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

EVANGELOS VASILAKIS, Chalmers University of Technology, Sweden

VASSILIS PAPAESTATHIOU, Foundation for Research and Technology Hellas (FORTH), Greece

PEDRO TRANCOSO and IOANNIS SOURDIS, Chalmers University of Technology, Sweden

65

DRAM caches have shown excellent potential in capturing the spatial and temporal data locality of applications capitalizing on advances of 3D-stacking technology; however, they are still far from their ideal performance. 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. For instance, caching DRAM cache tags adds constant latency to every access; accessing the DRAM cache using the TLB calls for OS support and DRAM cachelines as large as a page; reusing the last-level cache (LLC) tags to access the DRAM cache limits LLC performance as it requires indexing the LLC using higher-order address bits. In this article, we introduce *Decoupled Fused Cache*, a DRAM cache design that alleviates the cost of tag accesses by fusing DRAM cache tags with the tags of the on-chip LLC without affecting LLC performance. In essence, the *Decoupled Fused Cache* relies in most cases on the LLC tag access to retrieve the required information for accessing the DRAM cache while avoiding additional overheads. Compared to current DRAM cache designs of the same cacheline size, *Decoupled Fused Cache* improves system performance by 6% on average and by 16% to 18% for large cacheline sizes. Finally, *Decoupled Fused Cache* reduces DRAM cache traffic by 18% and DRAM cache energy consumption by 7%.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Memory and dense storage**;

Additional Key Words and Phrases: Caches, 3D stacking, DRAM, processor, memory

ACM Reference format:

Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache. *ACM Trans. Archit. Code Optim.* 15, 4, Article 65 (January 2019), 23 pages.

<https://doi.org/10.1145/3293447>

1 INTRODUCTION

As 3D-stacking technology becomes more widely used, DRAM and a processor can be integrated into the same package, allowing higher memory bandwidth and substantially lower energy per transferred bit [2]. Compared to the narrower channels of traditional off-chip DRAM, 3D-stacked

This work is supported by the European Commission under the Horizon 2020 Program through the ECOSCALE project (grant agreement 671632) and by the European Research Council (ERC) under the MECCA project (Contract No. 340328). Authors' addresses: E. Vasilakis, P. Trancoso, and I. Sourdis, CSE dept., Chalmers University of Technology, Rännvägen 6, Gothenburg, Sweden; emails: {evavas, ppedro, sourdis}@chalmers.se; V. Papaefstathiou, Foundation of Research and Technology Hellas (FORTH), 100 Nikolaou Plastira str., Vassilika Vouton, Heraklion, Greece; email: papaef@ics.forth.gr. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM

1544-3566/2019/01-ART65

<https://doi.org/10.1145/3293447>

Fig. 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.

DRAM achieves 2 to 4 times higher bandwidth [22]. 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 (DCs) exploit the coarse-grained spatial locality of applications and reduce the number of accesses to main memory; however, they are still far from their ideal access latency [26]. This in turn may negatively impact system performance as it adds a 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 1, a simple 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) [33], which uses the LLC tags to access the DRAM cache, and *ATCache*, which is a DRAM cache with an on-chip tag cache (DCTC) [13], 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 fits all applications [19, 23, 28]. 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 cachelines offer better prefetching and overall better performance when the workloads exhibit spatial locality [16, 18]. 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 [4].

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 [13]. 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 [19]. 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 [6, 14, 15]. Other techniques such as *Alloy Cache* and *Compound Access Scheduling* collocate DC data and tags in the same DRAM row to allow faster accesses [21, 28]. These designs require either 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 article, 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 [33]. 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 this article 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 of and comparison against related approaches show that *Decoupled Fused Cache* achieves better performance and energy efficiency.

The remainder of this article is structured as follows: Section 2 uses a motivating example to introduce background information and highlight the challenges addressed in this work. In Section 3, the proposed *Decoupled Fused Cache* design is presented. Section 4 offers the evaluation and comparison of our work. Section 5 discusses in more detail related work on DRAM caches and in particular on designs that aim at reducing DC tag lookup latency. Finally, Section 6 summarizes our conclusions.

2 BACKGROUND AND MOTIVATION

Considering a system as the one illustrated in Figure 2 with an inclusive DC placed between the on-chip 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

Fig. 2. System overview.

conventional way and subsequently contrast it with our previous *Fusioncache* design, pointing to its advantages and drawbacks addressed by *Decoupled Fused Cache*.

