Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap

This work efficiently enables conventional block sizes for very large die-stacked DRAM caches with two innovations: it makes hits faster with compound-access scheduling and misses faster with a MissMap. The combination of these mechanisms enables the new organization to deliver performance comparable to that of an idealistic DRAM cache that employs an impractically large SRAM-based on-chip tag array.

Gabriel H. Loh
Advanced Micro Devices

Mark D. Hill
University of Wisconsin—Madison

Die-stacking technology enables multiple layers of DRAM to be integrated with multicore processors. A promising use of stacked DRAM is as a cache, because its capacity would likely be insufficient to serve as all of main memory (except perhaps in specific market segments). However, a 1-Gbyte DRAM cache with 64-byte blocks could require 96 Mbytes of tag storage. Placing these tags on chip is impractical, larger than on-chip Level-3 (L3) caches. However, putting them in DRAM can be slow (two full DRAM accesses for tags and data). Larger blocks and subblocking are possible, but they're less robust due to fragmentation.

This work efficiently enables conventional block sizes for very large die-stacked DRAM caches with two innovations, as in our paper for the 44th Annual IEEE/ACM International Symposium on Microarchitecture.[1] First, via compound-access scheduling, 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, an efficient hardware structure to track the presence or absence of cache blocks, which eschews stacked-DRAM accesses on all misses. As in extreme subblocking, our implementation of the MissMap stores a vector of block-valid bits for each page in the DRAM cache. Unlike conventional subblocking, the MissMap points to many more pages than can be stored in the DRAM cache (making the effects of fragmentation rare) and doesn't point to the exact physical location of a block (but defers to the off-chip tags). For the evaluated large-footprint commercial workloads, the proposed cache organization delivers 92.9 percent of the performance benefit of an ideal 1-Gbyte DRAM cache with an impractical 96 Mbytes of on-chip static RAM (SRAM) tag array.

Challenges of implementing large caches

Adding DRAM to a processor to implement a large cache might sound simple,

Published by the IEEE Computer Society

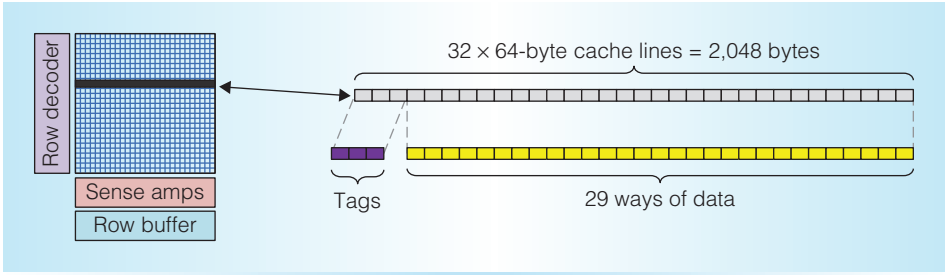

Figure 1. Mapping a cache set's tags and data to a single DRAM row. Three 64-byte blocks store the 29 tags that correspond to the data located in the remaining 29 64-byte blocks. Tags and data are colocated in the same DRAM row for faster, more efficient access.

but several challenges prevent the immediate widespread adoption of stacked-DRAM caches. Using DRAM as a cache requires the implementation of a tag store. A tag entry for a single cache line could require up to 5 or 6 bytes. For example, assuming 48-bit physical addresses, the tag itself is approximately 4 bytes, and the tag entry could include other metadata (such as least recently used [LRU] counters, coherence state, and sharer information). A 128-Mbyte DRAM can store 2^{21} 64-byte cache lines, which, at 6 bytes of tag overhead each, results in a total tag array size of 12 Mbytes. This is already larger than most L3 caches today. For a 1-Gbyte DRAM, this tag overhead increases to 96 Mbytes.

