Dense Footprint Cache: Capacity-Efficient Die-Stacked DRAM Last Level Cache *

Seunghye Shin
Dept. of Electrical and
Computer Engineering
North Carolina State
University, NC, USA
sshin6@ncsu.edu

Sihong Kim
Samsung Electronics
South Korea
sihong.kim@samsung.com

Yan Solihin
Dept. of Electrical and
Computer Engineering
North Carolina State
University, NC, USA
solihin@ncsu.edu

ABSTRACT

Die-stacked DRAM technology enables a large Last Level Cache (LLC) that provides high bandwidth data access to the processor. However, it requires a large tag array that may take a significant portion of the on-chip SRAM budget. To reduce this SRAM overhead, systems like Intel Haswell relies on a large block (Mblock) size. One drawback of a large Mblock size is that many bytes of an Mblock are not needed by the processor but are fetched into the cache. A recent technique (Footprint cache) to solve this problem works by dividing the Mblock into smaller blocks where only blocks predicted to be needed by the processor are brought into the LLC. While it helps to alleviate the excessive bandwidth consumption from fetching unneeded blocks, the capacity waste remains: only blocks that are predicted useful are fetched and allocated, and the remaining area of the Mblock is left empty, creating *holes*. Unfortunately, holes create significant capacity overheads which could have been used for useful data, hence wasted refresh power on useless data.

In this paper, we propose a new design, *Dense Footprint Cache* (DFC). Similar to Footprint cache, DFC uses a large Mblock and relies on useful block prediction in order to reduce memory bandwidth consumption. However, when blocks of an Mblock are fetched, the blocks are placed contiguously in the cache, thereby eliminating holes, increasing capacity and power efficiency, and increasing performance. Mblocks in DFC have variable sizes and a cache set has a variable associativity, hence it presents new challenges in designing its management policies (placement, replacement, and update). Through simulation of Big Data applications, we show that DFC reduces LLC miss ratios by about 43%, speeds up applications by 9.5%, while consuming 4.3% less energy on average.

*This research is supported in part by NSF grant CCF-1116540 and by Samsung Electronics.

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 ACM 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.

MEMSYS '16, October 03-06, 2016, Alexandria, VA, USA

© 2016 ACM. ISBN 978-1-4503-4305-3/16/10...\$15.00

DOI: <http://dx.doi.org/10.1145/2989081.2989096>

CCS Concepts

•Hardware → Memory and dense storage;

Keywords

Die-stacked DRAM; last-level cache; replacement policy

1. INTRODUCTION

The advance in chip fabrication has enabled more and more cores to be integrated on a single chip. The increase in the number of cores is placing a significant pressure on the memory bandwidth demand, creating a bandwidth wall problem where memory bandwidth increasingly becomes a performance limiting factor [21]. Fortunately, die-stacked memory has alleviated this bottleneck by providing high bandwidth memory near the processor. Through die-stacking, DRAM can be stacked on top of the processor die (3D) [6, 28] or on a separate die connected through silicon interposer (2.5D) [2], providing an additional memory capacity to the traditional off-chip DRAMs, but with much higher bandwidth and slightly lower interconnect latency. A stacked DRAM may be used as a Last Level Cache (LLC) [9, 10, 15, 16, 20, 29, 8, 1, 12, 5, 19], as a part of the main memory [18, 3], or both at the same time. As part of the main memory, a stacked DRAM incurs no tag overheads but requires careful page allocation and migration to keep hot pages in the stacked DRAM. In addition to the need of hardware support for profiling, frequent migrations are intolerably expensive; e.g., the costs of TLB shutdown [18]. In contrast, as an LLC, a stacked DRAM is automatically managed by hardware but incurs tag overheads.

One of main challenges of using a stacked DRAM as the LLC is that it requires a large tag array. If the tag array is implemented in SRAM, tag access will be very fast, but it consumes a large portion of the SRAM budget. For example, for a 256MB 16-way associative DRAM LLC operating on 64-bit addresses, the tag array requires a 32MB SRAM budget, which is prohibitively expensive. There are two general approaches in dealing with this problem. In one approach, tags are co-located in the stacked DRAM together with data [16, 20, 8, 5]. While it completely solves the tag array overhead problem, this approach comes at a very significant cost in latency and bandwidth. A typical cache access requires three (rather than one) DRAM accesses: the first one to read out the tag from the stacked DRAM, the second access to read out data upon a cache hit, and the third one to update the state and LRU bits. Recognizing

the problem, recent proposals try to mitigate this overhead by using way predictions or traffic filter [8, 5, 1], but they reduce bandwidth overheads at the expense of lower hit rates or still suffers high miss latency while improving hit latency. Thus, in [8], the authors report performance improvement only when the LLC size is very large (1GB or larger).

The second approach to solve the tag array overhead problem is to rely on large block sizes. A large block (major block or *Mblock*) in the range of a few kilobytes significantly reduces the number of cache blocks and thus the tag array size [10, 9, 8]. For example, if the block size is 2KB instead of 64B, the 256 MB LLC only requires 1MB in SRAM budget for the tag array. This approach is used in the Intel Haswell system [7]. The main drawback of this approach is cache capacity and bandwidth waste: insufficient spatial locality in most programs result in only a few small blocks (i.e., 64B blocks) being accessed out of a 2KB Mblock that is fetched and allocated in the LLC. There are several strategies to mitigate the waste. In CHOP [10], a "hot page" filter prevents cold Mblocks from getting allocated in the LLC. Another strategy is to subdivide the Mblock into smaller blocks where only blocks predicted to be needed by the processor are brought into the LLC [9, 8]. Thus, an Mblock may not be fully filled: only blocks that are predicted useful are fetched and allocated, and the remaining area of the Mblock are left empty (*holes*). Such a strategy removes bandwidth waste but not the cache capacity waste.

Holes are costly with stacked DRAM for two reasons. First, holes incur capacity overheads which could have been used for useful data. In addition, it incurs power overheads because DRAM rows containing holes still need to be refreshed, even when no valid data is present in holes. To eliminate the drawback, in this paper we propose a new design that we refer to as *Dense Footprint cache* (DFC), similar to [24]. Like Footprint cache, DFC uses a large Mblock and relies on useful block prediction in order to reduce memory bandwidth consumption. However, when blocks of an Mblock are fetched, they are placed contiguously in the cache, thereby eliminating holes. By eliminating holes, DFC avoids wasted LLC capacity and hosts more valid blocks in the stacked DRAM. However, due to eliminating holes, the Mblocks in DFC have variable sizes and a cache set has a variable associativity. This introduces unique challenge to the cache placement and replacement policy.

A large incoming Mblock may require multiple consecutive resident Mblocks to be evicted. Traditional replacement policies such as LRU are not applicable in this case because they assume there is only one victim Mblock. Thus, we propose three new cache replacement policies for DFC: *LRU+*, *MaxAvg*, and *MaxMin*. *LRU+* starts with LRU Mblock and expands to adjacent Mblocks. *MaxAvg* and *MaxMin* calculate the average and minimum stack distance positions of all candidate Mblock groups and select a group with the maximum score. We analyze these replacement policies and derive their behavior bounds mathematically and examine their performance through simulations. In addition, we also examine augmenting the replacement policy with placement policies such as best fit and worst fit.

Another contribution of this paper is a new footprint prediction update policy. In Footprint cache, a footprint of an Mblock is updated when the Mblock is evicted. However, when the LLC is large, Mblocks stay too long in the cache and hence the footprint predictor learns very slowly.

