Dirty-Block Tracking in a Direct-Mapped DRAM Cache with Self-Balancing Dispatch

DONGWOO LEE, Seoul National University

SANGHEON LEE and SOOJUNG RYU, Samsung Advanced Institute of Technology

KIYOUNG CHOI, Seoul National University

Recently, processors have begun integrating 3D stacked DRAMs with the cores on the same package, and there have been several approaches to effectively utilizing the on-package DRAMs as caches. This article presents an approach that combines the previous approaches in a synergistic way by devising a module called the dirty-block tracker to maintain the dirtiness of each block in a dirty region. The approach avoids unnecessary tag checking for a write operation if the corresponding block in the cache is not dirty. Our simulation results show that the proposed technique achieves a 10.3% performance improvement on average over the state-of-the-art DRAM cache technique.

CCS Concepts: • **Hardware** → **Dynamic memory**;

Additional Key Words and Phrases: DRAM Cache, 3D-stacked memory, memory bandwidth

ACM Reference Format:

Dongwoo Lee, Sangheon Lee, Soojung Ryu, and Kiyoun Choi. 2017. Dirty-block tracking in a direct-mapped DRAM cache with self-balancing dispatch. *ACM Trans. Archit. Code Optim.* 14, 2, Article 11 (May 2017), 25 pages.

DOI: <http://dx.doi.org/10.1145/3068460>

1. INTRODUCTION

A 3D stacked DRAM is an emerging technology that has a higher bandwidth compared to a conventional DRAM memory. Several standards of 3D stacked memory have already been proposed in the industry, such as High Bandwidth Memory (HBM) [JEDEC 2013], Wide-I/O [Kim et al. 2012], and Hybrid Memory Cube (HMC) [Jeddeloh and Keeth 2012]. The 3D stacked DRAM can be integrated in a package with a processor to provide a higher memory bandwidth to the processor. However, the area of the package restricts the capacity of the memory, and thus it has much smaller capacity than an off-package memory [Loh and Hill 2011].

Much research has been carried out to utilize the characteristics of 3D stacked DRAM. One such approach uses 3D stacked DRAM as a cache of an off-package memory to exploit the high bandwidth of the 3D stacked DRAM [Loh and Hill 2011; Qureshi and Loh 2012; Sim et al. 2012]. The main obstacle of this approach is the large size of tags for a DRAM cache. If the size of a DRAM cache with 64-byte blocks is 1GB, then its tag size is 96MB, which is impractical to be stored in an on-package SRAM. To solve

This work was supported by Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd.

Authors' addresses: D. Lee and K. Choi, Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826; emails: dongwoolee@dal.snu.ac.kr, kchoi@snu.ac.kr; Korea. S. Lee and S. Ryu, Samsung Advanced Institute of Technology, Suwon 17113, Korea; emails: sheon0.lee@samsung.com, soojung.ryu@samsung.com.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/05-ART11 \$15.00

DOI: <http://dx.doi.org/10.1145/3068460>

this problem, LH cache is proposed [Loh and Hill 2011], where the tags and the data of a DRAM cache are stored in a stacked DRAM. In this technique, the DRAM cache is organized as a set-associative cache and the tags and the data of a set are placed in the same row of the stacked DRAM. Because its miss penalty is huge, LH cache uses a miss map structure to reduce the overhead of the cache miss.

Qureshi and Loh [2012] address that a direct-mapped cache organization is more efficient for DRAM cache than a set-associative one and propose a technique called alloy cache. Alloy cache is organized as a direct-mapped cache, and it combines the tag and data (TAD) of a cache block as a single unit to read or write. The technique requires only one read operation to retrieve the TAD of a requested cache block from the stacked DRAM, which shortens the access latency of the DRAM cache. Moreover, alloy cache adopts an instruction-based hit-miss predictor to eliminate the latency overhead of a miss map used in LH cache.

Meanwhile, there is an approach to utilizing underused bandwidth of an off-package memory, which is named as mostly clean cache [Sim et al. 2012]. The technique, called self-balancing dispatch, can dispatch a read request to either an on-package 3D stacked DRAM or an off-package memory for clean data to balance the bandwidth utilization of both memories. To discriminate dirty data, the technique uses a dirty-region tracker that maintains the write-intensive address regions in page granularity and also adopts a region-based hit-miss predictor to remove the overhead of the miss map in LH cache. The idea is effective, but there is still room for further improvement since it is based on a set-associative cache, which is shown to be less effective than a direct-mapped cache.

In this article, we propose a DRAM cache technique that adopts the concept of self-balancing dispatch and that of direct-mapped cache at the same time to further improve the performance of a system. Based on an observation that directly combining the two techniques is inefficient due to the overhead of dirty-region tracking in a direct-mapped DARM cache, we devise a dirty-block tracker (DiBT) that maintains bit vectors to indicate the dirtiness of blocks in a dirty region. It enables filtering out unnecessary read operations for clean data on eviction of a victim from the tracker. It also enables eviction of any dirty region that has no dirty blocks. The DiBT can be used for both a set-associative and a direct-mapped DRAM cache using the self-balancing dispatch technique. To better exploit a direct-mapped cache organization, we design a DiBT that identifies a dirty region as a group of consecutive blocks in a row or in multiple rows of the on-package DRAM cache (near memory, or NM) instead of the off-package DRAM (far memory, or FM). The former is called DiBT based on the NM address (NA-DiBT) and the latter is called DiBT based on the FM address (FA-DiBT). The modification in NA-DiBT looks subtle but results in a big difference. It removes a read operation prior to a write operation (write request or block fill to the DRAM cache), which is necessary in FA-DiBT to check the dirtiness of the block to be evicted even if it is actually clean.

However, the optimization using NA-DiBT can cause a side effect on the accuracy of a hit-miss predictor (explained in Section 3.3) used in the technique because it does not read the DRAM cache if the block is clean when a cache miss is predicted, which can provide wrong feedback to the predictor. To solve this problem, we devise a sampling hit-miss predictor that periodically issues a read operation to DRAM cache when a cache miss is predicted, which effectively remedies the problem.

Following are our contributions in this article:

- We address an approach that combines self-balancing dispatch with direct-mapped DRAM cache, which has the potential to improve the performance of a system. We show that a naïve method of directly applying the self-balancing dispatch to a direct-mapped DRAM cache is inefficient due to the overhead of a dirty-region tracker.

Fig. 1. DRAM cache organization in a DRAM row: (a) set-associative cache and (b) direct-mapped cache.

- We propose FM-DiBT, which maintains dirty information of blocks in a dirty region. The technique can remove some unnecessary read operations of DRAM cache for an eviction of a dirty-region entry from the tracker at an overflow. Moreover, it can silently evict a dirty region that has no dirty blocks. FM-DiBT can be used for both direct-mapped and set-associative DRAM cache.
- We propose NA-DiBT, which tracks a dirty region as a group of blocks in a row or in multiple rows of the DRAM cache rather than the off-package DRAM. It removes unnecessary reads for dirty checking and thus saves a lot of bandwidth of a DRAM cache. NA-DiBT can be used only for direct-mapped DRAM cache.
- We devise a sampling hit-miss predictor, which effectively remedies the problem of biased prediction of NA-DiBT.

We evaluate our technique with 1GB DRAM cache. The architecture that implements the proposed technique consists of a direct-mapped DRAM cache, self-balancing dispatcher, and DiBT. FA-DiBT and NA-DiBT show average performance improvements of 6.0% and 10.3%, respectively, over the state-of-the-art direct-mapped DRAM cache technique.

