-cacheline to be placed on the same LLC set. *Decoupled Fused Cache* overcomes this limitation by decoupling the LLC tag in a way that resembles *Decoupled Sectored Caches* [30] yielding significant performance improvements – up to 100% for particular benchmarks and design points. In a nutshell, an LLC tag is associated with a DC-cacheline, which consists of several LLC cachelines, while the LLC management (validity, dirty, etc.) is performed (and related information is stored) at LLC cacheline granularity.
Concisely, the contributions of *Decoupled Fused Cache* are the following:

- A new cache hierarchy is proposed that fuses the on-chip LLC and DRAM cache tags to achieve zero-latency DC access without affecting LLC performance;
- The proposed solution supports any DC-cacheline size power of two multiple of the LLC-cachelines;
- An evaluation and comparison against related approaches showing that *Decoupled Fused Cache* achieves better performance and energy efficiency.

The remainder of this Chapter is structured as follows: Section 2.1 uses a motivating example to introduce background information and highlight the challenges addressed in this work. In Section 2.2 the proposed *Decoupled Fused Cache* design is presented. Section 2.3 offers the evaluation and comparison of our work. Section 2.4 discusses in more detail related work on DRAM caches and in particular on designs that aim at reducing DC tag lookup latency. Finally, Section 2.5 summarizes our conclusions.

2.1 Background and Motivation

Considering a system as the one illustrated in Figure 2.2 with an inclusive DRAM cache (DC) placed between the on-chip Last Level Cache (LLC) and main memory, we present a motivating example highlighting the challenges addressed in this work. We discuss first the functionality of a DC and LLC when organized in a conventional way and subsequently contrast it with our previous *Fusioncache* (FC) design pointing to its advantages and drawbacks addressed by *Decoupled Fused Cache* (DFC).

A conventional system with a DRAM cache (DC) would function as illustrated in the example of Figure 2.3a. Both the LLC and the DC use the upper part of an address as a tag and the remaining bits before the byte-offset for selecting a set as shown in Figure 2.3b (Figure 2.5 shows the addresses and fields of the LLC- and DC- cachelines used for the examples in Figures 2.3, 2.4 and 2.6).

Let us consider that the DC uses for its tags equal or fewer address bits than the LLC assuming it has equal or larger cachelines and number of sets. Then, in an inclusive cache hierarchy, a cacheline stored in the LLC will be also stored in the DC as part of a DC-cacheline. As a consequence, (part of) an LLC tag would identify a DC-cacheline and would be stored in the DC tag-array, too. For example,

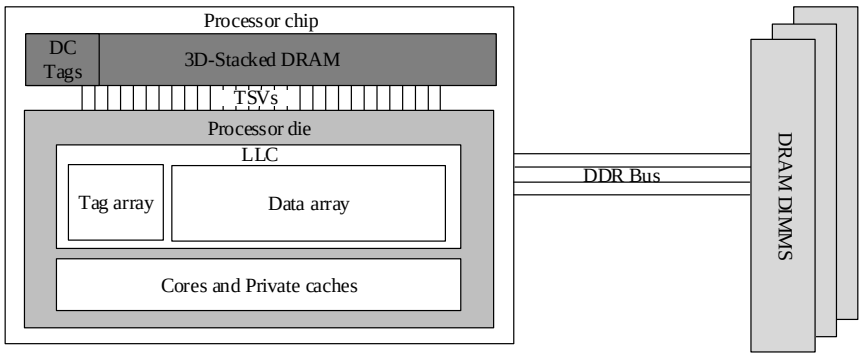
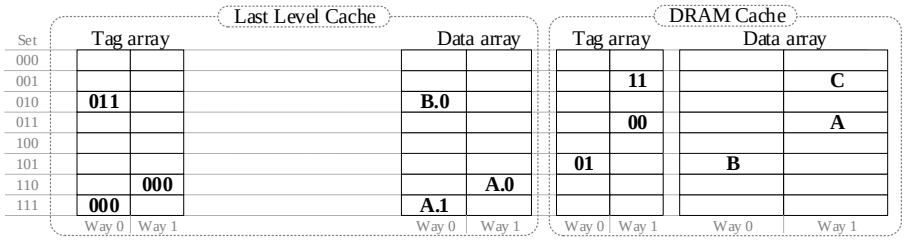


Figure 2.2: System Overview.

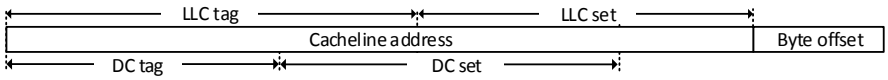
in Figure 2.3a, we can observe that the upper part of the tag of an LLC-cacheline (i.e., 011 for LLC-cacheline *B.0*) is the same with the DC-tag of the respective DC-cacheline (i.e.,01 for DC-cacheline *B*). Moreover, considering some spatial locality, it is reasonable for a cache to host consecutive cachelines which would have the same tag. Then, the tag of such cachelines would be repeated in multiple (consecutive) sets of the tag array as shown in the LLC of Figure 2.3a. For instance, consecutive LLC-cachelines *A.0* and *A.1*, which belong to the same DC-cacheline (*A*), store their identical LLC-tag (000) twice in the LLC tag array in different sets. In summary, we can observe that: firstly, (parts of) the LLC tags are also stored in the DC tag-array, and secondly, the LLC tags for consecutive LLC-cachelines are duplicated in multiple sets of the LLC tag array.

Fusioncache is based on the first above observation to reduce the DC tag access latency. It appends LLC tag array entries with information for accessing their respective DC-cacheline. Thereby, LLC accesses that would miss in the LLC, i.e., *B.1* in the above example, but their tags are stored in the LLC tag-array would need no further DC-tag access. However, an LLC access that falls to a particular DC-cacheline may hit one of several LLC sets as observed in Figure 2.3a. In this example, an access to DC-cacheline *A* may go through the LLC set that stores either *A.0* or *A.1*. In order to ensure that a single LLC access can provide a definite answer about the DC tags, *Fusioncache* restricts all LLC-cachelines that belong to the same DC-cacheline to be placed in the same LLC set as shown in Figure 2.4a. Our second above observation, that the LLC tags for consecutive LLC-cachelines are duplicated in multiple sets of the LLC tag array, also comes to *Fusioncache*'s advantage, as the tag for these LLC-cachelines is then stored only once saving space in the LLC tag array and increasing the number of DC tags that can be stored in the LLC. For example, the tag of DC-cacheline *C* is stored in the LLC of *Fusioncache* of Figure 2.4a without a corresponding LLC-cacheline and can be used for accessing the DC. In order to enforce this LLC-cacheline placement *Fusioncache* uses higher order address bits for indexing LLC as depicted in Figures 2.4b and 2.5. As explained in our previous work, this design choice restricts the effective LLC associativity and limits performance for particular memory access patterns especially in large DC-cacheline sizes [41].

Taking a closer look to the *Fusioncache* example of Figure 2.4a, an LLC-tag array entry stores the DC-tag and, besides the standard fields needed for the management of the two caches (validity, dirty, etc), it also stores the DC-way of the corresponding DC-cacheline, the offset of the stored LLC cacheline, and a pointer to the way of the

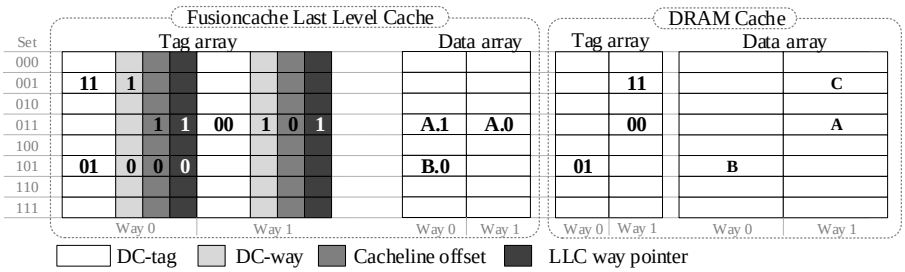


(a) Location of LLC-cachelines and tags in conventional cache.

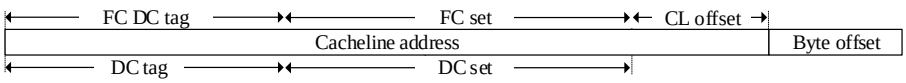


(b) Conventional cache address breakdown

Figure 2.3: Conventional cache example and address breakdown.

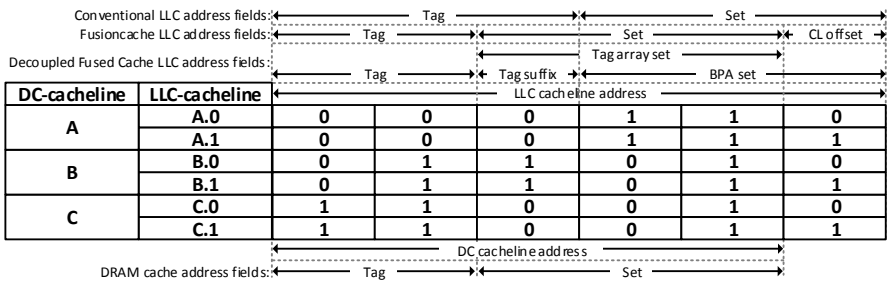


(a) Location of LLC-cachelines and tags in *Fusioncache* (FC)



(b) *Fusioncache* address breakdown

Figure 2.4: *Fusioncache* example and address breakdown.



DC-cacheline	LLC-cacheline						
A	A.0	0	0	0	1	1	0
	A.1	0	0	0	1	1	1
B	B.0	0	1	1	0	1	0
	B.1	0	1	1	0	1	1
C	C.0	1	1	0	0	1	0
	C.1	1	1	0	0	1	1

Figure 2.5: Address fields for indexing and tag matching for conventional LLC, *Fusioncache* LLC, *Decoupled Fused Cache* LLC, and DRAM cache.

LLC tag-array that stores the corresponding LLC-tag. In our example, LLC cachelines *A.0* and *A.1* have their tag stored in only one entry of the LLC-tag array (way-1 of the corresponding set). The same tag (00) is used for accessing the DC in way 1 as indicated by the DC-way field. The LLC cacheline offset is 1 and 0 for the LLC-cachelines *A.1* and *A.0*, respectively. Finally, the field pointing to the way that stores the tag for these two LLC-cachelines is 1 for both entries as their tag (00) is stored in way-1 of the LLC-tag array. Accessing the LLC requires matching each tag, as indicated by the LLC-way pointer, and each LLC-cacheline offset. When a DC-tag is matched but the requested LLC-cacheline is not present in the LLC, the DC data array can be accessed directly by use of the DC-way information, otherwise a DRAM access for the DC-tag is required before accessing the DC data array.

Effectively, *Fusioncache* indexes the LLC as if it was a cache with a DC-cacheline size and uses an offset to identify the particular LLC-cacheline. Thereby, the LLC tag array acts like a cache of DC-tags storing DC-tags used in previous LLC accesses, even if all corresponding LLC-cachelines have been evicted. However, the modified indexing of the LLC and the placement restriction of all LLC-cachelines that belong to the same DC-cacheline to reside on the same LLC set affects performance. When the number of LLC-cachelines per DC-cacheline is higher than the LLC associativity, sequential accesses to the same DC-cacheline would exhaust the LLC set and result in unwanted evictions. On the contrary, in a conventional LLC, the LLC-cachelines of the same DC-cacheline map to different LLC sets.

As explained in the next section, *Decoupled Fused Cache* addresses this problem allowing the LLC of a fused cache to operate as in the conventional way, using lower order address bits for indexing and still storing DC-tags in the LLC-tag array.

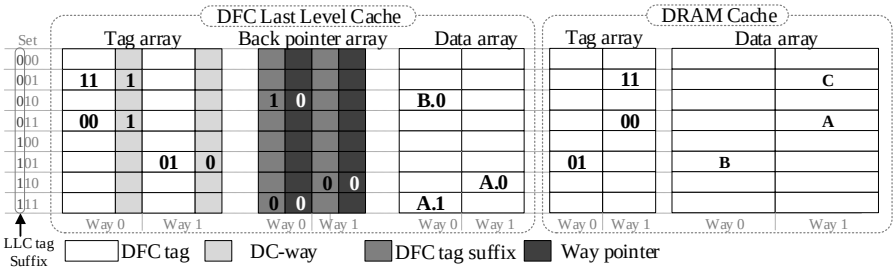
2.2 Decoupled Fused Cache design

The *Decoupled Fused Cache* (DFC) is based on the same two observations exploited by the *Fusioncache*. It takes advantage of the redundancy in the tags within the LLC as well as across the LLC and DC tag arrays and uses the LLC tag-array to store information about the location of data in the DRAM cache (DC). In the common case, this allows DFC to access the DC data array without looking up its tag array. As opposed to *Fusioncache*, DFC does not restrict LLC-cachelines of the same DC-cacheline to sit on the same LLC set. This is achieved by decoupling the location of LLC tags from the location of the LLC-cachelines in the LLC data array in a way that resembles Decoupled Sector Caches [30].

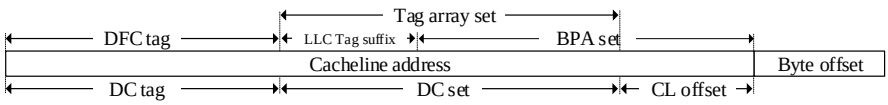
In DFC, tags in the LLC are stored in a *tag-array*, which is indexed as if it were a DC-tag array. Then, a second, *Back Pointer Array* (BPA), which follows the indexing of the LLC data array, is used to store the information for the LLC management (valid, dirty, LRU bits of the corresponding LLC data entry). In addition, each entry of the BPA points to the tag array entry, which stores the tag of the corresponding LLC cacheline. As explained below, pointing from the BPA to the tag-array requires information about the correct way of the tag-array as well as the LLC tag suffix[1]. Note that both the tag and back pointer arrays have equal number of sets and ways as the LLC data-array.

Using the above indirection, DFC decouples the location of tags and data in the LLC. In doing so, it allows the LLC-cachelines to be placed as in a conventional LLC

[1]That is the address bits that need to be appended to the DC-tag in order to form the LLC-tag.



(a) Location of LLC-cachelines and tags in Decoupled Fused Cache (DFC).



(b) DFC address breakdown.

Figure 2.6: Decoupled Fused Cache example and address breakdown.

and its tags to be organized as in a DC tag array. Then, each entry of the tag array can store information for the DC-cacheline associated with the stored tag. In particular, the location of the corresponding DC-cacheline and some DC management fields. As a consequence, the DFC avoids DC-tag accesses for all the tags stored in the LLC without restricting LLC-cacheline placement and hence without affecting the LLC performance.

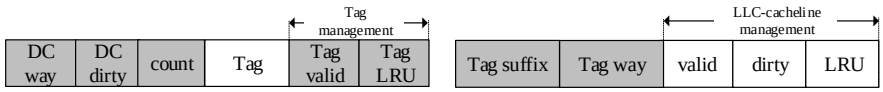
Figure 2.6a illustrates the DFC functionality for the same example used in the previous section to demonstrate the FC and conventional cache. Figure 2.5 shows the addresses as well as the respective address fields used for indexing and tag-matching for the LLC- and DC-cachelines used in the examples in Figures 2.3a, 2.4a, and 2.6a for a conventional cache, *Fusioncache*, and *Decoupled Fused Cache*, respectively. Notice that DFC keeps the same data placement in the LLC as in the conventional cache of Figure 2.3a. As opposed to FC, LLC-cachelines that belong to the same DC-cacheline are placed in different sets in DFC (e.g *A.0* and *A.1*). At the same time, DFC keeps only one tag for all cachelines that belong to the same DC-cacheline in the LLC, economizing space and ensuring that DRAM cache information is retrieved with a single LLC lookup.

Next, the details of DFC are discussed, explaining first the organization of the tag arrays, then, the indexing and tag matching mechanism and finally analyzing the hardware cost of the DFC design.

2.2.1 DFC tag arrays:

DFC splits the tag array of the LLC in two parts which are indexed by different parts of the cacheline address, the resulting two arrays are:

- The ***Tag Array*** which holds the tags for the DC-cachelines.
- The ***Back Pointer Array*** (BPA) which holds pointers that associate every LLC-cacheline with a tag by specifying the set and way in the tag array in which it is located.



(a) A DFC Tag Array entry.

(b) A DFC back pointer array entry.

Figure 2.7: DFC Tag array and BPA entries, shaded fields indicate additions for DFC.

DFC introduces some extra fields in the tag arrays beyond those in a conventional cache. These extra fields facilitate locating the LLC-cachelines using DC-cacheline granularity tags and additionally they store the information needed to access the DRAM cache (DC). These extra fields are shaded in Figures 2.7a and 2.7b for the *tag array* and the *Back Pointer Array* (BPA), respectively. These fields are:

- **Tag Array:**
  - **Tag valid, Tag LRU:** Additional valid and replacement policy fields for the tags, since each tag can be associated with several LLC-cachelines, the tag array must handle validity and replacement independently of the LLC data array.
  - **DC way:** The way in the DRAM cache set in which the DC-cacheline identified by this tag resides.
  - **DC dirty:** Dirty bit for the DC-cacheline in the DRAM cache, since LLC-cacheline evictions are directly forwarded to the DRAM cache, a dirty bit is needed to keep track of written blocks locally in the LLC tag array. This also allows to update the DRAM cache tags (in DRAM) only when a tag is evicted from the LLC tag array.
  - **count:** A counter to keep track of the number of LLC-cachelines that reference a tag. This field is optional as DFC can operate correctly without it, however, as we explain later in our evaluation in Section 2.3, it helps avoid unnecessary BPA lookups when a tag has to be evicted from the tag array.
- **Back Pointer Array:**
  - **Tag suffix:** Since each tag can be associated with multiple LLC-cachelines, each cacheline can potentially belong to any tag in a subset of the sets of the tag array. That subset is identified by the tag-suffix part of the address (Figure 2.6b) and must be stored in the back pointer array for every LLC-cacheline.
  - **Tag way:** To fully identify the correct tag for an LLC-cacheline in the tag array the way in the set must also be stored in the BPA.

2.2.2 DFC Indexing:

Figure 2.6b shows the breakdown of an address and the bit fields used to index the tag array and the Back Pointer Array (BPA) in the *Decoupled Fused Cache* LLC.

- **BPA Set:** The BPA is indexed using the same indexing bits as conventional caches, these are the Least Significant (LS) bits after the byte offset of the address.

- **Tag array set:** The tag array is indexed by the same bits of the address that would index a cache with a cacheline size equal to the DC-cacheline size. These are the LS bits right after the byte offset and LLC-cacheline offset (CL Offset) parts of the address. The CL Offset depends on the ratio of LLC-cachelines per DC-cacheline and can be 2 to 6 bits for 128 Byte to 4KB DC-cacheline sizes, respectively.

With this indexing, the LLC-cachelines are placed in the LLC data array in the same sets as they would be placed in a conventional LLC. Just like conventional caches, consecutive LLC-cachelines that are parts of the same DC-cacheline will be placed in consecutive sets. In contrast to FC, the LLC-cachelines of the same DC-cacheline are not forced into the same set and thus the set conflicts introduced by FC are avoided.

The tag that identifies an LLC-cacheline can only reside in a subset of the sets in the tag array, the size of this subset depends on the ratio of LLC-cachelines per DC-cacheline. For 64-Byte LLC-cachelines and 4KB DC-cachelines the DC-tag for an LLC-cacheline can reside 64 different sets, for 128-Byte DC-cachelines it can reside in two different sets. As demonstrated by Figure 2.6b, the *tag array set* is comprised of the Most Significant (MS) bits of the *BPA Set* and the *LLC tag suffix* parts of the address. In the DFC example in Figure 2.6a, the tag for a cacheline located in set 110 can only be located in sets 011 and 111 depending on the LLC tag suffix. For cacheline *A.0* the LLC tag suffix is 0 (as shown in Figure 2.5) and so the tag is located in set 011 of the tag array. This indexing allows DFC to decouple the tags from the LLC-cachelines that reference them in order to **(a)** save space in the tag array of the LLC to store additional information about the location of DC-cachelines in the DRAM cache and **(b)** to access that information with a single lookup.

2.2.3 DFC tag matching:

Figure 2.8 depicts the block diagram of the DFC LLC showing the address parts used for indexing the individual arrays as well as for matching the tag-array and BPA. In addition, Figure 2.9 offers a flowchart showing the steps of a DFC access for every possible case: LLC hit (Ⓐ), DC hit without tag lookup (Ⓑ), DC hit with tag lookup (Ⓒ), and DC miss (Ⓓ). When a request is made to the LLC, particular parts of the address are used separately as shown in Figure 2.6b. The *Tag array set* part of the address is used to index the tag array and at the same time, the BPA is indexed with the *BPA Set* field of the address. After both arrays have been indexed and the respective sets have been read, their contents must be matched to determine a cache hit or miss.