We solve this problem by allowing footprints to be updated even when Mblocks are still resident in the cache. We refer to this as *Early Footprint Update* policy.

Finally, we evaluate DFC using a set of big data applications, and found that it reduces LLC miss ratio by 43% compared to Footprint cache, which translates into an average speedup of 9.5%.

The remainder of the paper is organized as follows. Section 2 discusses related work, Section 3 overviews the DFC, Section 4 presents DFC design details, Section 5 describes the evaluation methodology, Section 6 and 7 discuss evaluation results and key findings. Section 8 concludes this work.

2. RELATED WORK

Researchers have recognized the usefulness of stacked DRAM for a long time. Several earlier studies demonstrated the high bandwidth benefit of using stacked DRAM as the entire main memory [27, 11, 13, 14]. More recent studies recognize that off-chip DRAM will exist alongside stacked DRAM, hence they have proposed to use the stack DRAM as a part of the main memory [18, 3]. Concurrently, other studies explored the design and usefulness of using the stacked DRAM as a hardware managed last level cache (LLC) [9, 10, 15, 16, 20, 29, 8, 1, 12, 5].

In the context of using stacked DRAM as the LLC, there was a recognition that the SRAM budget used for the tag array of the LLC is substantial and thus needs to be reduced [10]. In one approach, tags are co-located in the stacked DRAM together with data (LHcache [16], Alloy cache [20], Unison cache [8], and Bi-modal cache [5]). As discussed earlier, while it completely solves the tag array overhead problem, this approach increases latency and bandwidth significantly, as a typical cache access requires three (rather than one) DRAM accesses: one to read out the tag, another to read out data, and another to update the state and LRU bits. To reduce such a high hit latency, Unison and Bi-modal cache rely on way predictions [8, 5]. However, a cache miss (or way prediction miss) still require multiple times DRAM accesses, and when coupled with high cache miss rates in LLC, the performance overhead is still considerable [8]. In [8], the authors reported that Footprint cache as showing better performance than Unison cache when the DRAM LLC, unless the LLC is very large (1GB or larger). There is also a proposal to reduce bandwidth overheads in these cache designs at the expense of lower hit rates [1].

The second approach is to rely on large block sizes, such as the 1KB Mblock size in the Intel Haswell 128MB L4 cache [7]. The main drawback of using a large Mblock is bandwidth consumption due to insufficient spatial locality in most programs, where only a fraction of the bytes in an Mblock is actually used by the processor. To mitigate that, CHOP [10] uses a "hot page" filter to avoid allocating cold Mblocks from being allocated in the LLC. Another solution, used in Footprint cache [9] and Unison cache [8], subdivides the Mblock into blocks and only brings blocks that are predicted useful into the LLC [9]. Blocks that are not brought in are left empty (*holes*). Footprint cache [9] is the closest related work to this paper. In contrast to Footprint cache, our proposal *Dense footprint cache* (DFC) eliminates holes by storing valid blocks contiguously. This improves the usage of cache capacity. As a result, although virtually an Mblock size is 2KB, its physical size in the stacked DRAM

LLC is variable. In a way, DFC is a compressed form of Footprint cache. With the same DRAM budget, DFC achieves the performance comparable to twice the size of Footprint cache.

Our approach DFC’s co-location of multiple Mblocks in a cache line resembles decoupled sectored cache [23]’s co-location of multiple blocks, applied to a much larger granularity. However, apart from the granularity, there is a key difference. In the decoupled sectored cache, blocks are not stored in a de-fragmented format, they are simply co-located. Thus, two blocks can be co-located only when their non-hole sub-blocks do not overlap. In DFC, Mblocks are actually stored in a de-fragmented format, hence it is not important for blocks in one Mblock to strictly correspond to a hole in another Mblock for them to be co-located. This ensures higher co-location success. Finally, a similar idea to DFC can be found in a patent [24], however the patent only provides organization of the cache but not its management, and does not provide quantitative evaluation.

3. OVERVIEW

As discussed earlier, the tag array overhead is a major cost of stacked DRAM LLC. One important requirement for the tag array is that it must be accessible at a low latency in order to facilitate quick determination of LLC hit or miss. This is one of the primary reasons why the tag array of the LLC is implemented in SRAM in the Intel Haswell system [7]. In most stacked DRAM products, such as the Hybrid Memory Cube (HMC), the stacked DRAM access latency is only slightly faster than the off-chip DRAM access latency, primarily because design choices were made to prioritize density over fast access time in both cases. A solution is to increase the cache block size so that it reduces the number of blocks to keep track and makes SRAM tag array is viable. There is a limit to this approach, however. If the LLC is very huge, e.g. 1GB or larger, the required SRAM budget for the tag array is going to be too large. However, we believe that diminishing return from cache capacity will likely cap the LLC capacity much below 1GB, leaving the remainder of the stacked DRAM capacity as a part of the main memory to avoid duplicating not useful data in cache and main memory.

In the Footprint cache, an Mblock size of several KBs is used (e.g. 2KB). Each 2KB Mblock is divided into 32×64 -byte blocks. Figure 1(a) illustrates a Footprint cache. The cache has a SRAM tag array and DRAM data array. The tag array shows which blocks in an Mblock are valid (represented by ‘1’s), and which ones are holes (represented by ‘0’s). Correspondingly, the data array shows valid blocks as solid-colored cells and holes as crossed-out cells. The figure shows 4 Mblocks: A, B, C, and D.

Suppose that a request comes from the processor or L1/L2/L3 cache (Step 1). After checking the tag array, suppose that the block is found on Mblock A, as indicated by the valid bit vector. Each bit in the valid bit vector represents whether the block is valid or is a hole in the Mblock. In this example, the block is then returned to the processor (Step 2a). Let us consider a different situation, where the access is to Mblock E which cannot be found in the Footprint LLC. Here we have an Mblock miss, the Footprint History Table (FHT) is looked up. The Footprint in the FHT indicates that in the past, only the first two blocks of the Mblock are accessed by the processor. Hence, only these two blocks will be fetched

Figure 1: Illustration of the Footprint cache [9] (a), and Dense Footprint cache (b).

from the main memory (Step 3) and allocated in the LLC. An Mblock that is evicted to make room for the new Mblock updates the FHT, where the valid bits are recorded as the footprint for the evicted Mblock (Step 4). Note that as an overview, the discussion is somewhat simplified here. The actual Footprint cache uses two bits (valid and dirty) per Mblock to indicate whether an Mblock is valid or not, clean or not, and has been demanded or not. Also, the FHT is indexed using the hash of the PC of the missing load/store, and the address being missed. The Footprint cache may also be set associative.

Figure 1(b) illustrates our proposal: *Dense footprint cache* (DFC). The DFC in the figure co-locates up to two Mblocks in the same cache line (or DRAM row). The figure shows the tag array now consists of more Mblocks per cache line. Each Mblock not only has the tag and valid bit vector, but also has a starting location that indicates the offset in the cache line where the physical location of the Mblock starts. In the figure, Mblock A and B are co-located in one cache line. Mblock C and D overflow the second cache line, hence only one Mblock (i.e. Mblock D) can be kept in the cache. Note that the tag array of DFC has the same size as in Footprint cache, however the data array is half the size of Footprint cache. The total number of Mblocks that can be placed in the DFC falls somewhere between the size of Footprint cache and twice its size, depending on the average number of valid bits of Mblocks.