2. BACKGROUND

Recently, die-stacked DRAM technology has emerged as a realistic solution to the “memory wall” problem [Loh 2008]. Several industrial standards have already been announced, such as High Bandwidth Memory (HBM) [JEDEC 2013], Hybrid Memory Cube (HMC) [Jeddeloh and Keeth 2012], and Wide I/O [Kim et al. 2012]. A die-stacked DRAM consists of multiple DRAM layers and numerous through-silicon vias (TSVs) that connect these layers. These new types of memory can be integrated with a processor in a package of 3D or 2.5D form factor. The full 3D integration puts a die-stacked memory on top of a processor and the stacked layers communicate directly with each other using the TSV technology. The 2.5D integration utilizes a silicon interposer technology and places a processor and a die-stacked memory side by side in a package. This on-package memory can provide much higher bandwidth to a processor compared to the conventional off-package memory. However, its size is restricted by the area of a package, so its capacity is much smaller than that of an off-package memory. Many researchers have proposed ideas for an efficient utilization of an on-package DRAM, and one of the main approaches is using it as a cache of an off-package memory, which is software transparent [Loh and Hill 2011; Qureshi and Loh 2012; Sim et al. 2012].

2.1. Loh-Hill DRAM Cache

The main problem of using an on-package DRAM as a cache of an off-package memory is the large size of cache tags. If the size of a DRAM cache with 64-byte blocks is 1GB, then its tag size is 96MB, which is impractical to be stored in an on-package SRAM. Loh and Hill [2011] solve this problem by storing tags in an on-package DRAM with data of the cache. Loh-Hill (LH) DRAM cache is a set-associative cache where tags and data of a set are placed on the same row as the on-package DRAM, as shown in Figure 1(a). The miscellaneous information of a block such as validness, dirtiness, and LRU is stored with its tag in the same DRAM row.

Loh-Hill DRAM cache uses an SRAM miss map structure to check whether requested data exist in the cache or not, so that it can identify cache hit/miss before accessing a DRAM cache. However, in the case of a cache hit, a read operation for tags of blocks in a set is necessary to determine the position of data in a DRAM row before processing the request. In the case of a cache miss, a memory read request is dispatched to the off-package memory and then returned data are passed to the requester and also used to fill the cache. Before this cache block-fill, a read of the DRAM cache is required to determine a victim block and to identify whether the victim block is dirty or not. If it is dirty, it should be written back to the off-package memory. This kind of read operation for a victim block is also essential for a memory write request.

2.2. Alloy Cache

Loh-Hill DRAM cache needs an additional read operation to a DRAM cache even in the cache-hit case, which is inefficient. Alloy cache addresses this problem and solves it by combining the tag and data (TAD) of a cache block as a single unit to read and write [Qureshi and Loh 2012]. Alloy cache is organized as a direct-mapped cache, as shown in Figure 1(b). It requires only one read operation to retrieve the tag and the data of a request cache block from the stacked DRAM. On cache hit of a read request, this saves one read operation compared to LH cache.

Alloy cache adopts an instruction-based hit-miss predictor to eliminate the latency overhead of the miss map in LH cache. In the case of a memory read request, if the prediction is cache hit, the request is dispatched only to an on-package DRAM cache. On the other hand, if the prediction is cache miss, the request is dispatched to both on- and off-package memory concurrently to reduce the access latency incurred by wrong prediction. It looks inefficient to dispatch the request to the on-package DRAM cache on a predicted miss but actually is not. One read of the cache is required on a miss anyway for write-back of the victim from the cache block-fill. In the case of a memory write request, alloy cache also requires one read of the on-package DRAM cache to see if write-back of a victim block is needed, which is similar to Loh-Hill cache.

Because alloy cache is a direct-mapped cache, its cache-hit rate is lower than that of a set-associative cache such as LH cache. However, since its hit latency is much smaller, the overall performance of alloy cache is better than that of LH cache.

2.3. Mostly Clean DRAM Cache

Although the bandwidth of an off-package memory is narrower than that of an on-package DRAM, it is in general not fully utilized when the on-package DRAM is used as a cache, since the cache generates traffic to the off-package memory only when there is a cache miss. Mostly clean DRAM cache addresses this issue of utilizing the underused bandwidth of the off-package memory [Sim et al. 2012]. The cache, which is based on a set-associative cache, dispatches a read request to an off-package memory (instead of an on-package DRAM cache) if the requested data are clean and the off-package memory bandwidth is underutilized. In addition, the cache uses a region-based hit-miss predictor to remove the overhead of the miss map in LH cache.

The self-balancing dispatch should be restricted to clean data for data consistency. For this reason, mostly clean DRAM cache utilizes a dirty-region tracker to track a write-intensive dirty region, as shown in Figure 2. The cache consists of two parts. One is a dirty list in a set-associative structure, which maintains dirty regions. If the dirty list contains a tag of a page, then the page is treated as a dirty region. The other part is a dirty-region detector consisting of a set of counting Bloom filter tables, which is indexed by different hash functions and the address of a page. If there is a write

Fig. 2. A dirty-region tracker used in mostly clean cache.

operation, the indexed counters increase their values, and if all the counters exceed a predefined threshold, then the detector classifies the page as a write-intensive region. If the classified write-intensive page does not exist in the dirty list, a victim page is selected and replaced with the new dirty page and the values of the counters are halved. The blocks in the victim page are read out from the DRAM cache and written back to the off-package memory if they are in the cache and are dirty.

Overall, mostly clean cache works as follows. In the case of a memory read request, the mostly clean cache first checks the dirty-region tracker to see if the requested data are included in a dirty region. If the data are in a dirty region, then the request is dispatched to the on-package DRAM cache. If it is identified as clean data, a region-based predictor predicts a hit/miss of the request and dispatches the request to an off-package memory if the prediction is cache miss. The data from the off-package DRAM is used to fill the DRAM cache. If it predicts a cache hit, the balancing dispatch scheme determines where to dispatch the memory request, on-package DRAM or off-package memory, by considering the current bandwidth utilization of both memories.

In the case of a memory write request, the mostly clean cache dispatches the write request only to the DRAM cache if the requested data exist in a dirty region (write-back cache) and dispatches the request to both memories if the requested data are not in a dirty region (write-through cache). Before processing the write to the cache, one read of the on-package DRAM cache is required to identify a hit or miss and, in the case of a miss, to identify the dirtiness of the victim block. If it is a miss and the victim is dirty, then the block should be written back to the off-package memory. Note that the dirtiness information is stored in the DRAM cache with tags in the mostly clean cache.

3. DIRECT-MAPPED DRAM CACHE WITH SELF-BALANCING DISPATCH

The previous approaches have strengths and weaknesses. Alloy cache shows that a direct-mapped organization is more suitable to a DRAM cache whose tags are stored in the DRAM. However, it has underutilized bandwidth of the off-package memory. On the other hand, mostly clean DRAM cache better exploits the bandwidth of the off-package memory, but it is based on a set-associative DRAM cache, which is inefficient compared to a direct-mapped cache. In this article, we devise new DRAM cache techniques that implement the self-balancing dispatch technique on a direct-mapped DRAM cache to improve the performance of a system. First, we introduce a naïve approach that

Fig. 3. A flow chart for a read request in a naïve approach.

directly applies the self-balancing dispatch scheme to a direct-mapped cache and show the problems of this approach.