The matching consists of three steps:

- **Tag match:** The tags in the tag array set are compared against the *tag* field of the address for a match (Figures 2.8 and 2.9 ❶).

- **Suffix Match:** The *tag suffix* field of the set of the BPA are compared with the corresponding part of the address (*tag suffix*) for a match (Figures 2.8 and 2.9 ❷).

- **Way Match:** The way of the matching tag in the Tag array is compared with the *Tag way* field of each matching suffix in the BPA (Figures 2.8 and 2.9 ❸). In case of a match at this stage then this is an LLC hit Ⓐ, otherwise an LLC miss (Figure 2.9 Ⓑ,Ⓒ,Ⓓ).

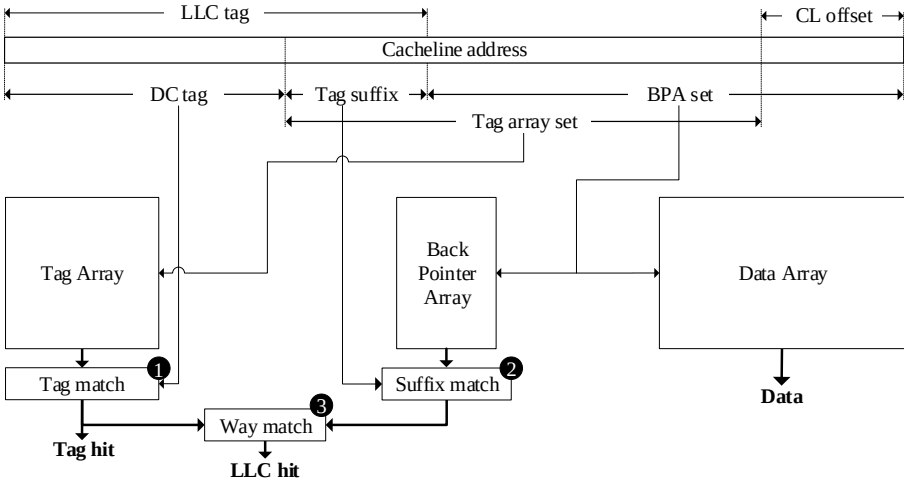


Figure 2.8: DFC address breakdown, indexing and tag matching process.

The *Tag match* ❶ and *Suffix Match* ❷ can be performed in parallel as they are independent of each other. The *Way Match* ❸ step however can start only when the previous two steps have completed. If the requested LLC-cacheline is located in the LLC then there is a LLC hit as shown in Figure 2.9 Ⓐ. Otherwise, an LLC miss will be handled in one of the following two ways depending on whether the tag of the requested address is in the tag array of the LLC:

[a] In case the tag is located in the tag array (there was a match at the *Tag match* ❶ stage) but there were no LLC-cachelines pointing to that tag (*Suffix match* ❷ or *Way match* ❸ failed), then the DC data array can be accessed directly (Ⓑ) using the *DC-way* field of the tag array entry that matched in the *Tag Match* ❶ stage. The physical address of the LLC-cacheline in the DC data array can be calculated from the set and way of the DC-cacheline in the DC. The set can be directly inferred from the physical address of the DC-cacheline and the DC-way is stored in the LLC tag array. The new LLC-cacheline read from the DC is subsequently stored in the LLC data array and its corresponding BPA entry is updated to point to the tag in the LLC tag array. Additionally, the respective fields of the tag entry are updated, in this case, only the replacement (LRU) and *count* bits. The *LRU* of the tag is updated to show that this was the most recently accessed tag in the set. The *count* field is incremented to show that one additional LLC-cacheline is now associated with this tag.

[b] In case the tag is not located in the LLC tag array (no match in the *Tag match* ❶ stage), then the DC tag-array stored in DRAM needs to be accessed (Figure 2.9 ❹). Thereby, it is determined whether the DC-cacheline is located in the DC (Ⓒ) or there is a DC miss and the requested cacheline should be read from the main memory (Ⓓ). In case of a DC miss, a suitable victim DC-cacheline is selected from the DC set using LRU replacement policy and written back to main memory if dirty. The DC-way of the DC-cacheline is then stored in the LLC tag array along with its tag (Figure 2.9 ❺). All subsequent misses of LLC-cachelines that belong to this DC-cacheline can be fetched from the DC directly without accessing the DC tags in DRAM (Figure 2.9 Ⓑ).

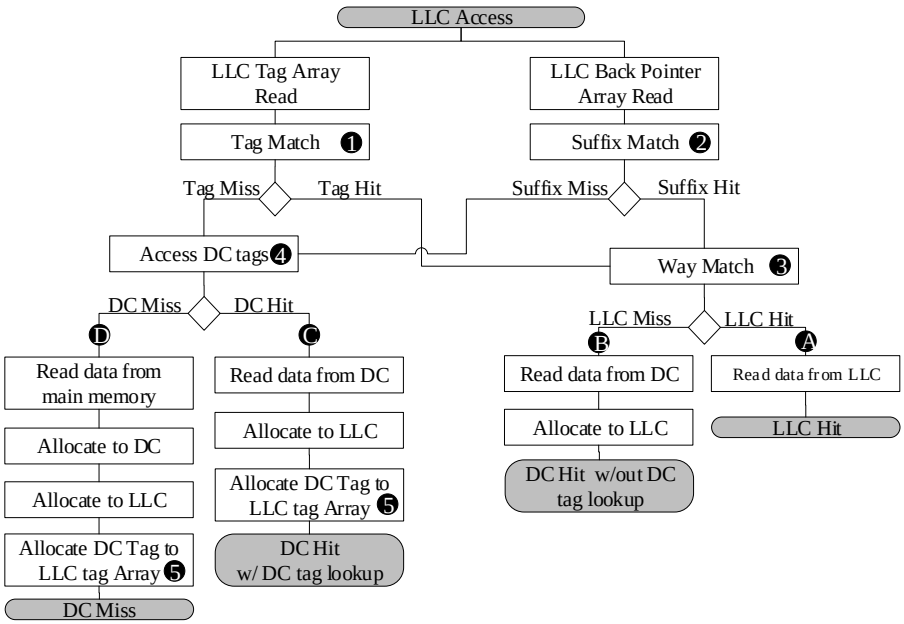


Figure 2.9: Decoupled Fused Cache access flowchart.

2.2.4 DFC Tag Evictions:

When a tag is evicted from the DFC tag array any LLC-cachelines that reference that tag must be evicted from the LLC as well, otherwise they will be orphaned and their *Tag-suffix* and *tag-way* fields in the BPA will point to a stale tag in the tag array. Considering a ratio of N LLC-cachelines per DC-cacheline, in the worst case there might be as many as N LLC-cachelines that must be evicted in N different LLC data array sets. To avoid looking up all the sets that could potentially hold an LLC-cacheline that is associated with a tag, we introduce a counter for every tag to account for the number of these LLC-cachelines, this counter is updated whenever an LLC-cacheline is fetched to or evicted from the LLC. Introducing this counter makes evictions more efficient and, as shown in our experiments, more than 99.5% of the time, with LRU replacement policy for the tags, the counter for the victim tag is zero. A counter equal to zero means that no LLC-cachelines need to be evicted from the LLC because of a tag eviction but also that the corresponding sets in the BPA need not be searched for such LLC-cachelines at all.

Furthermore, when a tag is evicted from the DFC tag array the DC tag array must be updated. The dirty status of the DC-cacheline that corresponds to the evicted tag is copied and the LRU of the DC set is updated. This is necessary since by design all LLC-cacheline writebacks from the LLC to the DC do not need to access the DC tags. Subsequently, the dirty state of the DC-cacheline is stored along with the tag in the tag array (*DC dirty* in Figure 2.7a) and the DC tags are updated only when a tag is evicted from the DFC tag array.

2.2.5 Configurable DC-cacheline size

As shown in our evaluation (Section 2.3) different workloads achieve their best performance with different DC-cacheline sizes. DFC can be configured (at boot time) to accommodate different DC-cacheline sizes ranging from 128 Bytes (two times the LLC-cacheline size) to 4KB (OS page size)[2]. This requires the additional DFC related fields on the LLC tag array and BPA to be provisioned for the worst case size, as shown in the next paragraph. Supporting variable DC-cacheline sizes allows DFC to better fit the needs of a particular workload and maximize performance.

2.2.6 DFC Hardware Overhead:

DFC re-organizes the LLC tag array and changes the indexing and tag matching mechanisms of the LLC. Furthermore, DFC requires the addition of some extra fields in the LLC tag array and splitting it into two separate arrays, these are the *tag array* and the BPA. In this Section we discuss the hardware cost of DFC and in particular its overhead in the LLC tag array.

When calculating the overhead of DFC we must take into account the characteristics of the DC and also the ratio of LLC-cachelines per DC-cacheline. Let the ratio of LLC-cachelines per DC-cacheline be $R \in [2-64]$, the LLC associativity be A, and the DC-associativity be B for the rest of our analysis.

The extra fields needed in the LLC tag arrays are:

- **DC-Tag Array:**
  - **DC dirty:** One bit for the dirty state of the DC-cacheline.
  - **DC way:** Log_2B bits for the DC way where the DC-cacheline is located.
  - **count:** Log_2R+1 bits for the counter of LLC-cachelines that reference this tag (from 0 to R LLC-cachelines).
  - **Tag valid:** Valid bit for the tag.
  - **Tag LRU:** Log_2A LRU bits for Tag replacement.
- **Back Pointer Array:**
  - **Tag suffix:** Log_2R bits that identify the set in which the tag of the LLC-cacheline is located.
  - **Tag way:** Log_2A bits to identify the way in which the tag of the LLC-cacheline is located in its set.

The above listed fields account for an total of $2\times\log_2R+2\times\log_2A+Log_2B+3$ bits per LLC-cacheline. However we can further reduce the cost by one bit per LLC-cacheline by using the valid bit as a part of the counter and offsetting the count by one. Additionally, the tag in a DFC is $\log_2R$ bits smaller than a conventional LLC tag. Thus, the additional space overhead of DFC is: $\log_2R+2\times\log_2A+Log_2B+2$. To support different DC-cacheline sizes, we must account for the worst case overhead of the fields in the Tag Array and the BPA, this is the overhead for the 4KB DC-cacheline.

To quantitatively present the hardware cost in the LLC we use a realistic example that matches our experimental setup configuration. Lets consider a system with 48

[2]Larger DC-cacheline sizes can be supported at little additional cost but we consider up to 4KB DC-cacheline size to keep within the granularities considered in competing designs.

Table 2.1: DFC SRAM overhead.

DC-cacheline size	DC-cacheline / LLC-cacheline ratio	Added Bits per Entry	LLC extra cost %
128 Byte	2	15	224KB (2.5%)
256 Byte	4	16	240KB (2.9%)
512 Byte	8	17	256KB (3.1%)
1024 Byte	16	18	272KB (3.3%)
2048 Byte	32	19	288KB (3.5%)
4096 Byte	64	20	320KB (3.7%)
configurable (128 Byte-4KB)	2-64	20	320KB (3.7%)

bit physical addresses, 64-Byte LLC-cachelines and a 16-way LLC with 8192 sets (total LLC capacity of 8MB). The 6 Least Significant (LS) bits are the byte offset in an LLC-cacheline and are not used for accessing the cache since it operates at LLC-cacheline granularity. The next 13 bits are used to index the 8192 sets of the cache (2^{13} sets). This means that each tag in the tag array is 29 bits long. For a 16 way LLC we also need 4 bits for LRU replacement policy and 2 more bits for valid and dirty flags. The total size for an entry in a tag array of a 8MB 16-way cache is thus 35 bits and the total size of the tag array is 560KB. We also assume a 512MB, 16-way set associative DC as in our evaluation.

Table 2.1 Shows the overhead of DFC in terms of additional storage required in the DFC tag array for every different supported DC-cacheline size. The worst case overhead of the DFC design is 320KB for a 8MB LLC which accounts for a 3.7% area overhead.

The hardware overhead of DFCs indexing and tag-matching mechanisms is very small compared to a conventional LLC in terms of additional space required in the LLC tag array. As far as lookup latency is concerned, the modified indexing and tag-matching mechanisms do not impose extra latency to the cache access compared to a conventional LLC. Steps ❶ and ❷ in Figure 2.8 are faster than a traditional LLC tag lookup because the number of compared bits is smaller. Step ❸, which adds to the latency of steps ❶ and ❷ in practice adds a 32-bit product of sums logic delay. This delay does not add a cycle to the LLC access time as it is within the available slack estimated by *Cacti* [42] after accounting for its logic latency in the same technology node.

2.3 Evaluation

In this section we present the evaluation of the proposed *Decoupled Fused Cache* and compare with state-of-the-art designs that target the tag access cost for DRAM caches. We first present our experimental setup followed by the results of our evaluation in terms of performance and energy consumption for a series of single- and multi-threaded benchmarks for different DC-cacheline sizes.

Table 2.2: System configuration.

Cores	4 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
DRAM Cache	512 MB, 16-way, 2 128-bit channels, 8ns tRCD, 10ns tCAS, 1.6 GHz (DDR 3.2GHz)
Tag cache	256 KB (272KB including its tag array), 8-way, 5 cycles access latency
Main DRAM	8 GB, 2 channels, 64 bit bus, 14ns tRCD, 14ns tCAS, 800 MHz (DDR 1.6GHz)

Table 2.3: Main DRAM and 3D-DRAM energy consumption.

Parameter	3D-DRAM	Main Memory DRAM
RD/WR + I/O energy	6.4pJ/bit	33pJ/bit
ACT/PRE DRAM Row	15nJ	15nJ

2.3.1 Experimental Setup

Our evaluation is performed using an in-house simulator based on *Pin* [43] following the interval-based simulation methodology [44] for the processor and cycle-accurate modelling of the cache and memory system. We simulate a four core processor with private L1 and L2 caches, a shared on-chip last level cache (LLC) and a DRAM cache (DC). Table 2.2 presents the configuration of our system[3]. We use *Cacti v6.5* to determine the access times for the caches and tag arrays [42]. For the main DRAM and 3D-DRAM timing and energy consumption we use the parameters provided by [21]. The DRAM energy parameters are shown in Table 2.3. To estimate the energy consumption of the processor cores we use McPAT [45].

We evaluate our design with both single- and multi-threaded workloads. For single-threaded workloads, we selected a representative subset of the SPEC2006 [46] benchmarks following the guidelines of Phansalkar et al. [47]. For multi-threaded workloads we used the *OpenMP* version of the *NAS Parallel Benchmark* suite [48,49].

We simulate one billion instructions for every thread after a warmup period of 100 million instructions. For the NAS benchmarks we select the simulated portion immediately after the initialization phase of each benchmark, while for the SPEC benchmarks we use *simpoints* to select a representative slice of one billion instructions [50]. The benchmarks used and their memory footprint are shown in Table 2.4.
Finally, our evaluation considers the following design points:

- **Baseline:** A system with no DRAM cache.
- **DRAM Cache (DC):** A system with a DRAM cache and tags-in-DRAM.
- **DRAM Cache with Tag-Cache (DCTC):** a DRAM cache system with tags in DRAM and an on-chip SRAM cache of the DC-tags similar to *ATCache* [19]. The size of the DCTC SRAM tag cache is equal to the size of the SRAM overhead incurred by DFC.

[3] All latencies reported in cycles concern processor clock cycles

- **Fusioncache (FC):** a system with Fusioncache [41].
- **Decoupled Fused Cache (DFC):** a system with the proposed *Decoupled Fused Cache*.
- **Zero Tag Overhead DC:** a system with a DC for which the DC-tag access comes for free without any latency or traffic cost (after the LLC tags have been accessed to determine an LLC miss).

DC, DCTC, and FC are the most relevant competing designs as they are able to support various DC-cacheline sizes. A DC with zero tag overhead is our reference point showing the theoretical limits of the potential performance gain of our approach. Other techniques are not directly included in our comparison because they pose particular design restrictions as described in Section 2.4.

2.3.2 Performance

The performance improvement over the baseline of four above systems that utilize a DRAM cache is first measured per DC cacheline size. Then, the best cacheline size of these four systems is selected per benchmark and compared. Finally, we compare for DCTC, FC and DFC the percentage of DC accesses that did not require a DC tag access in DRAM, as well as their generated DC traffic.

Figure 2.10 shows the performance improvement in terms of Instructions per Cycle (IPC) for each design over the Baseline (without DRAM cache). For this part of our evaluation we consider DC-cacheline sizes ranging from 128 Bytes to 4KB; smaller sizes are not supported by the FC and DFC as they require the DC-cacheline to be at least twice the size of the LLC-cacheline, which is 64 Bytes. Each graph also presents the average performance improvement for all benchmarks as well as for the SPEC and NAS benchmarks separately (*AVG-ALL*, *AVG-SPEC*, and *AVG-NAS*). Each different plot in Figure 2.10 represents a different DC-cacheline size (128 Bytes to 4KB).

Table 2.4: Benchmarks and their memory footprint (in MB).

Bechmark	Footprint(GB)
SPEC (single-threaded)	
leslie3d	80
libquantum	32
mcf	1315
omnetpp	146
lbm	402
NAS (multi-threaded)	
bt.C	720
cg.C	440
dc.A	520
ft.C	510
is.C	1100
lu.C	350
mg.C	2550
sp.C	770
ua.C	360

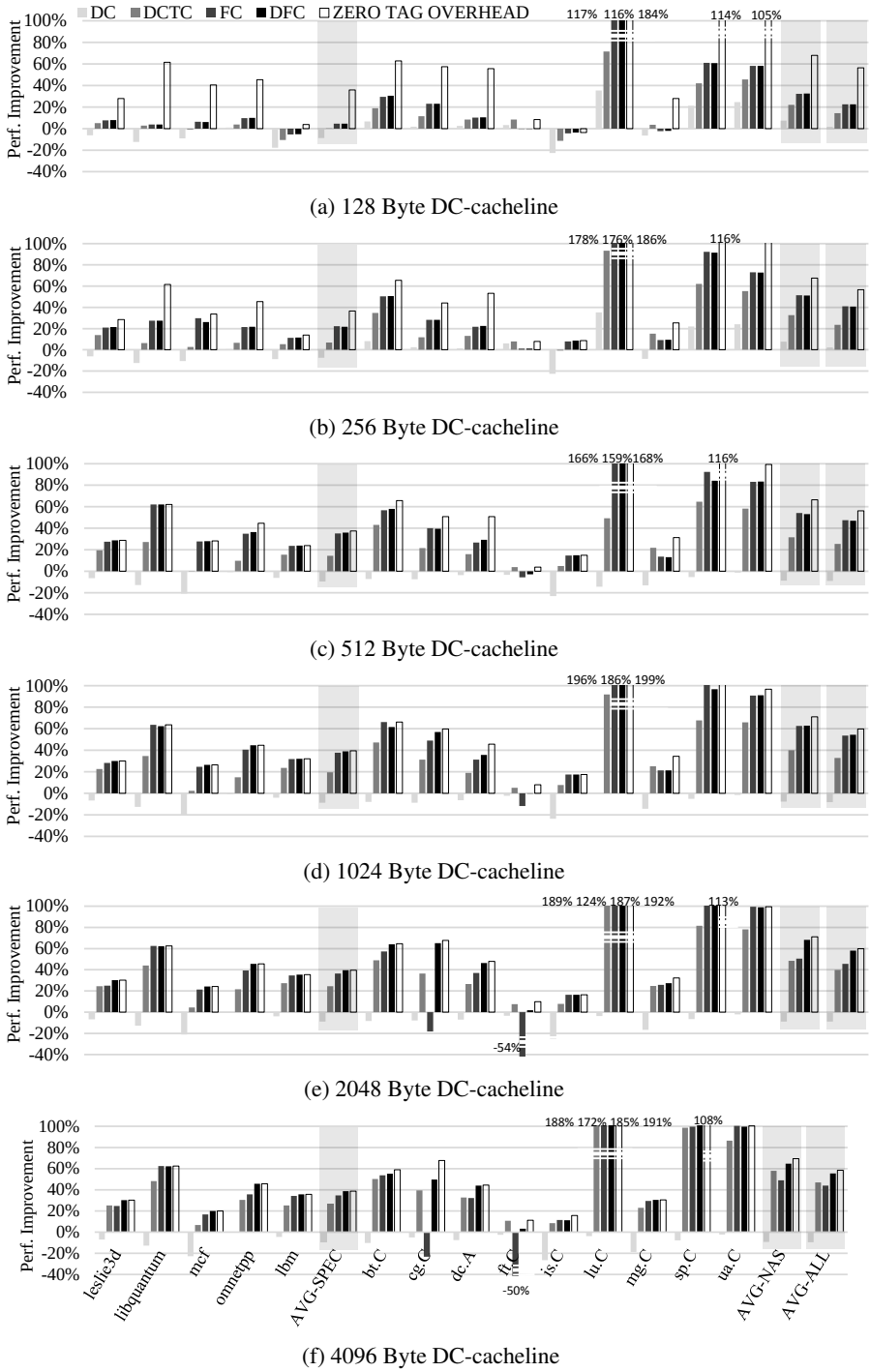


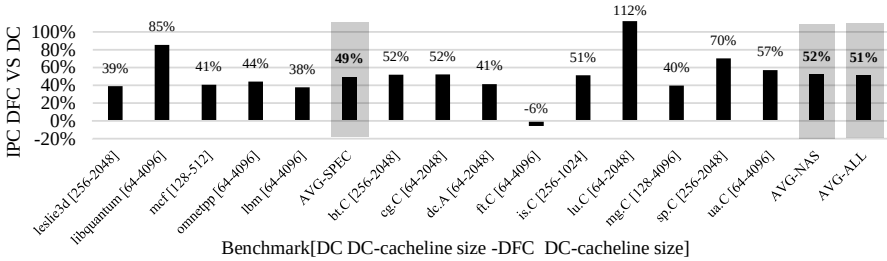
Figure 2.10: Performance improvement over the baseline of the DFC and related DC designs utilizing different DC-cacheline sizes.