As shown in the figure, DFC may still contain holes. However, there are two key differences between holes in DFC versus ones in Footprint cache. In Footprint cache, holes arise due to *internal fragmentation* within an Mblock, where some blocks are not valid within an Mblock. With DFC, there are

no holes due to internal fragmentation; holes are caused by *external fragmentation*, where co-located Mblocks do not occupy the entire cache line. Note also that the holes in Footprint cache are *permanent*: holes in an Mblock stay for the entire residency of the Mblock in the cache, and disappear only when the Mblock is evicted. In contrast, holes in DFC are *temporary*, and can disappear when a new Mblock that can utilize the space is allocated in the cache. For example, in the figure, if a new Mblock with a size of two blocks is brought into the second cache set, the holes will disappear. Because of these two major differences in nature of holes in Footprint cache vs. DFC, there are much fewer holes in DFC, resulting in better space efficiency (fewer cache capacity misses) and better power efficiency (refreshes are applied mostly to valid data).

While the cache becomes denser, there are several unique challenges for designing a DFC. Unlike the Footprint or conventional caches, the physical size of Mblocks is different because the number of blocks that are allocated in the DFC differs across Mblocks. This introduces several new challenges to the cache placement and replacement policies. Let us consider a case where an incoming Mblock needs to be allocated in the cache. Suppose evicting an existing Mblock creates sufficient space but there are several possible Mblocks with different sizes that can be chosen. In this case, the cache replacement policy faces a similar problem to the one faced by memory management in an operating system (OS), where the requested memory size may not be equal to the sizes of blocks in the free space. Several options, such as *best fit*, *worst fit*, *first fit*, or *random fit* are possible. In addition to such options, a question arises of how to incorporate traditional replacement policy information, such as stack position, when choosing which Mblock to evict.

Let us also consider a case where multiple Mblocks need to be evicted to make sufficient space for the incoming Mblock. Now we not only face the fit and stack position considerations, but we also face a plurality of stack positions to consider. For example, we may be able to evict a group of two Mblocks that have stack positions three and eight, versus a group of two Mblocks that have stack positions five and six. The first group of Mblocks contain the least recently used Mblock (stack position 8) but also a more recently used Mblock. The second group of Mblocks contain two of some of the least recently used blocks. A new placement and replacement policy is needed to choose which group of Mblocks to evict.

A final problem that we deal with DFC is an observation of a problem that we discover with the FHT update policy. The FHT is updated only when an Mblock is evicted. The blocks that are actually accessed during the Mblock’s residency in the LLC form a new footprint in the FHT entry for the Mblock. Such a policy makes sense in that when there is a miss to the Mblock in the future, the footprint provides good prediction of what blocks to fetch and allocate in the LLC. However, we found that as the LLC size grows, Mblocks stay longer in the cache and are evicted less frequently, which reduces footprint updates. This slows down the ability for the FHT to learn footprints. To overcome this slow footprint learning, we propose a new footprint update technique called *early footprint update* (EFU), which allows footprints to be updated even when the Mblock still resides in the cache. We showed that EFU improves footprint learning and improves the footprint prediction coverage. The following section will

discuss these design issues in more detail.

4. DENSE FOOTPRINT CACHE DESIGN

In DFC, Mblocks have different actual sizes, so we need a new cache placement/replacement policy. There are two types of cache misses in DFC: an *Mblock miss* where the access is to an Mblock that is not found in the cache, and a *block miss* where the access is to a block that is not found in an Mblock that is currently cached. An Mblock miss arises due to cold/capacity/conflict, while a block miss arises from inaccurate prediction of the footprint of an Mblock such that the Mblock was allocated in the cache without having the needed block. We will discuss the Mblock miss first.

While all Mblocks share the same virtual size (e.g. 2KB), their actual sizes vary and depend on how many blocks are allocated in the cache. When a large incoming Mblock needs to be allocated, we may need to evict several Mblocks to make room for the incoming Mblock. The traditional least recently used (LRU) replacement policy is not directly applicable when dealing with multiple victim Mblocks. A relevant question is: how should we take into account stack recency positions when evicting multiple Mblocks? Another question is: how should fragmentation be taken into account in the eviction? And finally: how should we handle block miss for an Mblock that is already in the cache?

4.1 Cache Replacement Policy

As mentioned earlier, potentially several Mblocks need to be evicted to provide space for an incoming Mblock. The first constraint that we choose is to require the evicted Mblocks to be consecutive, in order to avoid shifting the remaining Mblocks to create contiguous space large enough for the incoming Mblock. Shifting is very expensive in terms of latency and power consumption and hence we would like to avoid it.

An alternative to shifting is to split the incoming Mblock into non-contiguous parts so that it fits in the non-contiguous space freed up by evicting non-contiguous victim Mblocks. Breaking an Mblock rather than allocating it contiguously in a cache line creates unnecessary complexities in the design, as the tag array has to be designed to record multiple starting addresses. Reading from an Mblock requires assembling from non-contiguous parts, and additional computation to locate the desired block in an Mblock. Thus, we also avoid this.

Given our design choice’s preference of evicting contiguous Mblocks, we can now narrow down the replacement policy by considering the stack positions of the victim Mblocks. Here we consider three different policies: *LRU+*, *MaxAvg*, and *MaxMin*. *LRU+* straightforward: it first selects the LRU Mblock as a victim. If that does not free up sufficient space, its neighboring Mblock (either to the left or to the right) is selected for eviction as well. Which neighboring Mblock is selected? *LRU+* selects the less recently used Mblock between the two neighboring Mblocks to add to the LRU Mblock. If these two Mblocks still have not freed sufficient space, we repeat the process by expanding the victim by another neighboring Mblock until the total eviction size becomes equal or larger than the requested block size. Figure 2 (top diagram) illustrates *LRU+* replacement policy. The figure shows the tag array of a cache line, showing there are 8 Mblocks currently cached in the line: A–H. Each Mblock’s LRU stack position is shown with a number indi-

Figure 2: Comparing three possible DFC replacement policies: LRU+, MaxAvg, and MaxMin.

cating MRU (0), 2nd MRU (1), and so on until LRU (7). The LRU Mblock is D. Other fields, such as the starting location of an Mblock, its state, and which blocks are present, are omitted in the figure.

Now suppose that there is a miss to a new Mblock that is twice as large as any of the existing Mblocks. Thus, two Mblocks need to be evicted. With LRU+, we expand the victim group from Mblock D to its neighboring Mblock C or Mblock E. Since Mblock E is closer to the LRU (stack position 2), the two Mblocks selected for eviction are Mblocks D and E. Note, however, Mblock E is the third MRU block, hence it is risky to evict it since the processor may still need it. To reduce the risk, we propose two additional policies: MaxAvg and MaxMin.

In the MaxAvg, groups of Mblocks are scored based on their average stack distance positions, which is shown in the numbers above the Mblocks. For example, a group of two Mblocks A and B have an average stack distances of $\frac{4+6}{2} = 5$, a group of two Mblocks B and C have an average stack distances of $\frac{6+1}{2} = 3.5$, etc. After these average scores are calculated, the group of Mblocks having the largest score is selected for eviction. In the figure, Mblocks A and B have the highest average score, so they are selected for eviction. Note that overall, MaxAvg is less risky as they evict Mblocks much closer to the LRU than with LRU+. The rationale for MaxAvg is straightforward: it tries to find a group of Mblocks to evict that, on average, is the least recently used group.