A conventional system with a DC would function as illustrated in the example of Figure 3(a). 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 3(b) (Figure 5 shows the addresses and fields of the LLC and DC cachelines used for the examples in Figures 3, 4, and 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, in Figure 3(a), 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 that 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 3(a). 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 first, (parts of) the LLC tags are also stored in the DC tag array, and second, 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 whose 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 3(a). 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 4(a). 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 4(a) 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 4(b) and 5. As explained in our

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

(b) Conventional cache address breakdown

Fig. 3. Conventional cache example and address breakdown.

(a) Location of LLC-cachelines and tags in *Fusioncache* (FC) (b) *Fusioncache* address breakdownFig. 4. *Fusioncache* example and address breakdown. Fig. 5. Address fields for indexing and tag matching for conventional LLC, *Fusioncache* LLC, *Decoupled Fused Cache* LLC, and DRAM cache.

previous work, this design choice restricts the effective LLC associativity and limits performance for particular memory access patterns, especially in large DC cacheline sizes [33].

Taking a closer look at the *Fusioncache* example of Figure 4(a), 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 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 affect 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.

3 DECOUPLED FUSED CACHE DESIGN

The *Decoupled Fused Cache* 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 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 an equal number of sets and ways as the LLC data array.

¹That 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.

Fig. 6. Decoupled Fused Cache example and address breakdown.

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 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 6(a) illustrates the DFC functionality for the same example used in the previous section to demonstrate the FC and conventional cache. Figure 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 3(a), 4(a), and 6(a) 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 3(a). 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, we discuss the details of DFC, 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.

DFC Tag Arrays

DFC splits the tag array of the LLC into 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, and
- 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.

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 DC. These extra fields are shaded in Figures 7(a) and 7(b) for the *tag array* and the *BPA*, respectively. These fields are as follows:

Fig. 7. DFC tag array and BPA entries; shaded fields indicate additions for DFC.

- **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 one 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 4, 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 6(b)) 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.

DFC Indexing

Figure 6(b) shows the breakdown of an address and the bit fields used to index the tag array and the 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 in 64 different sets; for 128-byte DC cachelines it can reside in two different sets. As demonstrated by Figure 6(b), the *tag array set* is composed 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 6(a), 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 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 (1) save space in the tag array of the LLC to store additional information about the location of DC cachelines in the DRAM cache and (2) to access that information with a single lookup.

DFC Tag Matching

Figure 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 9 offers a flowchart showing the steps of a DFC access for every possible case: LLC hit (A), DC hit without tag lookup (B), DC hit with tag lookup (C), and DC miss (D). When a request is made to the LLC, particular parts of the address are used separately as shown in Figure 6(b). 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 8 and 9 ①).
- **Suffix match:** The *tag suffix* field of the set of the BPA is compared with the corresponding part of the address (*tag suffix*) for a match (Figures 8 and 9 ②).
- **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 8 and 9 ③). In case of a match at this stage, this is an LLC hit (A), otherwise an LLC miss (Figure 9 B, C, D).

The *tag match* ① and *suffix match* ② can be performed in parallel as they are independent of each other. The *way match* ③ 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 an LLC hit, as shown in Figure 9 A. 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:

- (1) In case the tag is located in the tag array (there was a match at the *tag match* ① stage) but there were no LLC cachelines pointing to that tag (*suffix match* ② or *way match* ③ failed), then the DC data array can be accessed directly (B) using the *DC way* field of the tag array entry that matched in the *tag match* ① 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.
- (2) In case the tag is not located in the LLC tag array (no match in the *tag match* ① stage), then the DC tag array stored in DRAM needs to be accessed (Figure 9 ④). Thereby, it

Fig. 8. DFC address breakdown, indexing, and tag matching process.

Fig. 9. Decoupled Fused Cache access flowchart.

is determined whether the DC cacheline is located in the DC (C) or there is a DC miss and the requested cacheline should be read from the main memory (D). 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 9 ⑤). 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 9 ⑥).

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 the 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 7(a)) and the DC tags are updated only when a tag is evicted from the DFC tag array.

Configurable DC Cacheline Size

As shown in our evaluation (Section 4), 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.