Compared to DC with equal DC-cacheline size, DFC offers 19% to 73% (55% on average) better performance across all benchmarks; the performance gap increases with the DC-cacheline size. Furthermore, DFC compared to a DCTC with the same cacheline size yields 6% to 16% (11% on average) better performance. Compared to FC, DFC is 6% better on average. In particular, DFC shows similar performance for DC-cacheline sizes up to 1KB, but for larger DC-cacheline sizes, where FC performed poorly for some benchmarks, DFC is 16% to 18% faster. Focusing on cacheline sizes of 2KB and 4KB, it can be observed that DFC overcomes the limitations of FC in larger DC-cacheline sizes. For example, in *cg* and *ft* NAS benchmarks it is obvious that while FC does not perform well for large DC-cacheline sizes, DFC clearly mitigates the effect of increased LLC evictions introduced by the FC LLC indexing scheme. Finally, DFC achieves 80% to 99% (93% on average) the performance of a theoretical DC with zero tag-access overheads and equal cacheline size.

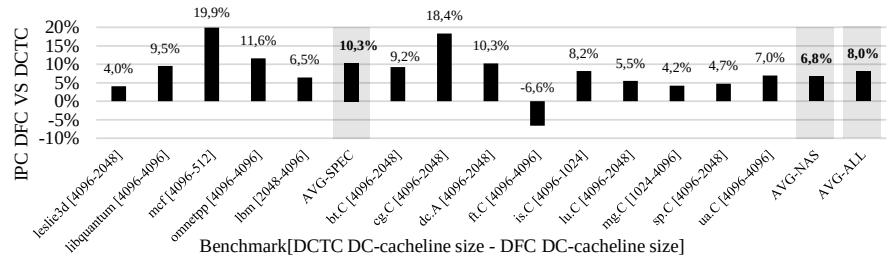
Figure 2.11 compares DFC to the competing designs (DC, DCTC, and FC) at the DC-cacheline size for which each design achieves its best performance. It also compares it to the DC with zero tag lookup overheads that uses DC cachelines of 4KB, which would be the best performance a Tagless DRAM Cache would possibly achieve [21]. Still, a Tagless DC design would introduce OS modifications and fix the DC-cacheline size to the OS page size as explained in Section 2.4. At the horizontal axis of each plot the name of each benchmark can be found and in brackets the DC-cacheline sizes of which each design achieves its best performance. For example, in Figure 2.11a *leslie3d[256-2048]* means that DC achieves its best performance using DC-cachelines of 256 Bytes while DFC maximizes performance using 2048 Byte cachelines. These results verify our initial statement in Section 2 that DC-based designs maximize their performance using different cacheline sizes for different benchmarks. This further highlights the importance of a design that is able to support different DC-cacheline sizes.

Figure 2.11a shows that DFC is a clear winner compared to DC, the only case where DC seems better that DFC is for the *ft.C* for which none of the DRAM cache designs showed any significant performance improvement compared to the baseline because of its streaming nature and little data reuse. Figure 2.11b shows the same trend when comparing DFC with DCTC where the performance difference can be as high as 19.9% in favour of DFC and on average 10.3% and 6.8% for the SPEC and NAS benchmarks, respectively. Figure 2.11c compares the best achieved speedup of DFC and FC. Although in some cases the performance of DFC is marginally lower than FC (up to -3.1% for *lu.C*), on average DFC performs better than FC and at best 10.8% for *cg.C*. Surprisingly, DFC achieves on average slightly better performance compared to a DC with zero tag lookup overhead using 4KB cacheline as shown in Figure 2.11d. This is because for some benchmarks DC-cacheline sizes other than the 4KB achieve better performance. This result shows that DFC is able to match the performance of a Tagless DC without any OS overheads and without fixing its granularity to the OS page size [21].

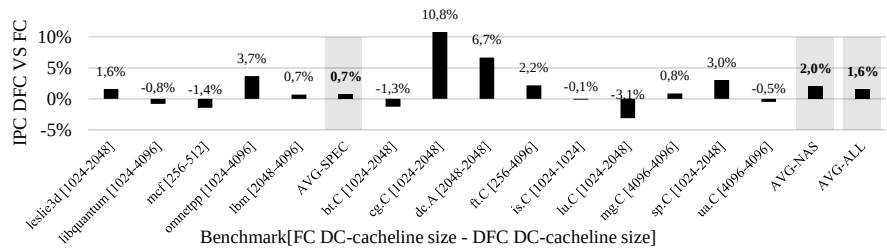
Figure 2.12 shows the average percentage of accesses that do not require a DRAM Cache tag lookup, this is equivalent to the tag-cache hit rate of DCTC. The average for all benchmarks per DC-cacheline size as well as the average across all DC-cacheline sizes for the SPEC and NAS benchmarks is shown. DFC can on average service 88% and 86% of the LLC misses directly for the NAS and SPEC benchmarks, respectively. This is similar to the respective FC results. On the other hand, the hit rate of the tag-cache in the DCTC design is 65% for the SPEC and 69% for the NAS benchmarks



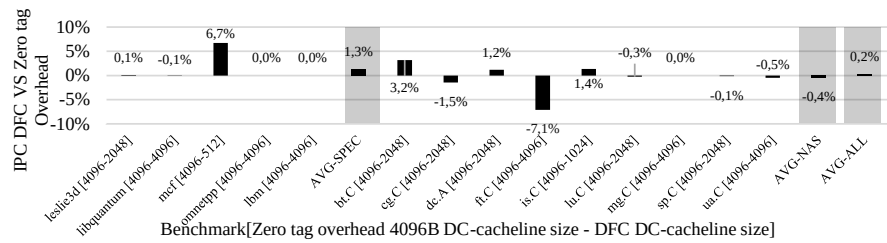
(a) DFC performance improvement compared to DC at best DC-cacheline size



(b) DFC performance improvement compared to DCTC at best DC-cacheline size



(c) DFC performance improvement compared to FC at best DC-cacheline size



(d) Performance improvement of DFC at best DC-cacheline size compared to Zero tag overhead at 4096B DC-cacheline size

Figure 2.11: Performance benefit of DFC compared to other designs at the best DC-cacheline size for each (the first number in the brackets after the benchmark name is the best DC-cacheline size for each competing design and the second is the best DC-cacheline size for DFC).

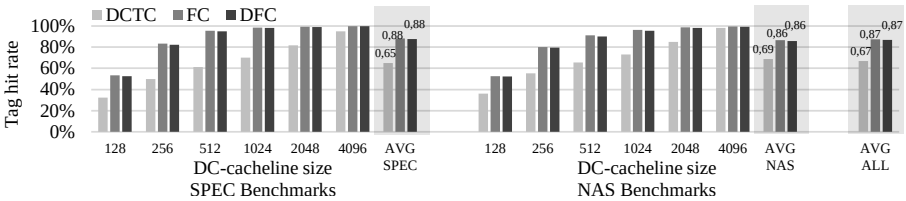


Figure 2.12: Percentage of accesses that do not require a DRAM Cache tag lookup.

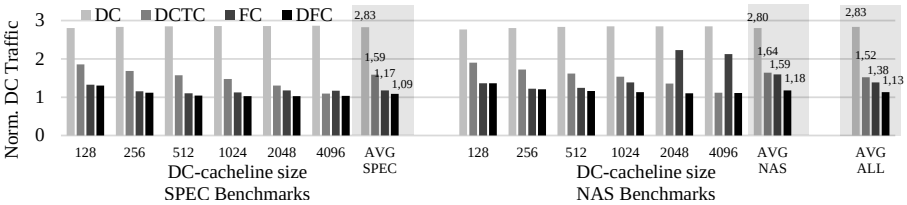


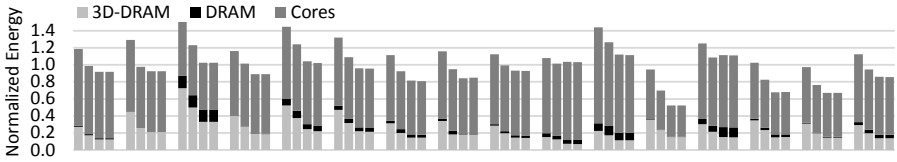
Figure 2.13: Average normalized DRAM Cache traffic.

respectively. This clearly shows the advantage of DFC over DCTC. Note, that the Tag cache size of the DCTC is similar to the SRAM overhead imposed by DFC as shown in Table 2.1. As explained below, a direct effect of the fewer DC-tag lookups is the reduced DFC traffic in the DRAM cache. In turn, the reduced DFC traffic alleviates the contention in the 3D-DRAM channels and as consequence offers lower overall DRAM cache latency and energy cost.

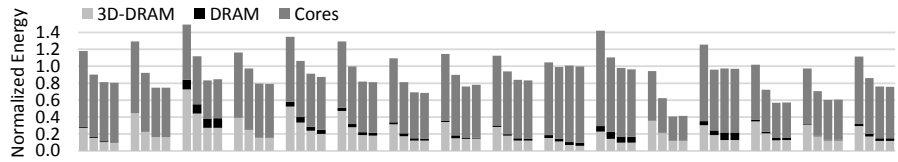
Figure 2.13 shows the average normalized DRAM cache traffic per DC-cacheline size of every design point in our evaluation for the SPEC and NAS benchmarks, respectively, as well as the average for each benchmark suite separately. DFC requires 32% and 28% (average 25%) less DRAM cache traffic compared to DCTC for the SPEC and NAS benchmarks, respectively. Compared to FC, DFC has on average 7,2% less DRAM cache traffic for the SPEC and 26% for the NAS benchmarks (18% on average). It is worth noting that in our experiments we observe that the latency overhead of DC-tag accesses is, besides the actual DRAM latency, also due to the increased contention in the DC channels. Reducing the DC traffic is further improving performance and in addition leads to lower energy consumption as explained below.

2.3.3 Energy efficiency

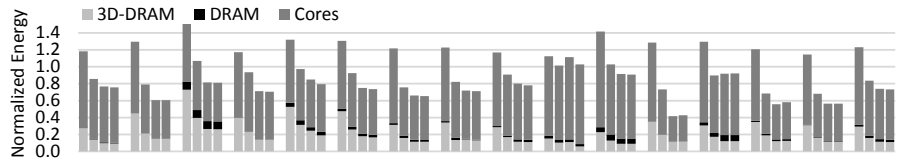
The energy consumption of the systems and its breakdown to cores, 3D-DRAM, and main DRAM energy cost is depicted per benchmark in Figure 2.14. The energy results are organized per DC-cacheline size and normalized to the energy consumption of the baseline system (without a DC). Considering designs with the same DC-cacheline sizes, DFC achieves 53% to 65% lower 3D-DRAM energy consumption (62% on average) than DC, 3.2% to 32.5% (24.5% on average) lower than DCTC, and 0.7% to 13% (7% on average) lower than FC. DFC's lower 3D DRAM energy cost is mainly due to its reduced DC traffic and improved system performance. A similar trend holds for the core energy cost, which is inversely proportionally to the performance of each design. Main memory energy consumption is similar across the designs and mostly negligible compared to core and 3d-stacked DRAM energy due to the use of a DRAM cache, which avoids most accesses to main memory. Overall, the total system energy consumption of DFC is 0.2% to 10.6% (4.1% on average) lower compared to FC,



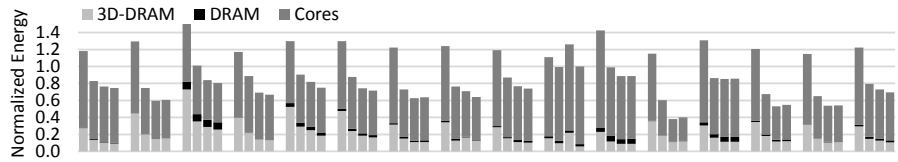
(a) 128 Byte DC-cacheline



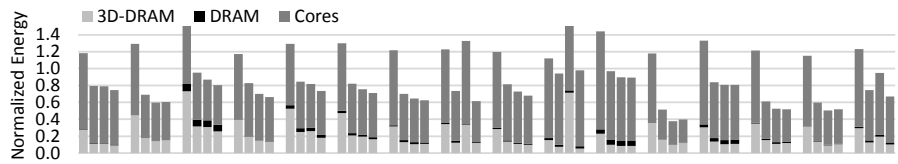
(b) 256 Byte DC-cacheline



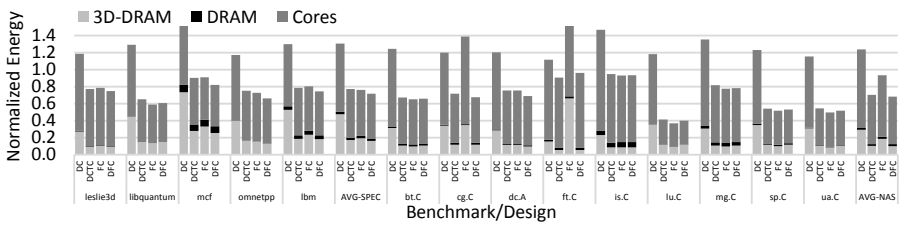
(c) 512 Byte DC-cacheline



(d) 1024 Byte DC-cacheline



(e) 2048 Byte DC-cacheline



(f) 4096 Byte DC-cacheline

Figure 2.14: Energy consumption from 3D-Stacked DRAM Cache (3D-DRAM), Main Memory (DRAM), and cores normalized to Baseline.

4.5% to 16% (12.3% on average) lower compared to DCTC and 25% to 45% (39% on average) lower than the simple DC design, considering equal cacheline size. Note that the energy overhead of (i) the BPA in the DFC, (ii) the tag-cache in the DCTC, and (iii) the larger LLC tag array in the FC are in general negligible and always less than 0.5% of the total energy consumption and thus cannot be depicted separately in the energy figures, however they are included in the cores' energy consumption.

2.4 Related Work

There are several existing DRAM cache (DC) designs that try to reduce their tag access overheads. Several choose to store DC tags only in the DRAM and employ various DRAM access and placement approaches to minimize the tag access latency. Others use various cache designs to keep a subset of the DC tags on chip. Another alternative is to utilize the page table mechanism to access the DRAM cache. One more technique reuses the LLC tag array to locate data in DRAM cache. Each of the above has its own strengths and weaknesses as explained below.

Alloy Cache attempts to reduce the tag access latency by proposing a direct mapped DRAM cache where the tag is placed along the data in DRAM [24]. This way, both tag and data are accessed in a single DRAM burst and since the cache is direct mapped the data can only reside in one location. This approach reduces the access overhead for DC hits as it avoids a DRAM row activation, but still imposes an unnecessary DRAM access when DC misses. To reduce the effect of this disadvantage, *Alloy Cache* uses a *Memory Access Predictor* for cache misses. *Alloy Cache* forces a direct mapped DRAM cache, which is very sensitive to conflicts and the tags still have to be read from DRAM in every access. This increases the overall DRAM cache traffic and energy consumption. Another proposed solution that combines tag and data accesses is to co-locate the tags for an associative DC in the same DRAM row as the data. This technique keeps the DRAM row open after the tag read and subsequently reads the data using *Compound Access Scheduling*. In this case, the data can be read without requiring a second DRAM row activation in case of a hit [12]. Co-locating the tags for each DRAM cache set in the same row and accessing them with *compound access scheduling* can be used with set-associative caches, however it still imposes higher DRAM traffic and high overhead for DRAM cache misses. Additionally, this design is limited to small cache line sizes because the tags and data of an entire set must fit in the same DRAM row (2KB-4KB). This causes considerable space waste for cache line sizes bigger than 64 - 128 Bytes. Co-locating tags and data has been utilized as a means to minimize the overhead of DC tag lookups in several other works, usually coupled with various predictors that aim to avoid tag lookups altogether [14, 25, 51].

ATCache uses a small SRAM cache for the DC tags [19]. In case of a tag-cache hit the access latency for the tags is in par with tags in SRAM while not incurring high area overhead on the processor chip. As the tag-cache access latency is in the critical path of any DRAM cache access, the tag-cache needs to be small. Chou et al. proposed a *Neighboring Tag Cache*, which buffers the tags of recently accessed adjacent cache lines as a means to reduce the DRAM cache traffic [22]. Hameed et al. propose a small and low latency SRAM/DRAM Tag-Cache structure that can quickly determine whether an access to the large L3/L4 caches is a hit or miss [52]. Another Tag-cache technique is presented by Meza et al. for Hybrid main memories composed of DRAM as a cache to non-volatile memories [53]. *ATCache* and other

Tag-cache solutions are limited by the temporal locality of DRAM cache set accesses and very sensitive to the tag-cache latency as it is always added to the critical path of every access. Micro-Sector cache [54] uses a Decoupled-Sectored [30] cache tag organization for the DC along with a new allocation and replacement unit called *micro sector* and a spatial locality aware replacement algorithm to improve space utilization in sectored DRAM caches. Decoupling the DC tags also improves the space utilization of the tag-cache this design utilizes.

For DRAM caches with page-granularity cachelines, the most prevalent work is the *Tagless DRAM Cache*. This work proposes a fully associative DRAM cache that is addressed directly without any tag access. This is done by changing the OS page tables and the TLBs to support DRAM cache addresses instead of main memory addresses [21]. The *Tagless DRAM Cache* design is effective but it only works for page based cache designs and requires big data transfers and awareness of data locality to support systems with big-pages. Additionally, it requires significant operating system (OS) support. Banshee [55] is another system which utilizes the TLBs and OS page-tables to locate data in a DC in its effort to optimize for both in-package and off-package DRAM bandwidth efficiency.

For multi-node systems storing a coherence directory on-chip would be prohibitively expensive. *CANDY* [18] re-purposes the existing on-die coherence directory as a DC *coherence buffer* to cache recently accessed directory entries similar to how our design re-purposes the LLC tag array. C^3D [17] attacks the same problem as *CANDY* by keeping DRAM caches clean to avoid the need to ever access remote DRAM caches on reads and by using a non-inclusive on-chip directory.

Finally, *FusionCache* presents a technique that utilizes the redundancy in the LLC tags to store information about the location of DC-cachelines in the LLC tag array. *FusionCache* achieves this by changing the LLC cache indexing to force LLC-cachelines that belong to the same DC-cacheline to be mapped to the same LLC set. This approach increases the number of distinct tags stored in the LLC tags array by splitting the LLC tags in upper (DC-cacheline tag) and lower tags (LLC-cacheline offset in the DC-cacheline) and by de-duplicating the LLC upper tags with the use of way-pointers [41]. *FusionCache* exploits the spatial locality of cache accesses well but degrades performance in some cases for large DC-cacheline sizes (over 1KB) because of the modified indexing in the LLC which causes more set conflicts.

Contrary to existing work, DFC mitigates the DC tag access overheads without imposing significant design restrictions. More precisely, DFC does not require any OS support, it does not limit DC associativity, it does not impose additional overheads in every access, and does not affect LLC performance. Still DFC offers zero tag access overhead in the common case, and can dynamically (at boot time) support variable DC-cacheline sizes.