3.1. A Naïve Approach

A naïve approach to combining the previous two techniques is directly adopting the self-balancing dispatch technique used in a mostly clean cache to an alloy cache. It is based on a direct-mapped alloy DRAM cache, and thus the basic unit to read and write is an alloy of TAD of a block. Blocks are placed in the DRAM cache such that the blocks in a row have consecutive cache indices to improve the row buffer hit rate. It also uses the instruction-based hit-miss predictor (i.e., MAP-I [Qureshi and Loh 2012]) used in the baseline alloy cache. A dirty-region tracker used in a mostly clean cache is attached to the baseline without a modification.

The operation of the naïve cache is similar to that of a mostly clean cache. Figure 3 shows a flow chart for a read request in the naïve approach. The shaded part comes from an alloy cache, and the other part comes from a mostly clean cache. In the case of a read request, the naïve cache first searches the dirty-region tracker to see if the request is in a dirty region. If it is, the naïve cache dispatches the request to an on-package DRAM cache like a mostly clean cache. If it is in a clean region, the instruction-based hit-miss predictor used in an alloy cache makes a prediction. If the prediction is cache miss, the cache dispatches the request to both an on-package DRAM cache and an off-package memory like an alloy cache, which reduces latency when the prediction is wrong. The data from the off-package DRAM are used to fill the DRAM cache. If the prediction is cache hit, then the self-balancing dispatch technique probes the number of requests in bank queues in the on-/off-package memories and estimates the latency for the request to be processed. If the estimated latency of the DRAM cache is shorter than that of the off-package memory, the approach sends the request to the cache. Otherwise, it sends the request to the off-package memory. This function of balancing dispatch is exactly the same as the original mostly clean cache.

Processing a write request in the naïve approach is also similar to the mostly clean cache. Figure 4 shows a flow chart for a write request in the naïve approach. The naïve cache uses the write-back policy for a request to a dirty region and the write-through policy for a request to a clean region. Before processing a write to the cache, one read

Fig. 4. A flow chart for a write request in a naïve approach.

of the on-package DRAM cache is always required to check for hit/miss and dirtiness of the victim block in case of a miss.

3.1.1. Problems of the Naïve Approach. The dirty-region tracker proposed in the mostly clean cache maintains a dirty region as a page granularity. Therefore, in the case of a dirty-region eviction, the tracker should write back all dirty blocks that are included in the evicted dirty region from the DRAM cache to the off-package memory. To check for the dirtiness of these blocks, the tracker should read the dirty bits of all cache blocks in the dirty region, incurring a lot of overhead. For example, assuming a 4KB page with block size of 64B, checking for dirtiness requires 64 reads for one eviction of a dirty region.

Such an overhead can be mitigated in a set-associative DRAM cache, such as mostly clean cache. This is because the blocks in a page are placed in different cache sets and these sets are interleaved in multiple DRAM rows in different channels, ranks, and banks in a set-associative DRAM cache, and as a result, the reads for a dirty page eviction are distributed over multiple DRAM banks [Sim et al. 2012]. On the other hand, the situation is different in a direct-mapped DRAM cache in row interleaving, where consecutive cache blocks are placed in a same DRAM row. As a result, the reads for a dirty-region eviction are concentrated to a few DRAM banks, and thus, they interfere with demand requests to these banks, resulting in a performance degradation of the system, even though these reads mostly hit in the row buffer.

In order to provide quantitative evidence to our argument, Figure 5 shows the average performance overhead of the dirty-region tracker in alloy caches under different address mappings of the DRAM cache over 11 benchmarks (refer to Section 4.2). The difference between “Alloy” and “Alloy with Dirty-Region Tracker” in each bar (i.e., -6.1% and -4.9%) indicates the performance overhead of the dirty-region tracker.¹ In this experiment, “Row Interleaving” maps 28 consecutive DRAM cache sets to a single on-package DRAM row (which represents the original alloy cache), while “Block Interleaving” interleaves consecutive DRAM cache sets into different on-package DRAM rows in different channels, ranks, or banks (which mimics the address mapping of set-associative DRAM caches).

¹Note that, to show a pure overhead of the dirty-region tracker, we do not apply self-balancing dispatch in these evaluations.

Fig. 5. Performance overhead of the dirty-region tracker in alloy caches under different address mappings of the DRAM cache. The results are normalized to those of the alloy cache with Row Interleaving (Row Interleaving: 28 consecutive sets are stored in an on-package DRAM row, Block Interleaving: consecutive sets are interleaved into different on-package DRAM rows).

Fig. 6. Breakdown of read operations to the DRAM cache in the naive approach.

From the results, we can draw two conclusions. First, row interleaving performs much better than block interleaving in the alloy cache. This is expected because row interleaving optimizes the row buffer locality of sequential access. Second, the performance overhead of introducing the dirty-region tracker is actually higher in row interleaving than in block interleaving (i.e., 6.1% in row interleaving vs. 4.9% in block interleaving). This is because of concentrated read traffic caused by dirty-region evictions, as explained previously. This motivates the need for a new mechanism that tracks dirty pages in *direct-mapped* DRAM caches.

Figure 6 shows a breakdown of read operations to the DRAM cache in the naive approach. There are four types of read operations sent to the DRAM cache:

- Type 1: read request from a processor (i.e., DRAM read caused by a cache miss in an on-chip cache)
- Type 2: read operation prior to a write request for hit/miss/dirtiness check
- Type 3: read for a dirty-region eviction when the block is actually dirty
- Type 4: read for a dirty-region eviction when the block is miss or clean

The last case (Type 4) is redundant and consumes the bandwidth meaninglessly. Some applications like GemsFDTD and omnetpp in Figure 6 have a relatively large portion of this type of read. Such read operations can be avoided if the dirtiness of a block in a DRAM cache can be identified without reading the tag of the block. Other types of read also have a chance to be avoided. For example, in the case of Type 1, if the read request is predicted as a cache miss, the request is dispatched to the off-package

Fig. 7. The structure of FA-DiBT.

memory. The read request is also sent to the on-package DRAM cache to see if it is really a miss. If it is, then the victim block is written back to the off-package memory if it is dirty. Thus, the read is redundant if the block is clean. In the case of Type 2, the read operation is also redundant if the block existing in the cache is clean.

3.2. Dirty-Block Tracker (DiBT)

This section explains solutions to the problems explained in the previous subsection. First, we present a dirty-block tracker based on far-memory address, called FA-DiBT, to relieve the overhead of the dirty-region tracker used in the naïve approach. Then we present an improved version of DiBT based on near-memory address, called NA-DiBT, to better exploit the direct-mapped organization of the DRAM cache, which eliminates redundant reads of the DRAM cache. NA-DiBT has a biased-miss prediction problem, and as a solution, we present a sampling hit-miss predictor.

