

A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch

Jaewoong Sim* Gabriel H. Loh† Hyesoon Kim* Mike O’Connor† Mithuna Thottethodi‡

*School of ECE
Georgia Institute of Technology

*College of Computing

{jjaewoong.sim,hyesoon.kim}@gatech.edu

†AMD Research

Advanced Micro Devices, Inc.

{gabe.loh,mike.oconnor}@amd.com

‡School of ECE

Purdue University

mithuna@purdue.edu

Abstract

Die-stacking technology allows conventional DRAM to be integrated with processors. While numerous opportunities to make use of such stacked DRAM exist, one promising way is to use it as a large cache. Although previous studies show that DRAM caches can deliver performance benefits, there remain inefficiencies as well as significant hardware costs for auxiliary structures. This paper presents two innovations that exploit the bursty nature of memory requests to streamline the DRAM cache. The first is a low-cost Hit-Miss Predictor (HMP) that virtually eliminates the hardware overhead of the previously proposed multi-megabyte MissMap structure. The second is a Self-Balancing Dispatch (SBD) mechanism that dynamically sends some requests to the off-chip memory even though the request may have hit in the die-stacked DRAM cache. This makes effective use of otherwise idle off-chip bandwidth when the DRAM cache is servicing a burst of cache hits. These techniques, however, are hampered by dirty (modified) data in the DRAM cache. To ensure correctness in the presence of dirty data in the cache, the HMP must verify that a block predicted as a miss is not actually present, otherwise the dirty block must be provided. This verification process can add latency, especially when DRAM cache banks are busy. In a similar vein, SBD cannot redirect requests to off-chip memory when a dirty copy of the block exists in the DRAM cache. To relax these constraints, we introduce a hybrid write policy for the cache that simultaneously supports write-through and write-back policies for different pages. Only a limited number of pages are permitted to operate in a write-back mode at one time, thereby bounding the amount of dirty data in the DRAM cache. By keeping the majority of the DRAM cache clean, most HMP predictions do not need to be verified, and the self-balancing dispatch has more opportunities to redistribute requests (i.e., only requests to the limited number of dirty pages must go to the DRAM cache to maintain correctness). Our proposed techniques improve performance compared to the MissMap-based DRAM cache approach while simultaneously eliminating the costly MissMap structure.

1. Introduction

Advances in die-stacking technologies have made it possible to integrate hundreds of megabytes, or even several gigabytes, of DRAM within the same package as a multi-core processor [1, 8, 11, 20] or a GPU [19]. To avoid dependencies on operating system vendors, maintain software transparency, and provide benefit to legacy software, recent papers have suggested that using die-stacked DRAM as a large cache is a compelling approach [11, 12].

To reduce the overhead of supporting the tags in large, die-stacked DRAM caches, recent work has considered embedding the tags directly alongside the data within the DRAM array, which avoids the need for a dedicated external SRAM tag array (e.g., 96MB for a 1GB DRAM cache) [4, 11]. Placement of the tags within the die-stacked

DRAM itself incurs a costly DRAM access even in the case where the request eventually misses in the cache. Loh and Hill proposed the MissMap, a multi-megabyte structure (much less overhead than a dedicated SRAM tag array) that allows the DRAM cache controller to skip the DRAM cache access when there is a cache miss [11].

While the MissMap provides a far more practical approach than using a massive SRAM tag array, its implementation cost is still likely to be prohibitively high to allow it to be deployed in commercial products (e.g., 4MB for a 1GB DRAM cache). Furthermore, the access latency of the MissMap is not trivial (the original paper used a latency of 24 cycles, which is added to *all* DRAM cache hits *and* misses). In this work, we point out that the MissMap approach is overly conservative (i.e., maintaining precise information about the DRAM cache’s contents is not necessary) and that it is actually possible to *speculate* on whether a request can be served by the DRAM cache or main memory. We introduce a light-weight, low-latency *Hit-Miss Predictor* (HMP) that provides 97% accuracy on average, with a hardware cost of less than 1KB.

We also propose a self-balancing dispatch (SBD) mechanism that dynamically steers memory requests to either the die-stacked DRAM cache or to the off-chip main memory depending on the instantaneous queuing delays at the two memories. While the stacked DRAM can provide higher bandwidth than the off-chip memory, overall system bandwidth would be greater yet if *both* die-stacked and off-chip memories could be efficiently exploited at the same time.

While the HMP and SBD techniques can potentially streamline the design of a DRAM cache, these approaches are only useful if they can still ensure correct execution. The source of potential complication comes from dirty/modified data in the DRAM cache. Both the HMP and SBD can potentially send a request to main memory when the DRAM cache contains the most-recent, modified value. Returning the stale value from off-chip memory could then lead to incorrect program execution. Beyond a basic mechanism to validate predictions, we also introduce a hybrid write policy that forces the majority of the DRAM cache to operate in a write-through mode, and only enables write-back for a limited set of pages that have high write traffic. This results in a DRAM cache that is mostly clean, thereby allowing the DRAM cache to avoid waiting on HMP prediction verification and creates more opportunities for SBD to freely send requests off-chip.

2. Background

2.1. DRAM Architectures

Most conventional caches are implemented with SRAM technology, whereas this work considers die-stacked DRAM. DRAM consists of arrays of bit-cells, where each bit-cell is comprised of a capacitor to store charge and an access transistor to enable reading/writing of the cell. Accessing bit-cells in a DRAM requires storing the bit-cell values in a row buffer, and all read and write operations effectively

operate directly on the row buffer (rather than the actual bit-cells). When, the row is no longer needed (or often when a different row is requested), the contents of the row buffer are written back into the original row of bit-cells and then a new row may be accessed.

The DRAM access mechanism is quite different from SRAM arrays. In the case of DRAM, an entire bank is occupied while the row is open, and therefore any requests to other rows in this bank will be delayed until the current operations complete (although operations in independent banks may proceed concurrently subject to DRAM timing and bus constraints). In an SRAM, the access paths are more easily pipelined, and so even if a request has been sent to a particular bank, subsequent requests need only wait a few cycles before they can proceed.

2.2. Die-stacked DRAM Caches

Caches store two types of information: tags and data. In conventional SRAM-based caches, these are stored in two physically distinct structures (the tag and data arrays, respectively). For a DRAM cache, one could consider an SRAM-based tag array, as shown in Figure 1(a), but previous estimates have shown that such a structure would require tens of megabytes of SRAM, and therefore this approach is not practical considering that current L3 cache sizes are typically only around 8 MB [2, 3].

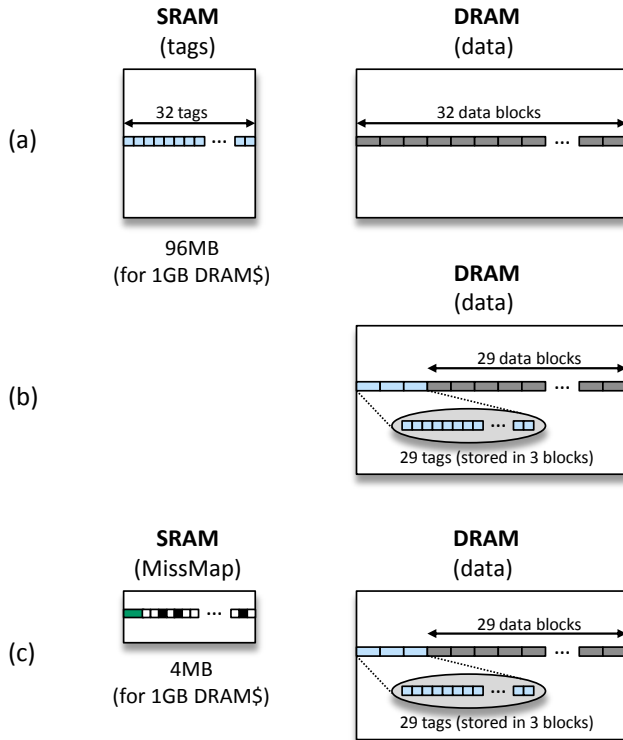


Figure 1: DRAM cache organizations using (a) an SRAM tag array, (b) tags embedded in the DRAM, and (c) tags in DRAM with a MissMap.

