Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches

Gabriel H. Loh[⋆]
[⋆]AMD Research
Advanced Micro Devices, Inc.
gabe.loh@amd.com

Mark D. Hill[†⋆]
[†]Department of Computer Sciences
University of Wisconsin – Madison
markhill@cs.wisc.edu

Abstract

Die-stacking technology enables multiple layers of DRAM to be integrated with multicore processors. A promising use of stacked DRAM is as a cache, since its capacity is insufficient to be all of main memory (for all but some embedded systems). However, a 1GB DRAM cache with 64-byte blocks requires 96MB of tag storage. Placing these tags on-chip is impractical (larger than on-chip L3s) while putting them in DRAM is slow (two full DRAM accesses for tag and data). Larger blocks and sub-blocking are possible, but less robust due to fragmentation.

This work efficiently enables conventional block sizes for very large die-stacked DRAM caches with two innovations. First, we make hits faster than just storing tags in stacked DRAM by scheduling the tag and data accesses as a compound access so the data access is always a row buffer hit. Second, we make misses faster with a MissMap that eschews stacked-DRAM access on all misses. Like extreme sub-blocking, our implementation of the MissMap stores a vector of block-valid bits for each "page" in the DRAM cache. Unlike conventional sub-blocking, the MissMap (a) points to many more pages than can be stored in the DRAM cache (making the effects of fragmentation rare) and (b) does not point to the "way" that holds a block (but defers to the off-chip tags).

For the evaluated large-footprint commercial workloads, the proposed cache organization delivers 92.9% of the performance benefit of an ideal 1GB DRAM cache with an impractical 96MB on-chip SRAM tag array.

General Terms

Design, Performance

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Design Styles—*Cache memories*

1. INTRODUCTION

Die-stacking technologies provide a way to tightly integrate multiple disparate silicon die with high-bandwidth, low-latency interconnects. A likely initial use of die stacking will be to integrate a large amount of DRAM with a conventional multicore processor chip. The implementation could involve vertical stacking [1, 2] or horizontal/2.5D stacking on an interposer [3], as illustrated in Figure 1, but in either case the processors are provided with a high-bandwidth, low-latency path to the stacked memory. Estimates of stacked DRAM capacity vary from a few tens or low hundreds of megabytes [4, 5] up to a few gigabytes [6, 7]. Even with a few gigabytes of stacked DRAM, this capacity may not be sufficient to support the entirety of a system's main memory, except perhaps for some lower-end and mobile platforms [8]. For the enterprise server market in particular, systems can contain many tens or even hundreds of gigabytes of DRAM.

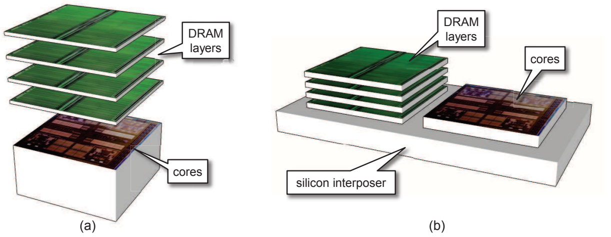


Figure 1: Die-stacked DRAM with a multicore processor chip implemented as (a) a vertical stack, and (b) side-by-side on a silicon interposer.

Without being able to stack the entire system's memory, there are two primary ways to use stacked DRAM. One approach is to expose the stacked DRAM to the operating system (OS) by directly mapping the stacked DRAM into the global physical address space. The OS is then responsible for deciding what memory pages should be placed in the faster, high-bandwidth stacked DRAM and what remains in conventional off-chip memory, migrating pages as necessary. This approach presents several challenges to near-term adoption. The OS typically does not have access to detailed usage statistics to determine the most frequently accessed pages that should be mapped to stacked DRAM. Remapping pages is a heavyweight operation that involves page table updates, TLB invalidations, and copying multiple pages of data. Adding such support to commercial operating systems will likely require several years of software development, requiring coordination among companies.

The second usage of stacked DRAM is as a very large last-level cache. This approach benefits from being software-transparent, and therefore it is not dependent on new versions of OS's, and could be deployed as quickly as a chip can be designed. Section 2 discusses challenges with DRAM cache variants that use large cachelines, sub-blocking, or tags in DRAM, as well as showing the potential of a stacked-DRAM cache with impractically large on-chip tags.

Section 3 presents our stacked-DRAM cache with two key innovations. First, we make hits faster than just storing tags in stacked DRAM by scheduling the tag and data accesses as a compound access, so the data access is always a row buffer hit. Second, we make misses faster with a MissMap that eschews stacked-DRAM

MICRO 11, December 3-7, 2011, Porto Alegre, Brazil
Copyright 2011 ACM 978-1-4503-1053-6/11/12 ...$10.00.

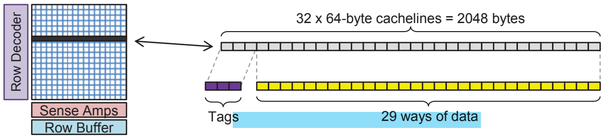


Figure 2: Mapping the tags and data of a cache set to a single DRAM row.

accesses on all misses. Like extreme sub-blocking, the MissMap stores a vector of block valid bits for each "page" in the DRAM cache. Unlike conventional sub-blocking, the MissMap (1) points to many more pages than can be stored in the stacked DRAM (making the effects of fragmentation rare) and (2) does not point to the way that holds a block (but defers to the off-chip tags).

2. CHALLENGES OF IMPLEMENTING LARGE CACHES

Adding DRAM to a processor to implement a large cache may sound simple, but there are several challenges that prevent the immediate wide-spread adoption of stacked-DRAM caches. Using DRAM as a cache requires the implementation of a tag store. A tag entry for a single cacheline may require up to five or six bytes. For example, assuming 48-bit physical addresses, the tag itself is approximately four bytes, and the tag entry may include other metadata (e.g., LRU counters, coherence state, and sharer information). A 128MB DRAM can store 2^{21} 64-byte cachelines, which at six bytes of tag overhead each, results in a total tag array size of 12MB. This is already larger than most L3 caches today. For a 1GB DRAM, this tag overhead increases to 96MB.

2.1 Large Cacheline Sizes

The tag overhead for very large caches has been noted by other researchers, and a common approach for avoiding this overhead is to increase the size of the cacheline [4, 9, 10]. For example, using a 4KB cacheline reduces the total number of cachelines to only 32,768 for a 128MB cache. Very large cachelines can have fragmentation problems; in the worst case, only a single 64-byte subblock will be used from each cacheline. Transferring 4KB of data at a time can also cause significant off-chip bus contention, leading to substantial queuing/back-pressure delays throughout the cache hierarchy. Similarly, large cachelines can cause severe false-sharing in multi-threaded applications, although the fragmentation problem tends to be of greater concern [11]. Copying unused blocks back and forth also wastes bandwidth and power [12].

Supporting 4KB cachelines has other challenges. DRAM arrays use row buffers that are typically only 1KB or 2KB. Therefore, using 4KB cachelines can require that the data be mapped across more than one physical DRAM row. Accessing a cacheline will now require occupying multiple banks and issuing more DRAM commands. The additional command activity leads to more contention on the command buses, and this can also lead to further inefficiencies due to DRAM timing constraints.

2.2 Sub-blocking

Sub-blocked caches can alleviate fragmentation and false sharing problems [13]. Like caching very large lines, multiple conventional (e.g., 64-byte) cachelines are grouped together into a single aligned "super-block." The tag entry maintains a single address tag for the entire large cacheline, but provides valid and coherence bits for each individual sub-block. This increases the overhead of the tag array compared to a simple large-cacheline approach. Assuming an overhead of eight bits per 64-byte sub-block, each tag entry now requires 68 bytes. A 1GB sub-blocked DRAM cache needs 18.4MB (larger than most current L3 caches).