DFC Hardware Overhead

DFC reorganizes 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:** $\log_2 B$ bits for the DC way where the DC cacheline is located

²Larger 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 1. DFC SRAM Overhead

DC Cacheline Size	DC Cacheline/ LLC Cacheline Ratio	Added Bits per Entry	LLC Extra Cost %
128 bytes	2	15	224KB (2.5%)
256 bytes	4	16	240KB (2.9%)
512 bytes	8	17	256KB (3.1%)
1,024 bytes	16	18	272KB (3.3%)
2,048 bytes	32	19	288KB (3.5%)
4,096 bytes	64	20	320KB (3.7%)
Configurable (128 bytes–4KB)	2–64	20	320KB (3.7%)

- **count:** $\log_2 R + 1$ 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:** $\log_2 A$ LRU bits for tag replacement
- **Back Pointer Array:**
 - **Tag suffix:** $\log_2 R$ bits that identify the set in which the tag of the LLC cacheline is located
 - **Tag way:** $\log_2 A$ bits to identify the way in which the tag of the LLC cacheline is located in its set

The above listed fields account for a total of $2 \times \log_2 R + 2 \times \log_2 A + \log_2 B + 3$ bits per LLC cacheline. However, we can further reduce the cost by 1 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 $\log_2 R$ bits smaller than a conventional LLC tag. Thus, the additional space overhead of DFC is $\log_2 R + 2 \times \log_2 A + \log_2 B + 2$. 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. Let's consider a system with 48-bit physical addresses, 64-byte LLC cachelines, and a 16-way LLC with 8,192 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 8,192 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 an 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 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 an 8MB LLC, which accounts for a 3.7% area overhead.

The hardware overhead of DFC's 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 ① and ② in Figure 8 are faster than a traditional LLC tag lookup because the number of compared bits is smaller. Step ③, which adds to the latency of steps ① and ② 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* [11] after accounting for its logic latency in the same technology node.

4 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 multithreaded benchmarks for different DC cacheline sizes.

4.1 Experimental Setup

Our evaluation is performed using an in-house simulator based on *Pin* [24] following the interval-based simulation methodology [7] for the processor and cycle-accurate modeling of the cache and memory system. We simulate a four-core processor with private L1 and L2 caches, a shared on-chip LLC, and a DC. Table 2 presents the configuration of our system.³ We use *Cacti* v6.5 to determine the access times for the caches and tag arrays [11, 34]. For the main DRAM and 3D DRAM timing and energy consumption, we use the parameters provided by [19]. The DRAM energy parameters are shown in Table 3. To estimate the energy consumption of the processor cores, we use McPAT [20].

We evaluate our design with both single- and multithreaded workloads. For single-threaded workloads, we selected a representative subset of the SPEC2006 [10] benchmarks following the guidelines of Phansalkar et al. [27]. For multithreaded workloads, we used the *OpenMP* version of the *NAS Parallel Benchmark* suite [1, 29].

We simulate 1 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 *simplpoints* to select a representative slice of 1 billion instructions [31]. The benchmarks used and their memory footprint are shown in Table 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* [13]. The size of the DCTC SRAM tag cache is equal to the size of the SRAM overhead incurred by DFC.
- **Fusioncache (FC):** A system with Fusioncache [33].
- **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 5.

4.2 Performance

The performance improvement over the baseline of the 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

³All latencies reported in cycles concern processor clock cycles.

Table 2. System Configuration

Cores	4 cores, out-of-order, 4-way issue/commit 3.2GHz
L1 cache	Private, 64KB, 4-way, 1 cycle access latency
L2 cache	Private, 256KB, 8-way, 9 cycles access latency
L3 cache	Shared 8MB, 16-way, 14 cycles access latency
DRAM cache	512MB, 16-way, two 128-bit channels, 8ns tRCD, 10ns tCAS, 1.6GHz (DDR 3.2GHz)
Tag cache	256KB (272KB including its tag array), 8-way, 5 cycles access latency
Main DRAM	8GB, 2 channels, 64-bit bus, 14ns tRCD, 14ns tCAS, 800MHz (DDR 1.6GHz)

Table 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

Table 4. Benchmarks and Their Memory Footprint (in MB)

SPEC (Single Threaded)		NAS (Multithreaded)			
Benchmark	Memory	Benchmark	Memory	Benchmark	Memory
leslie3d	80	bt.C	720	lu.C	350
libquantum	32	cg.C	440	mg.C	2,550
mcf	1,315	dc.A	520	sp.C	770
omnetpp	146	ft.C	510	ua.C	360
lbm	402	is.C	1,100		

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 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 10 represents a different DC cacheline size (128 bytes to 4KB).