3.2.1. Dirty-Block Tracker Based on Far-Memory Address (FA-DiBT). A dirty-region tracker proposed in mostly clean cache maintains a dirty region as a page granularity, and if there is a new write-intensive dirty region, it may evict a victim dirty page. The eviction of a victim region requires reads of a cache as many as the number of blocks in the page to be evicted. To reduce the number of read operations, we propose a DiBT, which detects a dirty region as a page granularity, but it additionally maintains the dirtiness of each block in the page. Therefore, the approach using DiBT does not need to maintain dirty bits inside the DRAM cache. The structure of DiBT is shown in Figure 7. The cache consists of two parts: dirty-region detector and dirty-block table. The dirty-region detector consists of a set of counting Bloom filter tables, which are indexed by different hash functions of a page address. The dirty-block table, having a structure similar to a set-associative cache, maintains dirty regions as well as the dirtiness of the blocks in those regions. Each dirty region has a set of blocks in a consecutive address space in the off-package memory, and that is why the proposed DiBT is called DiBT based on far-memory address. The baseline is similar to the dirty-region tracker used in a mostly clean cache. The difference is in the rows of dirty bits attached next to page tags in the dirty-block table. The dirty bits represent the dirtiness of each block in the dirty page. The size of a row is the same as the number of blocks in a page.

In the naïve approach, the dirtiness of a region can be identified with the dirty-region tracker, but the dirtiness of a cache block in the region can be identified only by the

Fig. 8. An operation flow for a write request in the technique using FA-DiBT: (a) write-back if the request is made to a dirty region but it is clean; (b) write-through if the request is made to a nondirty region.

dirty bit stored in the DRAM cache. However, in the FA-DiBT, the dirtiness of a cache block can be identified by looking up the dirty-block table at the bit position of the block.

Figure 8 shows the operation of FA-DiBT for a write request. After receiving a write request ①, the dirty-block tracker searches the dirty-block table to see if the request is to a dirty region and if the block is marked as dirty in the dirty-block tracker ②.

If it is to a dirty region (i.e., the page address is in the table) and the target block is marked as dirty in the dirty-block tracker, then the tracker simply sends the request directly to the on-package memory because the request is guaranteed to hit in the cache.

On the other hand, if the request is to a dirty region but the target block is not dirty, then the tracker takes the write-back policy, as shown in Figure 8(a). It first reads the tag of the block in the cache to check for a hit or a miss ③. In case of a hit, the tracker writes the block and sets the dirty bit ④. In case of a miss (the existing block is a victim), the tag is used to look up the dirty-block table to see if the victim is dirty (it is in the dirty region and the dirty bit is set) ⑤.² If it is dirty, the tracker writes

²Even though the tag is already read from the DRAM cache, we still have to look up the dirty-block table because the dirtiness information of DRAM cache blocks is stored only in the dirty-block table and DRAM cache tags do not contain such information (see the beginning of Section 3.2.1).

Fig. 9. An operation flow for an eviction of a victim dirty region in a dirty-block tracker (DiBT).

the victim block to the off-package memory ⑥ and then writes the new block into the cache ④. Otherwise (the victim is clean), it just writes the new block into the cache.

If the request is not to a dirty region, the technique takes the write-through policy by sending the request to both memories ③-1, ③-2, as shown in Figure 8(b). The write request sent to the on-package DRAM cache ③-1 goes through the same process as the write-back policy, as shown in Figure 8(a).

If the region of a request is not listed in the tracker but is identified as a new write-intensive region, then the tracker finds a victim region and replaces it with the new one. Figure 9 shows how a victim region is evicted. Once a region is newly identified as write intensive by the counting Bloom filter tables ①, the tracker finds a victim among the dirty regions in the dirty-block table ②, clears all blocks in the victim, and replaces it with the new write-intensive region. To clear the victim, read operations of the DRAM cache are necessary to evict real dirty blocks in the DRAM cache ③. Because the dirty-block tracker maintains dirtiness of the blocks in the victim, it is possible to send a read operation only to real dirty blocks in the cache, which eliminates unnecessary reads of clean blocks. Note that we do not need to access the DRAM cache just for modifying a dirty bit because the dirty information is maintained only in the dirty-block tracker.

The technique of FA-DiBT allows a silent eviction of an empty dirty region, which improves the efficiency in maintaining the dirty-block table in the tracker. For example, if there is a cache miss during a read operation, a victim block is written back to an off-package memory if it is dirty, resulting in a decrease of dirty blocks in a region. Such an eviction of dirty blocks from a dirty region can make the region empty of dirty blocks. In that case, the dirty region can be evicted silently from the dirty-block table. It makes an empty space in the table, making it easy to add a new dirty region.

3.2.2. Dirty-Block Tracker Based on Near-Memory Address (NA-DiBT). In this section, we present another dirty-block tracker called dirty-block tracker based on near-memory address, where each dirty region consists of a set of consecutive blocks in the on-package memory (DRAM cache). It still inherits features of DiBT, but more efficiently utilizes the direct-mapped cache organization.

The dirty-block tracker presented in the previous subsection tracks a dirty region as a page granularity based on the assumption that write operations are concentrated to a consecutive address space or a page (dirty region). A direct-mapped cache can track such a dirty region as a group of consecutive cache blocks (block group) placed closely

Fig. 10. An example of data mapping to a direct-mapped cache.

in a few DRAM cache rows because it caches data in a page to consecutive blocks.³ Thus, a dirty region can also be well represented by a block group in the on-package DRAM cache, instead of a consecutive address space (page) in the off-package memory.

Figure 10 shows an example of data mapping to a direct-mapped cache. For the purpose of explanation, let's assume that a decimal address is used and the size of a page is 10. A block group can have an arbitrary size, although it is set to 10 in our example. Data blocks of address 111 and 219 in the off-package DRAM are cached to location 11 and 19 in the DRAM cache, but their values are modified in the cache. In the case of FA-DiBT, if the tracker detects page 11 as a dirty region, it stores page number 11 as a page tag and sets only the dirty bit corresponding to the data block of address 111. On the other hand, NA-DiBT detects a dirty region as a block group, and thus it detects block group 1 as a dirty region. In this case, the tracker sets dirty bits corresponding to cache location 11 and 19.

Figure 11 shows the structure of NA-DiBT. The changes from FA-DiBT are as follows. First, the counting Bloom filter tables for detecting a write-intensive region are indexed by a block group address instead of a page address. Second, the dirty-block table in the tracker uses the address of a block group for tag checking and stores dirtiness of blocks in a block group instead of a page. These modifications are subtle but enable the technique to remove unnecessary read operations for dirty checking prior to write operations.

Figure 12 shows an operation flow of NA-DiBT for a write request to a clean block. If there is a write request ①, the tracker searches the dirty-block table to see if it is to a dirty-region ②. It takes write-back for a request to a dirty region and a clean block, as shown in Figure 12(a), and write-through for a request to a clean region, as shown in Figure 12(b), which is the same as FA-DiBT. However, unlike FA-DiBT, which always requires a read operation prior to a write request (the read operation is to check the tag to see if it is a hit or a miss; if it is a miss, the tag is used as an index to the dirty-block table for checking the dirtiness of the block existing in the cache), NA-DiBT can

³This is not true for set-associative cache, where neighboring blocks in a set are in general from different pages.

Fig. 11. The structure of NA-DiBT.

Fig. 12. An operation flow of NA-DiBT for a write request to a clean block: (a) write-back if the request is made to a dirty region but it is clean; (b) write-through if the request is made to a nondirty region.

Fig. 13. A flow chart for a read request in NA-DiBT.

eliminate the read operation if the region or block is clean. This is possible since it can identify dirtiness of a block by looking up the dirty-block table with the block address. It does not recognize whether it is a cache hit or not, but it does not matter since a clean block can be overwritten without writing it back to the off-package memory regardless of a hit or a miss.