The sub-blocked cache uses only bandwidth and power to fetch requested sub-blocks, compared to a large-cacheline approach that transfers the entire cacheline. Sub-blocking still does not address the problem of tracking only a limited number of cachelines. For workloads with low spatial locality, the large cachelines can result in high miss rates. Selective caching of the most frequently used cachelines can alleviate some of this effect [9, 10], but applications with large active working sets will still suffer.

2.3 Combining Tags and Data in the DRAM

An alternative approach stores the tags directly in the DRAM array with the data, as shown in Figure 2. A 2KB DRAM row could store up to thirty-two 64-byte blocks, but the row can also be partitioned into twenty-nine 64-byte cachelines, with the remaining 192 bytes for tags. The 29 data blocks need $29 \times 6 = 174$ bytes for their tag entries. For this configuration, there are 18 bytes left unused, which could be employed for better replacement policies, profiling, or other uses, but we do not explore these other opportunities in this work. Storing tags in the stacked DRAM can support arbitrarily large DRAM caches (in contrast to a separate SRAM tag array that scales linearly in size with the DRAM capacity), although at the cost of the DRAM capacity ($\sim$9.4% in this example).

While storing tags in DRAM has been discussed by prior work [4, 10], it has been largely dismissed because of the assumed latency impact. Accessing the cache now requires one DRAM access on a cache miss, and two for a cache hit: once for the tag, and once for the data. Even though stacked-DRAM implementations may be faster than conventional off-chip DRAMs, their latencies are still considerably longer than SRAMs. Furthermore, the DRAM will require a third access to update replacement information (e.g., LRU counters) and/or coherence state in the tag entry. While this extra DRAM access is off of the critical path of serving requests, it decreases DRAM bank availability, causing more bank contention.

2.4 Potential of Large DRAM Caches

Figure 3 shows the performance impact of implementing very large caches ranging from 128MB up to 1GB on four memory-intensive commercial workloads suffering from high memory contention (see Section 4 for details on simulation methodology). All results in the figure are normalized to a baseline eight-core processor with an 8MB L3 cache and no stacked DRAM. The top curve (△) shows the performance for a DRAM L4 cache supported by an ideal SRAM tag array (i.e., 12MB tag array for the 128MB L4, and 96MBs for the 1GB L4). The bottom curve (□) is for an L4 that stores the tags in the DRAM array, where a miss takes one full DRAM access, a hit takes two full accesses, and additional traffic for spilling, filling, and tag updates are all modeled. The ideal-SRAM tags provide an upper bound on what we can hope to achieve with DRAM caching. The bottom curve paints an ugly picture for storing tags in the DRAM. Similar results were shown by previous studies assuming constant latencies for the DRAM accesses [4, 10].

3. A PRACTICAL DRAM CACHE WITH CONVENTIONAL BLOCK SIZES

In designing our DRAM cache, we have a few objectives regarding performance and overheads: (1) Support 64-byte cachelines to avoid fragmentation and minimize the bandwidth of wasted transfers. (2) Keep SRAM overhead as low as possible. (3) On a cache hit, the latency should be as close as possible to a single DRAM

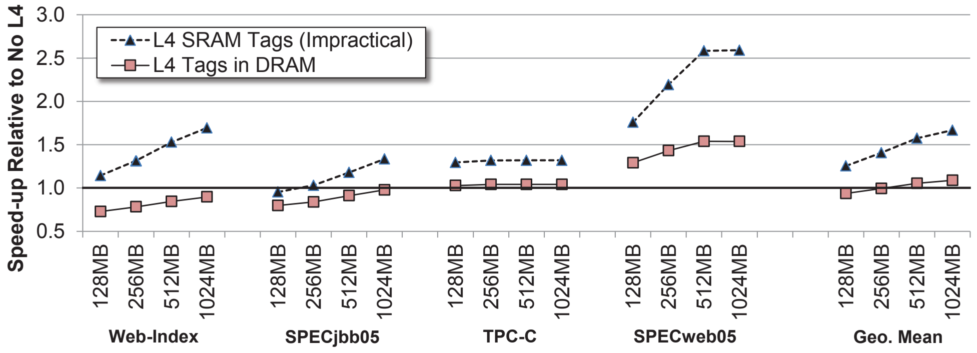


Figure 3: Performance benefits of large on-chip caches. See Section 4 for simulation methodology.

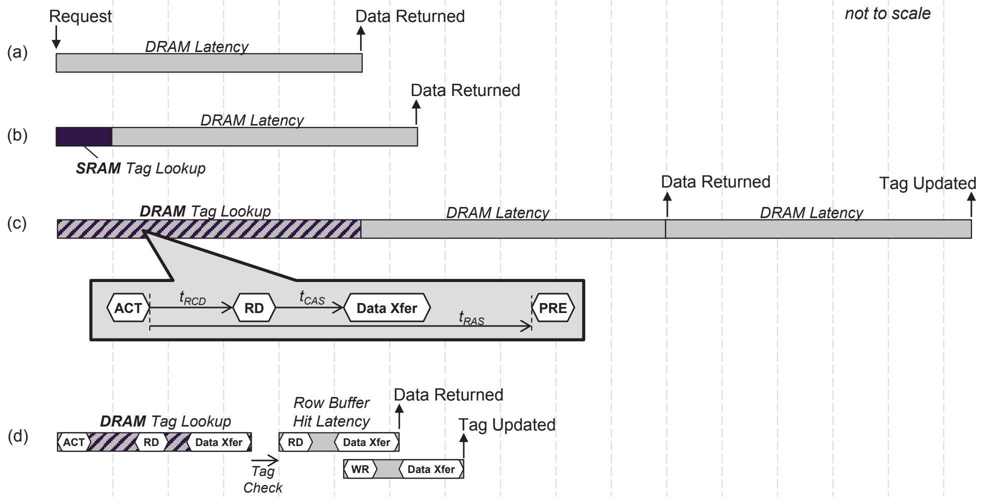


Figure 4: Timing diagrams for a DRAM cache hit. (a) Ideal case that requires *a priori* knowledge about the location of the requested block. (b) Using an SRAM tag array. (c) Placing tags in the DRAM, but using pessimistic in which where every DRAM access requires the worst-case access latency. (d) Placing tags in the DRAM with row buffer aware compound access sequencing.

access. (4) On a cache miss, the request should proceed to main memory as quickly as possible without a stacked-DRAM access. To support objectives (1) and (2), we will make use of a tags-in-DRAM organization with 64-byte cachelines. In the following sections, we explain how we attempt to achieve goals (3) and (4) despite the latency problems of the tags-in-DRAM organization.

3.1 Reducing Hit Latency

Ideally, a DRAM cache hit requires only a single DRAM access latency, as shown in Figure 4(a). This is very difficult to achieve because a tag lookup is typically required to determine the actual location (i.e., physical way or column) of the requested data.Figure 4(b) shows the case in which an SRAM tag lookup provides this information, and provides an overall latency close to the ideal case.

The results from Figure 3 showed that the latency of performing two DRAM accesses per cache hit caused some serious performance deficiencies compared to the ideal DRAM caches. This access sequence is illustrated in Figure 4(c). In real systems, DRAM latencies vary and depend on factors such as row buffer locality, command scheduling, and DRAM timing constraints.

To read data from a DRAM, a sequence of commands must be issued from the memory controller. Assuming the requested row is not already open, the memory controller issues an activation (ACT) command that retrieves the selected row and latches the values in a row buffer. The memory controller can then issue a read (RD) command, causing the selected data words to be transmitted across the data bus. Eventually, the memory controller must also close the row by issuing a precharge (PRE) command that writes the contents of the row buffer back to the DRAM bitcell array. The inset of Figure 4(c) illustrates this command sequence. Note that if a requested row has already been loaded into the row buffer, then the activation command can be skipped (called a row buffer hit).