Instead, recent research has considered organizations where the tags and data are directly co-located within the die-stacked DRAM, as shown in Figure 1(b) [4, 11]. While this eliminates the unwieldy SRAM tag array, it introduces two more problems. First, naively accessing the cache could double the DRAM cache latency: one

access to read the tags, followed by another access to read the data (on a hit). Second, for cache misses, the cost of the DRAM cache tag access is added to the overall load-to-use latency.

Loh and Hill observed that the tags and data reside in the same DRAM row, and so the actual latency of a cache hit can be less than two full accesses by exploiting row buffer locality [11]. That is, the DRAM row is opened or activated into the row buffer only once, and then tag and data requests can be served directly out of the row buffer at a lower latency compared to two back-to-back accesses to different rows. They also proposed a hardware data structure called a MissMap that precisely tracks the contents of the DRAM cache, as shown in Figure 1(c). Before accessing the DRAM cache, the MissMap is first consulted to determine whether the requested cache block is even resident in the cache. If the block is not in the cache (i.e., miss), the request can be sent directly to main memory without incurring the tag-check cost of the DRAM cache. For a 512MB DRAM cache, the MissMap needs to be about 2MB in size (which provides tracking of up to 640MB of data), and a 1GB cache would need a 4MB MissMap. While Loh and Hill argue that part of the L3 cache could be carved out to implement the MissMap, using the AMD Opteron™ processor that consumes 1MB of its L3 to implement a “Probe Filter” as an example [2], it seems unlikely that designers would be willing to sacrifice *half* of their L3 to implement the MissMap.¹

3. Motivation

In this section, we identify inefficiencies with the previously proposed DRAM cache organizations. First, we explain why the MissMap is overly conservative, which ultimately leads us to consider more speculative techniques with significantly lower overheads (both in terms of hardware cost and latency). Second, we describe scenarios where a conventional cache organization under-utilizes the available aggregate system bandwidth, which motivates our proposal for a *Self-Balancing Dispatch* mechanism. Third, we discuss how the presence of dirty/modified data in the DRAM cache can potentially limit how aggressively we can speculate on or rebalance DRAM cache requests.

3.1. The Overkill of the MissMap

The MissMap tracks memory at page (or other coarse-grain) granularity. Each MissMap entry consists of a tag that stores the physical page number, and a bit-vector that records which cache blocks from this page are currently resident in the DRAM cache. The bit vector is precisely maintained such that each time a new cache block is inserted, its corresponding bit in the vector will be set; conversely, when a cache block is evicted, its bit will be cleared. Furthermore, if a MissMap entry is evicted, then *all* dirty lines from the corresponding victim page must also be evicted and written back.

Loh and Hill mentioned that it is possible to allow the MissMap to have false positives [11]. That is, if the MissMap says that a block is present in the DRAM cache when in fact it is not, then there is only a performance impact as the system needlessly pays for the latency of the DRAM cache before going to main memory. However, if the MissMap reports that a line is *not* present when in fact it is (false negative), the request would be sent to main memory and returned to the processor. If the DRAM cache contains this block in a dirty state, then this can lead to incorrect program execution.

On a DRAM cache miss (whether the MissMap said so or not), the system sends the request to main memory. When the response

¹ Assuming a 4MB MissMap to support a 1GB DRAM cache and a baseline L3 cache size of 8MB. If such a system employed a Probe Filter as well, then only 3MB out of the original 8MB L3 would actually be available as a cache!

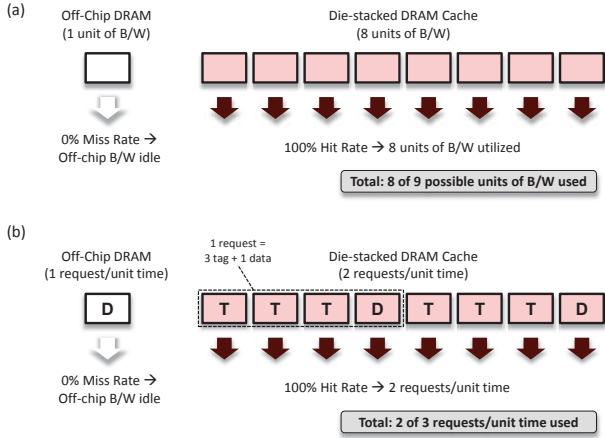


Figure 2: Example scenario illustrating under-utilized off-chip memory bandwidth in the presence of very high DRAM cache hit rates when considering (a) raw bandwidth in Gbps, and (b) in terms of request service bandwidth.

returns, the data are sent back to the L3 and the processor, and the data are also installed into the DRAM cache.² Prior to the installation of a new cache block, a victim must be selected. Furthermore, if the victim has been modified, then it must also be written back to main memory.

Note that when selecting a victim, the DRAM tags are checked. Therefore, if the system issued a request to main memory even though a modified copy of the block is in the DRAM cache, this can still be detected at the time of victim selection. Given this observation, the constraint that the MissMap *must* not allow false negatives is overly conservative. False negatives are tolerable so long as responses from memory are not sent back to the processor before having verified that a dirty copy does not exist in the DRAM cache.

Based on these observations, we propose a DRAM cache organization that can speculatively issue requests directly to main memory regardless of whether the decisions are “correct” or not. Section 4 describes a predictor design that exploits spatial correlation and the bursty nature of cache traffic to provide a light-weight yet highly accurate DRAM cache hit-miss predictor.

3.2. Under-utilization of Aggregate System Bandwidth

Die-stacked DRAM can potentially provide a substantial increase in memory bandwidth. Previous studies have assumed improvements in latency of $2\times$, $3\times$ and as much as $4\times$ compared to conventional off-chip DRAM [8, 11, 20]. At the same time, the clock speed can be faster, bus widths wider, and independent channels more numerous [9, 11]. Even with a rough estimate of half the latency, twice the channels, and double-width buses (compared to conventional off-chip memory interfaces), the stacked DRAM would provide an $8\times$ improvement in bandwidth. In an “ideal” case of a DRAM cache with a 100% hit rate, the memory system could provide an eight-fold increase in delivered bandwidth, as shown in Figure 2(a). However, the off-chip memory is completely idle in this scenario, and that represents 11% ($\frac{1}{1+8}$) of the overall system bandwidth that is being wasted.

Figure 2(b) shows the same scenario again, but instead of raw bandwidth (in terms of Gbps), we show the *effective bandwidth* in

²For this study, we assume that all misses are installed into the DRAM cache. Other policies are possible (e.g., write-no-allocate, victim-caching organizations), but these are not considered here.

terms of requests serviced per unit time. Note that a request to main memory only requires transferring a single 64B cache block, whereas a request to a tags-in-DRAM cache requires transferring three tag blocks (64B each) and finally the data block. Therefore, the sustainable effective bandwidth of the DRAM cache is only twice that of the off-chip memory ($8\times$ the raw bandwidth, but $4\times$ the bandwidth-consumption per request). In this case, a 100%-hit rate DRAM cache would leave 33% of the overall effective bandwidth unused ($\frac{1}{1+2}$). While the DRAM cache typically does not provide a 100% hit rate, hits often come in bursts that can lead to substantial queuing delays from bank and bus contention.

Apart from the available bandwidth, bank and bus conflicts at the DRAM cache can lead to increased queuing delays, some of which could potentially be mitigated if some of these requests could be diverted to the off-chip memory. In practice, other timing constraints, resource conflicts, and specific access patterns and arrival rates would affect the exact amount of bandwidth available for both the DRAM cache and the off-chip memory. However, this simple example highlights that there will be times where the system will have some idle resources, and we propose a Self-Balancing Dispatch technique to capitalize on these resources.

3.3. Obstacles Imposed by Dirty Data

Dirty data in the DRAM cache can severely restrict the aggressiveness of speculatively sending requests to main memory, as the copy in main memory is stale and its usage can result in incorrect executions. Likewise, dirty data prevents the system from exploiting idle main-memory bandwidth because accesses to dirty data must be sent to the DRAM cache regardless of how busy the DRAM cache or how idle the off-chip memory is. This also raises the question as to how the system can know ahead of time that a request targets a dirty cache line without having first looked up in the cache to see if the line is present and dirty. A key contribution of this work is a new way to operate the DRAM cache (which could be applied to other types of caches) such that most of the cache will be clean, and for the majority of the cache, we can guarantee its cleanliness without having to check the cache’s tags. This removes major limitations for both cache hit speculation and Self-Balancing Dispatch.