2.5 Conclusions

In this Chapter, *Decoupled Fused Cache* (DFC) was presented, a design that stores information about the contents of the DRAM cache in the LLC. DRAM cache tag lookups are then avoided for most LLC misses. *Decoupled Fused Cache* overcomes the limitations of our initial *Fusioncache* design implementing a decoupled LLC tag array so to not penalize LLC performance for large DC-cacheline sizes. DFC supports any DC-cacheline size power-of-two multiple of a LLC-cacheline (up to 4KB

in our experiments), which is configurable at boot time. Our evaluation shows that DFC improves performance by an average of 55% and 11% compared to a simple DRAM cache (DC) and a DRAM cache with on-chip tag-cache (DCTC), respectively. Compared to our initial *Fusioncache* design, DFC is on average 6% faster and in large DC cacheline sizes 16-18% faster, because, as opposed to the FC, it does not affect the LLC efficiency. DFC increases the number of accesses to the DRAM cache that do not require a tag lookup from 67% for DCTC to 87%. DFC further reduces DRAM cache traffic, by 7% in the SPEC benchmarks and 26% in NAS benchmarks compared to FC, and by one and two thirds compared to DCTC and simple DC, respectively. This traffic reduction as well as its improved performance allows DFC to reduce DRAM cache energy by 7% compared to FC, by 24.5% versus DCTC, and by 62% compared to the simple DC.

Chapter 3

LLC-guided Data Migration in Hybrid Memory Systems

Various memory technologies can be considered in the design of multicore memory systems, each having different performance tradeoffs. Conventional DRAM offers good capacity, decent memory latency, but low bandwidth [56]. Emerging non-volatile technologies provide higher capacity but limited bandwidth and very high access latency [57, 58]. Finally, 3D-stacked DRAM is a promising solution against the bandwidth wall [59] as it delivers higher bandwidth, but falls short on the capacity aspect [60, 61]. As a consequence, it is more difficult to build a memory system based exclusively on 3D-stacked DRAM technology.

Many architectures suggest the use of 3D-stacked DRAM as a cache layer between the last level SRAM cache and the main memory [12, 14, 23, 24]. DRAM caches have been very effective in improving the performance of latency-sensitive workloads. However, they deny considerable main memory capacity, which could otherwise be utilized by capacity limited workloads [34]. Furthermore, DRAM caches come with their own design challenges as the conventional design choices of SRAM caches need to be re-considered for Giga-scale DRAM caches, especially in terms of granularity and metadata placement [19–22, 26, 51].

Another way to exploit the bandwidth advantage of 3D-stacked DRAM without wasting the capacity is to include it in a flat address space, hybrid memory system combined with conventional DDR memory. Adding data migration support between the two types of memories is critical for capitalizing a significant performance gain [7, 9, 10, 32, 62–64]. Then, the memory system is composed of a 3D-stacked *Near Memory* (NM), placed above the processor die, and a conventional DDR, *Far Memory* (FM), located off-chip. The design of such a hybrid memory system is also challenging. The performance impact of migration relies on the accuracy of identifying *hot* memory segments to migrate as well as on balancing the introduced overheads. Data migration requires address remapping which is in the critical path of a memory request. Moreover, it introduces additional traffic which competes directly with the processor memory accesses.

In this work, we propose LLC-guided Migration (LGM), a novel scheme for data migration in hybrid memory systems aiming both at improving the selection of migrated data as well as at reducing their traffic overheads. The first objective is achieved by using the LLC to guide the selection of memory segments to be migrated

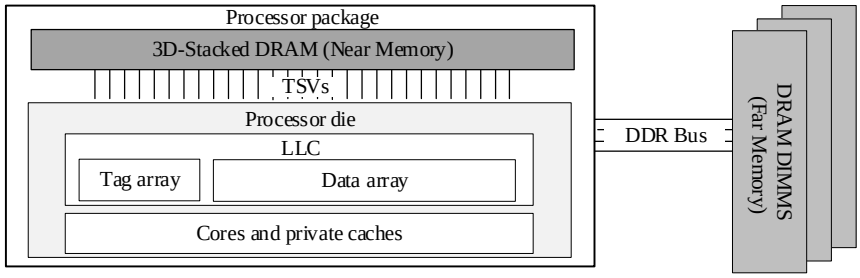


Figure 3.1: A hybrid memory system.

by detecting high spatial and temporal locality. More precisely, the LLC is used to identify memory segments that have a large number of cachelines on-chip. This is an indication for potential future reuse, which gets stronger when these cachelines are dirty. Employing the LLC to select segments for migration ensures that these segments are at that moment –at least partly– present in the last level cache (LLC). This can be used for achieving the second objective of reducing migration traffic. We observe that when a fraction of a memory segment is located in the LLC, it can be omitted from the migration to reduce the migration traffic, as long as the LLC evicts it back to memory.

Concisely, the contribution of this Chapter is a novel data-migration scheme for hybrid memory systems that:

- Employs the LLC to detect locality and leverages it for migrating data with higher potential for reuse;
- Reduces the migration traffic overhead by avoiding to migrate data that reside in the LLC;
- Increases the benefits of the above migration traffic reduction because LGM makes more likely the selected data to be in the LLC when they migrate.

The rest of the Chapter is organized as follows: Section 3.1 discusses some background and the motivation behind this work. Section 3.2 describes the proposed migration scheme. Section 3.3 presents our experimental setup and system configuration. Section 3.4 offers our evaluation results and comparison and finally Section 3.5 draws our concluding remarks.

3.1 Background and Motivation

This section presents related work on data migration for flat address space hybrid memory systems as well as some motivating results to identify overheads and potential for improving performance over the current state-of-the-art. Before that we first give a general description of the system considered and the data migration problem.

Figure 3.1 shows a *hybrid memory system* comprised of Near Memory (NM) and Far Memory (FM). NM can be 3D-stacked DRAM located either on top of the processor die, or in the same package through an interposer or any other high-bandwidth channel. FM can be conventional DDR DRAM connected to the processor chip through a DDR bus. The 3D-stacked DRAM offers substantially higher bandwidth, but has limited capacity, therefore it is complemented with off-chip DDR DRAM which adds more capacity (in our experiments 16× more than the NM) but at a much lower

Table 3.1: Categorization of existing works with respect to various aspects.

Design	Flexibility	Granularity	Remapping	Data Selection	Migr. Trigger	Driver
HMA [32]	All to All	OS Page	OS page tables	Access counters	Time Interval	HW and SW
PoM [7]	Congr. groups	Segment	Table in NM + cache	Competing counters	Threshold	HW
CAMEO [34]	Congr. groups	Cacheline	Table in NM + predictor	Anything accessed	On access	HW
SILC-FM [10]	Congr. groups	Segment/ cacheline hybrid	Table in NM + predictor	Aging counters	Threshold	HW
MemPod [9]	All to All	Segment	Table in NM + cache	MEA	Time Interval	HW
LGM	All to All	Segment	Table in NM + cache	LLC-guided	Threshold	HW

bandwidth (8× lower than NM in our evaluation). Such hybrid organization of the two technologies has the potential to compose a better memory system [7, 9, 10, 32, 62, 63] provided that data can migrate between the FM and the NM. Additionally, support for address remapping is needed to redirect memory requests to the actual location of the segment as well as a mechanism for orchestrating the movement of migrating data.

3.1.1 Related Work

There exists a large body of prior work on data migration for hybrid memory systems. We hereby attempt to identify the main features that differentiate these works and point out our design decisions and novelty with respect to these features. The most representative works are categorized in Table 3.1.

An important categorization of existing works is with respect to the flexibility of remapping data between the FM and the NM. This design choice is also tightly associated with the remap table size and complexity. Some allow all-to-all remapping [9, 32], while others consider congruence groups [7, 10, 34] restricting the migration as they may create conflicts when there is spatial locality [10, 34]. We consider an all-to-all approach more interesting, as it allows any page to migrate between the two memories increasing the migration options. In addition, congruence groups do not scale well when the ratio between FM size and NM size increases [10, 34]; in our work we consider memory systems where the FM is substantially larger than NM, i.e., 16:1.

Another important component is the granularity of the migratable memory segment. Previous works have considered various options such as: OS pages [7], large memory segments [9, 32], and single cachelines [10, 34]. Dynamic granularity has been also considered in [65]. As expected, coarser granularities are simpler to manage but require

careful selection of "dense" data to avoid memory waste, while finer granularities can fit "sparse" data more efficiently but are more expensive to implement. As in Mempod [9], we consider half an OS page (2KB) as the size of a memory segment that can migrate.

Various mechanisms have been used for address remapping in order to keep track of the migrated data. Some techniques involve the OS page tables to keep track of the remapping [32]. Most approaches use a remap table stored in NM and rely on predictors [10, 34] or caching [7, 9] to reduce the memory accesses for remapped address lookups. We opt for using a remap table and an on-chip cache for it as it is less intrusive and flexible to support all-to-all remapping.

The core component of every strategy, is the way a memory segment is selected for migration. Most approaches use counters to keep track of accesses to memory segments [32] or counters for every segment within a group [7, 10]. So far, the most promising approach has been the activity tracking mechanism proposed by Mempod [9], which uses the *Majority Element Algorithm* (MEA) [33]. MEA has been shown to predict the hottest pages within an interval with high accuracy and at minimal hardware cost. In this work, we propose a new selection mechanism which is based on the LLC state to select segments with good potential for future reuse. In addition, our approach offers an excellent timing for selecting memory segments to migrate, which contributes significantly to reducing migration overheads.

Different approaches trigger migrations in different ways. Many of them do it on time intervals [9, 32], while others do it on an event, e.g. CAMEO migrates at every memory access that is in a far memory [34]. Some approaches trigger migrations when the values of selection counters go beyond some threshold [7, 10]. Our mechanism also uses thresholds that are compared with the number of valid and dirty cachelines per memory segment stored in the LLC.

Finally, another aspect that characterizes the different approaches is whether the migration mechanism is based on software or hardware, or a combination of the two. Some migration mechanisms rely on the OS with some hardware support to identify the working set and orchestrate the migration [32], others only involve the hardware and are transparent to the OS [7, 9, 10, 34]. Our approach is transparent to software avoiding OS modifications and more complex and slower software-based techniques.

The best performing related works are MemPod [9] and SILC-FM [10]. Attempting to compare them with each other, it can be observed that MemPod achieves higher speedup than SLIC-FM versus the same competing techniques. Moreover, MemPod offers an all-to-all remapping flexibility, while SILC-FM is restricted by congruence groups which do not scale well when the size of the FM increases versus the NM size. Consequently, we model and compare against Mempod considering it the best current state-of-the-art competing design.

The main novelty of our approach is the following: Firstly, the migration overheads are reduced by avoiding traffic for cachelines already present in the LLC. Secondly, the quality of selecting of data for migration is improved. Even more important is that segments are selected for migrations when a large fraction of them resides in the LLC, this timing further reduces the migration traffic.

3.1.2 Motivation

Here, we attempt to quantify the performance potential of data migration schemes as well as to study various migration overheads. We identify and analyze the following overheads:

- The latency cost of looking-up remapped addresses, and
- The migration traffic between the processor and the NM as well as between the processor and the FM.

The migration approach used in this "limit study" is the current state-of-the-art Mempod approach with 2KB granularity of memory segments applied in a system like the one described in the beginning of this section, in which the FM is 16× larger than the NM and has 8× lower bandwidth. The detailed experimental setup as well as the benchmarks used here are the same as in our final evaluation and are described in Section 3.3.

The first metric considered is the average speedup across the benchmarks listed in Table 3.3. We evaluate the design points listed below and normalize the results to a baseline system without data migration support:

- Mempod [9],
- Mempod with zero overhead remapped address lookup (ZO-Remap),
- Mempod with zero migration traffic for NM (ZMT-NM),
- Mempod with zero migration traffic for FM (ZMT-FM),
- Mempod with zero migration traffic to both NM and FM (ZMT),
- Mempod with zero overheads for address remap lookup and for migration traffic (ZO-All),
- A system with a 3D stacked DRAM large enough to host the entire main memory (Ideal).

As shown in Figure 3.2, Mempod offers a speedup of 1.3× on average, but an *Ideal* memory system would reach a speedup over the baseline of 2.35×. This indicates an upper limit in improving performance with data migration and shows opportunity for improvements over the current state-of-the-art. Looking closer in the migration overheads, it can be observed that removing the latency of remapped address lookups from Mempod (ZO-Remap) would increase its speedup over the baseline to 1.38×. Furthermore, removing the migration traffic in the NM (ZMT-NM) would have negligible performance impact because NM offers high bandwidth and parallelism. On the contrary, removing the migration traffic from the FM (ZMT-FM) would improve Mempod's speedup to 1.39×. Moreover, removing all overheads of migration traffic (ZMT) offers a speedup of 1.41×. Finally, removing all overheads, both for remapping and for data traffic (ZO-All) would offer a speedup of 1.5×.

In order to understand better the impact of the migration traffic we measure and analyze the traffic break-down. Figure 3.3 shows the NM, FM and total traffic, average for all benchmarks, in a system with Mempod. The traffic is categorized as follows: Processor traffic (PT), Migration traffic (MT), Remapped Address lookup traffic (RT).

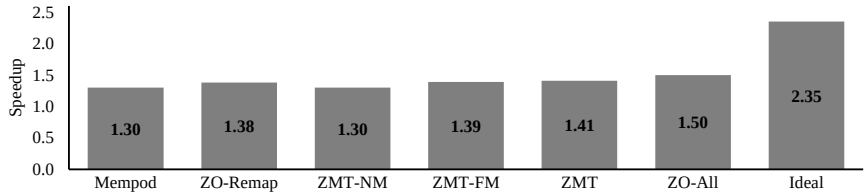


Figure 3.2: Average speedup over no migration.

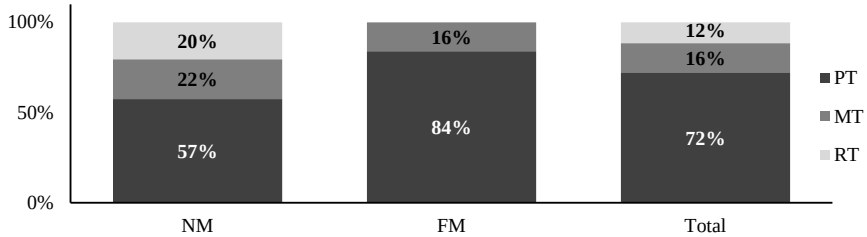


Figure 3.3: Average Traffic Breakdown in a Mempod system [9].

The traffic induced by migration (MT and RT) is significant and about 28% of the total traffic. This traffic is almost evenly divided between RT and MT traffic, but it is important to note that all RT goes exclusively to the NM, which has 8× higher bandwidth than the FM. More precisely, 22% of NM traffic is due to migration of data and 20% due to the lookup of the remapped addresses. FM does not suffer from RT and its MT overhead is 16%. In conclusion, the traffic overhead of data migration is significant and has an impact in performance as discussed above and presented in Figure 3.2.

In summary, this "limit study" offers the following motivating conclusions: There is significant potential for improving performance of hybrid memory systems with data migration beyond the current state-of-the-art (up to about 1.8×). To a large extend, this is due to the method for selecting data to migrate; to a lesser but significant extent (by about 1.15×), it is due to reducing the migration overheads.

3.2 LLC-guided Migration

This section presents our scheme for LLC-guided data migration (LGM) for hybrid memory systems. Our proposed approach is based on the observation that there is significant amount of state and content in the last-level cache (LLC) to be exploited for performing efficient data migration between the *Near* and the *Far* memories (NM and FM) in a hybrid memory system.

Section 3.2.1 describes our proposed technique for identifying segments that are good candidates for migration to the NM. Section 3.2.2 illustrates how to reuse data segments that reside entirely or partially in the LLC to avoid unnecessary accesses to the NM and FM. In Section 3.2.3 we present the details of our LGM architecture and propose some changes in the LLC to efficiently support our migration scheme.

3.2.1 Segment selection for migration

The NM offers high bandwidth but is a scarce resource while the FM which provides the bulk of the memory capacity has limited bandwidth. Therefore, the migration selection algorithm is of crucial importance for the performance of a hybrid memory system. The ultimate goal of all migration algorithms is to select the memory segments with the highest degree of spatial and temporal locality and place them to the NM in order to facilitate faster data reuse and at lower energy cost. In analogy, the on-chip cache hierarchies utilize sophisticated cache replacement algorithms to identify data reuse at the cacheline granularity. It is apparent that both types of algorithms have overlapping goals, however they are currently operating in isolation. Driven by this observation we advocate that there is significant amount of state and data in the LLC to be exploited by migration algorithms and take informed decisions about the memory segments that should be placed in NM.

The most important difference between cache replacement and data migration algorithms stems from the granularity of operation. The caches operate at the cacheline granularity (typically 64-bytes) while data migration schemes operate at the segment granularity which involves a plethora of cachelines (typically a power-of-two number); the segments in our system consist of 32 cachelines (2KB). The coarser granularity of segments poses a major challenge to the migration mechanisms which necessitates the selection of segments with very high degrees of spatial locality. Moreover, the migration traffic, i.e. the swapping of memory segments between FM and NM, puts an additional burden to the bandwidth-limited FM and calls for techniques to alleviate the extra bandwidth pressure.

To this end, we propose an LLC-guided data migration (LGM) scheme that migrates memory segments, from the FM to the NM, based on the number of cachelines present in the LLC and their state. The number of a segment's cachelines that are present (*valid*) in the LLC provides an indication about the degree of spatial and temporal locality for the segment. Moreover, segments that have several *dirty* cachelines are of particular interest because they indicate further traffic (*write-backs*) to memory in the future.

Migration Selection:

Our proposed migration selection algorithm monitors the number of *valid* and *dirty* cachelines and maintains a *Valid_Counter* and a *Dirty_Counter* for each FM segment that resides in the LLC. In order to trigger migrations of memory segments from FM to NM we introduce two thresholds: (a) Valid_Threshold: the minimum number of present cachelines of an FM segment in the LLC and (b) Dirty_Threshold: the minimum number of modified cachelines of an FM segment in the LLC.

The number of LLC cachelines that belong to an FM segment is constantly changing as a result of processor cache misses and LLC cache replacement decisions. The *Valid_Counter* of each FM segment is incremented upon LLC insertions and decremented upon LLC evictions of associated cachelines. The *Dirty_Counter* of each FM segment is incremented upon write-backs to the LLC from the cache levels above and decremented upon LLC evictions of associated cachelines. When the *Valid_Counter* exceeds the *Valid_Threshold* or the *Dirty_Counter* exceeds the *Dirty_Threshold* then a migration request is submitted to the Migration Controller (MIC) that performs the required operations in the background; after this point the associated segment is not

monitored. The detailed hardware support for tracking and migrating segments is described in Section 3.2.3.

Adaptive Thresholds:

The use of a *Valid_Threshold* and a *Dirty_Threshold* to select FM segments and trigger migrations is intuitive, however, deciding the actual values for these thresholds is a subtle issue. Setting the thresholds to high values, i.e. close to the number of cachelines in the segment (32 in our design), suggests migration only when the segments are almost entirely present in the LLC and this may lead to very conservative or late decisions. On the other hand, setting very low threshold values suggests aggressive decisions which may cause excessive migrations of segments without sufficient spatial locality. To balance between these two extremes and guard against excessive migration traffic, our proposed design is adaptive, i.e. adjusts the thresholds dynamically, and as such it can accommodate the behaviors of different applications and of different phases within an application.

Our adaptive threshold mechanism monitors the number of migrations within a fixed time interval and changes the threshold values only at interval boundaries. At the beginning, both thresholds start at the highest value *MAX* (32 in our design) to request migrations only for segments that have ideal spatial locality. If no migrations happen in an interval, then the thresholds are lowered to relax the previous requirement. To avoid overly relaxed thresholds, we set a low limit *MIN* (4 in our design) to filter out segments with a small amount of spatial locality However, several threshold values in the *[MIN,MAX]* range can be very optimistic, leading to excessive migrations and in effect congestion that negatively impacts performance. To guard against excessive migrations, we define a *high-watermark* as the highest acceptable number of migrations during an interval and use it for throttling. This *high-watermark* does not limit the actual number of migrations within an interval but is instead used at interval boundaries to raise the thresholds if they become aggressively low; effectively the *high-watermark* controls bandwidth allocation for migrations.