We assume that a single physical DRAM row holds both tags and data, as shown in Figure 2. Our proposed DRAM uses a simple modification of the memory controller's scheduling algorithm by treating the separate tag and data lookups as a *compound access*. On a DRAM cache lookup, the memory controller first issues activation and read commands as usual to load the requested cache set into a DRAM row buffer and read the tag information. The key step is that after the controller issues the tag read command, it reserves the row buffer to prevent any other requests from closing the row.

In the case of a cache hit, the position of the matching tag indicates the column address of the cache data in the row buffer, and the memory controller can immediately retrieve the data. By reserving

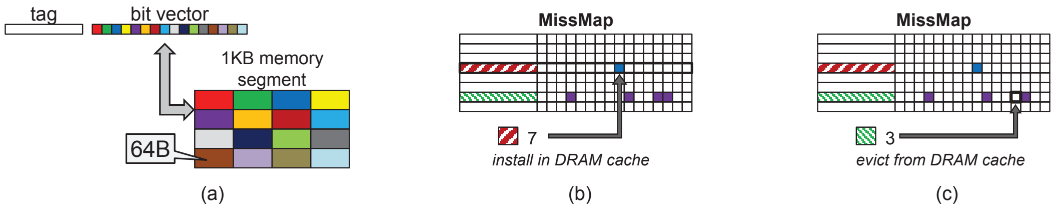


Figure 5: (a) MissMap entry covering a 1KB memory segment. (b) Setting a MissMap bit when installing a line in the DRAM cache, and (c) clearing a MissMap bit when evicting a line from the DRAM cache.

the row buffer, we have guaranteed a row buffer hit for the data access, as illustrated in Figure 4(d). Comparing this to the SRAM tag case of Figure 4(b), it turns out that by scheduling the tag and data lookups as a compound access, the net latency of our tags-in-DRAM approach is not much different. We have effectively traded SRAM lookup latency for row buffer hit latency. Furthermore, the controller can keep the row reserved so any necessary tag updates also hit in the row buffer. While this optimization may appear quite simple in retrospect, the failure to observe this opportunity caused previous researchers to dismiss the placement of tags in DRAM.

For a cache miss, the row buffer is unreserved after the tag check completes. A precharge can then be issued to close the row to allow other cache lookups on this bank to proceed. Reserving the row causes the bank to be unavailable to other requests for an additional bus cycle, but this is a small price to pay to guarantee row buffer hits for the data accesses. Furthermore, the MissMap technique described in the next section ensures that hits are the common case.

3.2 MissMap: Avoiding Accesses on a Miss

Our last design objective is to avoid the DRAM cache access on a miss. There are two basic approaches to this problem. The first is to keep track of all of the contents of the DRAM cache in some data structure, and if the requested block cannot be found in the data structure, then we have a miss. Unfortunately, this seems like it is the same as implementing a tag array with the same problems regarding the size of the required table. The second approach is to keep track of all memory locations *not* in the cache, but the fraction of the memory space not cached is much larger than what is cached.

A conventional tag array serves *two* primary purposes. The first is to track the contents of the cache; i.e., exactly what blocks currently reside in the cache. The second, is to record the *location* of each block in its set. This is implicitly tracked by maintaining a one-to-one correspondence between the physical ways of the tag array and the ways of the data array. Our insight is to use the precious on-chip SRAM to perform only the first task.

3.2.1 The MissMap

The location information implicit in a conventional tag array is not actually needed to answer the question of whether an access results in a cache miss. To efficiently track which blocks are currently stored in the DRAM cache, we decouple the block-residency and block-location problems. A simple *MissMap* data structure answers queries about cache block residency, and then the location problem is handled by the in-DRAM tags. In our current implementation, each MissMap entry tracks the cachelines associated with a contiguous, aligned segment of memory, such as a page. Each MissMap entry contains a tag corresponding to the address of the tracked memory segment, and a bit vector with one bit per cacheline. Figure 5(a) shows a MissMap entry for a 1KB segment.

Each time the processor inserts a new cacheline into the DRAM cache, the processor also looks up a MissMap entry corresponding to the segment containing the new cacheline (allocating a new entry if necessary) and sets the bit in the entry corresponding to the inserted cacheline, as shown in Figure 5(b). When the processor evicts a cacheline from the DRAM cache, the bit in the MissMap entry will be cleared, as shown in Figure 5(c). The MissMap maintains a consistent record of the current DRAM cache contents, and so by checking to see if a cacheline's MissMap bit is zero, the processor can quickly determine that there is a cache miss. Similarly, if no entry can be found for the segment, this means no cachelines from the entire segment are currently in the cache. In this fashion, DRAM cache misses bypass the DRAM cache lookup entirely; a consequence is that the lookups that do go to the DRAM cache are always hits, thereby increasing the effectiveness of scheduling the lookups as compound accesses (Section 3.1).

Note that the bit-vector MissMap entry is just one possible implementation, and it represents just one possible embodiment of the general approach. MissMaps in general can be imprecise in answering cacheline residency queries so long as they are conservative (e.g., with Bloom filters). That is, if a cacheline is present in the DRAM cache, then the MissMap must always accurately report this. If a cacheline is not present, then incorrectly reporting that it is present only represents a performance opportunity loss, but it will not cause incorrect program execution.

3.2.2 MissMap Sizing and Reach

At first blush, the MissMap looks very similar to the tag array for a large-cacheline cache and just performs sub-blocking in disguise. The critical difference is that the MissMap is sized to be able to track *more* memory than can fit in the DRAM cache, whereas the tag-array for a large-cacheline cache tracks *exactly* the number of cachelines that fit in the cache. Figure 6(a) shows an example 4KB cache that uses 1KB cachelines. The tag array consists of only four entries corresponding to the four large cachelines. Sub-blocks corresponding to cachelines other than the four currently cached lines cannot be stored, as depicted by the sub-blocks (shapes) sitting outside the cache area in the figure.

Figure 6(b) shows a similar 4KB cache, but with 64-byte cachelines and tags in the DRAM array. The main DRAM cache is still organized with tags in the DRAM, so it is not limited to suffer from only being able to handle a small number of unique cachelines like the large-cacheline approach. The MissMap still provides a compact, yet accurate, representation of the current DRAM cache contents, but can only be used to answer queries about block residency. This example uses small cache and MissMap sizes for illustrative purposes. Consider a 4KB segment size that contains 64 cachelines. The tag size is 36 bits, and the MissMap's bit vector is 64 bits, for a total of 12.5 bytes. Assuming a storage budget of 2MB, we can store approximately 167,000 MissMap entries. Each

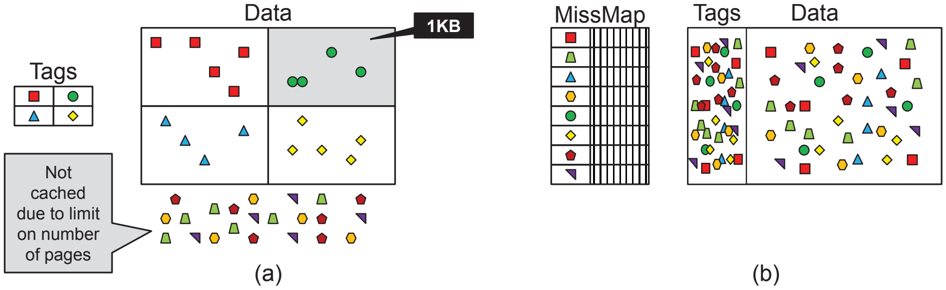


Figure 6: (a) Example large-cacheline cache consisting of four 1KB cachelines and four corresponding tag entries. (b) Example tags-in-DRAM cache with MissMap.

entry can track up to 4,096 bytes, and so the collective reach of the MissMap can cover 655MB of memory. To implement a 2MB MissMap, we propose to simply steal some space from the existing on-chip L3 cache. This is similar to how AMD's Magny-Cours processor uses part of its L3 cache for a probe filter [14]. The MissMap lookup is serialized with the DRAM cache lookup, so the overall DRAM cache latency is increased by the equivalent of an L3 cache data array access. We did evaluate a version where we dynamically monitored the DRAM cache hit rate, and if the rate increased past a threshold, then we would perform both MissMap and DRAM-cache accesses in parallel since in the common case we would have a cache hit. For our workloads, this had very little impact on performance and so we do not discuss it any further.