Large cache line sizes

Other researchers have noted the tag overhead for very large caches, and a common approach for avoiding this overhead is to increase the cache line size.[1-3] For example, using a 4-Kbyte cache line reduces the total number of cache lines to only 32,768 for a 128-Mbyte cache. Very large cache lines can have fragmentation problems; in the worst case, only a single 64-byte subblock will be used from each cache line. Transferring 4 Kbytes of data at a time can also cause significant off-chip bus contention, leading to substantial queuing and backpressure delays throughout the cache hierarchy. Similarly, large cache lines can cause severe false sharing in multithreaded applications, although the fragmentation problem tends to be of greater concern.[4] Copying unused blocks back and forth also wastes bandwidth and power.[5] Moreover, supporting 4-Kbyte cache lines that are larger than typical DRAM row buffers (1 to 2 Kbytes) leads to more DRAM commands and contention among multiple banks.

Subblocking

Subblocked caches can alleviate fragmentation and false-sharing problems.[6] Like caching very large lines, multiple conventional (for example, 64-byte) cache lines are grouped together into a single aligned superblock. The tag entry maintains a single address tag for the entire large cache line, but provides valid and coherence bits for each individual subblock. Assuming an overhead of 8 bits per 64-byte subblock, each tag entry now requires 68 bytes. A 1-Gbyte subblocked DRAM cache needs 18.4 Mbytes. The subblocked cache uses only bandwidth and power to fetch requested subblocks, compared to a large-cache-line approach that transfers the entire cache line. Subblocking still doesn't address the problem of tracking only a limited number of cache lines. For workloads with low spatial locality, the large cache lines can result in high miss rates. Selective caching of the most frequently used cache lines can alleviate some of this effect,[1,2] but applications with large active working sets will still suffer.

Combining tags and data in the DRAM

An alternative approach stores the tags directly in the DRAM array with the data (see Figure 1). For example, a 2-Kbyte DRAM row could store up to 32 64-byte blocks, but the row can also be partitioned into 29 64-byte cache lines, with the remaining 192 bytes used for tags. The 29 data blocks

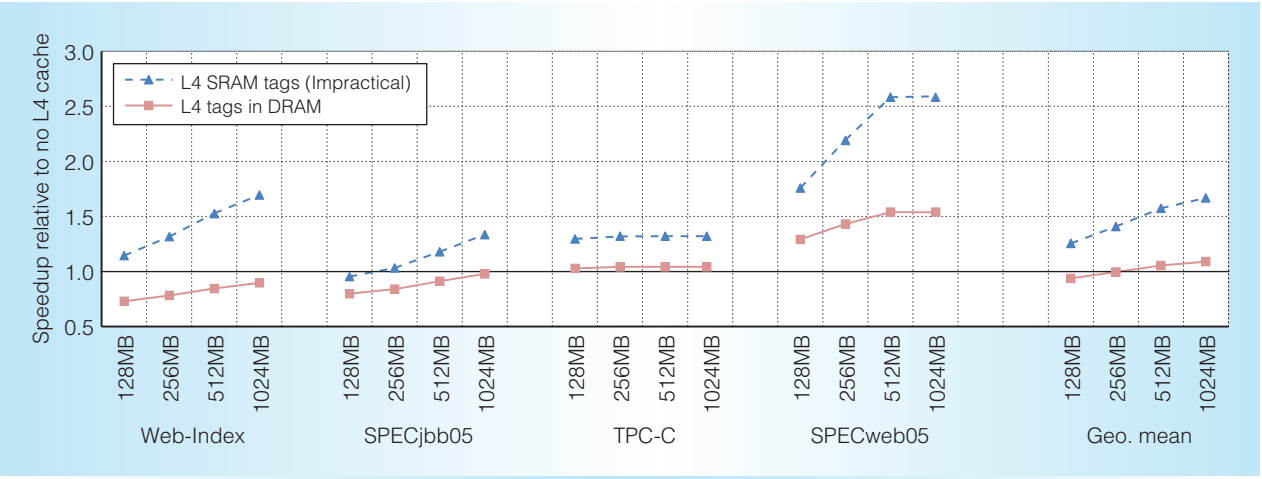

Figure 2. Performance benefits of large on-chip caches. The ''tags in DRAM'' results model a DRAM cache using pessimistic assumptions on DRAM accesses, and subsequently show poor performance. The ''SRAM tags'' results consider an impractical tag array implementation, but show that if this could be built, the performance potential is significant.

need 29 × 6 = 174 bytes for their tag entries. For this configuration, 18 bytes are left unused; these could be employed for better replacement policies, profiling, or other uses, but we don't explore these opportunities in this article. 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 (about 9.4 percent in this example).