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%

Fig. 10. Performance improvement over the baseline of the DFC and related DC designs utilizing different DC cacheline sizes.

on average) the performance of a theoretical DC with zero tag access overheads and equal cacheline size.

Figure 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 [19]. Still, a Tagless DC design would introduce OS modifications and fix the DC cacheline size to the OS page size as explained in Section 5. At the horizontal axis of each plot the name of each benchmark can be found and in brackets the DC cacheline sizes at which each design achieves its best performance. For example, in Figure 11(a), *leslie3d[256-2048]* means that DC achieves its best performance using DC cachelines of 256 bytes, while DFC maximizes performance using 2,048-byte cachelines. These results verify our initial statement in Section 1 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 11(a) shows that DFC is a clear winner compared to DC; the only case where DC seems better than 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 11(b) shows the same trend when comparing DFC with DCTC, where the performance difference can be as high as 19.9% in favor of DFC and on average 10.3% and 6.8% for the SPEC and NAS benchmarks, respectively. Figure 11(c) 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 a 4KB cacheline as shown in Figure 11(d). This is because for some benchmarks, DC cacheline sizes other than 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 [19].

Figure 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 and the average across all DC cacheline sizes for the SPEC and NAS benchmarks are 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, 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 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 a consequence offers lower overall DRAM cache latency and energy cost.

Figure 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 further improves performance and in addition leads to lower energy consumption as explained below.

(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

Fig. 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).

Fig. 12. Percentage of accesses that do not require a DRAM cache tag lookup.

Fig. 13. Average normalized DRAM cache traffic.

4.3 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 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 proportional 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, 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 (1) the BPA in the DFC, (2) the tag cache in the DCTC, and (3) 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.

5 RELATED WORK

There are several existing 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 the 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 [28]. 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

Fig. 14. Energy consumption from 3D stacked DRAM cache (3D DRAM), main memory (DRAM), and cores normalized to baseline.

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 the 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 colocate 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 [21, 23]. Colocating 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 to 4KB). This causes considerable space waste for cacheline sizes bigger than 64 to 128 bytes. Colocating 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 [8, 17, 32].

ATCCache uses a small SRAM cache for the DC tags [13]. In case of a tag cache hit, the access latency for the tags is on 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 *Neighboring Tag Cache*, which buffers the tags of recently accessed adjacent cachelines as a means to reduce the DRAM cache traffic [4]. Hameed et al. proposed 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 [9]. Another tag cache technique is presented by Meza et al. for hybrid main memories composed of DRAM as a cache to non-volatile memories [25]. *ATCCache* 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 [3] uses a Decoupled-Sectorized [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 sectorized 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 [19]. The *tagless DRAM cache* design is effective but 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 OS support. Banshee [35] is another system that 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 multinode systems, storing a coherence directory on-chip would be prohibitively expensive. *CANDY* [5] repurposes the existing on-die coherence directory as a DC *coherence buffer* to cache recently accessed directory entries similar to how our design repurposes the LLC tag array. *C³D* [12] 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 noninclusive 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 deduplicating the LLC upper tags with the use of way-pointers [33]. *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 it 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.

6 CONCLUSIONS

In this article, the *Decoupled Fused Cache* (DFC) was introduced, 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 as 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 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% to 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 the NAS benchmarks compared to FC, and by one-third 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% compared to DCTC, and by 62% compared to the simple DC.