3.2.3 *Handling MissMap Evictions*

When the processor inserts a cacheline into the DRAM cache and the MissMap does not contain a corresponding entry, then a new MissMap entry must be allocated. The MissMap is organized like a conventional set-associative cache, and a victim entry can be chosen using standard heuristics (we simply use LRU). One potential problem is that some bits in the victimized MissMap entry may still be set because cachelines from this memory segment are still resident in the DRAM cache. By overwriting the MissMap entry, we lose this information and the ability to accurately respond to residency queries. Therefore, whenever a MissMap segment is evicted, all corresponding cache blocks must be evicted to ensure that the updated MissMap still maps all cached blocks.

We employ selective writeback to maintain consistency between the MissMap and the DRAM cache. On a MissMap eviction, we use the segment's bit vector to determine which cache blocks are still resident in the DRAM cache. We do not track per-line dirty state in the MissMap, and so for each resident line, we perform a tag lookup from the in-DRAM tags. If the line is dirty, then a writeback is sent to main memory. After all dirty blocks have been written back, the victim MissMap entry can be reallocated to a new segment. In the worst case, in which all cachelines in a segment are dirty, the resultant burst of writebacks can cause significant off-chip traffic. In practice, the segment eviction overhead is much lower because the tag-check traffic is proportional to only the number of currently cached lines from this segment, and the writeback traffic is proportional to the actual number of dirty lines.

To further cut down on tag traffic during MissMap eviction, we add a single per-segment dirty bit to each MissMap entry. If any cacheline is ever written to within this segment, the dirty bit is set and tag checks must be performed when the processor eventually evicts this MissMap entry. If the per-segment dirty bit is not set, however, then the entire segment is clean and the MissMap entry can immediately be reallocated without any further checks.

Name	Description	Total Footprint (GB)
Web-Index	Internet/web indexing	2.98
SPECjbb05	Java server benchmark	1.20
TPC-C	Online transaction processing	1.03
SPECweb05	Web server benchmark	1.02

Table 1: Workloads used in this study.

Eviction of MissMap entries may induce some additional activity depending on the underlying coherence protocol or other cache assumptions. When performing a tag check to determine if a cacheline is dirty, if the line is in a state from which evicting it would normally generate additional coherence transactions (e.g., notifying the directory of the eviction), then these actions must still occur. New or slightly modified coherence protocols can be designed or optimized to better deal with a DRAM cache and the MissMap, but that is beyond the scope of this work.

4. EXPERIMENTAL RESULTS

This section first explains our benchmarks and simulation methodology, and then presents the performance results of our proposed DRAM cache architecture.

4.1 Methodology

For systems very large DRAM caches, interesting behaviors will only be observed when running applications with correspondingly large memory footprints. To this end, we used the four multi-threaded server workloads listed in Table 1. Each workload has a memory footprint in excess of 1GB. We used the gem5 simulator for our performance evaluations [15]. The simulator combines the Michigan M5 framework for functional simulation and system emulation [16] with the Wisconsin GEMS infrastructure to model the caches, coherency, and the memory system [17]. All of our simulations model an eight-core system with the parameters listed in Table 2. Each L2 cache is shared between a pair of cores, and the L3 is shared among all eight cores. We replaced the default GEMS DRAM model with a detailed DRAM timing simulator with a FR-FCFS memory controller [18] that issues each individual activate, read/write, and precharge command while honoring DDR3 timing constraints and performing data bus scheduling. All memory traffic is processed using physical addresses from real operating system virtual-to-physical memory allocations.

For the stacked-DRAM cache, we evaluated sizes from 128MB to 1GB. Similar to previous work, we assume the DRAM array has a lower latency than the off-chip memory; our stacked-DRAM

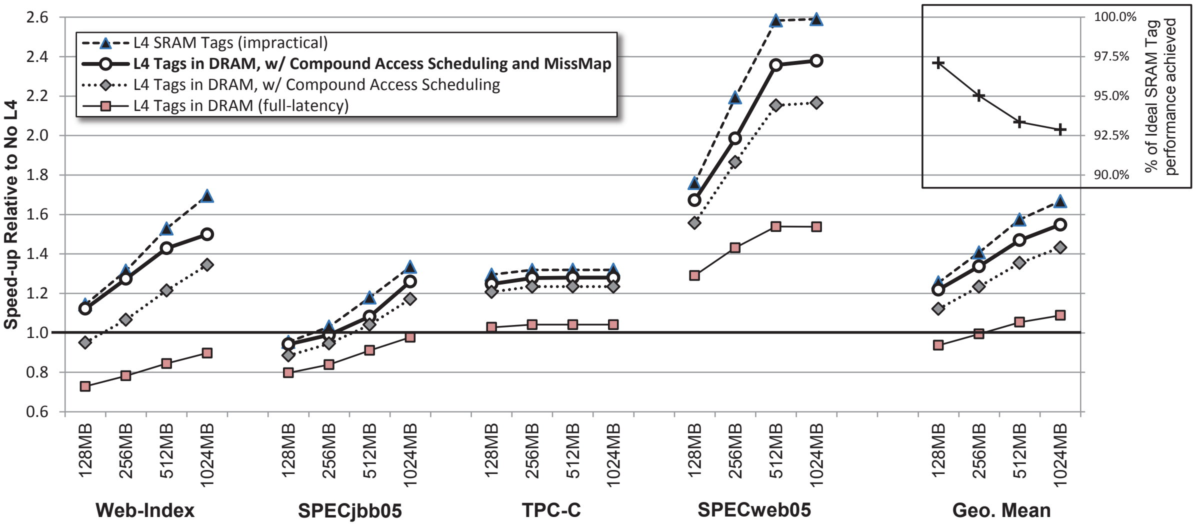


Figure 7: Performance impact of DRAM caches compared to a baseline with only an 8MB L3 cache. The inset shows how much using compound access scheduling and a MissMap with 4KB segments closes the performance gap between the baseline (no DRAM L4) and an ideal SRAM-tag implementation.

Processors	
Number of cores	8
Frequency	3.2GHz
Width	1 IPC
Caches	
I$ and D$	32KB, 2-way, 2 cycles each (per-core)
L2	2MB, 8-way, 10 cycles (per two cores)
L3	8MB, 16-way, 24 cycles (shared)
Off-Chip DRAM	
Bus frequency	800 MHz (DDR 1.6GHz)
Channels	2
Ranks	1 Rank per channel
Banks	8 Banks per rank
Row buffer size	2,048 bytes
Bus width	64 bits per channel
t_{CAS}-t_{RCD}-t_{RP}-t_{RAS}-t_{RC}	9-9-9-36-33
t_{WR}-t_{WTR}-t_{RTP}-t_{RRD}-t_{FAW}	10-5-5-5-30
Stacked DRAM	
Bus frequency	1.6GHz MHz (DDR 3.2GHz)
Channels	4
Banks	16 Banks per rank
Bus width	128 bits per channel

Table 2: System parameters used for the simulations in this study. Stacked DRAM parameters are the same as the off-chip DRAM parameters except where noted in the table.

timing latencies are approximately half of that compared to conventional off-chip DRAM,[1] whereas previous work assumed ratios of 1:3 [10] and 1:4 [4, 9]. A relatively slower off-chip memory makes the performance impact of a DRAM cache appear greater. The stacked DRAM also supports more channels, and more banks and wider buses per channel [19].