A block-fill operation in a cache read miss is similar to a write request. In FA-DiBT, a read of the cache prior to the block fill is required to check the dirtiness of the existing block in the cache. For example, FA-DiBT issues parallel reads to both on- and off-package memory if the prediction of a read request is cache miss, and the read to an on-package DRAM is used for dirty checking of the existing block in the cache. However, NA-DiBT does not issue a read to the cache when the prediction is cache read miss, as shown in Figure 13. Instead, it checks the dirtiness of a block by looking up the dirty-block table. If it is found to be clean, the data retrieved from an off-package memory can be used to fill the on-package DRAM cache safely. This optimization can reduce the bandwidth usage of the on-package DRAM cache as much as the number of predicted cache misses when compared to FA-DiBT.

3.3. Sampling Hit-Miss Predictor

The optimization using NA-DiBT can have an undesirable effect on the accuracy of the hit-miss predictor when the predictor predicts a cache miss. Most of the previously proposed hit-miss predictors are based on a feedback mechanism; there is a positive feedback for a correct prediction and a negative feedback for a wrong prediction. The correctness of the prediction is checked when the cache is actually read. FA-DiBT has no problem implementing such a predictor because it issues parallel reads to both on- and off-package DRAM on a predicted cache miss. However, NA-DiBT issues a read operation only to the off-package memory on a predicted miss case (i.e., the cache is not read and there is no chance to correct wrong miss predictions), resulting in a biased prediction. To solve this problem, we devise a sampling hit-miss predictor that periodically issues a cache read when the prediction is cache miss.

The sampling hit-miss predictor has the same hardware structure as the MAP-I predictor [Sim et al. 2012] used in the naïve approach. It consists of multiple bounded counter tables, one table for each processor core. The table is indexed by the address of each instruction that causes a memory request. In each indexed entry of the table, a bounded counter is used for a hit-miss prediction.

The key difference between MAP-I and our SP is on its counter update mechanism. Given a request, if the value of a counter is less than a predefined threshold value V_{TH} ,

Table I. System Configuration for Simulation

Processor	
Core	Out-of-order, 8 cores, 3GHz, 4-issue, 256 ROB
L1 I/D caches	4-way, 32KB I/D cache per core, 2-cycle
L2 cache	16-way, 8MB shared cache, 4-bank, 20-cycle
Off-package DRAM (16GB)	
Bus frequency	800MHz (DDR 1.6GHz)
Memory controller	2 memory controllers, 32-entry request queue per MC
Channel, rank, and bank	1 channel per MC, 2 ranks per channel, 8 banks per rank
Row buffer size	8KB
Bus width	64 bits per channel
tCAS-tRCD-tRP-tRAS-tFAW	14ns-14ns-14ns-35ns-30ns [Son et al. 2014]
Memory scheduling	PAR-BS [Mutlu and Moscibroda 2008]
On-package DRAM (1GB)	
Bus frequency	1.6GHz (DDR 3.2GHz)
Memory controller	4 memory controllers, 32-entry request queue per MC
Channel, rank, and bank	1 channel per MC, 1 rank per channel, 8 banks per rank
Row buffer size	2KB
Bus width	128 bits per channel
tCAS-tRCD-tRP-tRAS-tFAW	14ns-14ns-14ns-35ns-30ns [Son et al. 2014]
Memory scheduling	PAR-BS [Mutlu and Moscibroda 2008]

then the predictor predicts a cache hit and the request is sent to the cache. If it is a real cache hit, the counter decreases its value, and if there is a cache miss, the counter increases its value (in this mode, the counter is used for hit-miss prediction).

On the other hand, if the value of a counter is greater than or equal to V_{TH} , then the predictor predicts a cache miss. In this case, the request is sent to the off-package memory and the counter increases its value (in this mode, the counter is used to implement the sampling period). If the value of the counter hits the preset upper bound V_{UPPER} , then the predictor issues a read to the on-package DRAM cache to check the correctness of the prediction. If it is actually a cache hit, then the counter sets its value to $V_{TH} - 1$, so that the next prediction for the same index can be a cache hit. If it is indeed a cache miss, then the counter sets its value to V_{TH} , so that the predictor can continuously predict a cache miss for the same index. The proposed hit-miss predictor effectively remedies the biased prediction.

4. EVALUATION METHODOLOGY

4.1. Experimental Setup

We evaluate the performance of the proposed technique using the McSimA+ simulator [Ahn et al. 2013]. The details of the architectural configuration are summarized in Table I. The system has a 3GHz out-of-order eight-core processor, private 32KB four-way L1 instruction and data caches per core, and a shared 8MB 16-way L2 cache. The timing parameters of caches are calculated by using CACTI 6.5 [Muralimanohar et al. 2009]. The cache block size of all caches, including the DRAM cache, is 64 bytes. A 3D-stacked 1GB DRAM is integrated with the processor in a package. The DRAM is used as a cache of a 16GB off-package DRAM. The timing parameters of the on- and off-package DRAMs are adopted from the work in Son et al. [2014], where a modified version of CACTI-3DD [Chen et al. 2012] is used. The 16GB off-package DRAM consists of two channels, two ranks per channel, and eight banks per rank. Two memory controllers and an 800MHz 64-bit-wide bus are used for the off-package DRAM. The 1GB on-package DRAM consists of four channels, one rank per channel, and eight banks per rank. Four memory controllers and a 1.6GHz 128-bit-wide bus are used for the on-package DRAM. Each on-/off-package memory controller has a 32-entry request queue.

Table II. L2 Miss per 1,000 Instructions (MPKI)

Benchmark	L2 MPKI
mcf	59.61
soplex	36.28
libquantum	33.23
lbm	32.22
GemsFDTD	28.83
leslie3d	27.45
bwaves	22.86
omnetpp	20.82
mile	20.11
sphinx3	11.90
cactusADM	8.47

The memory controllers use parallelism-aware batch scheduling (PAR-BS) [Mutlu and Moscibroda 2008] as a scheduling algorithm and prioritize demand requests over the reads for dirty-region evictions within each PAR-BS batch. The on-package DRAM has $2\times$ more memory controllers and a $2\times$ faster, $2\times$ wider bus compared to the off-package DRAM, so that it can provide $8\times$ larger bandwidth.

4.2. Workloads

We use SPEC CPU 2006 benchmarks with reference input for the evaluation [Henning 2006]. We filter out benchmarks that have low access to memory by considering L2 MPKI and use 11 benchmarks as shown in Table II. We find a region of interest for each benchmark by using a SimPoint [Hamerly et al. 2005] and get a trace of 400 million instructions of each benchmark. We evaluate our technique by running eight copies of each benchmark on an eight-core processor in rate mode. Thus, the total number of instructions used is 3.2 billion, which is large enough to evaluate unified DRAM cache without a separate warm-up process.

5. RESULTS

We evaluate four techniques: the naïve approach that directly combines alloy cache and mostly clean cache, FA-DiBT, NA-DiBT, and NA-DiBT with a sampling hit-miss predictor (SP). We set alloy cache as a reference technique [Qureshi and Loh 2012] and use instruction per cycle (IPC) as a performance metric for the evaluation.