Although previous work has discussed storing tags in DRAM,[1,3] this approach 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 might be faster than conventional off-chip DRAMs, their latencies are still considerably longer than SRAMs. Furthermore, the DRAM often requires a third access to update replacement information (such as LRU counters) and coherence state in the tag entry. Although this extra DRAM access is off the critical path of serving requests, it decreases DRAM bank availability, causing more bank contention.

Potential of large DRAM caches

Figure 2 shows the performance impact of implementing very large caches, ranging from 128 Mbytes up to 1 Gbyte, on four memory-intensive commercial workloads suffering from high memory contention (see the ''Experimental evaluation'' section for details). We normalized all results in the figure to a baseline eight-core processor with an 8-Mbyte L3 cache and no stacked DRAM. The top curve shows the performance for a DRAM L4 cache supported by an ideal SRAM tag array (that is, 12 Mbytes of tag array for the 128-Mbyte L4 cache, and 96 Mbytes for the 1-Gbyte L4 cache). The bottom curve is for an L4 cache 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. Previous studies assuming constant latencies for the DRAM accesses showed similar results.[1,3]

A practical DRAM cache supporting conventional block sizes

In designing our DRAM cache, we had a few objectives regarding performance and overheads.

1. Support 64-byte cache lines to avoid fragmentation and minimize the bandwidth of wasted transfers.

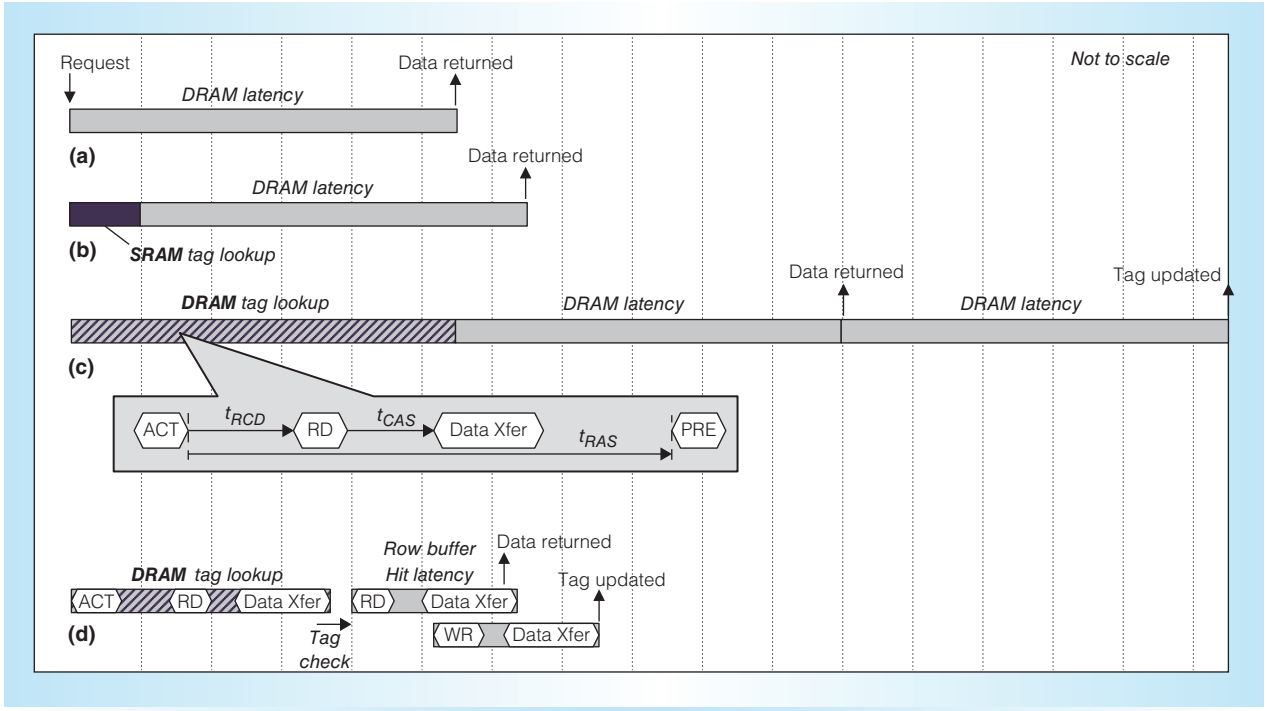

Figure 3. Timing diagrams for a DRAM cache hit: ideal case that requires a priori knowledge about the location of the requested block (a); using a static RAM (SRAM) tag array (b); placing tags in the DRAM, but using pessimistic constant-latency DRAM, in which every DRAM access requires the worst-case access latency (c); and placing tags in the DRAM with row-buffer-aware compound-access sequencing (d).