4. DRAM Cache Hit Speculation

The previously proposed MissMap provides precise tracking of DRAM cache contents, but as a result, the size (2-4MB) and latency (tens of cycles) of the structure introduce significant overheads. Section 3 explained how the DRAM cache can check for the existence of a dirty block at the time of a cache fill, and how this allows the DRAM cache to speculatively send requests to main memory so long as we ensure that the data are not returned to the processor until it has been verified that a modified copy is not also in the DRAM cache. In this section, we present the designs for lightweight and accurate region-based predictors that exploit the bursty nature of cache hits and misses [10].

4.1. Region-based Hit/Miss Prediction

Our region-based Hit/Miss Predictor (HMP_{region}) is structurally similar to a classic bimodal branch predictor [15]. The predictor itself consists of a table of two-bit saturating counters. For a DRAM cache with millions of cache blocks, it is not practical to directly index into the HMP_{region} table with a hash of the raw physical address; the aliasing and interference would render the predictor table nearly

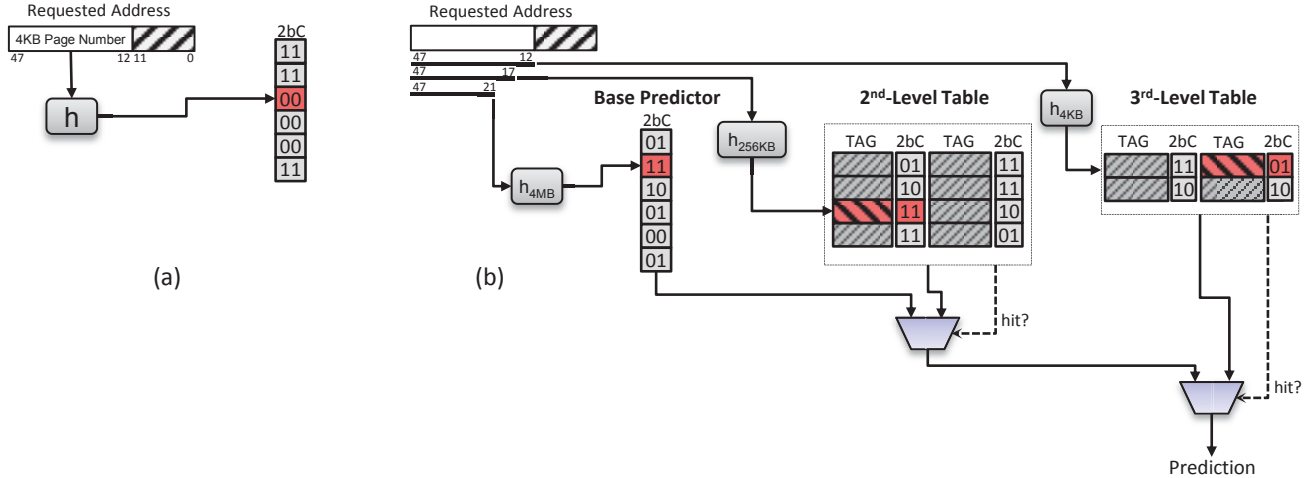


Figure 3: Hit-Miss Predictor designs: (a) one-level HMP_{region} , (b) multi-granular HMP_{MG} .

useless, or a gigantic table would be needed. Instead, we break up the memory space into coarser-grained regions (e.g., 4KB), and index into the HMP_{region} with a hash of the region’s base address as shown in Figure 3(a). This allows the HMP_{region} table to be much smaller, but it also means that *all* accesses within a region will follow the same prediction. The operation of the HMP_{region} is otherwise analogous to the bimodal predictor: DRAM cache hits increment the predictor, and misses decrement the predictor (saturating at 3 or 0, respectively).

The coarse-grained predictor organization of HMP_{region} is actually a benefit rather than a shortcoming. Accesses tend to exhibit significant spatial locality, especially at lower-level caches such as a large DRAM cache. Figure 4(a) shows the number of cache blocks present in the DRAM cache for one particular 4KB page of *leslie3d* (from the WL-6 workload) with respect to the number of accesses to this page (our methodology is explained in Section 7). Initially, nothing from this page is in the DRAM cache, but as the page is used, more and more lines are installed. During this installation phase, most accesses result in cache misses, and a simple 2-bit counter corresponding to this region would mostly predict that these requests result in misses. After this “warm up” phase, the footprint from this region is stable, and all subsequent accesses to this region result in hits. Again, a simple 2-bit counter would quickly switch over to predicting “cache hit” for this region and achieve high accuracy. When the application is finished with using this region, the contents will gradually get evicted from the cache, as shown by the drop back down to zero. At some future point,³ the page becomes hot again and the process repeats.

Figure 4(b) shows another 4KB region taken from *leslie3d* (from the same workload WL-6). This is just to illustrate that different regions and different applications may show different types of patterns, but so long as there exist sustained intervals where the curve is consistently increasing (mostly misses) or is consistently flat (mostly hits), then the simple 2-bit counter approach will be effective for making hit-miss predictions for the region.

The HMP_{region} approach is different from other previously proposed history-based hit-miss predictors. Past work has considered hit-miss predictors for L1 caches based on PC-indexed predictor

³The figure’s x-axis is based on accesses to the page. The time that elapses from the last access in the hit phase until the first access in the miss phase could easily span many thousands or even millions of cycles, but this all gets compressed here.

organizations [18]; such an approach may not be as easy as to implement for a DRAM cache because PC information is not typically passed down the cache hierarchy, may not exist for some types of requests (e.g., those originating from a hardware prefetcher), or may not be well defined (e.g., a dirty cache block being written back to the DRAM cache that was modified by two or more store instructions may have multiple PCs associated with it).

4.2. Multi-Granular Hit-Miss Predictor

The HMP_{region} predictor requires approximately one two-bit counter per region. For example, assuming a system with 8GB of physical memory and a region size of 4KB, the HMP_{region} would still need 2^{21} two-bit counters for a total cost of 512KB of storage. While this is already less than a 2-4MB MissMap, there is still room to further optimize.

We observed that even across large contiguous regions of memory spanning multiple physical pages, the hit-miss patterns generally remained fairly stable (that is, sub-regions often have the same hit-miss bias as other nearby sub-regions). While, in theory, different nearby physical pages may have nothing to do with each other (e.g., they may be allocated to completely independent applications), in practice memory allocation techniques such as page-coloring [17] tend to increase the spatial correlation across nearby physical pages. In our experiments, we found that memory would often contain large regions with *mostly* homogeneous hit/miss behavior, but smaller pockets within the larger regions would behave differently.

We propose a *Multi-Granular Hit/Miss Predictor* (HMP_{MG}) that is structurally inspired by the TAGE branch predictor [13], but operates on the base addresses of different memory regions (as opposed to branch addresses) and the different tables capture hit-miss patterns corresponding to different region sizes (as opposed to branch history lengths). Figure 3(b) shows the hardware organization of HMP_{MG} . The first-level predictor is similar to HMP_{region} , except that it makes predictions over very large 4MB regions. The second and third-level tables consist of set-associative tagged-structures that make predictions on finer-grained 256KB and 4KB region sizes, respectively. Each entry in the tagged tables consists of a (partial) tag and a two-bit counter for prediction. Tag hits in the tagged HMP_{MG} tables will override predictions from larger-granularity predictor tables.