4.2 Performance

Figure 7 shows the performance of various DRAM cache options, normalized to the baseline system with only an 8MB L3 cache. For reference, we include the "lower bound" configuration with a tags-in-DRAM cache assuming naive constant-latency DRAM accesses, and the "upper bound" configuration with 64-byte cachelines supported by an impractically large on-chip SRAM tag array; these two configurations are identical to those presented in Figure 3.

We first evaluate the impact of compound access scheduling in the context of a tags-in-DRAM cache organization. This configuration has lower overall cache capacity (29-way set associative rather than 32-way). When accounting for compound-access scheduling, the performance of this L4 cache stands in stark contrast to the results when assuming a pessimistic constant-latency DRAM. Across the results, this one simple optimization gets us more than halfway from the pessimistic lower bound to the ideal SRAM-tag upper bound. Compared to the L3-only baseline, this optimization enables the tags-in-DRAM approach to achieve 90.3% of the performance of the impractical SRAM-tag version for a 128MB cache, and 88.5% for the 1GB cache.

Next, we include the MissMap with 4KB segments. The MissMap occupies a little less than 2MB of storage, so we reduce the L3 size to 6MB (12-way). The results show that the MissMap closes the performance gap by approximately another one-half. Compared to the baseline of having no L4 cache, the combination of compound-access scheduling and the MissMap provides 92.9%-97.1% of the performance delivered by the ideal SRAM-tag configuration compared to having no DRAM L4 cache.

The baseline configuration employs a single rank per channel. For many server configurations, multiple ranks per channel are used to allow even greater memory access parallelism. We also conducted simulations using two and four ranks per channel. While the raw performance benefit of stacked DRAM reduces due to reductions in memory contention in the baseline cases, the relative benefit of compound access scheduling and the MissMap remain similar (i.e., they provide performance similar to the ideal SRAM tags implementation).

5. ANALYSIS

This section provides additional data regarding the impact on off-chip main memory activity, the eviction behavior of the MissMap, and further sensitivity studies for the MissMap-based DRAM cache. We also revisit caching with very large cachelines and discuss its limitations and opportunities.

[1] Timing parameters are in multiples of the bus clock, and the stacked-DRAM bus is clocked at twice the speed of the off-chip bus.

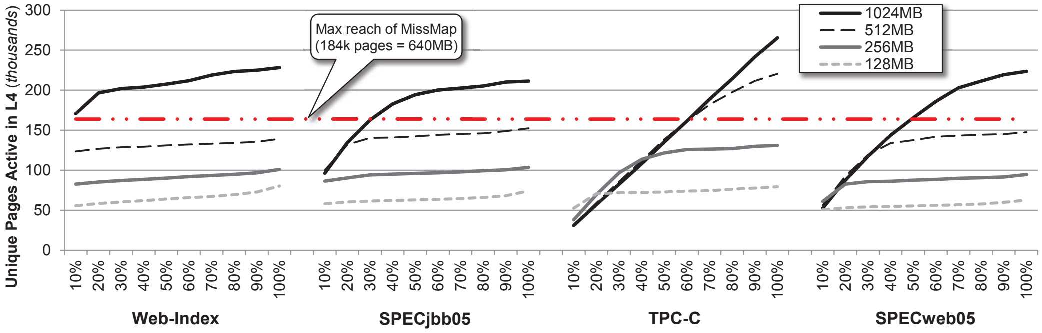
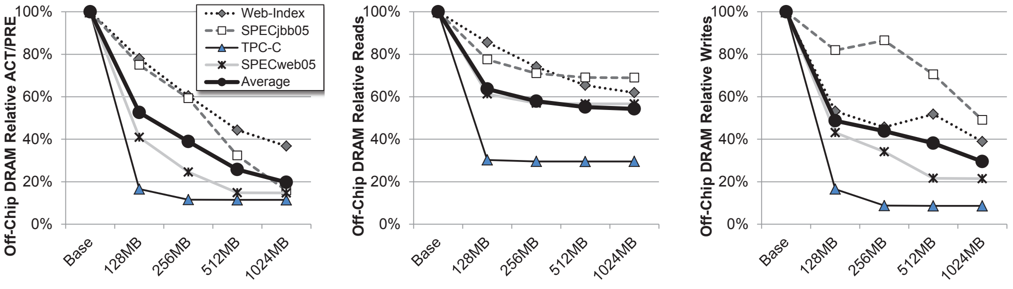


Figure 8: The number of unique 4KB segments represented in the DRAM cache, sampled at each cache insertion. The horizontal line marks the maximum reach of a 2MB MissMap.



(a) (b) (c)

Figure 9: The relative reductions in DRAM (a) activation and precharge, (b) read, and (c) write relative to an L3-only baseline.

5.1 MissMap Coverage

At smaller DRAM cache sizes, the MissMap has enough reach to track the majority of the DRAM cache's contents. The MissMap's effectiveness is predicated on the assumption that the number of unique segments active in the stacked-DRAM cache is less than the number of MissMap entries. When the number of segments exceeds this limit, the MissMap starts evicting entries, causing additional cache evictions and writebacks. To show how well this assumption holds, we tracked the number of unique 4KB segments with at least one cacheline in the DRAM cache for each workload. We collected this statistic on every installation of a new cacheline in the L4, and report this in Figure 8. For example, on SPECweb05, when using a 512MB DRAM cache, for 40% of the samples we observed 175,000 unique segments (or fewer) represented in the DRAM cache. Figure 8 also includes a reference line showing the best-case coverage of our 2MB MissMap. In most cases, apart from those involving the 1GB cache, the number of unique segments in the cache is less than the number of segments trackable by the MissMap. The coverage is not perfect because the MissMap is a set-associative structure that is still subject to conflict misses.

In the results of Figure 7, we do not increase the MissMap size with the DRAM cache size. The larger DRAM cache can hold onto cachelines from a larger number of unique segments, which in turn places more capacity pressure on the MissMap. Performance for the MissMap-based L4 cache continues to improve with larger DRAM sizes, but not at the same rate as the impractical SRAM-tag upper bound. In particular for the 1GB cache, a 2MB MissMap can only track 640MB of memory in the best case. Sensitivity studies in Section 5.4 explore the impact of varying the MissMap versus L3 cache allocation.

5.2 Off-chip Main Memory Effects

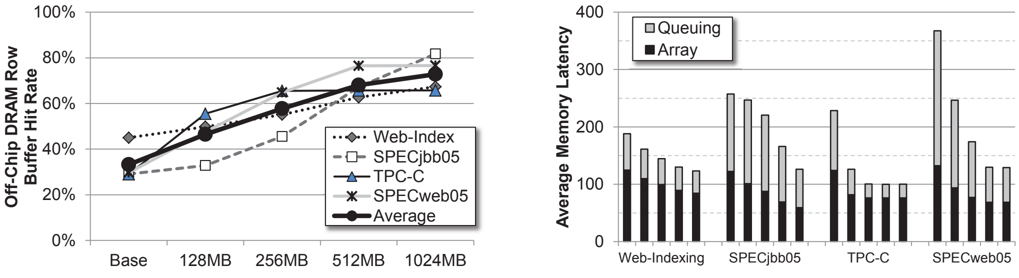
Part of the benefit of using DRAM caches comes from having faster cache hits served directly from the stacked cache. The conversion of off-chip traffic into in-stack cache hits has the additional benefit of reducing off-chip activity. This has direct performance and power implications.