REFERENCES

- [1] 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. 1991. The NAS parallel benchmarks. *Int. J. Supercomp. Appl.* 5, 3 (1991), 63–73.
- [2] 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. 2006. Die stacking (3D) microarchitecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO 39'06)*. 469–479.
- [3] M. Chaudhuri, M. Agrawal, J. Gaur, and S. Subramoney. 2017. Micro-sector cache: Improving space utilization in sectored DRAM caches. *ACM Trans. Archit. Code Optim.* 14, 1 (March 2017), Article 7, 29 pages. DOI: <https://doi.org/10.1145/3046680>
- [4] C. Chou, A. Jaleel, and M. K. Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 198–210.
- [5] C. Chou, A. Jaleel, and M. K. Qureshi. 2016. CANDY: Enabling coherent DRAM caches for multi-node systems. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. DOI: <https://doi.org/10.1109/MICRO.2016.7783738>
- [6] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castolino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. 1995. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Tech. J.* 7, 1 (Jan. 1995), 119–135. <http://dl.acm.org/citation.cfm?id=211554.211583>
- [7] D. Genbrugge, S. Eyerman, and L. Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *International Symposium on High-Performance Computer Architecture (HPCA'10)*. 1–12.
- [8] N. Guler, M. Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-modal DRAM cache: Improving hit rate, hit latency and bandwidth. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 38–50.

- [9] F. Hameed, L. Bauer, and J. Henkel. 2014. Reducing latency in an SRAM/DRAM cache hierarchy via a novel tag-cache architecture. In *Design Automation Conference (DAC'14)*. Article 37, 6 pages.
- [10] J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.
- [11] HP-Labs. 2017. An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. Retrieved from <http://www.hpl.hp.com/research/cacti/>.
- [12] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan. 2016. C3D: Mitigating the NUMA bottleneck via coherent DRAM caches. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49'16)*. IEEE Press, Piscataway, NJ, Article 36, 12 pages. DOI: <http://dl.acm.org/citation.cfm?id=3195638.3195681>
- [13] C.-C. Huang and V. Nagarajan. 2014. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *International Conference on Parallel Architectures and Compilation (PACT'14)*. 51–60.
- [14] MIPS Technologies Inc. 1996. MIPS R10000 Microprocessor Users Manual. Retrieved from ftp://ftp.sgi.com/sgi/doc/R10000/User_Manual/t5.ver.2.0.book.pdf.
- [15] Intel. 2018. Intel 64 and IA-32 Architectures Software Developer Manuals. Retrieved from <https://software.intel.com/en-us/articles/intel-sdm>.
- [16] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee. 2016. Efficient footprint caching for tagless DRAM caches. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 237–248.
- [17] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 25–37.
- [18] D. Jevdjic, S. Volos, and B. Falsafi. 2013. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, 404–415.
- [19] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. 2015. A fully associative, tagless DRAM cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 211–222. <https://doi.org/10.1145/2749469.2750383>
- [20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42'09)*. 469–480.
- [21] G. Loh and M. D. Hill. 2012. Supporting very large DRAM caches with compound-access scheduling and MissMap. *IEEE Micro* 32, 3 (May 2012), 70–78.
- [22] G. H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *International Symposium on Computer Architecture (ISCA'08)*. 453–464.
- [23] G. H. Loh and M. D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-44'11)*. 454–464.
- [24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 190–200.
- [25] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. 2012. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Comput. Archit. Lett.* 11, 2 (July 2012), 61–64.
- [26] S. Mittal and J. S. Vetter. 2016. A survey of techniques for architecting DRAM caches. *IEEE Trans. Parallel Distrib. Syst.* 27, 6 (June 2016), 1852–1863. DOI: <https://doi.org/10.1109/TPDS.2015.2461155>
- [27] A. Phansalkar, A. Joshi, and L. K. John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Archit. News* 35, 2 (2007), 412–423.
- [28] M. K. Qureshi and G. H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-45'12)*. 235–246.
- [29] Center for Manycore Programming Seoul National University. 2015. SNU NPB Suite. Retrieved from <http://aces.snu.ac.kr/software/snu-npb/>.
- [30] A. Sez nec. 1994. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. IEEE Computer Society Press, Los Alamitos, CA, 384–393.
- [31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.* 36, 5 (2002), 45–57.
- [32] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi. 2012. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–257.
- [33] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis. 2018. FusionCache: Using LLC tags for DRAM cache. In *Design, Automation Test in Europe (DATE'18)*. 593–596.

- [34] S. J. E. Wilton and N. P. Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits* 31, 5 (5 1996), 677–688.
- [35] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas. 2017. Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*. ACM, New York, 1–14. DOI : <https://doi.org/10.1145/3123939.3124555>

Received June 2018; revised October 2018; accepted November 2018