In the MaxMin, groups of Mblocks are scored based on their minimum stack distance positions, which is shown in the numbers above the Mblocks. For example, a group of two Mblocks A and B have a minimum score of $\min(4, 6) = 4$, a group of two Mblocks B and C have a minimum score of $\min(6, 1) = 1$, etc. After these minimum scores are calculated, the group of Mblocks having the largest score is selected for eviction. In the figure, Mblocks A and B have the highest minimum score, so they are selected for eviction. In this

example, MaxMin selects the same Mblocks as MaxAvg for eviction.

We need to have a framework of analysis to understand the replacement policies' behavior. For this purpose, we look several criteria. The first criteria is whether a replacement policy is *self contained*, meaning that it is guaranteed to be able to find a victim. If a replacement policy is self contained, it can be used alone. Otherwise, it has to rely on a different replacement policy as a backup mechanism to break a tie. The second criteria is which Mblocks a replacement policy guarantees not to evict. For example, can a replacement policy evict the MRU Mblock, second MRU Mblock, etc.? This question is important in that it helps us understand the worst case behavior of a replacement policy. An average case behavior is easy to obtain by running programs on a simulation model. However, knowing the replacement policy's worst case behavior gives us assurance that it will not produce pathological performance cases in some situations that are not covered by existing benchmarks, as well as helps explain poor performance numbers (if any) when running benchmarks on simulation models. Let us discuss these two criteria.

The LRU+ policy is self-contained, because it is always able to find an eviction victim. Starting from the LRU Mblock, it expands to one of two neighboring Mblocks with different stack positions. In contrast, MaxAvg and MaxMin are not self contained as they may produce the same replacement score for different groups of Mblocks, as shown in Figure 2: Mblocks C and D have the same MaxAvg score of '4' as Mblocks F and G, Mblocks B and C have the same MaxMin score of '1' as Mblocks E and F, etc. This means that if MaxAvg or MaxMin is deployed, we still need to use a back up mechanism to break the tie. The backup mechanism may be as simple as random, or as sophisticated as using an entirely different replacement policy.

The second criteria of the pathological eviction cases is much harder to evaluate. However, we can prove some properties by constraining our analysis using several simplifying assumptions. Specifically, we are going to assume that there are N Mblocks of the same size residing in one cache line. From these, we will choose a subset of R Mblocks to replace in order to make room for an incoming Mblock, where $1 \leq R < N$. Furthermore, we assume that a cache line wraps around, hence Mblocks on both edges of the cache line can be considered as an eviction group, even though they are not physically contiguous. Under these constrained situations, we will analyze LRU+, MaxAvg, and MaxMin, with regard to what Mblocks are guaranteed not to be selected for eviction.

For LRU+, the following property holds:

PROPERTY 4.1. *With LRU+, only the MRU Mblock is guaranteed not to be selected for eviction.*

The reason why only the MRU Mblock is guaranteed not to be evicted is that when we need to expand the eviction group, we have two Mblocks that can be considered: the left neighbor of the LRU Mblock and the right neighbor of the LRU Mblock. If one of these neighboring Mblocks is the MRU Mblock, then it will not be included into the eviction group, because the other neighboring Mblock will be selected instead. Thus, we can conclude that in the worst case situation, no protection is provided against pathological performance behavior where really useful Mblocks such as

the 2nd MRU, 3rd MRU, etc. may be evicted. If we relax the wrap-around cache line assumption, the situation becomes even worse. If the LRU Mblock is at either edge of the cache line, there is only way to expand the eviction group, and if the only neighboring Mblock is the MRU Mblock, it will be selected for eviction.

Let us now consider MaxMin. MaxMin has the following property:

PROPERTY 4.2. *With MaxMin, the $\lfloor \frac{N-1}{R} \rfloor$ most recently used Mblocks will not be selected for eviction.*

For example, in a cache line that selects 2-Mblock victim out of 8 Mblocks, $\lfloor \frac{8-1}{2} \rfloor = 3$ of the most recently used Mblocks are guaranteed not to be evicted. For a cache line containing 16 Mblocks ($N = 16$), 7 of the most recently used Mblocks are guaranteed not to be evicted, if the Mblock group size is 2, or 5 if the group size is 3. To prove it, suppose we start with the MRU Mblock and expand it to contain groups of R Mblocks. There are R such possible groups, and each group will have an eviction score of 0, which is the stack position of the MRU Mblock. Next, consider the 2nd MRU block and expand it to contain groups of R Mblocks. There are R such possible groups, however some of them may overlap with the ones for the MRU Mblock. The groups that overlap with the MRU Mblock will have a score of 0, but the ones that do not overlap will have a score of 1. Thus, for the 2nd MRU, it adds to anywhere between 1 to R unique groups. If we repeat the process until the k^{th} MRU, we have aggregated anywhere between $R+k-1$ groups to kR groups with the lowest eviction scores. As long as the number of these groups is smaller than N , then there is at least one group that has a higher eviction score. In other words, we need to satisfy the inequality $kR \leq N-1$, equivalent to $k \leq \frac{N-1}{R}$.

Essentially, MaxMin has a much stronger protection than LRU+. Whereas LRU only guarantees that the MRU Mblock will not be evicted (with wrap-around assumption), MaxMin guarantees almost half of the most recently used Mblocks from being evicted if the Mblock eviction group size is 2.

Let us now consider MaxAvg. Proving a property with MaxAvg is much more challenging mathematically. However, we can still infer a property of MaxAvg:

PROPERTY 4.3. *With MaxAvg, only the MRU Mblock is guaranteed not to be evicted when $R = 2$, no guarantee otherwise.*

To sketch a proof, we start by noticing that the average score for Mblock group itself averages to a constant. Suppose that the LRU stack positions of N Mblocks are $a_1, a_2, \dots, a_N$, where each a_i may range from 0 (MRU) to $N-1$ (LRU) for all i 's. There are exactly N potential Mblock group that can be selected for eviction: $(a_1, a_2, \dots, a_R)$, $(a_2, a_3, \dots, a_{R+1})$, $\dots$, $(a_N, a_1, \dots, a_{R-1})$. Each of these Mblock group has an average eviction score of $\frac{1}{R}(a_1 + a_2 + \dots + a_R)$, $\frac{1}{R}(a_2 + a_3 + \dots + a_{R+1})$, $\dots$, $\frac{1}{R}(a_N + a_1 + \dots + a_{R-1})$. The average of the eviction scores of Mblock groups turns out to be quite simple: $\frac{R \sum a_i}{RN} = \frac{\sum a_i}{N} = \frac{N-1}{2}$.

From here, it follows that any average score that is equal to or larger than $\frac{N-1}{2}$ may become an eviction victim. Is it possible to arrange a_i 's such that all Mblock groups have an equal eviction score? It turns out it is not possible, as we are dealing with LRU scores that are whole numbers. The

smallest difference in eviction score between two adjacent Mblock group is provably $\frac{1}{R}$. This means that in order for an Mblock group to be selected as a victim, its eviction score must be at least equal to $\frac{N-1}{2} + \frac{1}{R}$, the *threshold eviction score*.