Figure 9 shows the reduction in the number of activate and precharge commands, reads, and writes sent to the off-chip memory. (Every row that is opened must eventually be closed, so the numbers of ACT and PRE commands are equal.) All of these activities are normalized to an L3-only baseline. Every off-chip command avoided represents a savings in power due to the high-power circuits required to signal across the relatively long distance from the processor package to the memory DIMMs on the motherboard.

Figure 10(a) shows how larger DRAM caches (using the MissMap) increase the row buffer hit rate of *off-chip* main memory, which improves the average memory service time. The row buffer hit rate for a baseline configuration (8MB L3 only) is provided for reference. The reduction in memory traffic also helps reduce the queuing latency for each request (i.e., the time spent waiting at the memory controller before being issued to the off-chip DRAM). Figure 10(b) shows how the improved row buffer hit rates and the reduced traffic combine to improve memory latencies.

5.3 MissMap Eviction Behavior

Figure 8 showed that the number of active segments in the DRAM cache is usually low enough to be effectively tracked by the MissMap. There will remain some MissMap entries that are evicted (mostly due to conflict misses in the MissMap) that in turn can potentially generate large bursts of tag-lookup and writeback traf-



(a) (b)

Figure 10: (a) Row buffer hit rates for an L3-only baseline and DRAM cache using the MissMap. (b) Average memory latency for the same configurations; queuing delay is time spent from arrival at the memory controller until the first DRAM command has been issued for this request, and the array latency encompasses the remaining time until the data have been transmitted.

Workload Name	% Clean MissMap Evictions	Dirty Lines per Dirty Segment	
		128MB	1GB
Web-Index	99.7%	0.02	26.8
SPECjbb05	55.7%	1.06	29.3
TPC-C	74.7%	3.80	4.93
SPECweb05	72.2%	1.61	16.2

Table 3: Additional statistics on MissMap evictions.

fic. In practice, however, even when a MissMap entry has been evicted, the amount of additional traffic is quite low. Table 3 lists the percentage of all MissMap evictions in which the corresponding segment was completely clean (i.e., none of the cachelines were ever written to, and so the entry's dirty bit was clear). In these cases, no extra tag-lookup or writeback traffic will be generated. For SPECjbb, SPECweb, and TPC-C, approximately 56%-75% of all MissMap evictions were for clean segments. The Web-Index workload is an extreme case because it performs a large number of queries against a large, static indexing data structure. The DRAM cache capacity is mostly used to store portions of the web index, and so almost all MissMap evictions are for clean segments.

For the segments that contain modified cachelines, Table 3 also lists the average number of dirty lines written back per dirty segment. The number of dirty cachelines written increases with the size of the DRAM cache. This makes sense because with a larger cache, a larger fraction of a segment's cachelines will remain resident in the cache, so when the MissMap entry is evicted, there will likely be more cachelines still in the cache (including dirty ones). The reason the number of dirty lines per segment is so low in the small-cache case is that the dirty lines are evicted on their own due to the cache's regular replacement policy. When the overall segment MissMap entry gets evicted, the dirty bit will be set, but it will not find the dirty cacheline because that line was already evicted.

One of the reasons why the MissMap generates relatively little traffic due to evictions is in the mismatch between useful cacheline lifetimes and how long an entry stays in the MissMap. Figure 11 shows the average number of cycles for which an entry remains in the MissMap before eviction for a 128MB DRAM cache. The figure also shows the average number of cycles per cacheline from insertion until last use. Cachelines are brought into the cache, used, and then evicted long before the corresponding MissMap entry ever gets evicted. As a result, by the time a MissMap entry is old enough to get evicted, most of its cachelines have also long since been evicted. Therefore, the MissMap will have few bits set and correspondingly few tag-lookups and writebacks to perform.

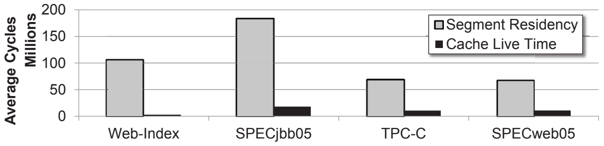


Figure 11: The average number of cycles that an entry stays in the MissMap, and the average number of cycles that a cacheline is alive in a 128MB DRAM cache.

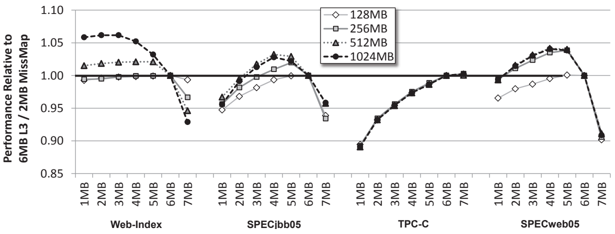


Figure 12: Performance sensitivity on the fraction of the L3 used for the MissMap. Sizes on the x-axis indicate the remaining capacity left for the L3 cache.

5.4 MissMap vs. L3 Tradeoff

For all MissMap performance evaluations so far, we have used 2MB the processor's 8MB L3 cache for the MissMap. Other partitionings are possible, and Figure 12 shows the performance impact of these other options. Apart from TPC-C, cannibalizing a larger portion of the L3 (e.g., 4MB/4MB) provides a small performance boost, especially for the bigger DRAM caches where having more MissMap entries can better track the contents of the larger cache. If too much of the L3 cache is used for the MissMap, then the decrease in L3 hit rates starts to cause performance to drop off again.

While these performance results suggest using a larger MissMap (4MB), we are uncomfortable recommending such a design choice because many servers run applications with smaller memory footprints that would benefit much more from the L3 cache. These results suggest further opportunities for the application or operating system to determine the partitioning. The incorporation of knowledge about application-level behavior and memory needs should provide better performance than a simple static configuration.

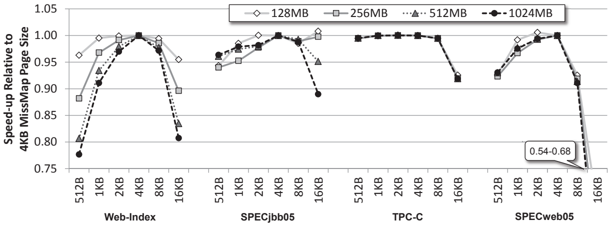


Figure 13: Performance sensitivity on the size of the segments tracked by the MissMap.

5.5 MissMap Segment Size

Throughout this paper (apart from a few illustrative examples), we have assumed that the MissMap tracks memory with a 4KB granularity. Figure 13 shows how performance changes with different MissMap segment sizes. All results in the figure are normalized to a 4KB segment size. Each MissMap entry has an approximately fixed overhead for storing the tag. Beyond that, the remainder of the storage is used for the valid-bit vector. Larger segment sizes effectively amortize the tag overhead over a longer bit-vector. Doubling the segment size doubles the effective reach per entry, reduces the number of MissMap entries for a fixed hardware budget, and results in an overall net increase in the best-case coverage of the MissMap. The problem is that the reduction in the number of MissMap entries limits the number of unique segments that the MissMap can track. Furthermore, segment sizes greater than 4KB consist of a contiguous span of the physical address space, which may have little correlation to the virtual address space. The large segment can contain multiple unrelated virtual pages that do not exhibit much, if any, mutual locality.

Going the other way, a reduction in the segment size reduces the coverage of the MissMap. Each halving of the segment size correspondingly halves the coverage of a MissMap entry. Unfortunately, the number of additional MissMap entries that can be supported is less than double, and so there is a net decrease in the overall coverage. For the vast majority of workloads and DRAM cache sizes considered, a segment size of 4KB performs best. This is likely not a coincidence with operating system page sizes. Various aspects of the software stack (e.g., compiler-optimized code) are tuned to handle data in 4,096-byte parcels.