2. Keep SRAM overhead as low as possible.
3. On a cache hit, keep the latency as close as possible to a single DRAM access.
4. On a cache miss, ensure that the request proceeds to main memory as quickly as possible without a stacked-DRAM access.

To support objectives 1 and 2, we used a tags-in-DRAM organization with 64-byte cache lines. In the following sections, we explain how we attempted to achieve goals 3 and 4 despite the tags-in-DRAM approach's latency problems.

Compound-access scheduling: Making hits faster

Ideally, a DRAM cache hit requires only a single DRAM access latency, as Figure 3a shows. This is difficult to achieve because a tag lookup is typically required to determine the actual location (that is, physical way or column) of the requested data. Figure 3b shows the case when an SRAM tag lookup provides this information and provides an overall latency close to the ideal case.

The results from Figure 2 showed that the latency of performing two DRAM accesses per cache hit caused some serious performance deficiencies compared to the ideal DRAM caches. Figure 3c illustrates this access sequence. 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 memory controller must issue a sequence of commands. Assuming the requested row isn't already open, the memory controller issues an activation command that retrieves the selected row and latches the values in a row buffer. The memory controller can then issue a read 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 command that writes the row buffer's contents back to the DRAM bit-cell array. The inset of Figure 3c illustrates this command sequence. If a requested row has already been loaded into the row buffer

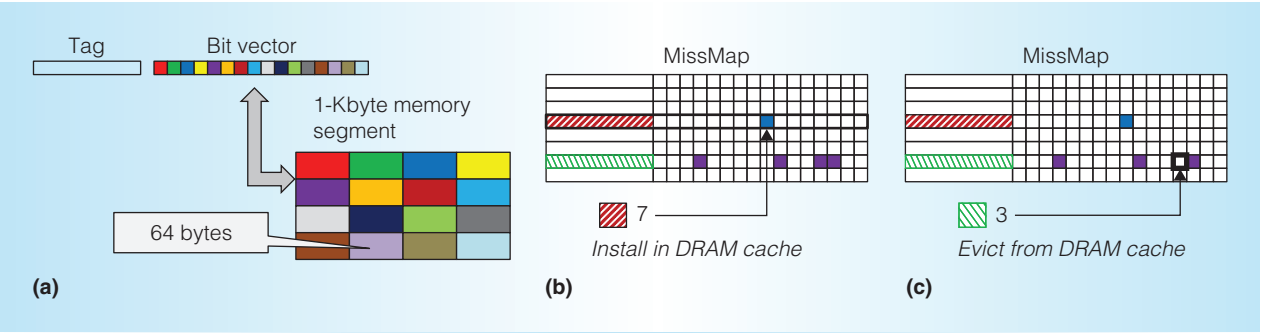

Figure 4. Structure of and operations on an example MissMap. MissMap entry covering a 1-Kbyte memory segment (a), setting a MissMap bit when installing a line in the DRAM cache (b), and clearing a MissMap bit when evicting a line from the DRAM cache (c).