In the special case where $R = 2$, the threshold eviction score simplifies to $\frac{N}{2}$. To prove that the MRU Mblock will never be evicted when $R = 2$, consider that even when the MRU Mblock (LRU score of 0) neighbors the LRU Mblock (score $N-1$), the eviction score of the MRU and LRU Mblocks is $\frac{N-1}{2}$, which is less than the threshold eviction score of $\frac{N}{2}$. Hence, the MRU Mblock is guaranteed to never be evicted. With regard to the 2nd MRU Mblock, its eviction score of 1, when it neighbors the LRU Mblock, the group eviction score is $\frac{N}{2}$, which meets the threshold eviction score. Thus, the 2nd MRU Mblock may be evicted.

For cases where $R > 2$, even the MRU Mblock may get evicted. For example, let us consider $N = 8, R = 3$. The MRU Mblock, when it neighbors the LRU (LRU score 7) and 2nd LRU Mblock (LRU score 6), their average eviction score is $4\frac{1}{3}$, which is larger than the threshold eviction score of $\frac{8-1}{2} + \frac{1}{3} = 3.83$.

Overall, in the worst case, the MaxAvg and LRU+ provide weak protection from evicting the recently used Mblocks. In contrast, MaxMin provides the strongest protection, since several MRU Mblocks will not be evicted, depending on N and R .

4.2 Placement Fit Policy

Related to the replacement policy is a question of whether we should take into account the fit when considering Mblocks to evict. Since the Mblock placement is similar to the memory allocation, we can use traditional memory allocation policies such as best fit and worst fit. However, unlike the memory allocation problem, the size of free space depends on the size of victim Mblocks chosen by replacement policy. Thus, in many cases the placement fit policy is only usable for breaking the tie between several victim Mblock groups with the same replacement scores.

4.3 Dealing with Block Miss

Besides of the placement/replacement policy required when we have an Mblock miss, there is another type of DFP cache miss: a block miss. This is a situation where the Mblock being accessed is found in the DFP cache, but the block being accessed is not present in the Mblock. This situation occurs when an Mblock access pattern changes that the block is not anticipated by the footprint predictor. The footprint predictor predicts blocks that are useful based on the Mblock's access pattern in the past. However, when the access pattern of the Mblock changes, the processor may access a block that was not accessed in the past and a block miss occurs.

When there is a block miss, the block must be inserted into the Mblock. How this insertion is handled depends on whether there is a hole to accommodate the new block or not. Depending on the location of the block insertion, there are several cases to consider. A hole may be there from fragmentation created by a previous Mblock allocation. For example, if a three-block Mblock was evicted to make room for a two-block Mblock, then a hole is created in its place. Figure 3 (top diagram) illustrates the case where there is a hole to accommodate the block. The figure shows two Mblocks (E and F), each having three blocks, co-located on

Figure 3: Block miss handling in DFC.

a single cache line. There are two holes in the cache line, one between the two Mblocks. Suppose that the last block in Mblock E is accessed and is missed. In this case, the Mblock E reclaims the hole, and expands to accommodate the new block. The valid bits are updated accordingly.

Figure 3 (middle diagram) depicts a different case involving Mblocks G and H. In this case, there is no hole for the Mblock G to expand, hence, Mblock H needs to be removed and re-inserted at a different location to make room for the new block. The valid bits of Mblock G and the starting position of Mblock H are updated accordingly. This case is more expensive than the first one, since re-insertion requires reading a large number of bytes from a row for one block miss and writing them back to the row when the row is closed. However, we found that it does not happen often due to high footprint predictor’s accuracy. Furthermore, the overhead itself is very minor because it is overlapped with the miss latency; hence there is no critical-path delay. Likewise, if the block being missed is in the middle of the Mblock, the entire Mblock is removed and then re-inserted.

Figure 3 (bottom diagram) depicts another case involving Mblocks I and J. In this case, there is no hole anywhere in the cache line for the Mblock I to expand, hence, an Mblock needs to be evicted to make room. In the figure, Mblock J is evicted, and this creates space for the new block along with introducing new holes. Note that in this case, the replacement policy can be invoked and this situation can be handled as if there is an Mblock miss. This case is potentially the most costly because it results in an Mblock being evicted. However, to avoid the complexity and latency for handling a full Mblock miss, and counting on the low number of occurrences of this case, we simply choose to evict the

Mblock that gets in the way for the Mblock that needs to expand.

4.4 Early Footprint Update

So far we have discussed how an Mblock and block miss is handled. We will now discuss another aspect of the DFC, which is when the footprint predictor is updated. Our DFP predictor shares a similarity with the original footprint predictor [9], but differs in a key aspect. The DFP predictor is similar in the way the footprint predictor is indexed, using a combination of a part of the memory address (spatial offset) and program counter. Using the PC and spatial offset helps in getting a footprint prediction for highly reused code that works on different data items yet exhibiting the same access pattern. Some examples of such a scenario includes database scan and join operations [9].

In the Footprint cache and our basic DFC design, the footprint of an Mblock is updated when the Mblock is evicted from the cache. Blocks that were accessed during the Mblock’s residency in the DFC became the footprint for the Mblock, causing them to be fetched the next time the cache misses on the Mblock. However, if the cache size grows, Mblocks stay longer in the cache and are evicted less frequently, which delays or even avoids footprint updates. This slows down learning for the footprint predictor, and reduces its prediction coverage. Especially, if we consider a scenario where data access pattern due to code reuse is applied over a large data set, this slow footprint learning reduces the opportunities for making footprint prediction. To overcome this slow footprint learning, we propose *early footprint update* (EFU), which allows footprints to be updated even when an Mblock is not evicted from the cache yet. One naive way to update the footprint prediction is whenever there is a new block that is accessed in the Mblock. However, this will result in too frequent footprint table updates, which increases occupancy and power consumption of the FHT. Furthermore, updating the FHT too soon is risky because incomplete footprints reduces footprint prediction accuracy. Thus, we should design footprint update criteria that is not too aggressive and yet still allow early updates.

To achieve that goal, the criteria for an early update that we use is the number of accessed blocks in an Mblock during its residency in the DFC. Over time, during an Mblock’s residency in the cache, more and more of its blocks are accessed. Once the number of blocks accessed reaches a threshold, the footprint is updated. We refer to this as Early Footprint Update (EFU). This allows the footprint predictor to learn footprints early, increasing the footprint prediction coverage. However, the footprint predictor is possibly updated using incomplete access pattern information, hence EFU trades off accuracy for coverage. As long as the benefit from increased coverage outweighs the drawback from reduced accuracy, EFU improves performance. Through evaluation, we found that using multiple threshold values works well. For example, we may have threshold values of 16, 24, and 32. When the number of accessed blocks reaches 16 for an Mblock, we update the footprint predictor. Later, when the number of accessed blocks reaches 24, we re-update the footprint predictor, etc.

4.5 Overhead analysis

Cache designs normally have a straightforward way to locate a cache block from its physical address. However, in

order to access a specific block in DFC, it requires to calculate the exact bit location and counting the number of preceding ones in a bit vector in tag array. This additional computation requires 4 levels of parallel adders along with a masking logic to mask succeeding bits after the bit location [17]. Considering the inverter delay as 13.9ps by 2007 paper [26], 4 adders with one masking logic is roughly less than 0.6 ns. We believe that such logic is probably going to cost way less than half nanosecond if we consider the technology scaling. Thus, a cache block look up overhead is negligible.