5.6 Comparison to Sub-blocked Caches

We also considered an L4 cache implementation using sub-blocking [13, 20]. We used 2KB blocks (to match the row-buffer size), consisting of thirty-two 64-byte sub-blocks. For the computation of the tag overheads of the sub-blocked caches, we assume that each block needs tag bits (about four bytes), a valid bit, and replacement bits (five bits for a 32-way set associative organization). Each sub-block has an overhead of one byte, consisting of coherence state and sharer information. The total overhead is 36.75 bytes per block, and the overall overhead is listed in Figure 14. Because the sub-blocked cache overhead increases with the L4 size, we include performance curves for both 2MB and 4MB MissMaps. For the 128MB and 256MB L4 cache sizes, the MissMap (with compound access scheduling) significantly outperforms the sub-blocked cache. At these sizes, block-level thrashing causes the sub-blocked cache to perform relatively poorly. At 512MB, the thrashing is reduced enough that the sub-blocked cache starts performing well, but at this point, the 2MB MissMap still delivers the same level of performance but at a much lower overhead (2MB versus 9.2MB). When the L4 size is increased to 1024MB, the sub-blocked cache finally outperforms the 2MB MissMap, but requires over 9× more tag overhead. The 4MB MissMap (with correspondingly reduced L3 cache capacity) provides the same average performance as the sub-blocked cache, but again at significantly lower overhead.

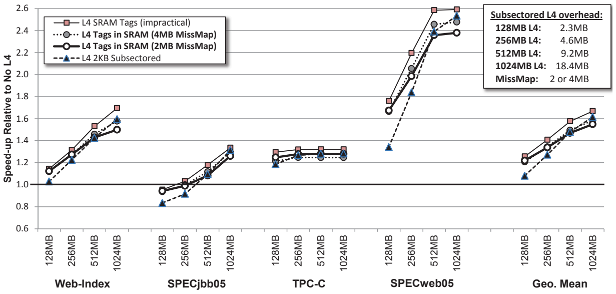


Figure 14: Comparison of L4 cache with 2MB and 4MB MissMaps against a sub-blocked L4 cache (using 2KB blocks).

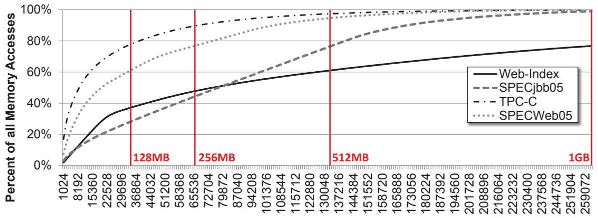


Figure 15: Cumulative distribution of the 4KB pages responsible for the largest number of memory accesses.

5.7 Comparison to Large Cachelines

Our earlier performance results did not provide a comparison against large-cacheline approaches. We implemented a DRAM cache with 4KB cachelines and a full SRAM tag array. We also implemented multiple versions of selectively filtered caches (i.e., CHOP) [9]. At least for our workloads, it appears that there is insufficient locality of hot pages to make 4KB cache lines work well.

The previous CHOP study focused on a 128MB DRAM cache using 4KB pages. Figure 15 shows the most frequently accessed 1GB worth of memory pages; the plot also includes a line demarcating the top 128MB's worth of memory. For the Web-Index and SPECjbb workloads, a perfect page cache that could omnisciently choose these most frequently accessed pages could still only serve 30-40% of requests. Due to the large size of the hot working set, setting the adaptive filter cache's "hotness" threshold too low generated too much off-chip traffic and would significantly degrade performance; setting the threshold too high left the DRAM cache largely unused and so provided little benefit over not having the cache at all. While we did not observe the same benefits of this approach as the prior work, their techniques will still be useful for workloads with high spatial locality. Future research may consider DRAM cache organizations that can simultaneously handle (or at least adaptively switch between) multiple caching granularities.

6. RELATED WORK

During the past several years, many researchers have studied ways to make use of 3D stacked memory. Most of these works either assume that all of main memory can be stacked [6, 7, 21, 22] or that the stacked DRAM is deployed as a very large last-level cache [4, 9, 10, 23, 24]. Other 3D memory research has explored the opportunities for stacked SRAM caches [5, 25–28], stacked non-volatile memories [29], and hybrids of memory technologies [30]. In the following, we will only focus on those works targeting the problem of implementing DRAM caches and other directly related works.

As already discussed, several earlier works have proposed organizing stacked DRAM with very large (i.e., page-sized) cache lines to reduce the tag overhead [9, 10]. While these approaches may be quite effective for workloads that demonstrate significant page-level spatial locality, we believe the DRAM cache organization proposed in this work is more flexible and generally applicable due to its use of conventional (small) cacheline sizes. Zhao et al. also considered modest levels of sub-blocking the cachelines (i.e., grouping together 2, 4 or 8 cachelines), but found that performance fell off beyond a sub-blocking level of two [4]. While not explicitly concerned with 3D-stacked DRAM caches, Zhang et al. considered large off-chip caches with the tags stored along with the data in the off-chip cache, and then used a small on-chip tag cache [31]. For workloads with large working sets, the tag cache will suffer from capacity problems. Other works have considered replacement policies for large DRAM caches [23] and the refresh implications of placing the DRAM cache on top of a hot processor chip [24], but these works do not deal with the tag management problems addressed by our work.

Sector or sub-block caches associate a tag with the data with a power of two number of cachelines [13, 20]. Seznec's decoupled sector cache allows a few tags (often two) to be associated with the same data storage to reduce fragmentation, especially with direct-mapped data arrays [32]. The MissMap goes much further, often associating a tag with 64 cache lines. Rothman and Smith's pool of subsectors approach has structural similarities to the MissMap [33]. The pool of subsectors uses a number of sectors with fewer cache lines (subsectors) than necessary to cache all of the data corresponding to all of the subsectors. Each sector maintains one hardware pointer per subsector to indicate the physical location of the allocated cache line (null pointer if the subsector is not cached). The MissMap is structurally different in that it uses simpler bit vectors rather than hardware pointers, which is enabled by the observation that tracking cache content versus content location can be decoupled. Despite any structural similarities with the decoupled sector cache and pool of subsectors, the MissMap is fundamentally different because it fulfills a different functional purpose: avoiding unnecessary stacked DRAM cache accesses.

Beyond die-stacking research, past work has proposed hardware mechanisms with similarities to our MissMap. Most of these related works have employed bit-vector style tracking for filtering purposes. Lin et al. use a "density vector" to suppress prefetch requests that are unlikely to be useful [34]. In Moshovos et al.'s Jetty work, they proposed one version (vector-exclude-jetty) that uses a small bit vector to represent short (4-8) contiguous sequences of memory blocks [35]. Jetty allows the coherence system to avoid some unnecessary cache snoops; our MissMap differs in that we maintain precise tracking of cache contents whereas Jetty only provides a prediction (but Jetty is also smaller). Cantin et al. proposed "region coherence" for a similar goal of reducing unnecessary coherence traffic between cores [36]. Their region coherence array (RCA) has some operational similarities with the MissMap, in particular the requirement of cache evictions when a MissMap/RCA entry gets evicted. Even though our implementation of the MissMap shares a lot of structural similarities with these and other works, the important insight used in this work is that the problem of tracking which blocks are in a cache can be separated from the problem of finding these blocks. The MissMap is one possible mechanism that leverages this observation to enable a scalable DRAM cache that uses normal cacheline sizes.

7. CONCLUSIONS

This work revisited the seemingly "bad" idea of storing tags along with data in a DRAM-based 3D stacked cache supporting 64-byte cachelines. By considering the actual operation of a DRAM device, rather than abstracting it away as a constant-latency delay, we show that scheduling DRAM-cache tag and data lookups as compound accesses can make this tags-in-DRAM cache organization practical. Furthermore, by decoupling the problem of tracking *which* cache lines are resident in the DRAM cache from the problem of tracking *where* in the cache these lines can be found, the MissMap allows us to compactly track a very large amount of memory. The combination of these two techniques delivers an overall DRAM cache architecture that provides a substantial fraction of the performance benefit only attainable by an ideal DRAM cache assisted with an impractically large SRAM tag array.