Our mechanism lowers the thresholds in a conservative way, always by one, when the number of migrations within an interval is zero. The two thresholds *Valid_Threshold* and *Dirty_Threshold* are handled separately. We favor migrations of *dirty* FM segments over the *clean* ones, because we certainly know that they will perform accesses in memory. The *Dirty_Threshold* has the highest priority and gets decremented until the lower limit *MIN* before the *Valid_Threshold* starts being decremented. On the other hand, we follow an aggressive policy to raise the thresholds when the number of migrations exceeds the *high-watermark*, i.e. very high migration rate. Our mechanism disables all migrations for *valid* segments by setting the *Valid_Threshold* to *MAX+1* (OFF) and permits migrations only for *dirty* segments with ideal spatial locality by setting the *Dirty_Threshold* to MAX. We take the decision to effectively "pause" migrations and allow the memory system to "recover" in order to avoid the adverse effects of congestion. After the recovery period, the algorithm will eventually lower the thresholds and permit migrations in future intervals. Listing 3.1 outlines our algorithm for adjusting the migration thresholds based on the number of migrations and the *high-watermark*.

```
if ( num-migrations > high-watermark ){
    Valid_threshold = MAX+1; // OFF
    Dirty_threshold = MAX;
} else if ( num-migrations == 0 ){
    if (Dirty_threshold > MIN)
      Dirty_threshold--;
    else if ( Valid_threshold > MIN )
      Valid_threshold--;
}
```

Listing 3.1: Adaptive migration threshold adjustment.

3.2.2 Reducing Migration Traffic

Segments that are currently located in FM and have several *dirty* cachelines in the LLC are particularly interesting because they can have lower migration cost in terms of memory traffic and energy. We observe that the migration process for such FM segments can be optimized because: (a) there is no need to read the "stale" subset of cachelines (those modified in the LLC) from the FM and (b) there is no need to update the complete segment in the NM immediately – the NM contents will become fully up-to-date when eventually these *dirty* cachelines are evicted from the LLC and written back to memory (in this case NM). For "dense" segments that have ideal spatial locality the migration traffic and energy are effectively halved. For "sparse" segments the migration traffic and energy are lowered proportionally to the number *dirty* cachelines of that segment in the LLC.

We also observe that the latter optimization can be applied to the *valid* but *clean* cachelines of a FM segment that resides in the LLC and is selected for migration. To implement this optimization and ensure correctness we merely need to mark the associated cachelines with an "always-write-back" flag in the LLC and use it during the eviction process.

The optimizations proposed above reduce the migration memory traffic and relieve the bandwidth pressure on both the bandwidth-limited FM and the high-bandwidth NM. However, the "always-write-back" optimization puts additional pressure on the LLC since it disables "silent evictions" of "clean" cachelines for FM segments that have been migrated. In spite of that, LLCs can handle additional accesses since they are typically multi-banked and have significantly higher bandwidth than the NM and the FM. In addition, reusing the cachelines present in the LLC constitutes a significantly more energy efficient option than reading from off-chip memory; off-chip memory accesses cost at least an order of magnitude higher energy than on-chip cache accesses. The detailed hardware support for these optimizations is described next.

3.2.3 Architecture

The architecture of our LGM approach is illustrated in Figure 3.4. The LGM migration mechanism is implemented between the LLC and the memory controllers. The LLC is designed to keep track of data at the granularity of a memory segment in addition to the LLC cacheline granularity. In doing so, the LLC keeps track of information about the number of (*valid* and *dirty*) cachelines in a segment. This information is then used by an LLC Migration Agent (LMA), which is the core module of our migration scheme and implements the proposed LGM selection algorithm, described in Section 3.2.1. The memory segments selected by the LMA for migration are sent to the Migration

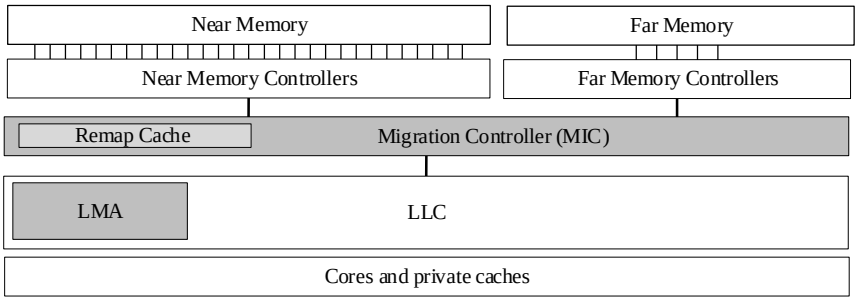


Figure 3.4: Block diagram of the LGM-based architecture.

Controller (MIC) module which is responsible for performing the data movement. It further manages the contents of the Remap Table, which is located in the NM and stores the address (re)mappings of memory segments. Next to the MIC, a cache of the remap table, the *Remap Cache*, is used to avoid excessive NM accesses to the full size table. Below, each of these modules is described. The MIC communicates with the LMA located in the LLC and with all the NM and FM memory controllers.

Last Level Cache (LLC):

There are many design options for an LLC, however a design that lends itself naturally for the basic functionality required by LGM is a *Decoupled Sectored Cache* (DSC) [30] as it keeps track of data both at the granularity of a cacheline and that of a segment, composed of multiple cachelines. DSCs, first introduced to prevent space waste in sectored caches [30], decouple the indexing of their *tag array* (TA) from the one of their *data array* allowing the first to handle tags at the granularity of a segment and the latter to handle data at the granularity of a cacheline. This decoupling is achieved via an indirection supported by a *Back Pointer Array* (BPA). Then, all cachelines that belong to the same segment reference the same tag through the use of back-pointers as depicted in Figure 3.5. The figure further illustrates that the BPA is indexed like the data array, using the lower address bits after the byte offset; however, for the tag array, higher order bits are used skipping the bits of the cacheline offset within the segment. Figures 3.6a and 3.6b show in white boxes the basic fields of the DSC TA entries and BPA entries, respectively. For every cacheline, the BPA stores its state (valid, dirty, replacement bits) and the *back-pointers* which associate it with a (segment) tag in the TA. The TA stores segment tags along with their accompanying state. To fully support the LMA functionality we augment the DSC with additional state as described in the following paragraphs.

LLC Migration Agent (LMA):

The LMA is an extension of the LLC controler. It selects FM segments to migrate to NM by monitoring the number of their *valid* and *dirty* cachelines using counters in the tag array. When these counters surpass their respective thresholds, LMA forwards the segment address to the MIC along with a bit-vector of all *present* cachelines of the segment in the LLC. At the same time, the LMA marks as “always write-back” all *present* cachelines of the segment so that they are written-back to main memory

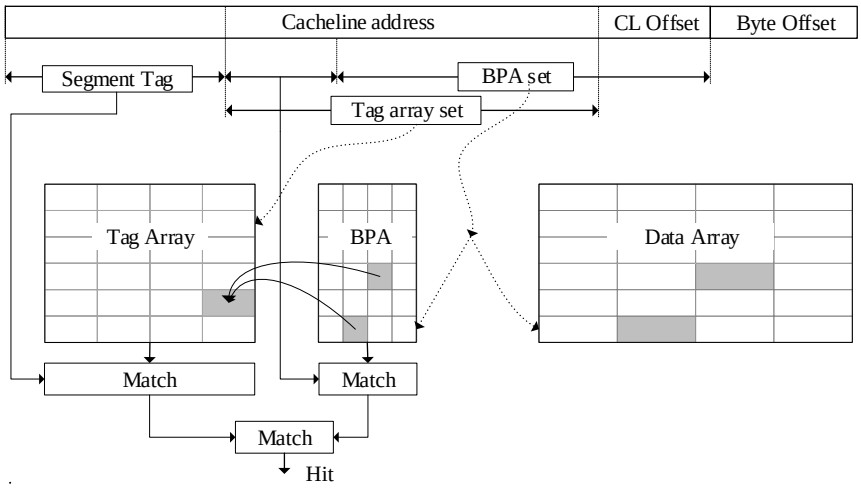
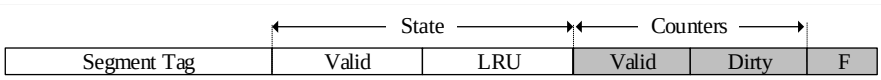
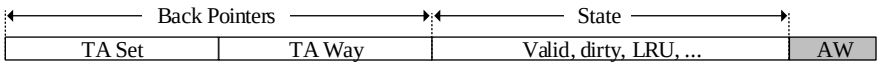


Figure 3.5: DSC address breakdown and indexing.



(a) DSC TA entry with counters and F bit.



(b) DSC BPA entry with AW bit.

Figure 3.6: DSC changes to BPA and TA entries.

when evicted from the LLC. In order to support the above LMA functionality, the LLC design is required to provide the following functionality:

1. Keep track of the state of FM segments that reside in the LLC (i.e. the number of valid and dirty cachelines)
2. Mark all LLC cachelines of a migrating segment which are skipped during migration as “always write-back” so that they are eventually written-back to memory when evicted from the LLC.
3. Provide MIC with a bit-vector of the cachelines present at the LLC when a segment is selected for migration.

For the first above LMA functionality, the LMA only needs to monitor changes in the state of FM segments. For that purpose we add a *Far-bit* in the LLC, which indicates that a particular segment is located in FM and should be monitored for state changes that might lead to migration. The *Far-bit*, shown as *F* in Figure 3.6a, is placed along with the segment tag in every TA entry. The location, either NM or FM, of a segment is piggy-backed in the response packet from the memory system and if the cache miss was served from FM then the *Far-bit* is set. The *Far-bit* is cleared when a segment is selected for migration to NM and thus it does not need to be monitored any further.

To track the state of FM segments in the LLC we add two counters that hold the number of *valid* and *dirty* cachelines for every segment. The *Valid_Counter* and the *Dirty_Counter* are placed in every TA entry, as shown in Figure 3.6a. These counters change as a result of processor cache misses, write-backs from the cache levels above, and LLC replacement decisions. Upon changes, the values of the *Valid_Counter* and the *Dirty_Counter* are compared with the *Valid_Threshold* and *Dirty_Threshold* respectively to select FM segments for migration. The size of the *Valid_Counter* and the *Dirty_Counter* is proportional to the number of cachelines per segment. For N cachelines per segment each counter is $logN + 1$ bits wide.

The second functionality of LMA is marking in the LLC all *present* cachelines of a segment when it is selected for migration. These cachelines are not transferred during migration of that segment and we must ensure that they are eventually updated in memory. For this purpose, we add an *Always Write-back* (AW) bit in every BPA entry as shown in Figure 3.6b. The AW bit is checked by the LLC controller whenever a cacheline is replaced and if set the cacheline will be written-back to memory regardless if it is dirty or not. We add this new bit instead of using already existing state bits, e.g. "dirty", to avoid interfering with coherence state bits and their complex transient states. Furthermore, we assume a Non-inclusive/Non-exclusive LLC for our design. Therefore, if the AW bit is set, a cacheline is always written back to memory when evicted from the LLC regardless of whether there exists a copy of that cacheline in lower cache levels. This approach simplifies the design, although it may lead to some unnecessary memory updates when the latest copy of a cacheline is not in the LLC.

The third functionality of LMA is to provide a bit-vector of all cachelines of a segment that are present in the LLC to the MIC when a segment is selected for migration. This is achieved by looking up the subset of the BPA that can potentially host cachelines of a segment[1]. Looking up the *valid* cachelines of a segment in the BPA and setting their *AW* bit is accomplished with a single pass when a segment is selected for migration. Furthermore, the BPA is significantly smaller than a conventional tag array, accesses to it are fast and consume little energy, and thus they do not impose significant overheads. Additionally, the search of BPA for cachelines of a segment terminates when LMA has located as many as indicated by the *Valid_Counter* of the particular segment.

To summarize, the changes we introduce to the Decoupled LLC for our migration mechanism are:

- The *LLC Migration Agent* (LMA) which consists of some additional logic in the LLC controller.
- A *Far-bit* to every TA entry (Figure 3.6a), which is used to indicate that a segment is currently in FM,
- A *Valid_Counter* and a *Dirty_Counter* to every TA entry (Figure 3.6a) that count *valid* and *dirty* cachelines for every segment and,
- An *Always Write-back* bit to every BPA entry (Figure 3.6b) which ensures that the cachelines of a segment that were skipped during migration to NM are eventually updated in memory.

[1] In the worst case, LMA needs to probe as many BPA sets as the number of cachelines that belong to a segment (32 in our case).

When considering a 16-way, 8MB LLC with 64-byte cachelines, 2KB segments, and 48 bits of physical address space we find that our modifications require only 4.3% additional area over a conventional LLC with the same characteristics.

LGM with a conventional LLC:

Using a DSC is the most cost effective option to support the functionality LGM requires. However, LGM can be implemented on top of a traditional LLC, albeit with higher area and energy cost. To do so, we must include a DSC-like tag-array in addition to the conventional LLC in order to keep track of the segments that reside in the LLC. This additional tag-array must store tags at segment granularity and also maintain a valid and dirty bit-vector for the cachelines of each segment in the LLC. The always-writeback bits should also be stored in the added tag-array for every cacheline in a segment. Besides the additional area overhead, the extra tag array also incurs energy costs as it needs to be accessed and updated in parallel to the LLC. Furthermore, the LLC controller must consult the always-writeback bits of the DSC-tag array on LLC evictions.

Banked DSCs:

A DSC still allows for a banked implementation. Banked caches interleave their sets across multiple banks for better access parallelism and throughput. In a DSC, this interleaving can be achieved by selecting the LSbits of the tag to identify the bank. Although the tag array and BPA are indexed by different parts of the address (*Tag array set* and *BPA set* in Figure 3.5, respectively), these address parts have an overlap. Using this overlapping address part as the most significant bits for indexing the tag-array and BPA ensures that the tag of a segment and its cachelines fall on the same group of consecutive sets (subset) of the cache.

The number of sets in a subset is equal to the number of cachelines per segment, and for our design it is 32. Thereby, a DSC can be split into multiple banks as long as a subset does not span multiple banks.

Migration controller (MIC):

The MIC is responsible for implementing migration decisions. It includes swap buffers to facilitate the migrations of segments between NM and FM. It also manages the address remapping mechanism which consists of a "remap table" stored in NM and cached on-chip in the *Remap Cache*. Moreover, it has an "inverted remap table", also stored in NM, which is used to find the physical address of any remapped segment. The "remap table" is indexed by the segment address and provides the physical address for each segment in the address space. The "inverted remap table" is indexed by a physical address and provides the address of the segment that currently resides there. The entry size in both tables is 4 bytes in our evaluated system and they are both stored in the lowest addresses of the NM. When the MIC receives a segment address from the LMA to migrate to NM it performs the following steps:

1. Selects a NM segment to be swapped-out (victim) using a *First-In-First-Out* (FIFO) policy.

2. Finds the physical address of the victim segment via the *Inverted Remap Table* which resides in NM.

3. Finds the location of the segment to be migrated to NM (hot) through the *Remap Table* also located in NM.

4. Issues memory requests to swap the two segments using the bit-vector.

5. Updates the *Remap Table* and the *Inverted Remap Table*.

6. Stalls any incoming processor requests to the "pending" segments while they are being swapped.

Periodically, at fixed time-interval, the MIC compares the number of migrations to the high-watermark and adapts the *Valid_Threshold* and *Dirty_Threshold* for the LMA.

To support the above functionality, the MIC requires simple hardware mechanisms such as: Migration buffers to store cachelines of segments undergoing migration, logical shifters to access the proper remap table indexes given an address, logic to issue memory requests, a small structure that keeps track of outstanding migrations, comparators for identifying requests to segments that are being currently swapped, a comparator for adjusting the migration thresholds and a simple interface to communicate these thresholds to the LMA.

3.3 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 3.2, our system configuration considers an eight core processor with private L1 and L2 caches and a shared last level cache (LLC). The memory system consists of 1GB of HBM and 16GB of DDR4, all 17GB of memory are available as a continuous flat physical address space. The memory is organized similar to Mempod [9] having two pods, each pod with one DDR4 channel (8GB) and four HBM channels (512MB). Each pod operates independently and there is no data movement across pods. The NM is mapped to the lower addresses of the physical address space. The addresses are interleaved across channels at a page granularity (4KB). We allocate the remap table and the inverted remap table to the NM.

Table 3.2: 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
3D-DRAM	HBM2 2GHz, 1 GB, 8 128-bit channels, 16 banks, tCAS-tRCD-tRP-tRAS: 7-7-7-17, RD/WR+I/O energy: 6.4pJ/bit, ACT/PRE energy: 15nJ
Main DRAM	DDR4-1600, 16 GB, 2 64-bit channels, 16 banks, tCAS-tRCD-tRP-tRAS: 11-11-11-28, RD/WR+I/O energy: 33pJ/bit, ACT/PRE energy: 15nJ
Remap Cache	64 KB per Pod, 4-way, 1 cycle access latency

Table 3.3: Benchmark characteristics.

High MPKI		
Bechmark	**Footprint(GB)**	**MPKI**
libquantum (MP)	0.3	24.8
milc (MP)	4.6	25.6
lu.D (MT)	3	25.9
lbm (MP)	3.2	30.2
bt.D (MT)	10.7	30.2
sp.D (MT)	11.2	30.2
gcc (MP)	0.7	31.3
soplex (MP)	0.5	31.4
mcf (MP)	9.5	72.9
cg.D (MT)	7.9	90.4
Medium MPKI		
Bechmark	**Footprint(GB)**	**MPKI**
ua.D (MT)	3.1	7.8
leslie3d (MP)	0.6	8.2
dc.B (MT)	4	8.5
bwaves (MP)	3.4	9
is.C (MT)	1.1	9.1
GemsFDTD (MP)	5.3	10.3
sphinx3 (MP)	0.2	12.1
mg.C (MT)	2.8	14.3
astar (MP)	1.5	15.4
omnetpp (MP)	1.2	19.2
Low MPKI		
Bechmark	**Footprint(GB)**	**MPKI**
calculix (MP)	0.3	1.3
h264 (MP)	0.1	1.4
xalancbmk (MP)	1.4	2.2
hmmer (MP)	0.1	2.3
dealII (MP)	0.4	2.9
ft.C (MT)	0.9	3.2
bzip2 (MP)	0.1	3.9
cactus (MP)	1.2	4.9
wrf (MP)	0.7	6.4
zeusmp (MP)	1.7	7.3

Simulator:

Our evaluation is performed using an in-house simulator based on *Pin* [43] following the interval-based simulation methodology [44] for the processor and cycle-accurate modelling of the memory system using DRAMSim2 [66]. We use *Cacti* to determine the access times for the caches [42]. The DRAM energy parameters are shown in Table 2.2. Through all of our experiments the memory is allocated randomly in the HBM or DDR4 proportionally to their capacity.

Workloads:

We evaluate our design with both multi-programmed (MP) and multi-threaded (MT) workloads. For the multi-programmed workloads we use benchmarks from SPEC2006 [46]. For the multi-threaded workloads we use the OpenMP version of the NAS parallel benchmarks [48] [67]. 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 more than one LLC miss per 1000 instructions (MPKI). 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 [50] while for the NAS benchmarks we simulate one billion instructions for each thread after the initialization phase. Table 3.3 shows the average LLC MPKI and the memory footprint for the simulated portion of each benchmark.

3.4 Evaluation

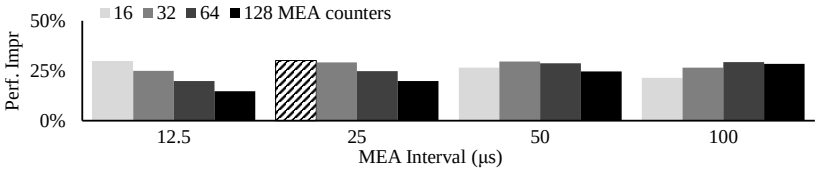
In this Section we present the results of our evaluation in terms of performance and energy efficiency. Our evaluation considers the following design points:

- **Baseline:** No migration mechanism.
- **Mempod (Mpod):** Mempod [9] migration using the Majority Elements Algorithm (MEA).
- **LLC Guided Migration (LGM):** Our proposed migration scheme described in Section 3.2.

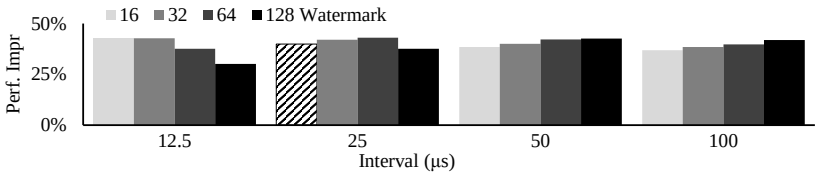
Throughout our evaluation we present results for every workload separately as well as the average over all workloads and the averages for High, Medium, and Low MPKI workload groups as defined in Table 3.3.