While the replacement policies discussed here are more complex than a regular LRU, fortunately it is only used for the LLC which is much less frequently accessed than other cache levels. Moreover, the victim selection itself will be very fast, because all operations can be performed in parallel. The victim selection starts with making victim Mblock groups, where all victim Mblock groups can be made in one time SRAM access in parallel with a few additions and comparisons in case of MaxMin and LRU+. Later on, looking up the victim among groups can be executed in parallel as well, then the smallest group satisfying the incoming macro block size is selected. Thus, only one time SRAM access to every macro block tag in the set will be dominant time and all other operations are processed in parallel. Furthermore, the replacement policy computation can be overlapped with the LLC miss latency, so that it does not generate any critical-path delay.

5. METHODOLOGY

For evaluating DFC, we primarily target scale-out memory intensive server applications/benchmarks from Cloud-Suite 2.0 [4], including Data Analytics (DA), Data Caching (DC), Data Serving (DS), Software Testing (ST), Web Search (WS), and Graph Analysis (GA). All these benchmarks are run on a Simics 3.0.31 virtual machine with AMD64 CPU and 16GB main memory, running Linux OS Ubuntu 10. We run each server benchmark for about a few hours until each server benchmark enters the phase of processing client's requests. Then, we use 3 billion memory instructions to produce a trace. In the simulation, 2 billion memory instructions are used to warm up the cache structures and 1 billion memory instructions are used for detailed simulation. The trace is designed to be sufficiently comprehensive to enable timing and power simulation. It contains basic information such as the memory instruction type, program counter, and address. The trace also contains dependence information between memory instructions, the number of non-memory instructions separating two memory instructions broken into ones (directly or indirectly) dependent on the preceding memory instruction and ones that are independent.

The traces are then used for simulation of a baseline machine model shown in Table 1. We model a detailed memory hierarchy model, and off-chip and stacked DRAM timing model using DRAMSim2 [22]. To estimate execution time, we implement an abstract Tomasulo pipeline. The pipeline has a 128-entry instruction window. An instruction is entered into the window as it is issued, and is executed once the instruction it depends on has completed execution. All instructions are assumed to take one clock cycle to execute, except for memory instructions of which their execution times depend on the hit/miss outcome at various caches

and DRAMs. Therefore, while simple, our pipeline model takes into account both memory data dependence stalls and structural hazard stalls. We envision a 16-core system with SRAM L1 and L2 caches private to each core, and a stacked DRAM L3 cache shared by all cores. Since our trace was collected from a single core, we assume the L3 cache to be partitioned among all cores equally, and simulate just one core and one partition of the L3 cache. The main memory is assumed to be off-chip DRAM. DRAM parameters for both the L3 cache and the off-chip DRAMs are shown in the table.

Processor	16 cores 3.2GHz, 1 IPC
L1 I\$ and D\$	32KB, 4-ways, 64B Block 2 cycles each
L2 cache	Variable size and latency (Table 2)
Off-chip DRAM	16GB DDR3-1600 (800MHz), 1 channel 8 Banks per rank 16KB row-buffer
Stacked DRAM	DDR-like interface (1600MHz), 4 channels 16 Banks per rank 8KB row-buffer 128 bits per channel
t _{CAS} -t _{RCD} -t _{RP} -t _{RAS} t _{RC} -t _{WR} -t _{WTR} -t _{RTP} t _{RRD} -t _{FAW}	11-11-11-28 39-12-6-6 5-24

Table 1: The baseline system configuration.

For the DRAM L3 (LLC) cache, we implement six different schemes: Footprint cache [9], Unison cache [8], block-based DRAM cache (i.e. DRAM cache with a traditional 64-byte block size), an oracle DRAM cache that always hit and has no tag space overheads, our scheme DFC, and for comparison purpose, no L3 cache. For Footprint cache and Unison cache, we follow the optimized parameters from [8], where the cache is 4-way set associative, each DRAM row is 8KB, each Mblock is 2KB for Footprint cache, and each Mblock is 960B for Unison cache. For DFC, we implement a design with 8KB DRAM row which can hold anywhere from four Mblocks to eight Mblocks.

LLC size (MB)	32	64	128	256	512
Unison					
L2 Size (MB)	16	16	16	16	16
Latency (cycles)	13	13	13	13	13
Footprint					
L2 Size (MB)	15.80	15.59	15.22	14.45	12.94
Latency (cycles)	13	13	13	12	11
DFC					
L2 Size (MB)	15.58	15.16	14.34	12.73	9.55
Latency (cycles)	13	13	12	11	10
64B Based					
L2 Size (MB)	13.94	12	8.25	1	None
Latency (cycles)	11	11	10	9	None

Table 2: L2 cache size configurations as affected by a limited SRAM budget.

To have a fair comparison, we use a fixed SRAM budget of 16MB (1MB per core), which can be allocated in different proportions between the L2 cache and the tag array of the L3 stacked DRAM cache. As a result of the constant SRAM budget, the L2 cache capacity and latency and the

Figure 4: Speedup ratios of Footprint cache (Footprint), Unison (Unison), our design (DFC), and small-block cache (64B block), for when the L3 cache size is 64MB (a) or 256MB (b), relative to system with no stacked DRAM L3 cache. Full Stacked DRAM is an oracle L3 cache with zero miss rate and zero tag overheads.

L3 tag capacity and latency vary for different techniques, as shown in Table 2. As shown in the table, we tested all cache capacities from 32MB to 512MB (2MB to 32MB per core). For the block-based stacked DRAM L3 cache, its high tag space requirement caps the L2 cache capacity to 16MB per core.

6. EVALUATION

6.1 Overall Performance

Figure 4 shows the speedup ratio of Footprint cache (Footprint), Unison cache (Unison), our design (DFC with MaxMin), and 64-byte block size cache (64B block), when the L3 cache capacity per core is 64B (part (a)) or 256MB (part (b)). For reference, an oracle L3 cache with zero miss rate and zero tag overheads (Full Stacked DRAM) is also added to both figures. The speedups are relative to a baseline of a system with only off-chip DRAM.

Let us examine the 64MB L3 cache first. Here we can see that the oracle L3 cache can achieve a geometric mean (gmean) of speedups of 19%, but the speedups decrease to 8.8% when the L3 cache size is limited at 64MB with the 64-byte block size. Note that Footprint cache performs better than Unison cache, consistent with results reported in the Unison cache paper [8]. Although the way predictor reduces its hit latency, it shows that it still suffers from its high miss

latency. According to Unison cache paper, it starts to perform better than Footprint cache in cache size larger than 1GB. However, considering the diminishing returns of larger caches, we believe that such large cache is not practical and leaving the remainder of the stacked DRAM for part of main memory is a more viable choice. With a small block size, at 64MB, the SRAM budget needed for the L3 tag array only reduces the L2 cache size slightly, from 16MB to 12MB, so it produces an acceptable performance. The Footprint cache produces speedups in some applications but slowdowns in others, leading to a gmean speedup of 3.4%. Footprint cache produces a significant slowdown in graph analysis (GA) because only a few blocks in an Mblock are useful; in Footprint cache, such an Mblock is mostly empty but still occupies a 2KB space in the cache. Such internal fragmentation of Mblocks is eliminated in DFC, hence DFC performs much better than Footprint cache for this application. In contrast to the Footprint cache, DFC has no internal fragmentation. However, DFC may incur some external fragmentation when there are holes in a cache line that are not sufficiently large to an additional Mblock. 64-byte block cache has no fragmentation whatsoever, so it slightly outperforms DFC in few benchmarks (ST, DS, and WS). However, DFC outperforms 64-byte block cache overall (9.4% vs. 8.8% gmean). Both significantly outperform the Footprint cache because of their efficient use of L3 cache capacity.