The overall structure of the HMP_{MG} provides a more efficient

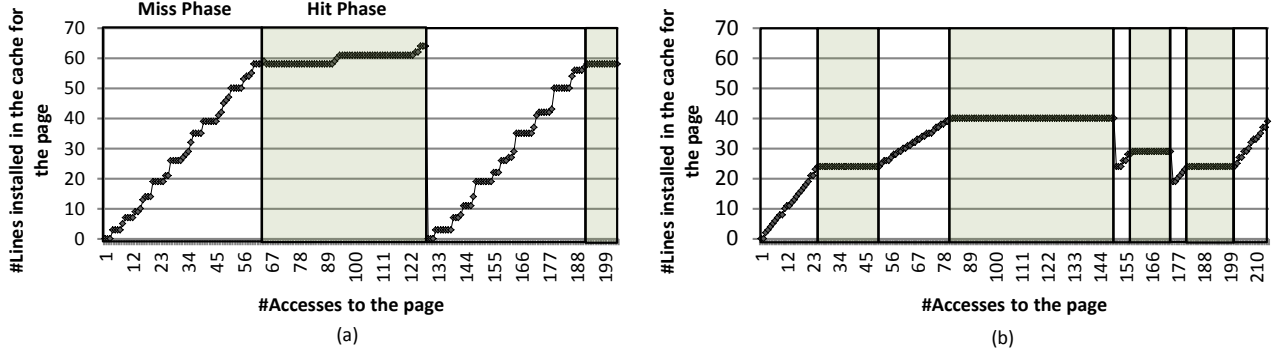


Figure 4: Hit and miss phases for two example pages from leslie3d (when run as part of the multi-programmed workload WL-6).

and compact predictor organization. A single two-bit counter in the first-level table covers a memory range of 4MB. In the single-level HMP_{region} predictor, this would require 1024 counters to cover the same amount of memory.

4.3. Predictor Operation

The entries in the HMP_{MG}'s base predictor are initially set to weakly miss or "1". To make a hit/miss prediction for a request, the base and tagged components are all looked up simultaneously. The base component makes a default prediction, while tagged components provide an overriding prediction on a tag hit.

The HMP_{MG} is updated when it has been determined whether there was a DRAM cache hit or not. The 2-bit counter of the "provider" component is always updated.⁴ On a misprediction, an entry from the "next" table is allocated (a victim is chosen based on LRU). For example, if the prediction came from the first-level table, then a new entry will be allocated in the second-level table. Mispredictions provided by the third table simply result in the corresponding counter being updated without any other allocations. The newly allocated entry's 2-bit counter is initialized to the weak state corresponding to the actual outcome (e.g., if there was a DRAM cache hit, then the counter is set to "weakly hit" or 2).

4.4. Implementation Cost

Table 1 shows the storage overhead of the HMP_{MG} configuration used in this paper. Compared to a MissMap that requires 2-4MB of storage, the HMP_{MG} only requires 624 bytes of total storage. A single predictor is shared among all cores. At this size, the entire L3 cache can once again be used for caching (as opposed to implementing a MissMap). Also important, the small size of the HMP_{MG} allows it to be accessed in a single cycle as it is smaller than even many branch predictors. Compared to the 24-cycle latency assumed for the MissMap [11], this provides significant benefits both for performance and implementability.

Hardware	Size
Base Predictor (4MB region)	1024 entries * 2-bit counter = 256B
2nd-level Table (256KB region)	32 sets * 4-way * (2-bit LRU + 9-bit tag + 2-bit counter) = 208B
3rd-level Table (4KB region)	16 sets * 4-way * (2-bit LRU + 16-bit tag + 2-bit counter) = 160B
Total	624B

Table 1: Hardware cost of the Multi-Granular Hit-Miss Predictor.

⁴The "provider" is the terminology used for the TAGE predictor to indicate the table from which the final prediction came from.

5. Exploiting Unused Bandwidth

As described in Section 3, there are scenarios where a burst of DRAM cache hits (or predicted hits for that matter) can induce significant DRAM cache bank contention while the off-chip memory remains largely idle. In this section, we describe a *Self-Balancing Dispatch* (SBD) mechanism that allows the system to dynamically choose whether (some) requests should be serviced by the DRAM cache or by the off-chip memory.

In an ideal case, every request could be routed to either the DRAM cache or to off-chip memory. If both memories had the same latency per access, then the system could simply look at the number of requests already enqueued for each and send the request to the one with fewer requests. However, the different memories have different latencies, and so the request should be routed to the source that has the lowest *expected* latency or queuing time. The expected latency for each memory can be estimated by taking the number of requests already "in line" and then multiplying by the average or typical access latency for a single request at that memory. Overall, if one memory source is being under-utilized, then it will tend to have a lower expected latency and the SBD mechanism will start directing requests to this resource. In the steady-state, the bandwidth from both sources will be effectively put to use.

Complications arise due to the fact that not every request can or should be freely routed to whichever memory has the lowest expected latency. If a request is for a dirty block in the DRAM cache, then routing the request to the off-chip memory is of no use (in fact, it just wastes bandwidth) because the data must ultimately come from the DRAM cache. If the HMP predicts that a request will miss in the DRAM cache, then there is likely little benefit in routing it to the DRAM cache (even if it has a lower expected latency), because if the prediction is correct, there will be a cache miss which in the end simply adds more latency to the request's overall service time.

The above constraints mean that SBD can only be gainfully employed for requests that would have hit in the DRAM cache where the corresponding cache block is not dirty. To determine whether a request will (likely) hit in the DRAM cache, we simply rely on the HMP. While the HMP is not perfectly accurate, mispredictions simply result in lost opportunities for SBD to make better use of the available bandwidth. To deal with dirty data, we will first simply assume that the DRAM cache makes use of a write-through policy to ensure that all blocks are always clean. Algorithm 1 below describes the basic SBD algorithm assuming a write-through cache. In the next section, we will show how to remove the strict write-through requirement to avoid the unnecessary write traffic to main memory.

Note in Algorithm 1, we do not count *all* of the requests that are

Algorithm 1 Self-Balancing Dispatch

0) Self-balancing dispatch operates only on (predicted) hit requests.

1) $N_{\text{Off-Chip}}$:= Number of requests already waiting for the same bank in the off-chip memory.

2) $L_{\text{Off-Chip}}$:= Typical latency of one off-chip memory request, excluding queuing delays.

3) $E_{\text{Off-Chip}}$:= $N_{\text{Off-Chip}} * L_{\text{Off-Chip}}$. (Total expected queuing delay if this request went off-chip.)

4-6) $N_{\text{DRAM_Cache}}$, $L_{\text{DRAM_Cache}}$, $E_{\text{DRAM_Cache}}$ are similarly defined, but for the die-stacked DRAM cache.

7) If $E_{\text{Off-Chip}} < E_{\text{DRAM_Cache}}$, then send the request to off-chip memory; else send to DRAM cache.

waiting to access off-chip memory, but we limit the count to those waiting on the same bank as the current request that is under SBD consideration (similar for the number of requests to the target off-chip DRAM cache bank). The above description uses the “typical” latency (e.g., for main memory we assume the latency for a row activation, a read delay (tCAS), the data transfer, and off-chip interconnect overheads; for the DRAM cache we assume a row activation, a read delay, three tag transfers, another read delay, and then the final data transfer). Other values could be used, such as dynamically monitoring the actual average latency of requests, but we found that simple constant weights worked well enough. Note also that these latency estimates only need to be close enough relative to each other; slight differences in the estimated expected latency and the actual observed latency do not matter if they do not lead to different SBD outcomes (i.e., an error of a few cycles will in most cases not cause the SBD mechanism to change its decision).

6. Maintaining a Mostly-Clean Cache

When a request is for a cached dirty block, the SBD mechanism has no choice but to send the request to the DRAM cache (it is possible that the HMP mispredicted it as a miss, but this would ultimately be detected and requires reading the data from the DRAM cache anyway). If the system could *guarantee* that a requested block is not cached *and* dirty, then SBD could more freely make bandwidth-balancing decisions with its effectiveness only constrained by the accuracy of the HMP.

6.1. Write-Through vs. Write-Back

We earlier discussed how employing a write-through policy for the DRAM cache can in fact ensure that all requests that hit in the cache are *not* for dirty blocks, but applying a write-through policy wholesale to the entire DRAM cache can result in significant increases in write-through traffic to main memory. Figure 5(a) shows the top most-written-to pages in the DRAM cache for the SPEC2006 benchmark *soplex*. The upper curve (dotted) shows the write traffic for a write-through policy, and the lower curve (solid) shows the write traffic for a write-back policy. The large differences between the curves indicate that the write-back policy achieves significant write-combining, and therefore employing a write-through policy could significantly increase write traffic to main memory. There are other scenarios, such as that shown in 5(b), where, even in a write-back cache, dirty lines are usually only written to once before they are subsequently evicted. However, on average across all of our workloads, we observed that a write-through DRAM cache results in $\sim 3.7\times$ greater write traffic to main memory than a write-back policy (although the amount varies significantly based on the exact workloads).