3.4.1 Design space exploration

Figure 3.7 shows the average performance results of our design space exploration for Mempod (3.7a) and LGM (3.7b). For Mempod we evaluate configurations for 12.5 up to 100μs intervals and 16 to 128 MEA counters. The best result for Mempod is achieved with 16 MEA counters and 25μs intervals. For LGM we examine intervals from 12.5 to 100μs and high-watermark ranging from 16 to 128 migrations. The best result is achieved for the configuration with 64 migrations high-watermark and 25μs interval. From this design space exploration we observe that Mempod is more sensitive to its parameters than our LGM design. This is because the number of counters in Mempod sets an upper limit on the number of migrations per interval as opposed to our flexible high-watermark approach which dynamically adapts the number of migrations without limiting them.



(a) Mempod.



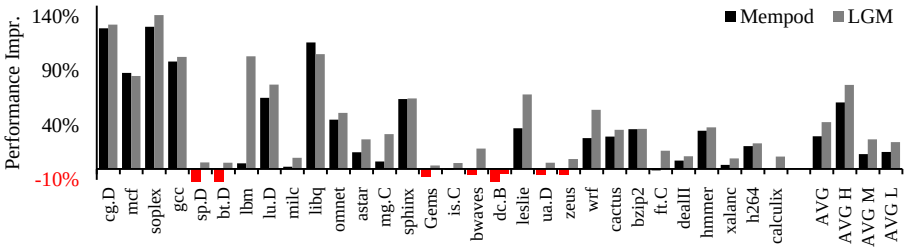
(b) LGM.

Figure 3.7: Design space exploration, average performance improvement over Baseline.

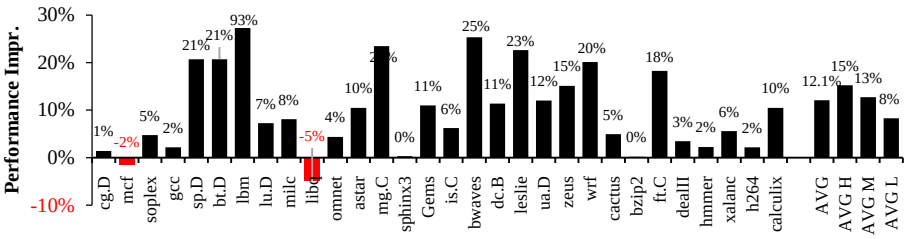
3.4.2 Performance

Figure 3.8a shows the performance of LGM and Mempod normalized to the Baseline. We can see that LGM outperforms the Baseline in all workloads except *dc.B* for which it degrades performance by 4% while Mempod degrades performance for seven workloads (up to 14% for *dc.B*). LGM improves performance on average by 43% over the baseline for all workloads, and by up to 77% on average for the most memory intensive workloads (*AVG H*). Figure 3.8b shows the performance improvement of LGM over Mempod. Our scheme can almost double performance (1.94× speedup) for one of the most memory intensive workloads (*lbm*). LGM only shows slightly worse performance than Mempod for two workloads: *mcf* and *libquantum* by 2% and 5% respectively. However, these two workloads already achieve very high performance improvement over the baseline for both Mempod and LGM. For workloads with large memory footprints and little data reuse (e.g. *sp.D*, *bt.D*, and *Gems*), Mempod degrades performance because of the significant overheads of data migration which are not justified by the reuse of migrated segments in NM. LGM on the other hand significantly increases performance because of the reduced migration overheads. Overall, the average performance improvement of LGM over Mempod is 12.1% for all workloads. For the memory intensive workloads the improvement is higher reaching 15% on average while for less memory intensive workloads improvement is smaller, 8% on average.

An interesting question arises regarding the individual performance benefits of each of the two main contributions of LGM, namely the selection of data to migrate and the reduction of migration traffic for data in the LLC. Although we do not present the detailed results here due to space limitations, we have answered the above question. A Mempod design with our idea of avoiding migrations when data are in the LLC achieves up to 7% and on average 2.3% speedup over Mempod. Moreover, an LGM with no support for avoiding migration of data that are stored in the LLC achieves up to 23% speedup and on average is 2% better than Mempod. The fact that the combination of the two gives LGM 12.1% better performance over Mempod indicates that our selection approach migrates data when they are available in the LLC reducing



(a) Performance improvement of Mempod and LGM over Baseline.



(b) Performance improvement of LGM over Mempod.

Figure 3.8: Performance improvement of (a) Mempod and LGM over Baseline and (b) LGM over Mempod.

significantly their traffic cost. Actually, on average 29 cachelines out of 32 in a segment are present in the LLC when the segment is migrating. This saves almost entirely the traffic overhead for bringing a segment to the NM and the only overhead that remains is the one of evicting a segment from the NM to the FM.

Sensitivity to remap cache size:

Figure 3.9 shows the impact of the remap cache size on Mempod and LGM performance (normalized over baseline). Both Mempod and LGM show a similar performance improvement as the size of the remap cache increases. For the rest of our evaluation we assume a 64KB remap cache as in Mempod [9].

Sensitivity to DRAM latency:

Figure 3.10 shows the impact of different DRAM latency characteristics on the performance of Mempod and LGM and an Ideal design where all data are present in the NM, the performance is normalized over the Baseline. From left to right we show the average speedup of the different designs for:

- DDR4: Both NM and FM having the same latency characteristics as the *Main Memory* in table 3.2.
- HBM: Both NM and FM having the same latency characteristics as the *3D-DRAM* in table 3.2 and,
- DDR4 + HBM: Each memory with different latency characteristics as in table 3.2.

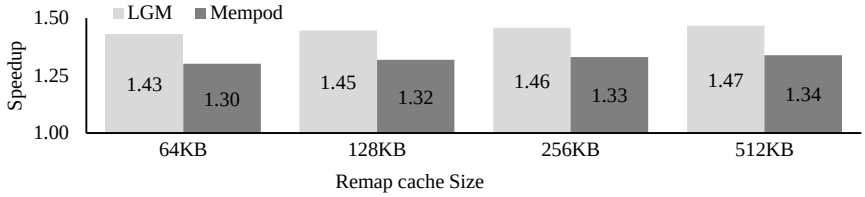


Figure 3.9: Average performance improvement over Baseline for Mempod and LGM for different remap cache sizes.

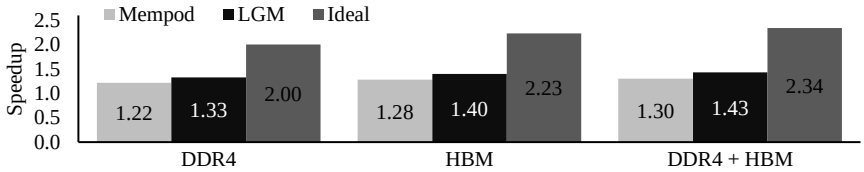


Figure 3.10: Average speedup over baseline for Mempod, LGM and Ideal design for different DRAM latency characteristics.

From these results we see that in all cases the performance advantage of LGM over Mempod remains similar regardless of the latency difference between NM and FM.

3.4.3 Traffic

To understand the reasons behind our LGM's better performance we need to examine the ratio of processor memory requests that are served by the NM over FM. Figure 3.11 shows the ratio of requests to NM over FM for Mempod and LGM. LGM serves on average 4.9 NM requests for every request served from FM, more than three times higher than the ratio achieved by Mempod. This is a clear indication that LGM's migrations are more effective. This effect is more pronounced in Lower MPKI workloads where LGM serves on average 9.1 requests from NM for every FM request, which is more than five times the ratio for Mempod. This is due to the fact that LGM, in contrast to Mempod, does not limit the number of migrations per interval. It is worth noting that for all workloads LGM always shows a ratio higher or equal to Mempod. Even in the cases where the ratio is not improved significantly, LGM still outperforms Mempod because of the more efficient data migrations overall.

Figure 3.12 shows the traffic for Mempod and LGM normalized to the baseline traffic. The traffic is separated in NM and FM *Processor Traffic* (PT), which concerns the memory requests coming from the processor, *Remap Traffic* (RT), which is the traffic imposed by the remap cache misses, and *Migration Traffic* (MT). For some workloads such as *milc* and *ua.D* the sum of NM and FM PT for LGM exceeds the traffic of the Baseline system, this happens when our migration mechanism marks as dirty, cachelines that would have not been dirty otherwise. These cases are just a few and with little impact on the performance of our design. As mentioned before, when comparing just NM and FM traffic between Mempod and LGM, for all workloads our design is able to "convert" more FM traffic into NM traffic. Nevertheless, the total traffic for LGM is not always less than Mempod. For example, for *lu.D*, *ua.D*, and especially for *dc.B* where the total traffic for LGM is up to 1.4 times the total traffic for Mempod. This is due to higher migration traffic. However, LGM migrates

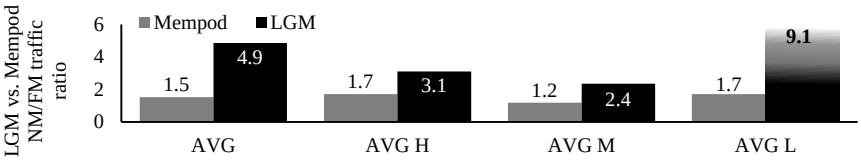


Figure 3.11: NM to FM processor request ratio of Mpod and LGM.

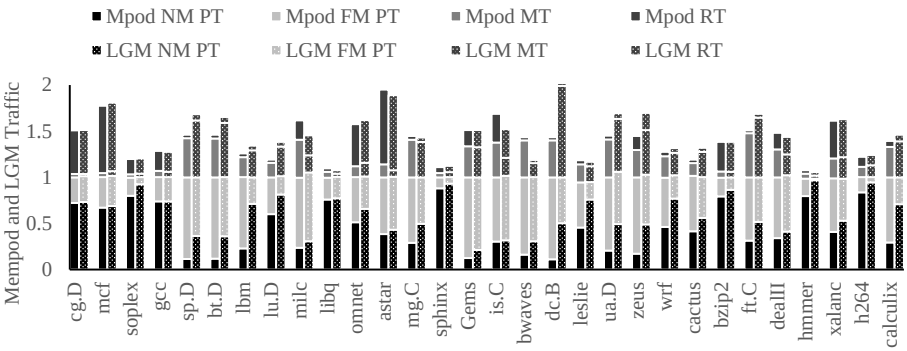


Figure 3.12: Memory traffic breakdown for Mempod and LGM in Processor Traffic in NM (NM PT) and FM (FM PT), Migration Traffic (MT) and remap traffic (RT) normalized to Baseline traffic

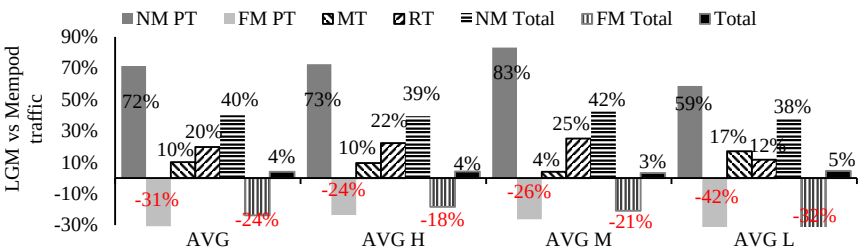


Figure 3.13: LGM Traffic vs. Mempod for processor traffic (PT) in NM/FM, migration traffic (MT), remap cache traffic (RT).

substantially more data to NM than the increase of its migration traffic due to imposing lower overheads. These additional migrations are then reused effectively resulting in an overall performance improvement over Mempod.

Figure 3.13 shows the average difference between LGM and Mempod for all of the above traffic classes and for the total traffic in NM (*NM Total*), FM (*NM Total*) and both (*Total*) for all workloads, as well as the average for each workload group. LGM increases the processor traffic to the NM by 72% while it decreases the processor traffic to the FM by 31%. Migration Traffic is increased by 10%, this increase in migration traffic is however small when considering that LGM performs on average two times more migrations than Mempod. This shows that migrations in our design come at almost half the cost compared to Mempod. We also notice a significant increase in the remap cache traffic. This is a direct consequence of the larger number of migrations

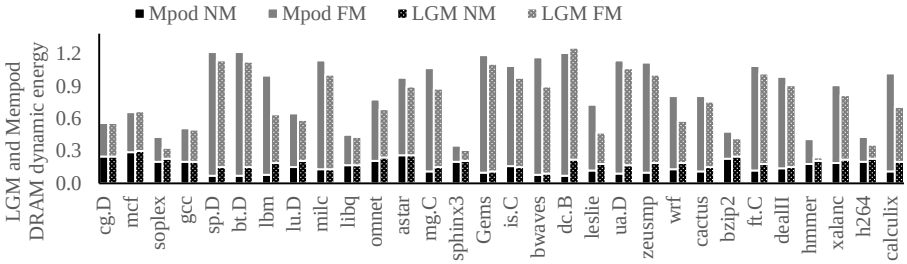


Figure 3.14: Memory dynamic energy consumption for NM and FM normalized to Baseline.

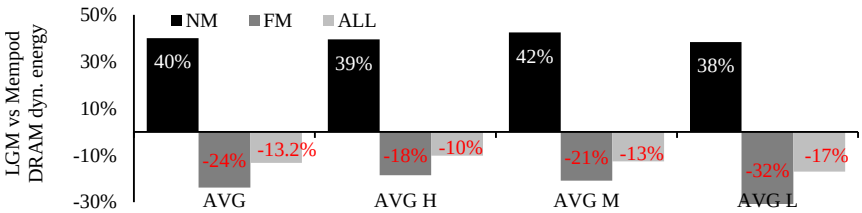


Figure 3.15: Difference in DRAM Dynamic energy consumption of LGM over Mempod.

since the remap tables are updated more often, leading to more remap cache writebacks and misses. LGM increases the overall system traffic by 4%, however note that this increase concerns only the fast and energy efficient NM for which the traffic increases by 40% on average while for the slower, more energy demanding FM the total traffic is reduced by 24% on average.

3.4.4 Energy Consumption

Figure 3.14 shows the dynamic energy consumption of NM and FM for all workloads for Mempod and LGM normalized to the baseline system. For most workloads both designs achieve lower total energy consumption than the baseline. The total dynamic memory consumption of LGM is lower than Mempod for all workloads except *dc.B*. The largest energy gain of LGM is 40% for *hmmer*. For *dc.B*, LGM performs more migrations to NM than Mempod resulting in a 2× increase in NM dynamic energy compared to Mempod however, it achieves a significant reduction in FM energy and the net effect is only 4% higher enegry than Mempod; note that here LGM offers 11% higher performance. For all other workloads, the fact that LGM moves more data to NM results in an increase in the NM dynamic energy, however, this decreases requests to FM. Since FM is more energy demanding than NM, the net effect is an important reduction in the total dynamic memory consumption. Figure 3.15 shows the difference in DRAM dynamic energy consumption of LGM over Mempod for NM, FM and total for both memories. Looking at the results for NM and FM separately we observe the same trends as the traffic served by each memory. The NM energy consumption increases by 40% on average while FM decreases by 24%. This behaviour is due to the substantially higher ratio of memory requests served by NM in our design and

because of the additional migration and remap cache traffic. Overall, this redistribution of traffic towards the more energy efficient NM results in a reduction of the memory system energy by 13.2% on average.

3.5 Conclusions

This Chapter presented LGM, an LLC-guided Migration scheme for flat address space hybrid memory systems. LGM employs the LLC to select memory segments for migration between the low-bandwidth far memory and the high-bandwidth near memory. Segments with large number of cachelines in the LLC indicate higher potential for reuse, especially if their cachelines are dirty. Letting the LLC select segments for migration achieves a convenient timing as a fraction of these segments is present in the LLC. Then, data under migration that are available in the LLC do not need to be read from the far memory and also do not need to be written to the near memory immediately, thereby, reducing migration traffic. Our experiments show that LGM reduces migration traffic to almost half and enables more data to be migrated more efficiently. This results in a three-fold increase on the ratio of memory requests serviced by the near memory. Overall, LGM improves performance by 12.1% and reduces memory system dynamic energy by 13.2% compared to the state-of-the-art.

Chapter 4

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

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 [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 [7–11, 35]; the second one uses NM as DRAM cache and FM as the main memory [12–26, 31].

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 [34]. 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 act as a cache for 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 Chapter is organized as follows: Section 4.1 presents related work and some motivating results for our proposed design. Section 4.2 describes the *Hybrid*^{2} architecture. Section 4.3 explains our experimental setup. Section 4.4 offers our evaluation results. Finally, Section 4.5 summarizes our conclusions.

4.1 Related Work and Motivation

Various memory technologies exhibit different trade-offs in terms of capacity, bandwidth, access latency, and cost [57, 58, 60, 61, 68, 69]. A promising direction towards a more efficient memory system design is to combine memory technologies with complementary 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 [60, 61]. Although the latest 3D-stacked DRAM memories can offer capacities that reach up to 24GB per stack (i.e. HBM2 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 NM and the capacity of FM. The first approach uses the NM as a DRAM cache and the FM as the main memory; the second one combines NM and FM in a flat address space and supports migration between them.

4.1.1 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 [12]. 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 DRAM access [24]. 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 [21]. These approaches are scalable but limit the design options of DRAM caches [31].

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 [19]. The Decoupled Fused Cache (DFC) 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 [31]. 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 [23]. Unison Cache follows the same approach, but stores the tags in DRAM for scalability to larger cache sizes [14]. Finally, Footprint Tagless DRAM Cache combines the Tagless with the footprint design [20]. 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 [55] to balance bandwidth utilization. BATMAN monitors the number of accesses to both 3D stacked DRAM and conventional DRAM and regulates data movement [70]. Finally, BEAR proposes mechanisms to limit the bandwidth used by secondary operations of DRAM caches [22].

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 [37].

4.1.2 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 [32]. 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 to 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 [34]. 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 [33]. 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 upon eviction. PageSeer proposes using the Memory Management Unit (MMU) to prefetch pages from Non Volatile Memory to conventional DRAM [11]. SILC-FM presents a more flexible group approach [10], it uses set-associative swap-groups and migrates 2 KByte blocks, but allows sub-blocks to interleave data in the 3D-stacked DRAM.

4.1.3 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

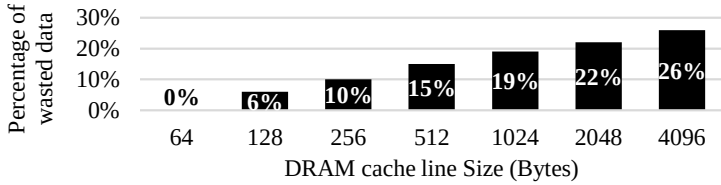


Figure 4.1: Average percentage of data brought in DRAM cache, but remained unused, with respect to cache line size.

memory in the address space. To preserve the memory space, data migration must swap data instead of just copying them like a cache. 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.

DRAM caches and migration schemes face some similar issues. One such 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 excessive bandwidth due to 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 4.1 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].

Migration schemes exhibit similar trade-offs with regard to migration aggressiveness and granularity of migrated data. 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 4.2 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 [35] and Chameleon (CHA) [8] and for caches we show the results for DFC [31], Tagless [21], 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. 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 degrade performance due to over-fetching. On the contrary, migration schemes do not have that risk as they do not bring

[1] Average results with 1GB DRAM cache for the benchmarks described in Section 4.3.

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 greater for smaller cache line sizes.

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. 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 entirely on the processor die offering short access latency, in addition it prevents the negative performance effects of large cache lines. Moreover, we choose our DRAM cache to be sectored and migrate data at sector granularity, this allows us to reduce the metadata overheads and at the same time not waste precious far memory bandwidth by only fetching the requested cache lines on cache misses. In order to put together caching and migration efficiently, our design needs to 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.

4.2 Hybrid Caching and Migration

Hybrid^2 is a hybrid memory system architecture that combines a DRAM cache with a flat address space migration scheme. It uses 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.

4.2.1 Hybrid^2 System Overview

Hybrid^2 exploits 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 area cost, while the data part of the DRAM Cache is kept in NM occupying a relatively small portion of it.