In the longer term, operating system-visible memory is probably still desirable. Many (non-server) applications have memory footprints that fit within the capacity of a stacked DRAM, so it would be simpler to directly map all of that program's memory directly into the stacked DRAM from the start of execution. This avoids the need to track the program to determine hot pages, and it also avoids the problems associated with caches (i.e., tag overheads, and transferring data from off-chip to the stacked DRAM and back).

Even for applications with working sets that do not fit in the stacked DRAM, there exists significant higher-level information (application-level) that can be leveraged to select the subset of memory that should be mapped to the stacked DRAM. Implementing this requires many changes throughout the software stack (operating systems, compilers, runtime, and/or the user applications themselves), and these changes take time. Such software-exposed stacked memory may start in the embedded and mobile domains where companies can own, control, and co-optimize the entire hardware-software stack. Until this happens in the high-performance domain, however, stacked DRAM will likely need to be deployed as a cache, and this work provides a reasonable way forward.

Acknowledgments

Work by Hill was performed largely while he was on sabbatical at AMD Research. At Wisconsin, Hill is supported in part by NSF (CNS-0720565, CNS-0916725, and CNS-1117280) and Sandia/DOE (#MSN123960/DOE890426).

8. REFERENCES

[1] S. Gupta, M. Hilbert, S. Hong, and R. Patti, "Techniques for Producing 3D ICs with High-Density Interconnect," in *Proc. of the 21st Intl. VLSI Multilevel Interconnection Conference*, Waikoloa Beach, HI, September 2004.

[2] Y. Xie, G. H. Loh, B. Black, and K. Bernstein, "Design Space Exploration for 3D Architecture," *ACM Journal of Emerging Technologies in Computer Systems*, vol. 2, no. 2, pp. 65–103, April 2006.

[3] Y. Deng and W. Maly, "Interconnect Characteristics of 2.5-D System Integration Scheme," in *Proc. of the Intl. Symp. on Physical Design*, Sonoma County, CA, April 2001, pp. 171–175.

[4] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Proc. of the 25th Intl. Conf. on Computer Design*, October 2007, pp. 55–62.

[5] 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 *Proc. of the 39th Intl. Symp. on Microarchitecture*, Orlando, FL, December 2006.

[6] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *Proc. of the 35th Intl. Symp. on Computer*

Architecture, Beijing, China, June 2008.

[7] 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 *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, Bangalore, India, January 2010, pp. 429–440.

[8] M. Facchini, T. Carlson, A. V. M. Palkovic, F. Catthoor, W. Dehaene, L. Benini, and P. Marchal, “System-Level Power/Performance Evaluation of 3D Stacked DRAMs for Mobile Applications,” in *Proc. of the Conf. on Design, Automation and Test in Europe*, Nice, France, April 2009, pp. 923–928.

[9] 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 *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, January 2010, pp. 1–12.

[10] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, “Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support,” in *Proc. of the ACM/IEEE Int’l Conf. for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, November 2010, pp. 1–11.

[11] J. Torrellas, M. S. Lam, and J. L. Hennessy, “False Sharing and Spatial Locality in Multiprocessor Caches,” *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 651–663, 1994.

[12] J. Jalminger and P. Stenström, “Improvement of Energy-Efficiency in Off-Chip Caches by Selective Prefetching,” *Microprocessors and Microsystems*, vol. 26, no. 3, pp. 107–121, April 2002.

[13] J. S. Liptay, “Structural Aspects of the System/360 Model 85, Part II: The Cache,” *IBM Systems Journal*, vol. 7, no. 1, pp. 15–21, 1968.

[14] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor,” *IEEE Micro Magazine*, pp. 16–29, March–April 2010.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “gem5: A Multiple-ISA Full System Simulator with Detailed Memory Model,” *Computer Architecture News*, vol. 39, June 2011.

[16] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” *IEEE Micro Magazine*, vol. 26, no. 4, pp. 52–60, July–August 2006.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset,” *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, November 2005.

[18] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, “Memory Access Scheduling,” in *Proc. of the 27th Intl. Symp. on Computer Architecture*, June 2000, pp. 128–138.

[19] 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 *Proc. of the Intl. Solid-State Circuits Conference*, San Francisco, CA, February 2011.

[20] M. D. Hill and A. J. Smith, “Experimental Evaluation of On-Chip Microprocessor Cache Memories,” in *Proc. of the 15th Intl. Symp. on Computer Architecture*, Ann Arbor, MI, June 1984, pp. 158–166.

[21] T. H. Kgil, S. D’Souza, A. G. Saidi, N. Binkert, R. Dreslinski, S. Reinhardt, K. Flautner, and T. Mudge, “PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor,” in *Proc. of the 12th Symp. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2006, pp. 117–128.

[22] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, “Bridging the Processor-Memory Performance Gap with 3D IC Technology,” *IEEE Design and Test of Computers*, vol. 22, no. 6, pp. 556–564, November–December 2005.

[23] G. H. Loh, “Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy,” in *Proc. of the 42nd Intl. Symp. on Microarchitecture*, New York, NY, December 2009, pp. 201–212.

[24] M. Ghosh and H.-H. S. Lee, “Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs,” in *Proc. of the 40th Intl. Symp. on Microarchitecture*, Chicago, IL, December 2007.

[25] K. Puttaswamy and G. H. Loh, “Implementing Caches in a 3D Technology for High Performance Processors,” in *Proc. of the Intl. Conf. on Computer Design*, San Jose, CA, October 2005.

[26] P. Reed, G. Yeung, and B. Black, “Design Aspects of a Microprocessor Data Cache using 3D Die Interconnect Technology,” in *Proc. of the Intl. Conf. on Integrated Circuit Design and Technology*, Austin, TX, May 2005, pp. 15–18.

[27] Y.-F. Tsai, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, “Three-Dimensional Cache Design Using 3DCacti,” in *Proc. of the Intl. Conf. on Computer Design*, San Jose, CA, October 2005.

[28] B. Black, D. Nelson, C. Webb, and N. Samra, “3D Processing Technology and its Impact on IA32 Microprocessors,” in *Proc. of the 22nd Intl. Conf. on Computer Design*, San Jose, CA, October 2004, pp. 316–318.

[29] W. Zhang and T. Li, “Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures,” in *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Raleigh, NC, September 2009, pp. 101–112.

[30] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell, “Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy,” in *Proc. of the 15th Intl. Symp. on High Performance Computer Architecture*, Raleigh, NC, February 2009, pp. 262–274.

[31] Z. Zhang, Z. Zhu, and X. Zhang, “Design and Optimization of Large Size and Low Overhead Off-Chip Caches,” *IEEE Transactions on Computers*, vol. 53, no. 7, pp. 843–855, July 2004.

[32] A. Seznec, “Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost,” in *Proc. of the 21st Intl. Symp. on Computer Architecture*, Chicago, IL, April 1994, pp. 384–393.

[33] J. B. Rothman and A. J. Smith, “The Pool of Subsectors Cache Design,” in *Proc. of the 1999 Intl. Conf. on Supercomputing*, Rhodes, Greece, June 1999, pp. 31–42.

[34] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak, “Filtering Superfluous Prefetches using Density Vectors,” in *Proc. of the 19th Intl. Conf. on Computer Design*, Austin, TX, September 2001, pp. 124–132.

[35] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, “JETTY: Snoop Filtering for Reduced Energy Consumption in SMP Servers,” in *Proc. of the 7th Intl. Symp. on High Performance Computer Architecture*, Monterrey, Mexico, January 2001, pp. 85–96.

[36] J. F. Cantin, M. H. Lipasti, and J. E. Smith, “Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking,” in *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005, pp. 246–257.