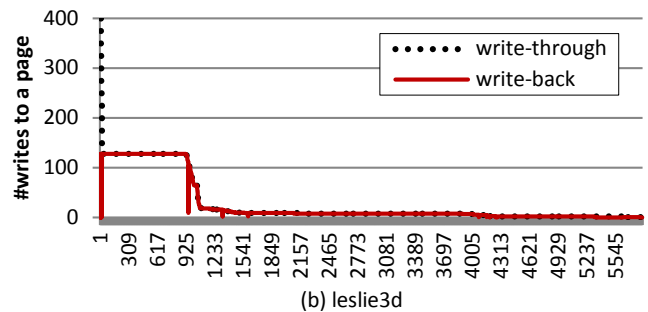
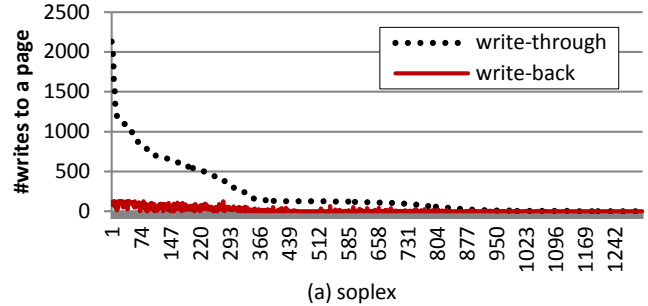


Figure 5: Number of writes for each page with write-through and write-back policy. The x-axis is sorted by top most-written-to pages.

Another important statistic is that, on average for our experiments, only about 5% of an application’s pages ever get written to. This indicates that in typical scenarios, the vast majority of the DRAM cache’s blocks are in fact clean. A write-through cache ensures cleanliness, but costs significantly more main-memory write traffic. A write-back cache minimizes off-chip write traffic, but then cannot provide any guarantees of cleanliness despite the fact that most blocks will in fact be clean.

6.2. The Dirty Region Tracker

We propose a hybrid write-policy for the DRAM cache where, by default, pages employ a write-through policy (to handle the common case of clean pages), but a limited subset of the most write-intensive pages are operated in write-back mode to keep the main-memory write traffic under control. To support this hybrid policy, we introduce the *Dirty Region Tracker* (DiRT). The DiRT consists of two primary components as shown in Figure 6. The first structure is a counting Bloom filter (CBF) that is used to approximately track the number of writes to different pages. On each write, the page address is hashed differently for each of the CBF tables, and the corresponding counters are incremented.⁵ When a page’s counters in *all three* CBFs exceed a threshold, then it is determined to be a write-intensive page (and each indexed CBF counter is reduced by half).

At this point, we introduce the second structure that is a *Dirty List* of all pages that are currently operated with a write-back policy. The Dirty List is a set-associative tagged structure where each entry consists of a tag to store a physical page number and 1 bit of storage to implement a not-recently-used (NRU) replacement policy. A page not currently in the Dirty List, but whose counters have exceeded the threshold, gets inserted into the Dirty List (and the NRU entry from the Dirty List is evicted). Note that when a page is evicted from the Dirty List, its write policy is switched back to write-through; at

⁵We use three CBFs with different hash functions, which increases the efficacy of identifying the most write-intensive pages due to the reduction in aliasing.

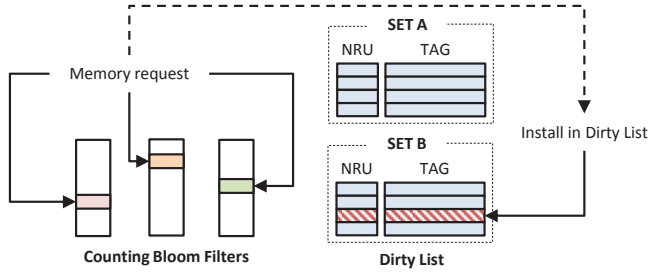


Figure 6: Dirty Region Tracker (DiRT).

this point, the system must ensure that any remaining dirty blocks from this page are written back to main memory. At first blush, this may seem like a high overhead, but a 4KB page only contains 64 cache blocks. Current die-stacked DRAMs already support 32 banks (e.g., 4 channels at 8 banks each [9]), and so the latency overhead is only two activations per bank (and the activations across banks can be parallelized) plus the time to stream out the data back to main memory. Note that all of these cache blocks will have very high spatial locality because they are all from the same page, so practically all of the writeback traffic will experience row buffer hits in main memory. Also, any clean blocks of course need not be written back. The detailed algorithm for DiRT management is listed in Algorithm 2.

Algorithm 2 DiRT Management

- 1) Check Dirty List for the written-to page; if it's there, update NRU replacement meta-data.
- 2) If not, increment each indexed counter in all three CBFs.
- 3) If all indexed counters are greater than threshold:
 - a) Evict the NRU entry from the Dirty List; writeback any associated dirty blocks.
 - b) Allocate the new page to the Dirty List.
 - c) Reduce each indexed CBF counter by half.

6.3. Putting the DiRT to Work

6.3.1. Streamlining HMP: The DiRT works synergistically with Hit-Miss Prediction. In parallel with the HMP lookup, a request can also check the DiRT to see if it accesses a guaranteed clean page. If the page is clean (i.e., not currently in the Dirty List), then requests that are predicted misses can be issued directly to main memory. When the value is returned, this data can be forwarded directly back to the processor without having to verify whether there was actually a dirty copy of the block in the DRAM cache because the DiRT has already guaranteed the block to be clean. Without the DiRT, all returned predicted-miss requests must stall at the DRAM cache controller until the fill-time speculation has been verified. During times of high bank contention, this prediction-verification latency can be quite substantial.

6.3.2. Streamlining SBD: When combining the DiRT with the SBD mechanism, the DiRT can guarantee that accesses to certain (most) pages will be clean, and so SBD can freely choose the best memory source to route the request to. When the HMP predicts a hit, the system first consults the DiRT's Dirty List. If the requested page is found in the Dirty List, then we do not know if the requested *block* is dirty or not (e.g., it could be one of the few clean blocks in a mostly-dirty page). In this case, SBD always routes the request to the DRAM cache. However, if the requested page is not in the Dirty List,

then the page (and therefore the requested block) is *guaranteed* to be clean, and therefore SBD can do as it wishes. Note that clean pages are the overwhelming common case (except for a few benchmarks), and so using the DiRT provides SBD with many more opportunities to make use of otherwise under-utilized off-chip bandwidth.

6.4. Putting It All Together

Figure 7 shows the decision flow chart for memory requests with all of the proposed mechanisms. One should note that Hit-Miss Prediction, SBD, and the DiRT can all be accessed in parallel (SBD can speculatively make a decision assuming an access to a clean, predicted-hit block). Furthermore, HMP and DiRT lookups could even be initiated early before the L2 hit/miss status is known as these components only require the requested physical address.

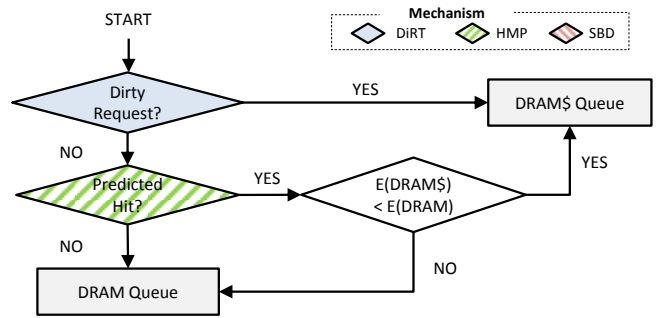


Figure 7: Decision flow chart for memory requests.

6.5. DiRT Implementation Cost

The DiRT is a slightly larger structure compared to the simple hit-miss predictors, but the overall hardware cost is still quite manageable (6.5KB, just 0.16% of our L2 data array size). Each of the three CBF tables has 1024 entries, and each entry consists of a five-bit saturating counter. We use a threshold of 16 writes to consider a page as write-intensive. For Dirty List, we use a 4-way set associative structure with 256 sets, so it supports up to 1024 pages operating in write-back mode at a time. Each entry of the Dirty List consists of 1-bit reference information for NRU replacement policy and a tag for the page. Other approximations (e.g., pseudo-LRU, SRRIP [7]) or even true LRU (this only requires 2-bit for a 4-way set associative structure) could also be used for the replacement policy, but a simple NRU policy worked well enough for our evaluations. In Section 8.7, we provide additional results while comparing our implementation with different DiRT organizations and management policies. For these estimates, we also conservatively assumed a 48-bit physical address (12 bits used for 4KB page offset), which increases our tag size. The total overheads are summarized in Table 2.