5.1. Performance

Figure 14 shows IPC values normalized to that of alloy cache. The naïve approach, FA-DiBT, NA-DiBT, and NA-DiBT with SP respectively show 2.4%, 6.0%, 7.2%, and 10.3% performance improvements on average over the reference technique. In summary, each legend item represents the following optimizations:

- **Naïve** represents the alloy cache with self-balancing dispatch. The performance difference of “Naïve” over the baseline shows the impact of using self-balancing dispatch with the alloy cache.
- **with FA-DiBT** represents “Naïve” with FA-DiBT. The performance improvement of “with FA-DiBT” over “Naïve” shows the benefit from eliminating unnecessary read operations on dirty-region evictions by adding per-block dirty bits for each entry.
- **with NA-DiBT** represents “Naïve” with NA-DiBT. Since NA-DiBT provides most of the benefits of FA-DiBT, comparing “with NA-DiBT” against “with FA-DiBT” shows the performance improvement achieved by removing a read operation prior to a write

Fig. 14. Speedup over the reference for the naïve approach, FA-DiBT, NA-DiBT, and NA-DiBT with a sampling hit-miss predictor (SP) (left to right).

operation (which was necessary in “with FA-DiBT” to see if the victim cache block is dirty).

- **with NA-DiBT+SP** represents “with NA-DiBT” combined with a sampling hit-miss predictor. This intuitively represents the performance benefit of using a sampling hit-miss predictor with NA-DiBT.

In the case of the naïve approach, some benchmarks such as sphinx3 and leslie3d show a significant performance improvement over the reference technique due to the self-balancing dispatch technique, but some benchmarks such as GemsFDTD and lbm show a performance degradation over the reference technique. As a result, the overall performance improvement of the naïve approach over the reference technique is not that significant. In the case of FA-DiBT, some benchmarks such as omnetpp and mcf show a drastic performance improvement, and the other benchmarks show a modest performance improvement over the naïve approach. When comparing NA-DiBT with FA-DiBT, some benchmarks such as lbm and bwaves show a distinct performance improvement, but some benchmarks such as omnetpp and sphinx3 show a performance degradation. As a result, the overall performance improvement of NA-DiBT over FA-DiBT is insignificant. The performance degradation of NA-DiBT due to the predictor problem for some benchmarks such as libquantum and sphinx3 disappears in NA-DiBT with SP. As a result, NA-DiBT with SP shows a significant performance improvement over the reference technique. When comparing NA-DiBT with and without SP, two benchmarks—cactusADM and lbm—show minor performance degradation with SP. For these applications, hit-miss prediction without sampling is already very accurate, so the sampling read itself only results in performance degradation.

5.2. Analysis

Figure 15 shows a breakdown of read operations to the DRAM cache by the cause in the four approaches: naïve approach, FA-DiBT, NA-DiBT, and NA-DiBT with SP. The number of read operations is normalized to that of the naïve approach. As mentioned in Section 3.1.1, there are four types of read operations to the DRAM cache:

- Type 1: read request from a processor
- Type 2: read operation prior to a write request for hit/miss/dirtiness check

Fig. 15. Breakdown of read operations to the DRAM cache by the cause in the naïve approach, FA-DiBT, NA-DiBT, and NA-DiBT with SP (left to right).

- Type 3: read for a dirty-region eviction when the block is actually dirty
- Type 4: read for a dirty-region eviction when the block is miss or clean

For some benchmarks such as *omnetpp*, the naïve approach has a large portion of Type 4 reads. However, the read operations of Type 4 are removed in FA-DiBT. This removal shows the effect of using DiBT, which generates reads only to dirty blocks in the cache for a dirty-region eviction. Comparing NA-DiBT with FA-DiBT, most benchmarks show reductions of Type 1 and Type 2 reads. The reason for the reduction of Type 1 is the elimination of a read operation of the cache in the case of a predicted miss. The reason for the reduction of Type 2 is the elimination of a read operation prior to a write request to a clean block in the cache. This result shows the effect of optimizations using NA-DiBT, but it also includes the side effect of biased prediction. Comparing NA-DiBT with and without SP, benchmarks such as *leslie3d* and *libquantum* show a growth of Type 1 reads. This result shows the effect of the sampling hit-miss predictor that effectively remedies the problem of biased prediction, increasing the number of hit predictions and thus increasing the number of read operations to the cache. It also includes the overhead of periodic sampling reads of the cache.

Figure 16 shows a breakdown of read requests from L2 cache in the four different approaches by decisions of three components: dirty-region tracker, hit-miss predictor, and self-balancing dispatcher. There are four cases for these read requests from L2 cache by the decisions. The dirty-region tracker first checks to see whether the request is to a dirty region or not. If it is to a dirty region (naïve approach) or the requested block is dirty (other approaches), the request is guided to the on-package DRAM cache (Case 1, “Dirty region” in the figure). Otherwise, the hit-miss predictor makes a decision on hit or miss. If the prediction is cache miss (Case 2, “Predicted miss” in the figure), then the request is sent to both on- and off-package DRAM in the naïve approach and FA-DiBT. It is sent only to the off-package DRAM in NA-DiBT. If the request is predicted as a cache hit, then the self-balancing dispatcher decides the direction of the request, to the DRAM cache (Case 3, “SBD (to Cache)” in the figure) or to the off-package DRAM (Case 4, “SBD (to DRAM)” in the figure).

Compared to the naïve approach, FA-DiBT shows a reduced portion of Case 1 read for most benchmarks, because the decision of FA-DiBT is made at the cache block granularity. Actually, the reduced portion is shifted to the hit-miss predictor. Comparing NA-DiBT to FA-DiBT, some benchmarks such as *cactusADM* and *lbm* show an increase

Fig. 16. Breakdown of read requests from L2 cache by decisions of three components, dirty-region tracker, hit-miss predictor, and self-balancing dispatcher, in the naïve approach, FA-DiBT, NA-DiBT, and NA-DiBT with SP (left to right).

of Case 1 read. This is because NA-DiBT maintains the dirtiness of a cache location instead of an actual data block. If a dirty block, which is not the requested one but mapped to the same cache location, exists in the cache, then a Case 1 read request will be sent to the DRAM cache since the dirty-block tracker does not know that it will be a miss.

Some benchmarks such as libquantum and omnetpp show an increasing portion of Case 2, because of the problem of the biased prediction. It shrinks the portion for the self-balancing dispatch to determine (Case 3 and Case 4). NA-DiBT with SP reduces the portion of Case 2 read, showing that the predictor effectively remedies the biased prediction problem.

Figure 17 shows a breakdown of write requests from L2 cache (i.e., write-back from L2 to DRAM) by a decision of the dirty-region tracker in the four different approaches. There are three cases for these write requests. If the dirty-region tracker finds a write request going to a dirty region that is maintained in the dirty-block table in the tracker, the request is guided to the DRAM cache (Case 1, “Dirty region” in the figure). If the tracker detects that the request is to a write-intensive region and thus identifies it as a dirty region, then the request is guided to a DRAM cache (Case 2, “New dirty region” in the figure). If the tracker finds that the request does not go to a dirty region, then the write request is guided to both on- and off-package DRAMs to guarantee a clean block in the cache (Case 3, “Write-through” in the figure).

In the comparison of NA-DiBT and FA-DiBT, the breakdown for the two is similar to each other. It indicates that NA-DiBT tracking a dirty region as a group of cache blocks identifies write-intensive regions as well as FA-DiBT tracking a dirty region as a page.

5.3. Prediction Accuracy

Table III shows cache hit/miss prediction accuracy for the four different approaches. There are four cases: true hit, false hit, true miss, and false miss. For example, if the prediction is cache hit and it is correct, then it is a true hit.