(called a *row buffer hit*), the activation command can be skipped.

We assume that a single physical DRAM row holds both tags and data, as Figure 1 shows. 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 cache data's column address in the row buffer, and the memory controller can immediately retrieve the data. By reserving the row buffer, we have guaranteed a row buffer hit for the data access, as Figure 3d shows. Comparing this to the SRAM tag case of Figure 3b, we have effectively traded the SRAM lookup latency for a row buffer hit's latency. Furthermore, the controller can keep the row reserved so that any necessary tag updates also hit in the row buffer. This optimization is the result of considering the entire sequence of tag and data accesses, and coordinating the command scheduling of all of the individual accesses as a single compound access; the overall scheduling strategy is called compound-access scheduling.

MissMap: Making misses faster

Our last design objective was to avoid the DRAM cache access on a miss. A conventional tag array serves two primary purposes: first, to track the cache's contents (that is, exactly what blocks currently reside in the cache), and second, to record the location of each block within its set. The conventional tag array implicitly tracks the cache's contents 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.

To efficiently track which blocks the DRAM cache currently stores, we decouple the block-residency and block-location problems. A simple MissMap data structure answers queries about cache block residency, and the in-DRAM tags handle the location problem.

In our current implementation, each MissMap entry tracks the cache lines 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 cache line. Figure 4a shows a MissMap entry for a 1-Kbyte segment.

Each time the processor inserts a new cache line into the DRAM cache, the processor also looks up the MissMap entry corresponding to the segment containing the new cache line (allocating a new entry if necessary) and sets the bit in the entry corresponding to the inserted cache line (see Figure 4b). When the processor evicts a cache line from the DRAM cache, the bit

in the MissMap entry will be cleared (see Figure 4c). The MissMap maintains a consistent record of the current DRAM cache contents; by checking to see if a cache line'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 cache lines from the entire segment are currently in the cache. In this fashion, DRAM cache misses bypass the DRAM cache lookup entirely.

At first blush, the MissMap looks similar to the tag array for a large line cache and just performs subblocking in disguise. The critical difference is that the MissMap is overprovisioned to track more memory than can fit in the DRAM cache, whereas the tag-array for a large line cache tracks exactly the number of cache lines that fit in the cache. The MissMap provides a compact, accurate representation of the current DRAM cache contents, but it can be used only to answer queries about block residency.

When the processor inserts a cache line into the DRAM cache and the MissMap doesn't contain a corresponding entry, the DRAM cache controller must allocate a new MissMap entry. 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 might still be set because cache lines from this memory segment still reside 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.

Experimental evaluation

Here, we explain our benchmarks and simulation methodology, and we present the performance impact of our proposed DRAM cache architecture.

Methodology

For systems with very large DRAM caches, we observe interesting behaviors only when running applications with correspondingly large memory footprints. To this end, we use four multithreaded server workloads, each having a memory footprint in excess of 1 Gbyte. We use the gem5 simulator to model an eight-core system.[7] Each pair of cores shares a 2-Mbyte L2 cache, and all eight cores share an 8-Mbyte L3 cache. For the stacked-DRAM cache, we evaluated sizes from 128 Mbytes to 1 Gbyte. Similar to previous work, we assume the stacked DRAM array has a lower latency than the off-chip double data rate (DDR3) memory; our stacked-DRAM timing latencies are approximately half of those for conventional off-chip DRAM. The stacked DRAM also supports more channels, as well as more banks and wider buses per channel.[8]

Performance

Figure 5 shows the performance of various DRAM cache options, normalized to the baseline system with only an 8-Mbyte L3 cache. For reference, we include a configuration with a tags-in-DRAM cache assuming naive constant-latency DRAM accesses, and a configuration with 64-byte cache lines supported by an impractically large on-chip SRAM tag array.