Hardware	Size
Counting Bloom Filters	3 * 1024 entries * 5-bit counter = 1920B
Dirty List	256 sets * 4-way * (1-bit NRU + 36-bit tag) = 4736B
Total	6656B = 6.5KB

Table 2: Hardware cost of the Dirty-Region Tracker.

7. Experimental Results

7.1. Methodology

Simulation Infrastructure: We use MacSim [6], a cycle-level x86 simulator, for performance evaluations. We model a quad-core processor with two-level SRAM caches (private L1 and shared L2) and an L3 DRAM cache. The stacked DRAM is expected to support more channels, banks, and wider buses per channel [9]. In this study, the DRAM cache has four channels with 128-bit buses, and each channel has eight banks, while the conventional off-chip DRAM has 2 channels, each with 8 banks and a 64-bit bus. Also, key DDR3 timing parameters with bank conflicts and data bus contention are modeled in our DRAM timing module. Table 3 shows the system configurations used in this study.

CPU	
Core	4 cores, 3.2GHz out-of-order, 4 issue width, 256 ROB
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	16-way, shared 4MB (4 tiles, 24-cycle)
Stacked DRAM cache	
Cache size	128MB
Bus frequency	1.0GHz (DDR 2.0GHz), 128 bits per channel
Channels/Ranks/Banks	4/1/8, 2KB row buffer
tCAS-tRCD-tRP	8-8-15
tRAS-tRC	26-41
Off-chip DRAM	
Bus frequency	800MHz (DDR 1.6GHz), 64 bits per channel
Channels/Ranks/Banks	2/1/8, 16KB row buffer
tCAS-tRCD-tRP	11-11-11
tRAS-tRC	28-39

Table 3: System parameters used in this study.

Workloads: We use the SPEC CPU2006 benchmarks and sample 200M instructions using SimPoint [14]. Then we categorize the applications into two different groups based on the misses per kilo instructions (MPKI) in the L2 cache. We restrict the study to workloads with high memory traffic; applications with low memory demands have very little performance sensitivity to memory-system optimizations and therefore expose very little insight (we did verify that our techniques do not negatively impact these benchmarks). Out of the memory-intensive benchmarks, those with average MPKI rates greater than 25 are in Group H (for **H**igh intensity), and of the remaining, those with 15 MPKI or more are in Group M (for **M**edium). Table 4 shows MPKI values of the benchmarks and their group.

Group M	MPKI	Group H	MPKI
GemsFDTD	19.11	leslie3d	25.85
astar	19.85	libquantum	29.30
soplex	20.12	milc	33.17
wrf	20.29	lbm	36.22
bwaves	23.41	mcf	53.37

Table 4: L2 misses per kilo instructions (L2 MPKI).

We select benchmarks to form rate-mode (all cores running separate instances of the same application) and multi-programmed workloads. Table 5 shows the primary workloads evaluated for this study. Section 8 also includes additional results covering a much larger number of workloads.

For each workload, we simulate 500 million cycles of execution. We verified that the DRAM cache is sufficiently warmed up: the

Mix	Workloads	Group
WL-1	4 × mcf	4×H
WL-2	4 × lbm	4×H
WL-3	4 × leslie3d	4×H
WL-4	mcf-lbm-milc-libquantum	4×H
WL-5	mcf-lbm-libquantum-leslie3d	4×H
WL-6	libquantum-mcf-milc-leslie3d	4×H
WL-7	mcf-milc-wrf-soplex	2×H + 2×M
WL-8	milc-leslie3d-GemsFDTD-astar	2×H + 2×M
WL-9	libquantum-bwaves-wrf-astar	1×H + 3×M
WL-10	bwaves-wrf-soplex-GemsFDTD	4×M

Table 5: Multi-programmed workloads.

DRAM cache access statistics at the end of the simulation show that the number of valid cache lines is equal to the total capacity of the cache, and the total number of evictions is $5 \times - 6 \times$ greater than the total cache capacity.

Performance Metric: We report performance using weighted speedup [5, 16], which is computed as:

$$\text{Weighted Speedup} = \sum_i \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{single}}}$$

The geometric mean is also used to report average values.

7.2. Performance

Figure 8 shows the performance of the proposed hit-miss predictor (HMP), self-balancing dispatch (SBD), and dirty region tracker (DiRT) mechanisms for multi-programmed workloads. For comparison, we use a baseline where the DRAM cache is not employed. We also compare our mechanisms with the previously proposed MissMap structure (denoted as MM in the figure). We model a MissMap with zero storage overhead; i.e., no L2 cache capacity is sacrificed for the MissMap, but the L2 latency is still incurred for the lookup.

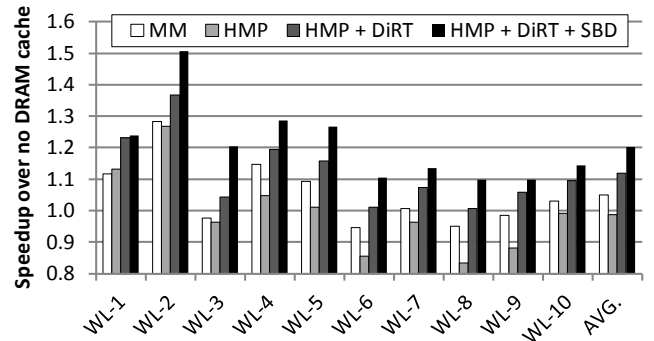


Figure 8: Performance normalized to no DRAM cache for MissMap (MM), and combinations of HMP, SBD, and DiRT.

We first evaluate the impact of hit-miss prediction (HMP) without DiRT. In this usage scenario, every predicted miss request serviced from off-chip memory must wait to be verified as *safe* (i.e., no dirty data in the DRAM cache). As a result, for most benchmarks, HMP without DiRT performs worse than MissMap. This is not necessarily a negative result when one considers that the HMP approach sacrifices the multi-megabyte MissMap for a much smaller sub-kilobyte predictor. Achieving even close to similar performance while removing such a large hardware overhead is still a desirable result. However, with DiRT support, HMP+DiRT performs even better than MissMap

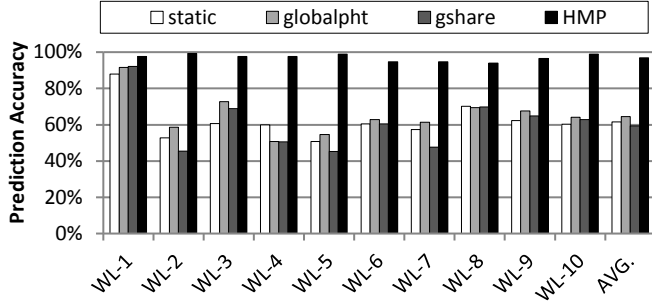


Figure 9: Prediction accuracy of HMP and its comparison with other types of predictors.

due to the elimination of fill-time prediction verifications for clean blocks (which are the common case). At this point, the performance benefit *over* MissMap is primarily due to the replacement of the 24-cycle MissMap latency with a 1-cycle HMP lookup.

Next, we apply the SBD mechanism on top of HMP+DiRT. As shown in the results, SBD further improves performance (often significant, depending on the workload). Compared to HMP+DiRT, SBD provides an additional 8.3% performance benefit on average. In summary, the proposed mechanisms (HMP+DiRT+SBD) provide a 20.3% performance improvement over the baseline. Also, compared to MissMap, they deliver an additional 15.4% performance over the baseline. On a last remark, one should note that the evaluated MissMap does not sacrifice the L2 cache (i.e., ideal), so our mechanisms would perform even better when compared to a non-ideal MissMap that reduces the effective SRAM cache size.

8. Analysis

In this section, we provide additional analysis on the proposed hit-miss predictor, self-balancing dispatch, and the dirty region tracker.