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 if the requested data are currently located in FM; Section 4.2.4 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

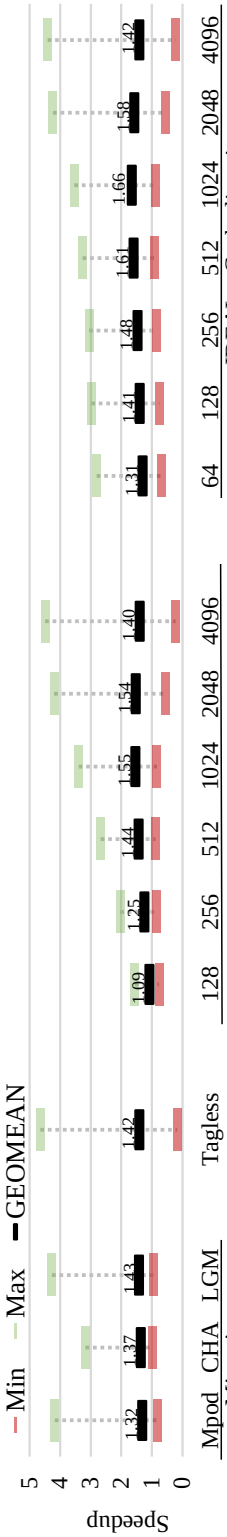


Figure 4.2: Minimum, Maximum, and Geometric mean of the speedup achieved by migration and DRAM cache designs.

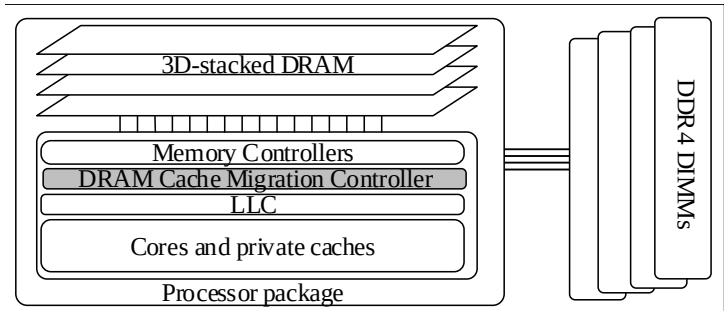


Figure 4.3: System Overview.

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, reducing 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 reducing the address translation and tag lookup overheads. Section 4.2.2 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 4.3 presents an overview of the system considered in this work. It consists of the processor, 3D-stacked DRAM and conventional DRAM. The conventional DRAM is the FM and the 3D-stacked DRAM is the NM. The shaded box is the DRAM Cache 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.

4.2.2 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 4.4. 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

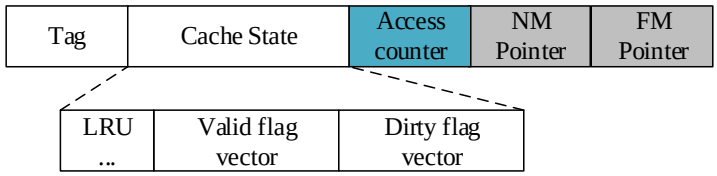


Figure 4.4: eXtended Tag Array Entry (XTA).

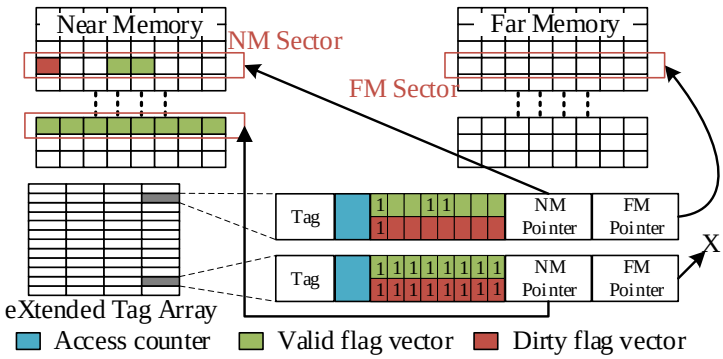


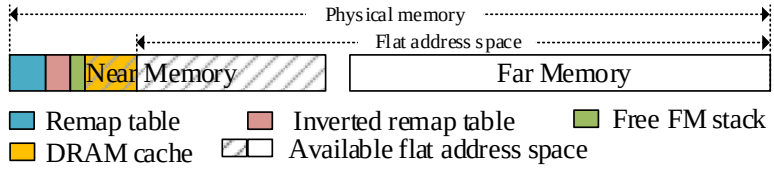
Figure 4.5: *Hybrid*^{2} XTA example.

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.

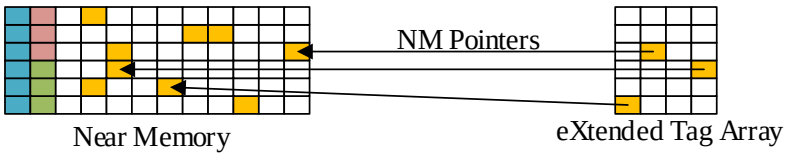
Figure 4.5 shows an example use of the XTA entries. The top entry corresponds to a sector that is partially present in the DRAM cache and thus not migrated to the NM, as such, some cache lines of that sector have been fetched to the NM, as denoted by the valid flag vector of the XTA entry. The dirty 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.

4.2.3 Memory space layout and metadata

Figure 4.6a shows the layout of the NM and FM, the coloured areas are reserved and the uncoloured area is available as a flat address space. Note that the sectors that



(a) Memory system logical address space layout.



(b) Near Memory Layout and XTA use.

Figure 4.6: Memory layout and reserved memory.

correspond to each XTA entry can be located anywhere in the lined area of NM. Figure 4.6b 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 sectors out of the NM and more details are provided in Section 4.2.5. 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 4.5.

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 the sectors of these locations 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. In our implemetnation, each entry of the remap table, inverted remap table, and Free-FM-Stack is 4 bytes. Overall, the space required for the remapping data structures is at most 3.5% of the NM capacity.

4.2.4 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 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 either be fully in the DRAM cache (there exists an entry in the XTA for them) or not at all. When a request arrives in the DCMC, the address is used to index the XTA to determine if the sector and

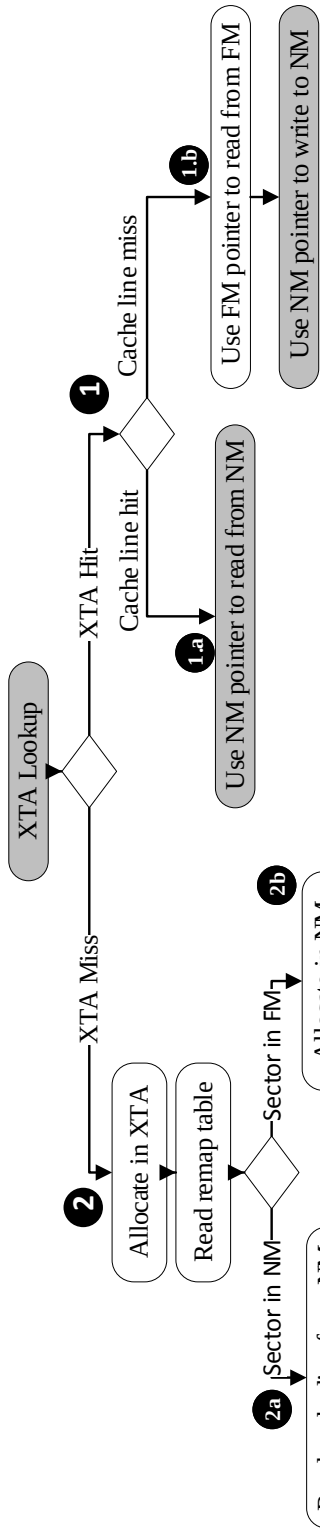


Figure 4.7: Memory access path.

specific cache line is available in the DRAM cache. There are four possible outcomes as shown in Figure 4.7, 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, in both cases 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 in NM 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 4.2.5). 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 in NM 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 4.14 – No Remap).

4.2.5 Allocating NM

In case of an XTA miss where the requested sector is in FM (2b in Figure 4.7) 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[2] 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 4.8.

[2]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.

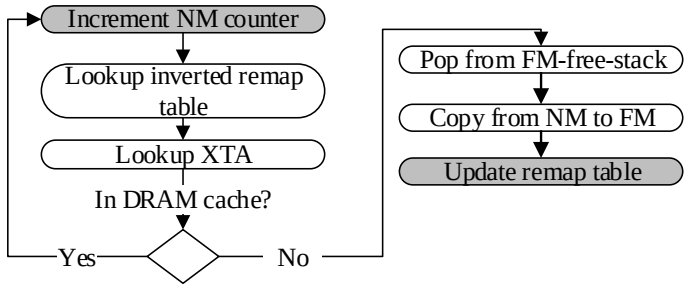


Figure 4.8: Allocating a sector in NM.

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 4.6a) 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 (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 location of the sector. 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. Moreover, it provides a better replacement decision than just FIFO, because often accessed sectors will probably reside in DRAM cache and thus not considered as victim sectors.

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.

4.2.6 DRAM cache evictions

The DRAM cache eviction logic is illustrated in Figure 4.9. It uses a standard LRU algorithm to decide which sector to evict. The DRAM cache can contain sectors that have already been migrated to the NM or 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, 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 for 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, 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

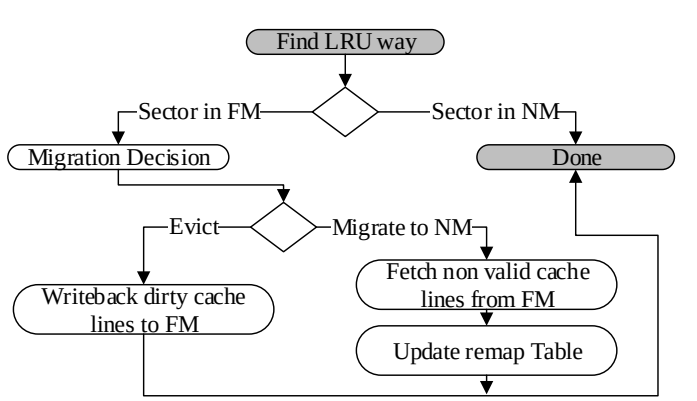


Figure 4.9: Eviction from DRAM cache.

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. Below, we present the algorithm for deciding between evicting a sector to FM or migrating it in NM.

4.2.7 Migration Decision

Figure 4.10 illustrates the algorithm used for deciding whether to migrate a sector to NM when evicted from the DRAM cache. First, it is checked whether the overhead of FM accesses caused by the migration in question can be afforded ❶. This is performed with the help of a counter, which is incremented upon processor FM accesses and decremented upon migration FM accesses. If the counter value is larger than the additional FM accesses needed for migrating the sector, then the sector is considered further, otherwise it is evicted to its existing FM location. In practice, this first check regulates migration traffic overheads and ensures a balance between processor and migration traffic. If this first check is successful, then it is checked whether the sector is (one of) the most accessed in the DRAM cache set ❷. If so, the sector is migrated, otherwise it is evicted to its existing FM location. Bellow we describe in detail the mechanism used for regulating migration traffic overheads as well as how the most accessed sector in a DRAM cache set is determined.

Migration traffic overheads:

There are two alternatives when evicting from the DRAM cache a sector which is *not* already migrated to NM: (i) either evict it back to FM, which requires all dirty cachelines to be written-back to FM, or (ii) migrate the sector to NM by fetching the remaining cache lines of the sector to NM. The overheads of migrating a sector in terms of FM accesses is the difference between the FM accesses required in each of the two above cases and is calculated based on the number of valid and dirty cache lines of the sector in the DRAM cache. Specifically, just evicting the sector to FM requires a number of FM accesses (E_{cost}) equal to the number of dirty cache lines which have to be written back to FM (N_{dirty}):

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

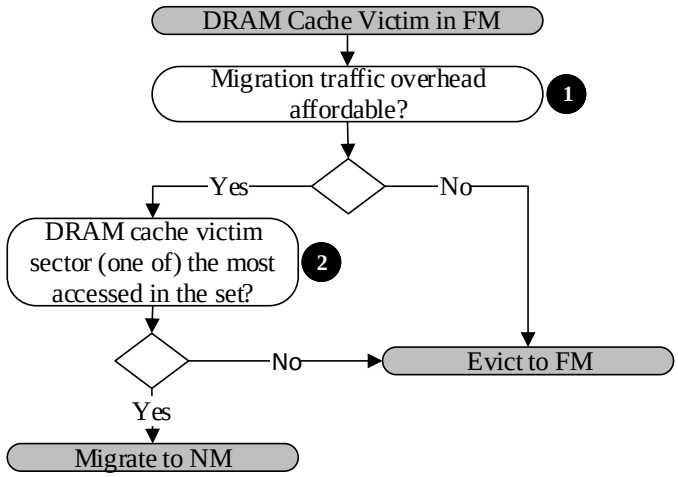


Figure 4.10: Migration decision.

If migrated to NM, the sector needs a number of FM accesses (M_{cost}) described by the following equation:

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

That is equal to the total number of cache lines in the sector (N_{all}) minus the valid ones already in the DRAM cache (N_{valid}), plus the cost of swapping out from NM a victim sector, which requires N_{all} write-backs to FM. Then, migrating a sector has an overhead of FM accesses (MO_{cost}) equal to:

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

where the constant "1" is added as a minimum migration cost. Then, the MO_{cost} can vary from 1 when all cache lines of a sector are valid and dirty, to $2 * N_{all}$ when only one cache line of a sector is valid and clean when evicted from the DRAM cache.

Balancing Migration and Processor traffic:

A single counter (called *FM access counter*) is employed to regulate migration accesses to FM with respect to processor FM accesses. This counter is incremented for every DRAM cache miss which must be fetched from FM. When a sector is migrated, the counter is decremented by its migration overhead (MO_{cost}). When deciding for a sector migration, its MO_{cost} is compared with the counter value and if it is smaller then the sector is considered further for migration. In essence, this counter sets an upper bound to the number of FM accesses spent for migration and is reset periodically (every 100K cycles) to adjust to workload phase changes.

Most Accessed Sector in the Set:

For every sector in the DRAM cache, a counter is maintained in the XTA which is incremented on every access to that sector (Figure 4.4). Upon eviction, the value

of the counter for the victim sector is compared against the counters of the other sectors in the cache set. This comparison concerns counters of a few bits (9 bits in our implementation) and happens during eviction so it is not in the critical path of a memory access. In case the counter value of the sector is greater or equal to all other counters in the set, then the sector is migrated. In case there is another sector in the set with a counter value greater than the evicted sector counter then the sector is evicted without migration. The counter is only incremented for sectors that have not been migrated to ensure there is no starvation in a set by NM sectors that have many accesses and therefore not evicted from the XTA. Furthermore, 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.

4.2.8 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 here, *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 4.2.5). 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.

Table 4.1: 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 [71]	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

4.3 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 4.1, 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, 1:8 and 1:4.

Simulator: Our evaluation is performed using a Pin-based in-house simulator [43] following the interval-based simulation methodology [44] for the processor and cycle-accurate modelling of the memory system using DRAMSim2 [66]. We use *Cacti* to determine the access times for the caches [42]. Through all of our experiments the memory pages are allocated randomly in the HBM or DDR4 proportionally to their capacity, for example for 1:4 NM:FM ratio, 1/5 are allocated to NM and 4/5 to FM. 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 multi-threaded (MT) workloads. For the multi-programmed workloads we use the SPEC2017 benchmark suite [72]. For the multi-threaded workloads we use the OpenMP version of the NAS parallel benchmarks [48] [67]. For each of the NAS benchmarks we use 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 Last Level Cache (LLC) capacity (8 MBytes). 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 [50] while for the NAS benchmarks we simulate one billion instructions for each thread after the initialization phase. Table 4.2 shows the average 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 4.4 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.

4.4 Evaluation

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* for a fair comparison. As described in Section 4.2.8 *Hybrid*^{2} could also use more free space in the same way as Chameleon.
- **LLC-guided data migration (LGM)** [35]. 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)** [21]. For the Tagless DRAM cache, we optimistically do not model any operating system overheads like the extra memory accesses on TLB misses or page-faults.
- **Decoupled Fused Cache (DFC)** [31]. 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.

4.4.1 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 [35]. 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 4.11 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 4.1). For the same DRAM cache size and cache line size, 2 KByte sectors perform better than 4 KByte sectors. Larger 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.

4.4.2 Performance

Figures 4.12a, 4.12b, and 4.12c show the geometric mean of the speedup over a baseline without NM for each MPKI class as well as for all benchmarks, for three different NM to FM ratios (1:16, 1:8, 1:4). From the results we see that all designs benefit from larger NM:FM ratios.

Table 4.2: 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

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

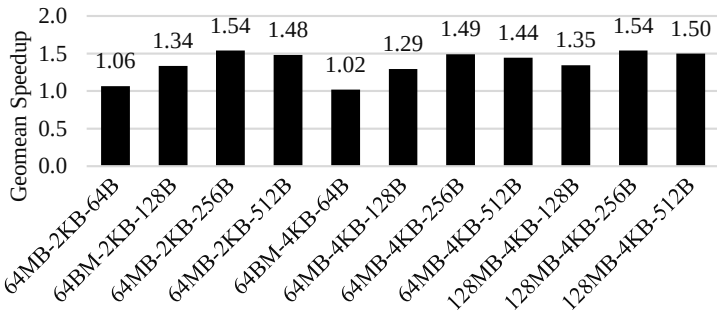


Figure 4.11: Design space exploration: Geometric mean of the speedup over baseline for different *Hybrid*^{2} configurations.

pronounced in bigger NM sizes.

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 which would degrade the performance of caches more severely than migration schemes.

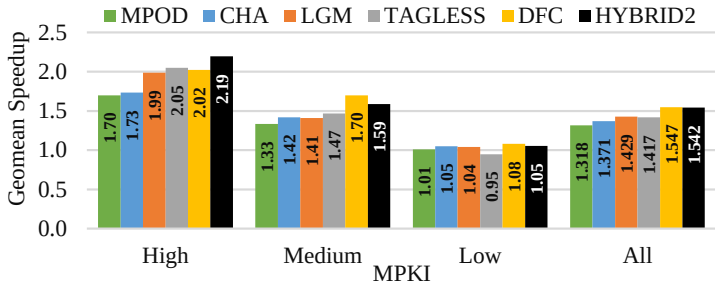
For the rest of this Section, we present detailed results for the 1:16 NM:FM ratio as it stresses all designs more due to the smaller NM size which is smaller than the memory footprint of most benchmarks.

Figure 4.13 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 benchmarks 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 because of their small memory footprint. Notice how large cache line 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.

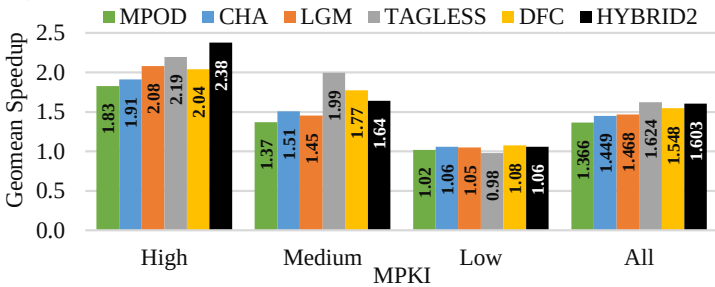
Overall, *Hybrid*^{2} outperforms other migration schemes and matches -or surpasses for high MPKI workloads- cache performance without wasting NM capacity.

Performance breakdown:

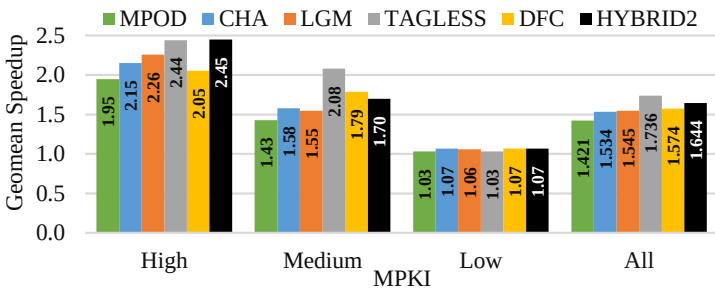
The performance of *Hybrid*^{2} can be attributed to both the DRAM cache and the migration components as well as the elimination of address translation overheads. To show the effects of each factor above we conducted a series of experiments. Figure 4.14 shows the geometric mean of the speedup achieved for a number of different designs.



(a) Geometric mean of the 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 4.12: Geometric mean of the speedup over baseline for high, medium, low MPKI, and all benchmarks for NM sizes of 1GB, 2GB and 4GB.