The overall accuracy of FA-DiBT is similar to that of the naïve approach. However, the overall accuracy of NA-DiBT noticeably decreases because of the biased prediction problem. It is shown that the prediction is biased toward miss. NA-DiBT with SP

Fig. 17. Breakdown of write requests from L2 cache by a decision of a dirty-region tracker in the naïve approach, the cache with FA-DiBT, the cache with NA-DiBT, and the cache using NA-DiBT with the sampling hit-miss predictor (left to right).

Table III. Prediction Accuracy for Different Techniques

Prediction	Cache Hit		Cache Miss		Overall Accuracy
	Correct	Wrong	Correct	Wrong	
Naïve approach	40.1%	3.5%	52.4%	3.9%	92.5%
Cache with FA-DiBT	40.0%	4.2%	51.8%	4.0%	91.8%
Cache with NA-DiBT	30.6%	8.8%	44.7%	16.0%	75.3%
Cache with NA-DiBT + SP ^a	40.7%	3.7%	49.8%	5.8%	90.5%

^aSP: sampling hit-miss predictor.

Table IV. Prediction Accuracy for Different V_{UPPER}

Prediction	Cache Hit		Cache Miss		Overall Accuracy
	Correct	Wrong	Correct	Wrong	
$V_{UPPER} = 127$	39.5%	3.0%	50.8%	6.7%	90.3%
$V_{UPPER} = 63$	40.7%	3.7%	49.8%	5.8%	90.5%
$V_{UPPER} = 31$	41.6%	4.7%	48.8%	5.0%	90.4%
$V_{UPPER} = 15$	42.0%	6.0%	47.7%	4.3%	89.7%
$V_{UPPER} = 7$	43.0%	8.7%	44.7%	3.6%	87.7%

recovers accuracy, showing that the proposed sampling hit-miss predictor effectively remedies the biased prediction problem.

5.4. Sensitivity to Sampling Hit-Miss Predictor to V_{UPPER}

Table IV shows prediction accuracy of the sampling hit-miss prediction with different V_{UPPER} values, 127, 63, 31, 15, and 7. V_{TH} is fixed to 4, which is around half of the smallest V_{UPPER} , 7 (3-bit counters are used for the naïve approach and FA-DiBT). The value of V_{UPPER} determines the cycle of sampling the DRAM cache. The predictor issues a sampling read operation to the DRAM cache for correcting wrong miss prediction more frequently with a smaller V_{UPPER} value.

In Table IV, the predictor forecasts more cache hits with smaller V_{UPPER} . The rate of correct hit prediction increases along with smaller V_{UPPER} value. However, the rate of wrong hit prediction increases more sharply. The reason is as follows. If a sampling

Fig. 18. Performance sensitivity of NA-DiBTs with SP to different dirty-block table sizes.

read of DRAM cache is cache hit, then the corresponding counter is set to $V_{TH} - 1$ and thus the following prediction becomes cache hit. Such a hit prediction is not as accurate as the conventional hit-miss prediction that considers history of cache hit/miss and still predicts cache miss after one cache hit if there have been a lot of cache misses.

Since the overall accuracy of the predictor with V_{UPPER} of 63 is the highest, we use it for the evaluation of our technique in this article.

5.5. Sensitivity to Sampling Hit-Miss Predictor to V_{UPPER}

Figure 18 shows the performance sensitivity of NA-DiBTs with SP to different dirty-block table sizes. The results are normalized to that of the alloy cache. In our configurations, 1024- to 8192-set dirty-block tables cover 16MB to 128MB of dirty regions in the DRAM cache at maximum. Thus, as can be seen in the figure, using too small dirty-block tables degrades the performance since smaller dirty-block tables can track fewer dirty blocks, resulting in too many dirty-region evictions. Among these configurations, we chose the 4,096-set dirty-block table in our evaluations as it shows nearly identical performance compared to the configuration with a $2\times$ larger dirty-block table. This dirty-block table configuration can track at most 64MB of dirty regions in the DRAM cache. Although this may seem small compared to the DRAM cache capacity, many workloads have a limited amount of dirty data in general (which agrees with the observation in Sim et al. [2012], where at most 16MB of dirty data are tracked for a 128MB DRAM cache).

5.6. Scalability

Our approach provides scalable latency and area overheads in terms of both off-package and on-package DRAM capacity. First, the off-package memory capacity does not affect the scalability of our schemes because FA/NA-DiBT tracks dirty blocks in the on-package DRAM cache, and thus, the size of FA/NA-DiBT is determined only by the capacity of on-package DRAM. Thus, our approach can easily support systems with hundreds of GBs of off-package DRAM with the same level of overhead.

Second, when our approach is implemented for larger DRAM caches, its overhead scales linearly with the DRAM cache capacity (e.g., 3.4MB of DiBT for a 16GB DRAM cache). Since NA-DiBT can be decomposed into multiple small instances, each of which covers part of the DRAM cache address space, its access latency is scalable to the DRAM cache capacity. Also, the area overhead of DiBT is smaller than that of the previous work on DRAM caches (e.g., 2MB MissMap for a 1GB DRAM cache [Loh and Hill 2011], 3.12MB tag storage for a 512MB cache [Jevdjic et al. 2013], etc.). If

Table V. Implementation Cost

	Naïve Approach
Hit-miss predictor	$8 * 512 \text{ entries} * 3\text{-bit counter} = 1.5\text{KB}$
Counting Bloom filters	$3 * 4\text{K entries} * 5\text{-bit counter} = 7.5\text{KB}$
Dirty list (DL)	$4\text{K sets} * 4\text{-way} * (1\text{-bit NRU} + 36\text{-bit tag}) = 74\text{KB}$
Total	83KB
Latency of DL	1ns (3 cycles at 3GHz)
	FA-DiBT
Hit-miss predictor	$8 * 512 \text{ entries} * 3\text{-bit counter} = 1.5\text{KB}$
Counting Bloom filters	$3 * 4\text{K entries} * 5\text{-bit counter} = 7.5\text{KB}$
Dirty-block table (DBT)	$4\text{K sets} * 4\text{-way} * (1\text{-bit NRU} + 36\text{-bit tag} + 64 \text{ dirty bits}) = 202\text{KB}$
Total	211KB
Latency of DBT	1.16ns (4 cycles at 3GHz)
	NA-DiBT and Sampling Hit-Miss Predictor
Hit-miss predictor	$8 * 512 \text{ entries} * 6\text{-bit counter} = 3\text{KB}$
Counting Bloom filters	$3 * 4\text{K entries} * 5\text{-bit counter} = 7.5\text{KB}$
Dirty-block table (DBT)	$4\text{K sets} * 4\text{-way} * (1\text{-bit NRU} + 36\text{-bit tag} + 64 \text{ dirty bits}) = 202\text{KB}$
Total	212.5KB
Latency of DBT	1.16ns (4 cycles at 3GHz)

such overhead is undesirable, we can reduce the size of DiBT at the cost of a slightly increased write-back ratio.

5.7. Implementation Cost

Table V shows the implementation cost for three approaches: naïve approach, FA-DiBT, and NA-DiBT with SP. In the naïve approach, the hit-miss predictor consists of eight counter tables for each core, where each table has 512 entries of 3-bit counters. The dirty-region tracker consists of counting Bloom filters and a dirty list for dirty regions, and it tracks dirty regions at 4KB granularity.⁴ The counting Bloom filters consist of three counter tables, each of which has 4K entries of 5-bit counters. The dirty list is a set-associative structure of 4K four-way sets, and each set consists of 1 bit for NRU and 36 bits for tag. In summary, the total implementation cost of the naïve approach is 83KB.