8.1. HMP: Prediction Accuracy

Figure 9 shows the prediction accuracy of the proposed predictor with comparison to some other types of predictors. *static* indicates the best of either static-hit or static-miss predictors, so the value is always great than 0.5. A reasonable predictor at least should be better than *static*. *globalpht* is the implementation of only one 2-bit counter for all memory requests, where it is incremented/decremented on a hit/miss. *gshare* is a gshare-like cache predictor (i.e., using the XOR of a requested 64B block address with a global history of recent hit/miss outcomes to index into a pattern history table).

First, the results show that our predictor provides more than 95% prediction accuracy on the evaluated workloads (average of 97%), which implies that the spatial locality-based hit/miss prediction is highly effective. Next, compared to *static*, we can see that the other predictors actually do not improve prediction accuracy much. For WL-1, all of the predictors perform well because the workload has a high hit rate and is simply easy to predict. But, if the hit ratio is around 50% as in other workloads, the other predictors perform poorly. For *globalpht*, one core may be consistently hitting while the other is consistently missing, and as a result the simple counter could ping-pong back and forth generating mispredictions. For *gshare*, the hit/miss history register provides poor information, and its inclusion often introduces more noise than useful correlations, resulting in overall lower prediction rates. In summary, HMP outperforms other predictors that use the individual 64B request address and/or history information of the actual outcomes.

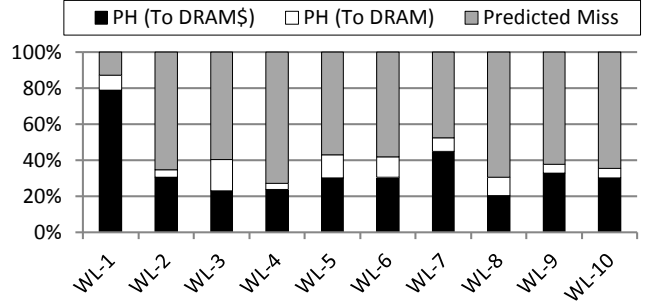


Figure 10: Issue direction breakdown. PH indicates predicted hit requests.

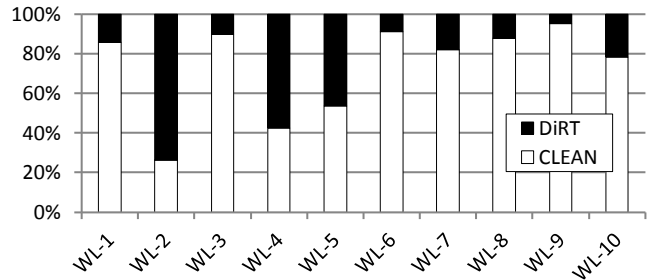


Figure 11: Percentage distribution of memory requests captured in DiRT.

8.2. SBD: Percentage of Balanced Hit Requests

Figure 10 shows the distribution of SBD’s issue decisions (i.e., DRAM or DRAM cache). The black bar (PH: To DRAM\$) represents the percentage of requests that are predicted hits and are actually issued to the DRAM cache, while the white bar (PH: To DRAM) represents the predicted-hit requests that were diverted to off-chip memory. Note that SBD does not work on the predicted-miss requests; thus, the requests in the (Predicted Miss) portion are always issued to off-chip memory.

At first thought, one might think that SBD does not operate on the benchmarks whose hit ratios are low (e.g., below 50%) because the amount of traffic to off-chip DRAM would be greater than that to the DRAM cache. Due to the bursty nature of memory requests, however, the instantaneous hit ratio and/or bandwidth requirements vary from the average values; thus, the balancing mechanism provided by SBD can still be beneficial even for the low hit-ratio workloads. In fact, as shown in the results, SBD was able to redistribute some of the hit requests for all of the benchmarks.

8.3. DiRT: Benefit and Traffic

Figure 11 shows the percentage distribution of write-through mode (CLEAN) and write-back mode (DiRT) memory requests. The “CLEAN” portion indicates the number of requests that are not found in the DiRT; thus, they are free to be predicted or self-balanced. The results show that DiRT allows a significant amount of memory requests to be handled without fear of returning a stale value. Note that without DiRT, every request that is a predicted miss (or a predicted hit but diverted to DRAM) needs to wait until it has been verified that the DRAM cache does not contain a dirty copy.

Figure 12 illustrates the amount of write-back traffic to off-chip DRAM for write-through, write-back, and the DiRT-enabled hybrid

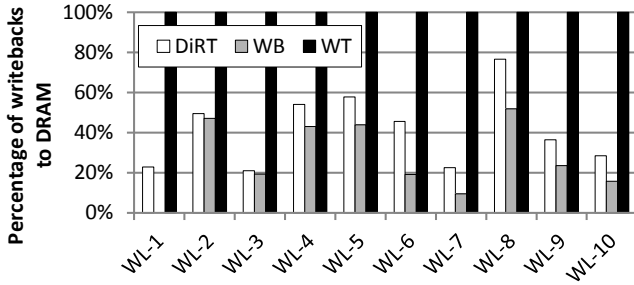


Figure 12: Write-back traffic to off-chip DRAM between write-through, write-back, and DiRT (WL-1 does not generate WB traffic), all normalized to the write-through case.

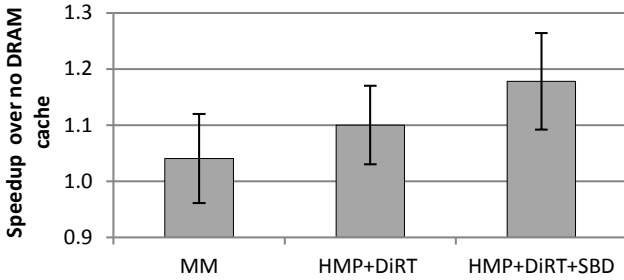


Figure 13: Average performance of MissMap and our proposed mechanisms over no DRAM cache baseline with +/-1 std. deviation for 210 workloads.

policy, all normalized to the write-through case. As shown in the results, a write-back policy performs a significant amount of write-combining and thereby greatly reduces the amount of write traffic to main memory as compared to a write-through policy. DiRT is not perfect and it does increase write traffic slightly compared to a true write-back policy, but the total write traffic from a DiRT-enabled DRAM cache is much closer to that of the write-back case than it is to write-through. The relatively small increase in write traffic due to the DiRT is more than compensated by the streamlined HMP speculation and increased opportunities for SBD.

8.4. Sensitivity to Different Workloads

To ensure that our mechanisms work for a broader set of scenarios beyond the ten primary workloads used thus far in this paper, we simulated all 210 combinations ($_{10}C_4$) of the ten Group H and Group M benchmarks. Figure 13 shows the performance results averaged over all of the 210 workloads, along with error-bars to mark one standard deviation. As shown in the figure, our mechanisms combine to deliver strong performance over the previously proposed MissMap-based DRAM cache approach.

8.5. Sensitivity to DRAM Cache Sizes

Figure 14 shows the performance of the proposed mechanisms with different sizes of DRAM caches. The results show that the benefit of MissMap, HMP+DiRT, and HMP+DiRT+SBD increases as the cache size grows. For all cache sizes, HMP+DiRT+SBD still performs best. In addition, the benefit of SBD increases as the DRAM cache size increases because the higher hit rate provides more opportunities for SBD to dispatch requests to main memory.

8.6. Sensitivity to DRAM Cache Bandwidth

In our evaluation, the ratio of peak DRAM cache bandwidth to main memory is 5:1 (2GHz vs. 1.6GHz, 4 channels vs. 2 channels, and

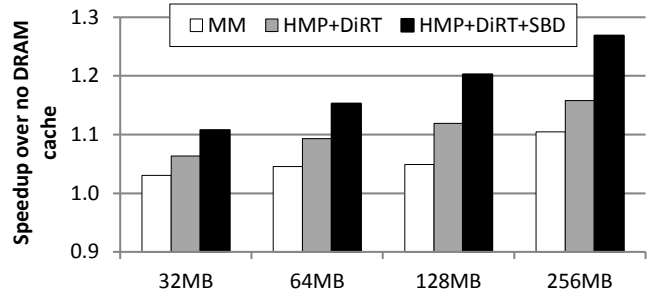


Figure 14: Performance sensitivity of the proposed mechanisms to different DRAM cache sizes.