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. 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

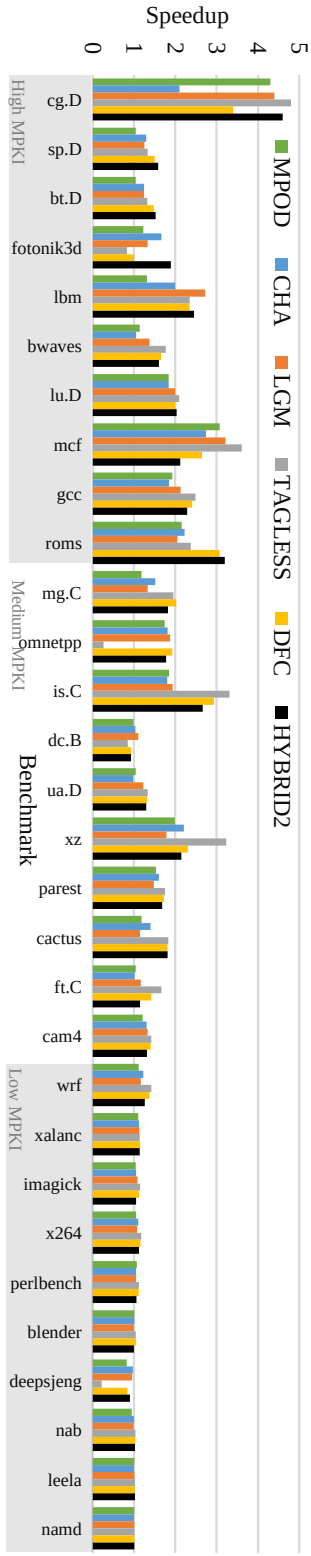


Figure 4.13: Speedup over baseline.

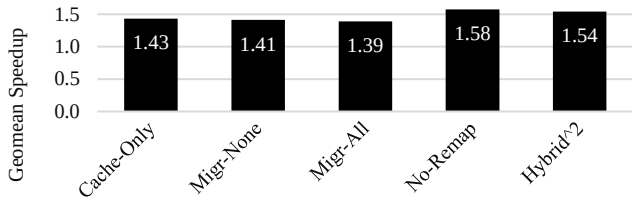


Figure 4.14: *Hybrid*^{2} Performance factors breakdown.

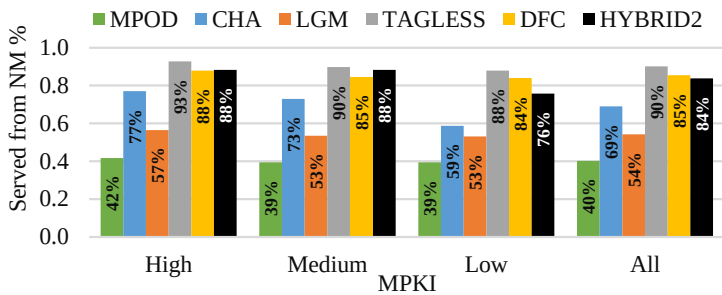


Figure 4.15: Geometric mean of normalized processor requests served from NM for benchmarks with high, medium, and low MPKI.

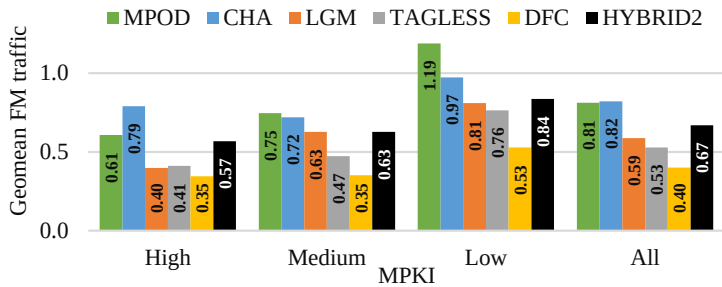


Figure 4.16: Geometric mean of normalized FM traffic for benchmarks with high, medium, and low MPKI.

effectively tackles the address translation overheads. 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.

NM Utilization:

Figure4.15 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

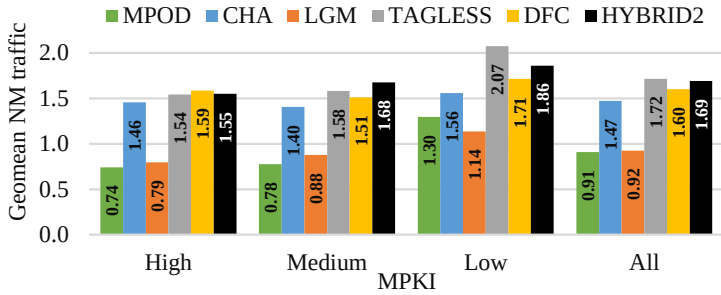


Figure 4.17: Geometric mean of normalized NM traffic for benchmarks with high, medium, and low MPKI.

designs at 90% while its performance is considerably lower. On average, 84% of processor requests in *Hybrid*^{2} are served from NM; this percentage is even higher for High and Medium MPKI workloads. DFC achieves a slightly higher ratio at 85%. *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 54% as its bandwidth saving mechanism allows it to migrate more aggressively, finally, Chameleon achieves the best of all migration designs at 69% on average.

4.4.3 Traffic

Figure 4.16 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 comapred to 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 of the baseline.

Figure 4.17 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 fewer processor requests that are served from NM (Figure 4.15).

4.4.4 Energy consumption

Figure 4.18 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.

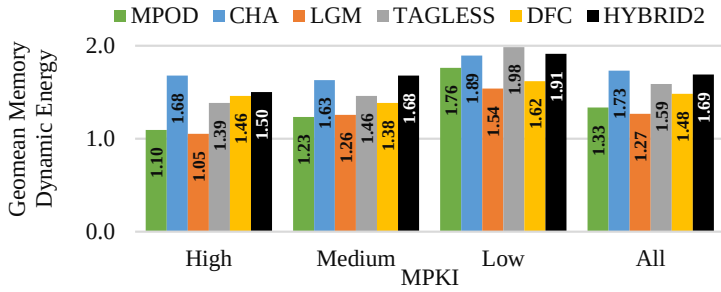


Figure 4.18: Geometric mean of normalized dynamic memory energy for benchmarks with high, medium, and low MPKI.

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}.

4.5 Conclusions

This Chapter 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 the memory system and implements transparent data migration in hardware. 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.

Chapter 5

Conclusions

This thesis considered the design of hybrid memory systems that consist of 3D-stacked and conventional off-chip DRAM. Conventional DRAM offers limited bandwidth at high capacity while 3D-stacked DRAM offers high bandwidth with limited capacity. The main question in the design of such systems is how to best utilize the 3D-stacked DRAM. Existing approaches use the 3D-stacked DRAM either as a cache or as part of a flat address space with data migration. Each of the two existing approaches comes with its own tradeoffs and inefficiencies.

DRAM caches suffer mainly from the tag lookup overheads as, because of their size, they require large tag arrays which cannot be stored on-chip. DRAM cache tags are most commonly stored along with the DRAM cache data in the 3D-stacked DRAM. Storing the tags in the 3D-stacked DRAM however incurs high access latency compared to on-chip tags. Furthermore, accessing the tags causes traffic which in turn increases the queuing latency as it competes for bandwidth with data accesses. Decreasing the tag lookup overheads can improve the performance of DRAM caches.

Data migration designs use the 3D-stacked DRAM capacity as part of the address space, because of this, data has to be swapped between the two memories in order to utilize the 3D-stacked DRAM. Swapping however, comes at a much higher traffic overhead compared to simply copying as in the case of caches. This traffic overhead presents a major performance bottleneck factor for data migration as it burdens both the 3D-stacked and conventional DRAM channels. Additionally, because of the high migration traffic overheads, data migration schemes have to be selective about which data to migrate to 3D-stacked DRAM. Ideally, only data with the best potential for future reuse must be migrated, spatial and temporal locality can be good indications for future reuse. Only limiting the migration bandwidth can substantially improve the performance of data migration designs, combining it with better migration data selection can compound the effects and lead to overall better system performance.

Both DRAM caches and data migration offer promising alternatives to utilizing the 3D-stacked DRAM, however, they represent two extremes in the design space. DRAM caches are more bandwidth efficient than data migration but they sacrifice the 3D-stacked DRAM capacity to achieve that, rendering it transparent to the rest of the system. Data migration preserves the 3D-stacked DRAM capacity at the cost of elevated traffic overheads and a dependence on the efficiency of the migration selection algorithm. Until recently, a memory system design that could bridge the gap between these two extremes had not been proposed.

5.1 Summary

Chapter 2 proposes *Decoupled Fused Cache* (DFC), a DRAM cache design that alleviates the cost of tag accesses by fusing DRAM cache tags with the tags of the on-chip LLC [31]. DFC proposes an LLC organization that resembles decoupled sectored caches [30] and adds a few extra fields that store information about the contents of the DRAM cache. One tag in the LLC corresponds to a DRAM cache line and at the same tag can be associated with several LLC cache lines. This way, after the first LLC miss for a cache line which belongs to a bigger DRAM cache line, the way of the DRAM cache line is stored in the LLC tag array. This allows accessing the DRAM cache directly after most LLC misses and for all writebacks. In essence, DFC relies in most cases on the LLC tag access to retrieve the required information for accessing the DRAM cache avoiding most tag lookups. DFC can support a configurable (at boot time) DRAM cache line size, which is a power-of-two multiple of the LLC cache lines. The only limitation of DFC lies in that the DRAM cache lines must be at least twice the size of LLC cache lines. DFC is based on our previous work, FusionCache [41], which was our first attempt to mitigate the tag lookup overhead for DRAM caches and overcomes its limitations.

Chapter 3 proposes *LLC-guided Data Migration* (LGM), a new data migration scheme for hybrid memory systems that improves data migration efficiency [35]. LGM improves data migration by using the LLC to achieve two complementary goals. The first goal is to lower the migration traffic overheads, this is achieved by skipping the cache lines of a data segment when it is migrated, if they are present at the LLC at the time of migration. To ensure correctness, the skipped cache lines are marked in the LLC so that they are always copied back to the 3D-stacked DRAM when evicted. The second goal is to improve the selection algorithm for selecting data to migrate to 3D-stacked DRAM. This goal is also facilitated by the LLC which we use to detect the spatial locality of data and their potential for future reuse. The locality and future reuse potential of coarse granularity segments is inferred based on the number of cache lines of each segment in the LLC and their state (dirty or not). Furthermore, to make the migration algorithm adapt to different workloads and workload phases, we employ a mechanism that changes the migration criteria at short time intervals based on the number of migrations during the previous interval. LGM is implemented in hardware and uses a decoupled sectored organization for the LLC in order to track data at the coarser (segment) granularity required for data migration. LGM supports all to all migration, which means that every memory segment can be located anywhere in the memory system without any mapping restrictions.

Chapter 4 proposes *Hybrid*^{2}, a new hybrid memory system architecture that combines a DRAM cache with a migration scheme [38]. *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. The small DRAM cache faithfully follows the working set of workloads, fetching data as they are requested by the processor, just like a larger cache would do, giving the benefits of caching to the memory system. At the same time, data is selectively migrated to the rest of the 3D-stacked DRAM space when evicted from the DRAM cache. The DRAM cache is sectored to allow for cheaper on-chip tag-storage, while fetching at cache line granularity to avoid over-fetching. Migration is managed at larger (sector) granularity which matches the DRAM cache sector size. To manage both caching and migration, *Hybrid*^{2} uses a small, on-chip, tag array which completely covers the DRAM cache contents and also

uses indirection to facilitate the address translation that is required for data migration. Additionally, indirection allows to avoid relocating data within the 3D-stacked DRAM on migration. To enable this, the 3D-stacked DRAM is split logically between cache and migration space, this allows the DRAM cache data to be placed anywhere in the 3D-stacked DRAM. To make more informed migration decisions, the DRAM cache is used as a staging area to select the data most suitable for migration. To decide which sectors to migrate when they are evicted from the DRAM cache, DFC uses an access counter for every sector in the DRAM cache to find the most accessed sectors in each cache set. Furthermore, the migration decision takes into account the spatial locality of each sector as well as the overall required traffic for migration.

5.2 Contributions

In order to improve the performance of hybrid memory systems that use the 3D-stacked DRAM as a cache, this thesis proposes *Decoupled Fused Cache*, a new DRAM cache architecture which:

- Mitigates the cost of accessing the DRAM cache tags while enforcing minimal design restrictions
- Supports a configurable (at boot time) DRAM cache line size, which is a power-of-two multiple of the LLC cache lines
- Compared to DRAM cache designs of the same cacheline size, improves system performance by 11% and reduces DRAM cache traffic by 25% and DRAM cache energy consumption by 24.5% on average.

When the 3D-stacked DRAM is used as a part of a flat address space with hardware data migration, this thesis proposes LLC-guided Data Migration, a data migration design which:

- Uses the on-chip LLC to guide the selection of data to be migrated to 3D-stacked DRAM.
- Reduces the migration traffic overheads by skipping cache lines already in the LLC.
- Outperforms current state-of-the art migration designs improving system performance by 12.1% and reducing memory system dynamic energy by 13.2%.

In order to combine the benefits of caching and migration, this thesis proposes *Hybrid*^{2}, a hybrid memory system design that:

- Uses a small part of the 3D-stacked DRAM as a cache so as not to waste most of the capacity.
- Supports migration for data when evicted from the DRAM cache.
- Depending on the capacity ratio of 3D-stacked DRAM to conventional DRAM, outperforms current state-of-the-art migration schemes by 6.4 to 9.1% on average and compared to caches, gives away only 0.3 to 5.3% of performance offering 5.9 to 24.6% more main memory capacity.

5.3 Future Work

There are several directions for future research that can improve and complement the work presented here. In the following, we identify and list some of them:

Involving the software in hybrid memory system management: All the designs proposed in this thesis are implementable in hardware and are transparent to the software layer. Being transparent to the software means that programmers and OS developers need not worry about the underlying architecture which enables easier adoption of new designs. However, this means that the hardware must infer any information about the software from the memory access patterns. Actively co-designing hardware and software to achieve better performance could lead to overall more efficient systems.

Non-volatile memories: It would be interesting to evaluate the impact of the designs and ideas proposed in this thesis on other memory technologies such as Non-Volatile Memories as these have radically different access characteristics to DRAMs. Furthermore, there are other factors that might affect design decisions such as wearing in flash-based memories and/or heat dissipation and peak power consumption for write operations for ReRAMs and STT-RAMs.

Non-Uniform memory systems: Another interesting research direction would be the design of hybrid memory systems for multi-node systems. With the emergence of chiplet-based designs the memory system strays from the single-point, monolithic abstraction. New hardware and software mechanisms might be required to provide a familiar programming paradigm for more distributed systems, even within the same chip.

Blurring the lines between memory and storage: Another possible research direction is to try to change the conventional views about memory and storage. Traditionally these two are managed in different ways by the software due to their different access characteristics. Storage is slow and managed explicitly by the operating system while memory is an integral part of computation and is managed mostly by the hardware. The emergence of non volatile memories with much better bandwidth and latency, combined with the vast capacity they can offer opens up new directions. To fully utilize this technology we have to re-visit the traditional abstractions, programming models, and operating systems. Hardware memory/storage architectures will have to provide new primitives which will enable managing such systems and make the software transition easier.

Bibliography

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH C.A. News*, vol. 23, no. 1, pp. 20–24, 1995.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA-42*, 2015, pp. 336–348.

[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*, 2015, pp. 105–117.

[4] M. Pavlovic, Y. Etsion, and A. Ramirez, "On the memory system requirements of future scientific applications: Four case-studies," in *IISWC*, 2011, pp. 159–170.

[5] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," in *ISCA-36*, 2009, pp. 371–382. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555801

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

[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*, Dec 2014, pp. 13–24.

[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)*, Oct 2018, pp. 533–545.

[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)*, Feb 2017, pp. 433–444.

[10] 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)*, Feb 2017, pp. 349–360.

[11] 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)*, Feb 2019, pp. 596–608.

[12] 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)*, Dec 2011, pp. 454–464.

[13] G. Loh and M. D. Hill, “Supporting very large dram caches with compound-access scheduling and missmap,” *IEEE Micro*, vol. 32, no. 3, pp. 70–78, May 2012.

[14] 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*, Dec 2014, pp. 25–37.

[15] 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*, Oct 2007, pp. 55–62.

[16] S. Mittal and J. S. Vetter, “A survey of techniques for architecting dram caches,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1852–1863, June 2016.

[17] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, “C3d: Mitigating the numa bottleneck via coherent dram caches,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 36:1–36:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195681

[18] 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)*, Oct 2016, pp. 1–13.

[19] 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)*, Aug 2014, pp. 51–60.

[20] 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)*, March 2016, pp. 237–248.

[21] 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)*, June 2015, pp. 211–222.

[22] 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)*, June 2015, pp. 198–210.

[23] D. Jevdjic, S. Volos, and B. 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*, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485957

[24] 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*, Dec 2012, pp. 235–246.

[25] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-modal dram cache: Improving hit rate, hit latency and bandwidth," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 38–50.

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

[27] Intel, "Intel 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/en-us/articles/intel-sdm, 2018.

[28] M. T. Inc, "MIPS R10000 Microprocessor User's Manual," ftp://ftp.sgi.com/sgi/doc/R10000/User_Manual/t5.ver.2.0.book.pdf, 1996.

[29] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich, "Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor," *Digital Tech. J.*, vol. 7, no. 1, pp. 119–135, Jan. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=211554.211583

[30] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ser. ISCA '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 384–393.

[31] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Decoupled fused cache: Fusing a decoupled llc with a dram cache," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 65:1–65:23, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3293447

[32] 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)*, Feb 2015, pp. 126–136.

[33] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, Mar. 2003. [Online]. Available: http://doi.acm.org/10.1145/762471.762473

[34] 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*, Dec 2014, pp. 1–12.

[35] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "LLC-guided data migration in hybrid memory systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019.

[36] Intel, "Intel Knights Landing," https://ark.intel.com/content/www/us/en/ark/products/codename/48999/knights-landing.html.

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

[38] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining caching and data migration in hybrid memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA) 2020*, 2020.

[39] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, 2006, pp. 469–479.

[40] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *Int. Symposium on Computer Architecture (ISCA)*, 2008, pp. 453–464.

[41] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Fusioncache: Using llc tags for dram cache," in *Design, Automation Test in Europe (DATE)*, March 2018, pp. 593–596.

[42] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[44] 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*, Jan 2010, pp. 1–12.

[45] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[46] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[47] A. Phansalkar, A. Joshi, and L. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 412–423, 2007.

[48] 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*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[49] C. f. M. P. Seoul National University, "SNU NPB Suite," http://aces.snu.ac.kr/software/snu-npb/, 2015.

[50] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/605397.605403

[51] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 247–257.

[52] F. Hameed, L. Bauer, and J. Henkel, "Reducing latency in an sram/dram cache hierarchy via a novel tag-cache architecture," in *Design Automation Conference (DAC)*, 2014, pp. 37:1–37:6.

[53] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, July 2012.

[54] M. Chaudhuri, M. Agrawal, J. Gaur, and S. Subramoney, "Micro-sector cache: Improving space utilization in sectored dram caches," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 7:1–7:29, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3046680

[55] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124555

[56] JEDEC, "DDR4 Specification," https://www.jedec.org/standards-documents/docs/jesd79-4a, [Online].

[57] 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*, vol. 98, no. 12, pp. 2201–2227, Dec 2010.

[58] Micron, "NVDIMM," https://www.micron.com/products/dram-modules/nvdimm, [Online].

[59] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," *SIGARCH*

Comput. Archit. News, vol. 37, no. 3, pp. 371–382, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555801

[60] H. M. C. Consortium, "Hybrid Memory Cube Specification 2.1," http://hybridmemorycube.org/, [Online].

[61] 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)*, May 2017, pp. 1–4.

[62] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995911

[63] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[64] D. Knyaginin, V. Papaefstathiou, and P. Stenstrom, "Profess: A probabilistic hybrid main memory management framework for high performance and fairness," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 143–155.

[65] J. H. Ryoo, L. K. John, and A. Basu, "A case for granularity aware page migration," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, 2018, pp. 352–362. [Online]. Available: http://doi.acm.org/10.1145/3205289.3208064

[66] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[67] SNU, "SNU NPB Suite," http://aces.snu.ac.kr/software/snu-npb/.

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

[69] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, Sep. 2017.

[70] C. Chou, A. Jaleel, and M. Qureshi, "Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17. New York, NY, USA: ACM, 2017, pp. 268–280. [Online]. Available: http://doi.acm.org/10.1145/3132402.3132404

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

[72] 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)*, Feb 2018, pp. 271–282.