Moving on to the 256MB L3 cache, the situation changes drastically. The 64-byte block L3 cache incurs much larger SRAM budget for the tag array (leaving the L2 cache with 1MB), resulting in slowdowns for half of the applications (DC, DS, and WS) and it performs worse than baseline overall. In contrast, DFC achieves 13.2% gmean speedup, and even reaching 20% for DA and DS. With 256MB, the L3 cache capacity is large enough to hold much of the working sets of the applications, such that even inefficient use of cache capacity in the Footprint cache allows it to perform quite well (11.1% gmean speedup).

In what seems like an anomaly in DA, DFC performs very slightly better than oracle. First, the L3 cache hit rates are already very high in DFC, hence there are not many L3 cache misses. Second, because the DFC is much smaller and has much fewer DRAM rows, there are cases where two Mblocks mapping to a single DRAM row, leading to more row buffer hits, and better performance in DFC.

Let us compare DFC vs. Footprint cache to understand better in what ways they differ. We will look at the cache miss ratio, footprint prediction accuracy and coverage. Table 3 shows the L3 cache miss ratios of Footprint cache vs. DFC. The data shows that DFC always achieve lower miss ratios vs. Footprint cache, achieving an average 10.9% better for 64MB L3 cache. The spread decreases to only 6.5% when the L3 cache size is 256MB, because more of the working sets of the applications fit in the L3 cache, reducing the capacity pressure that can be relieved by DFC.

The accuracy of Footprint cache and DFC prediction is shown in Table 4. The table shows accuracy (number of blocks of an Mblock fetched due to footprint prediction that are subsequently used by the processor divided by total blocks fetched), and under-prediction rate (number of blocks that are not fetched that are subsequently missed by the processor divided by total number of blocks fetched). Ideally, we want high accuracy and low under-prediction. Over-prediction (fetching too many blocks) is costly for DFC

App	64MB			256MB		
	FC	DFC	Spread	FC	DFC	Spread
DA	41.7	29.8	11.9	15.1	10.6	4.5
DC	34.9	19.0	15.9	13.3	10.2	3.2
GA	85.7	83.6	2.1	83.0	76.8	6.2
ST	48.2	41.5	6.7	31.1	18.4	12.8
DS	58.9	37.8	21.1	18.4	10.9	7.5
WS	53.1	45.5	7.6	17.7	12.8	5.0
Avg			10.9			6.5

Table 3: L3 miss ratios of Footprint cache (FC) vs. our DFC. All numbers are percentages.

	Footprint cache		DFC	
	Accur	Under	Accur	Under
DA	92.7	14.4	94.2	7.2
DC	70.1	38.2	73.3	27.4
GA	25.3	78.9	88.7	11.3
ST	69.4	37.5	63.6	38.1
DS	71.5	51.8	79.9	28.1
WS	69.1	47.3	72.0	36.0
	66.4	44.7	78.6	24.7

Table 4: Predictor accuracy comparison of Footprint cache vs DFC. All numbers are percentages.

because unused blocks consume bandwidth and L3 cache space, but is only slightly costly for Footprint cache as they do not change the L3 cache space usage. However, under-prediction is very costly for both schemes because if the processor accesses a block that has not been predicted useful, even when the Mblock is present, the block has to be fetched from the off-chip DRAM with great latencies. Thus, under-prediction is critical to performance. The table shows that DFC achieves about half the under-prediction rate as Footprint cache (44.7% vs. 24.7%). One reason for these excellent numbers for DFC is that Mblocks stay longer in the L3 cache due to more efficient usage of cache capacity, and thus have a much higher chance of capturing the entire footprint behavior of an Mblock. To illustrate an example, suppose the first half of an Mblock is accessed first, and the second half of an Mblock is accessed last. If during this time the Mblock resides in the cache, when the Mblock is evicted, its footprint is accurately captured and reflects all the blocks that were accessed. However, if the Mblock only resides for half as long, when it is evicted, its footprint only reflects one half of the access, and the next time it is brought in and evicted, the footprint reflects the other half. Thus, footprint learning and prediction is more stable and more accurate in DFC.

Energy consumption. For energy simulation, we measured the energy consumption from L2 cache, off-chip DRAM, and stacked DRAM and sum them. Compared to Footprint cache, DFC reduces the energy consumption from 0.8% to -22.5% at 64MB depending on benchmarks and average -4.4%. Most of energy saving originates from reducing off-chip memory traffic since DFC keeps Mblocks in the cache longer and hence incur fewer cache misses. A secondary effect comes from DFC’s higher footprint prediction accuracy which increases row buffer hit rate and reduces the number of DRAM row activations, which are the highest energy required operation [9, 8, 25]. However, the energy gains are

(a) Miss ratios

(b) Speedups

Figure 5: Comparing LRU+, MaxAvg, and MaxMin replacement policies for a 64MB DFC in terms of miss ratios (a) and speedups over Footprint cache (b).

decreased to -0.1% at 256MB dram cache, which is due to increased SRAM cache misses in DFC and narrowed DRAM cache miss difference between Footprint cache and DFC.

6.2 Replacement Policy Comparison

Figure 5 compares the LRU+, MaxAvg, and MaxMin replacement policies for a 64MB DFC in terms of miss ratios (a) and speedups over Footprint cache (b). The figure shows that replacement policies show comparable performance for DC, GA, and ST; but produce different performance for DA, DS, and WS. When the performance of different policies differ, MaxMin achieves the best performance (lowest miss ratios and highest speedups), followed by MaxAvg and finally by LRU+. As we discussed in Section 4.1, MaxMin exhibits the strongest guarantee of not evicting nearly half of the most recently used Mblocks, whereas LRU+ and MaxAvg do not even guarantee that the MRU Mblock will not be evicted. To investigate why MaxAvg slightly outperforms LRU+, we collect statistics on the number of times the MRU Mblock gets evicted. The statistics shows that MaxAvg evicts the MRU Mblock only 12.3–19.1% of the number of MRU Mblock evictions performed by LRU+. It also confirms that MaxMin never evicted MRU Mblock in any of these benchmarks.

6.3 Placement Policy

When an Mblock needs to be allocated in the cache, and several candidate groups of Mblocks have equal replacement scores, we can break the tie by using the best fit or worst fit

Figure 6: Miss-ratio Comparison for Early Footprint Update

policy. In worst fit, the largest Mblock group is evicted so that even when a part of the freed capacity is used for the incoming Mblock, the leftover portion may be large enough for allocation of another Mblock. In storage allocation literature, best fit is cited as leaving a lot of small segments that are not usable, such that worst fit fares better. In DFC, however, blocks evicted by worst fit may have to be refetched, incurring significant memory bandwidth and latency to refetch them. We test these policies and found that worst fit performs very slightly worse: it shows up to 4% higher miss ratio compared to best fit.