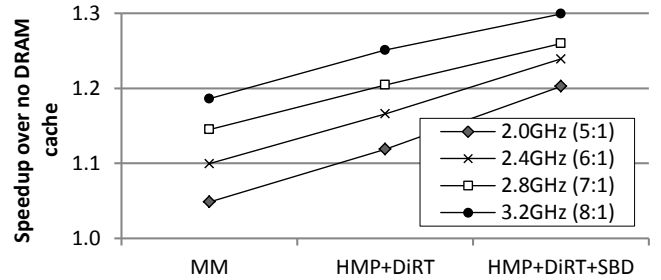


Figure 15: Performance sensitivity to different ratios of DRAM cache bandwidth to off-chip memory.

128-bit bus per channel vs. 64-bit bus per channel). While we believe that this is reasonable for plausible near-term systems,⁶ it is also interesting to see how the effectiveness of HMP and SBD scales under different bandwidth assumptions. Figure 15 shows the performance sensitivity when we increase the DRAM cache frequency from 2.0GHz (what was used so far in this paper) up to 3.2GHz. First, as shown in the results, the benefit of HMP does not decrease if future die-stacked DRAMs provide more bandwidth (relative to off-chip). As the DRAM-cache frequency increases, the cost of the 24-cycle MissMap increases relative to the DRAM-cache latency, and therefore HMP provides a small but increasing relative performance benefit as well. On the other hand, increasing the DRAM cache frequency reduces the *relative* additional bandwidth provided by the off-chip DRAM, thereby potentially decreasing the effectiveness of SBD. In our experiments, we do observe that the relative benefit of SBD over HMP reduces as the DRAM cache bandwidth increases, but overall, SBD still provides non-negligible benefits even with higher-frequency DRAM caches. Note that the die-stacked DRAM bandwidth may not grow too rapidly (such as 32:1), as adding more TSVs requires die area on the memory chips (which directly impacts cost), and increasing bandwidth via higher-frequency interfaces has power implications.

8.7. Sensitivity to DiRT Structures

Figure 16 shows the performance results as we vary the number of Dirty List entries (first four bars), assuming a fully-associative structure with LRU replacement. Note that such a structure would be difficult to implement for these sizes (e.g., true LRU on 1K entries).

⁶For example, current x86 processors tend to have two DDR3 memory channels (some have three or four, which would provide even more opportunities for SBD). The JEDEC Wide-IO standard provides four channels at 128 bits each, which is the same as our stacked DRAM configuration.

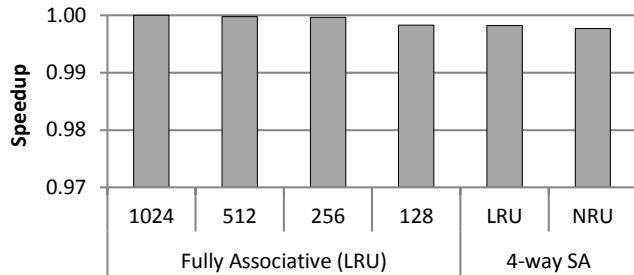


Figure 16: Performance sensitivity to different DiRT structures and management policies.

Overall, there is very little performance degradation even when reducing the size of the DiRT to only 128 entries, but we still chose to employ a 1K-entry table to reduce the performance variance across workloads.

The right side of Figure 16 also shows the results for 4-way set-associative implementations, each with 1K entries. The right-most bar (1K entry, 4-way, NRU) is the configuration used thus far in the paper and has the lowest implementation complexity and cost. Overall, the results show that even with our simplified DiRT organization, we lose very little performance compared to an impractical fully-associative, true-LRU solution.

9. Conclusion

The prior work by Loh and Hill provided an important step toward the realization of a more practical die-stacked DRAM cache solution. However in this paper, we have shown that there still exist inefficiencies in the prior solution. In particular, the assumption that precise cache-content tracking was needed led to a MissMap structure that was over-designed for the DRAM cache. By taking advantage of the simple observation that on a miss, tag reads for victim selection need to occur anyway, false-negative mispredictions can be verified to prevent returning stale data from main memory back to the processors. The ability to freely speculate enables our DRAM cache organization that avoids the hardware overheads of the MissMap.

We also observed that while the die-stacked DRAM may provide significant bandwidth, the off-chip memory bandwidth is still a valuable resource that should not be disregarded. Our Self-Balancing Dispatch approach allows our DRAM cache design to make better use of the system’s aggregate bandwidth. For both the HMP and SBD approaches, we found that life is significantly easier when we do not need to worry about dirty data. Completely abolishing dirty data from the DRAM cache with a write-through policy causes write traffic to increase tremendously. However, by bounding (and tracking) a limited number of pages in write-back mode, we could greatly amplify the effectiveness of both HMP and SBD techniques. Overall, we have proposed a significantly streamlined DRAM cache organization that improves performance over the state of the art while eliminating the large MissMap structure.

Beyond the ideas presented in this paper, there likely remain other research opportunities to further improve the performance and/or practicality of die-stacked DRAM caches. For instance, the motivational example showing under-utilized off-chip bandwidth also illustrates the high cost of placing the tags in the DRAM cache in terms of bandwidth. In particular, the raw $8\times$ higher bandwidth of the die-stacked DRAM (compared to off-chip) is reduced to only a $2\times$ increase in the effective bandwidth in terms of serviceable

requests per unit time. DRAM cache organizations that can make more efficient use of the DRAM cache’s raw bandwidth would likely provide further performance benefits. Studies on further improving the practicality of die-stacked DRAM caches, such as the interaction with cache coherence, are also good directions for future research.

Acknowledgments

We would like to thank Andreas Moshovos, the Georgia Tech HPArch members, and the anonymous reviewers for their suggestions and feedback. Part of this work was conducted while Jaewoong Sim was on an internship and Mithuna Thottethodi was on sabbatical leave at AMD Research. We gratefully acknowledge the support of the NSF CCF-0644183 (Thottethodi); and AMD, Sandia National Laboratories, and NSF CAREER award 1139083 (Kim).

References

- [1] B. Black, M. M. Annavaram, E. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. P. Shen, and C. Webb, “Die Stacking (3D) Microarchitecture,” in *MICRO-39*, 2006.
- [2] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor,” *IEEE Micro*, March–April 2010.
- [3] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stoloro, and A. Subbiah1, “A 22nm IA Multi-CPU and GPU System-on-Chip,” in *ISSCC*, 2012.
- [4] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, “Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support,” in *SC*, 2010.
- [5] S. Eyerhan and L. Eeckhout, “System-Level Performance Metrics for Multiprogram Workloads,” *IEEE Micro*, May–June 2008.
- [6] “Macsim simulator;” <http://code.google.com/p/macsim/>, HPArch.
- [7] A. Jaleel, K. Theobald, S. C. Steely, and J. Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),” in *ISCA-32*, 2010.
- [8] 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 *HPCA-16*, 2010.
- [9] J.-S. Kim, C. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, and Y.-H. Jun, “A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking,” in *ISSCC*, 2011.
- [10] H. Liu, M. Ferdman, J. Huh, and D. Burger, “Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency,” in *MICRO-41*, 2008.
- [11] G. H. Loh and M. D. Hill, “Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches,” in *MICRO-44*, 2011.
- [12] G. H. Loh, N. Jayasena, K. McGrath, M. O’Connor, S. Reinhardt, and J. Chung, “Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems,” in *SHAW-3*, 2012.
- [13] A. Seznec and P. Michaud, “A Case for (Partially) TAGged GEometric History Length Branch Prediction,” *Journal of Instruction-Level Parallelism*, 2006.
- [14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *ASPLOS-X*, 2002.
- [15] J. E. Smith, “A Study of Branch Prediction Strategies,” in *ISCA-3*, 1981.
- [16] A. Snavely and D. Tullsen, “Symbiotic Job Scheduling for a Simultaneous Multithreading Processor,” in *ASPLOS-IX*, 2000.
- [17] G. Taylor, P. Davies, and M. Farmwald, “The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism,” in *ISCA-12*, 1990.
- [18] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, “Speculation Techniques for Improving Load Related Instruction Scheduling,” in *ISCA-21*, 1999.
- [19] J. Zhao, G. Sun, Y. Xie, and G. H. Loh, “Energy-Efficient GPU Design with Reconfigurable In-Package Graphics Memory,” in *ISLPED*, 2012.
- [20] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, “Exploring DRAM Cache Architectures for CMP Server Platforms,” in *ICCD-25*, 2007.