FA-DiBT is similar to the naïve approach. One difference is that 64-bit dirty bits (= 4,096-byte page/64-byte cache blocks) are added to each entry of the dirty list to make a dirty-block table, which costs 128KB. Therefore, the total implementation cost of FA-DiBT is 211KB. We coordinate NA-DiBT to have a size similar to FA-DiBT for a fair comparison. The sampling hit-miss predictor has eight 512 entries of 6-bit counters, so that the total cost of NA-DiBT with SP is 212.5KB. This value is negligible considering the size of a DRAM cache.

We used CACTI 6.5 with 32nm technology and ITRS-LOP cells to model the latency of a dirty list (in the naïve approach) and dirty-block tables (in FA/NA-DiBT). We included this latency overhead of accessing the dirty list or the dirty-block tables, if any, in our simulation.

6. RELATED WORK

We have already explained the three most closely related block-granularity DRAM cache techniques in Section 2 [Loh and Hill 2011; Qureshi and Loh 2012; Sim et al. 2012]. In addition, BEAR [Chou et al. 2015] aims to reduce bandwidth consumptions of DRAM cache by miss fill, write-back probes, and miss detection, but it does not

⁴The rationale behind this is that a typical OS page size is 4KB, and thus, cache blocks inside a 4KB region are expected to share similar access characteristics.

consider redundant read prior to write for checking the dirtiness of a block in the DRAM cache. In ATCache [Huang and Nagarajan 2014], a small SRAM tag cache is adopted to avoid the high area overhead of maintaining tags of DRAM cache on SRAM. It shows a performance similar to a very fast tags-in-SRAM design. It is different from our technique based on a tags-in-DRAM design.

There are several page-granularity DRAM caches [Woo et al. 2010; Jevdjic et al. 2013, 2014]. In those approaches, the tag overhead is lower than that of a block-granularity DRAM cache. In a die-stacked DRAM architecture proposed by Woo et al. [2010], the technique prefetches an entire page while providing a cache access in a cache block size. The footprint DRAM cache [Jevdjic et al. 2013] allocates a page in a DRAM cache but fetches only blocks that will be accessed during the residency of the page in the cache, which can reduce the traffic of an off-package memory. The unison DRAM cache [Jevdjic et al. 2014] adopts a tag-in-DRAM concept similar to alloy cache [Qureshi and Loh 2012]. It applies the concept to the footprint cache [Jevdjic et al. 2013] to improve the scalability and the performance of a system. The bimodal DRAM cache [Gulur et al. 2014] organizes the data at two granularities, big blocks for data that have high spatial locality and small blocks for the rest. The technique can efficiently utilize the capacity of the DRAM cache by adaptively selecting the right granularity for individual blocks at runtime. The tagless DRAM cache [Lee et al. 2015] uses cache-map TLB that stores virtual-to-cache address mappings to remove tags of a DRAM cache. CAMEO [Chou et al. 2014] uses an on-package stacked DRAM as a part of memory with hardware management, and it can dynamically change the physical location of a cache line to retain recently accessed data in a DRAM cache.

The dirty-block index (DBI) technique [Seshadri et al. 2014] decouples the dirty bits from the tag store and groups the dirty information of blocks in the same DRAM row in the same DBI entry. It applies several optimizations to a cache by using decoupled dirty bits, such as aggressive DRAM-aware write-back, bypassing cache lookups, and heterogeneous ECC for clean/dirty blocks. The structure of DBI is similar to the dirty-block table in our FA-DiBT because the row tags of DBI entries are based on off-package memory addresses, but it is different from NA-DiBT, which is based on on-package DRAM addresses. Therefore, DBI cannot eliminate tag checking prior to a write operation when victim is not dirty. Actually, the work on DBI does not apply the technique to DRAM cache but only to on-chip last-level cache (if it is applied to DRAM cache, then the result will be similar to FA-DiBT).

7. CONCLUSION

We propose a DRAM cache technique that combines self-balancing dispatch with a direct-mapped cache organization. Based on the observation that a direct combination of them is inefficient due to the overhead of dirty-region eviction from the dirty-region tracker, we devise a dirty-block tracker (DiBT) that maintains the dirtiness of blocks in a dirty region. We also propose an improved version of DiBT called NA-DiBT that detects a dirty region as a group of cache blocks instead of a page. It inherits good features of DiBT and also exploits the characteristics of a direct-mapped cache organization. NA-DiBT can remove read operations for checking the dirtiness of blocks in the cache prior to write operations if the blocks are not dirty. This optimization can save bandwidth usage of the DRAM cache significantly. To mitigate the biased prediction problem caused by the proposed approach, we also devise a sampling hit-miss predictor. The simulation results show that our DRAM cache technique using NA-DiBT with the sampling hit-miss predictor improves the performance of a multicore system by more than 10% on average compared to the state-of-the-art direct-mapped DRAM cache technique.

REFERENCES

- Jung Ho Ahn, Sheng Li, O. Seongil, and Norman P. Jouppi. 2013. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, 74–85.
- Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'12)*. IEEE, 33–38.
- Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1–12.
- Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, 198–210.
- Nagendra Guler, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 38–50.
- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.
- John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
- Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 51–60.
- Joe Jeddleloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *Proceedings of the 2012 Symposium on VLSI Technology (VLSIT'12)*. IEEE, 87–88.
- High Bandwidth Memory JEDEC. 2013. DRAM (JESD235).
- Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 25–37.
- Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 404–415.
- Jung-Sik Kim, Chi Sung Oh, Hocheol Lee, Donghyuk Lee, Hyong Ryol Hwang, Sooman Hwang, Byongwook Na, Joungeon Moon, Jin-Guk Kim, Hanna Park, Jang-Woo Ryu, Kiwon Park, Sang Kyu Kang, So-Young Kim, Hoyoung Kim, Jong-Min Bang, Hyunyeon Cho, Minsoo Jang, Cheolmin Han, Jung-Bae Lee, Joo Sun Choi, and Young-Hyun Jun. 2012. A 1.2 V 12.8 GB/s 2 Gb mobile wide-I/O DRAM With 4 × 128 I/Os using TSV based stacking. *IEEE Journal of Solid-State Circuits* 47, 1 (2012), 107–116.
- Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. 2015. A fully associative, tagless DRAM cache. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 211–222.
- Gabriel H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the 35th International Symposium on Computer Architecture, 2008 (ISCA'08)*. IEEE, 453–464.
- Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 454–464.
- Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31.
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 63–74.
- Moinuddin K. Qureshi and Gabriel H. Loh. 2012. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 235–246.
- Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2014. The dirty-block index. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. IEEE, 157–168.

- Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. 2012. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 247–257.
- Young Hoon Son, O. Seongil, Hyunggyun Yang, Daejin Jung, Jung Ho Ahn, John Kim, Jangwoo Kim, and Jae W. Lee. 2014. Microbank: Architecting through-silicon interposer-based main memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 1059–1070.
- Dong Hyuk Woo, Nak Hee Seong, Dean L. Lewis, and Hsien-Hsin S. Lee. 2010. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 1–12.

Received May 2016; revised January 2017; accepted February 2017