6.4 Early Footprint Update

For the evaluation, we set up three early update thresholds, 16 blocks, 24 blocks, and 32 blocks. At every time the number of blocks in Mblock reaches one of thresholds, the Mblock’s footprint is updated in the footprint history table (FHT). Figure 6 shows the miss ratio improvement of DFC with EFU compared to DFC without EFU in 64MB stacked DRAM. Because the trend is almost similar in all three replacement policies, we simply select MaxMin to draw the graph. EFU helps to expedite footprint updates which can be delayed by the large DRAM size. Therefore, if the L3 size is large enough to slow down footprint updates, EFU can increase prediction coverage, which leads to lower miss ratio. From the figure, we can see that 0–2.2% reduction in miss ratio.

To investigate further, we measure footprint coverage (number of Mblock misses that can be predicted by the footprint predictor). With the EFU, we expect increased coverage. We found that coverage without EFU is already nearly 100% for DC, GA, and ST; hence EFU does not make much differences. However, for DA, DS, and especially WS, the footprint coverage increases measurably (2.1%, 2.6%, and 8.4%, respectively).

7. SENSITIVITY STUDY

Sensitivity to Additional Tags. In DFC, the number of tags should be carefully decided based on the SRAM budget because its performance is highly related to it. If DFC has too many tags, it can keep more Mblocks but also requires larger SRAM space for the additional tags that might not be used. In contrast, if the number of tags is too small, the L2 cache can be made larger, but the DFC can keep fewer Mblocks. Thus, we evaluated the impact of tag counts

Figure 7: Performance Comparison over number of tags difference per set

Figure 8: Performance Comparison under memory contention

per set. Figure 7 depicts the impact of tag count per set. Footprint cache has a fixed 4 tags per set, while for DFC we can vary the number of tags per set. The figure shows the speedups for different L3 cache size and number of tags, averaged over all benchmarks. We can make several conclusions. First, 8 tags performs the best overall across all benchmarks, especially for larger DFC. For smaller DFC, the impact from tag array overhead is small, so it favors 10 tags per set. Note that this means that overall, DFC can keep between up to 2 – 2.5× more Mblocks than Footprint cache.

Sensitivity to DRAM Speed. DRAM can process only one request in one channel. In our previous experiments, we expected that there is no contention in DRAM and only divide DRAM capacities by cores. However, if the number of cores grows, the request contention happens and a request should wait until its previous request is processed. Under heavily contended circumstances, we expect that DFC performs better because stacked DRAM handles the larger fraction of traffic compared to off-chip DRAM. We simulate the impact of DRAM contention by increasing tRC, the time interval between accesses to different rows in a bank. Footprint predictor issues burst of memory operations, but these are not affected by this limitation, because these requests affect the same DRAM row. Figure 8 shows the speedups of DFC normalized to Footprint cache. The figure shows that as the DRAM’s tRC increases, DFC’s speedups relative to Footprint Cache increase. This is expected, because DFC reduces the number of off-chip memory accesses.

8. CONCLUSION

In this paper, we have presented *Dense footprint cache* (DFC), a new last level cache design utilizing stacked DRAM. DFC minimizes the SRAM budget for the LLC tag array by utilizing large blocks (Mblocks) of a few KBs in size. It uses prediction of which 64-byte blocks within an Mblock will be used by the processor and only fetch them into the cache. These predicted-useful blocks are stored in a compact way, allowing capacity-efficient utilization of the LLC. Due to the variable actual Mblock space in the cache, DFC introduces new challenges in designing the replacement and placement policies. We have proposed three new replacement policies (LRU+, MaxAvg, and MaxMin), derived their properties analytically, and evaluated them with simulations. We have also proposed placement fit policies (best fit vs. worst fit), and early footprint update (EFU) policy. Through simulation studies with Big Data applications, we showed that DFC reduces LLC miss ratios by about 43%, speeds up applications by 9.5%, while consuming 4.3% less energy on average.

9. REFERENCES

- [1] C. Chou, A. Jaleel, and M. K. Qureshi. Bear: techniques for mitigating bandwidth bloat in gigascale dram caches. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, Jun. 2015.
- [2] Y. Deng and W. Maly. Interconnect characteristics of 2.5-d system integration scheme. In *Proceedings of the Intl. Symp. on Physical Design*, pages 171-175, Apr. 2001.
- [3] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010.
- [4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.
- [5] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.
- [6] S. Gupta, M. Hilbert, S. Hong, and R. Patti. Techniques for producing 3d ics with high-density interconnect. In *Proceedings of the 21st Intl. VLSI Multilevel Interconnection Conference*, Sep. 2004.
- [7] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth generation intel core processor. *IEEE Micro*, Vol. 34, Issue 2, April 2014.
- [8] D. Jevdjic, G. Loh, C. Kaynak, and B. Falsafi. Unison cache: A sacable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014.
- [9] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, Jul. 2013.
- [10] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, Jan. 2010.
- [11] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. Picoserver: Using 3d stacking technology to enable a compact energy efficient chip multiprocessor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.
- [12] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A fully associative, tagless dram cache. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, Jun. 2015.
- [13] C. Liu, I. Ganusov, and M. Burtcher. Bridging the processor memory performance gap with 3d ic technology. In *IEEE Design & Test of Computers*, Nov-Dec. 2005.
- [14] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th International Symposium on Computer Architecture*, Jun. 2008.
- [15] G. H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Proceedings of the 42nd International Symposium on Microarchitecture*, Dec. 2009.
- [16] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th International Symposium on Microarchitecture*, Dec. 2011.
- [17] P. Mehrotra and P. D. Franzon. Novel hardware architecture for fast address lookups. *IEEE Communications Magazine*, Vol. 40, Issue 11, Nov. 2002.
- [18] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *Proceedings of 21st International Symposium on High Performance Computer Architecture*, Feb. 2015.
- [19] M. Oskin and G. H. Loh. A software-managed approach to die-stacked dram. In *PACT '15 Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2015.
- [20] M. Qureshi and G. H. Loh. Fundamental latency trade-offs in architecting dram caches. In *Proceedings of the 45th International Symposium on Microarchitecture*, Dec. 2012.
- [21] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *Proceedings of the 36th International Symposium on Computer Architecture*, Jun. 2009.
- [22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Drsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, Vol. 10, Issue 1, Mar. 2011.
- [23] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *ISCA '94 Proceedings of the 21st annual international symposium on Computer architecture*, Apr. 1994.
- [24] Y. Solihin. Sectored cache with a tag structure capable of tracking sectors of data stored for a particular cache way. *U.S. Patent 8,874,849*, Oct 28 2014.

- [25] S. Volos, J. Picorel, B. Grot, and B. Falsafi. Bump: Bulk memory access prediction and streaming. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.
- [26] K. von Arnim, E. Augendre, C. Pacha, T. Schulz, K. T. San, F. Bauer, A. Nackaerts, R. Rooyackers, T. Vandeweyer, B. Degroote, N. Collaert, A. Dixit, R. Singanamalla, W. Xiong, A. Marshall, C. R. Cleavelin, K. Schrufer, and M. Jurczak. A low-power multi-gate fet cmos technology with 13.9ps inverter delay, large-scale integrated high performance digital circuits and sram. *VLSI Technology, 2007 IEEE Symposium on*, pages 106-107, Jun. 2007.
- [27] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, Jan. 2010.
- [28] Y. Xie, G. H. Loh, B. Black, and K. Bernstein. Design space exploration for 3d architecture. In *ACM Journal of Emerging Technologies in Computer Systems*, 2(2):65-103, Apr. 2006.
- [29] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring dram cache architectures for cmp server platforms. In *Proceedings of the 25th International Conference on Computer Design*, Oct. 2007.