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

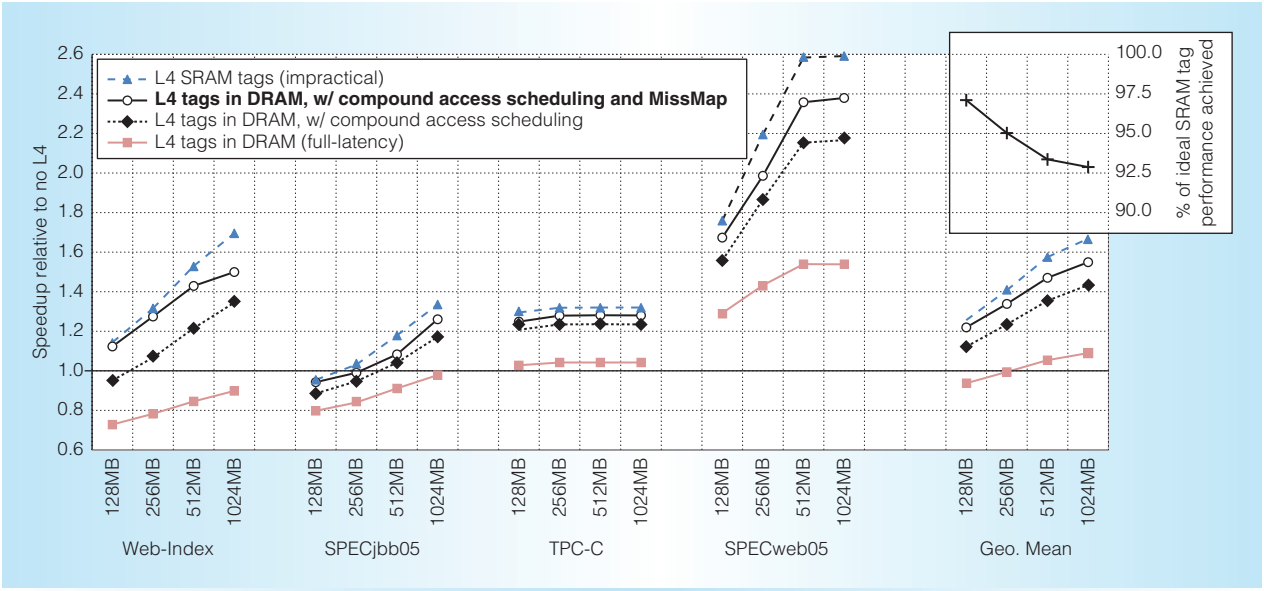

Figure 5. Performance impact of DRAM caches compared to a baseline with only an 8-Mbyte Level-3 (L3) cache. The inset shows how much using compound-access scheduling and a MissMap with 4-Kbyte segments closes the performance gap between the baseline (no DRAM L4 cache) and an ideal SRAM-tag implementation.

The performance benefits come from two sources. First, requests that hit in the stacked DRAM cache can be delivered with lower latency compared to off-chip accesses. Second, and less obvious, is that the traffic serviced by the DRAM cache reduces the contention for all remaining traffic that misses and goes off chip. Our original study provides several additional studies that analyze the impact of various design choices and examines how and why the proposed mechanisms work.[1]

Discussion

Die-stacking technologies, particularly for memory, are maturing rapidly; several prominent memory vendors such as Micron and Samsung have made recent announcements. As multicore processors and combined CPU/GPU systems exacerbate the memory wall problem,[9] we must find ways to effectively exploit upcoming die-stacked memories. Without such advancements, widespread industry adoption will be far less likely (or at least significantly delayed). The techniques we propose here provide a way to use large integrated DRAMs, but the impact and significance extends well beyond providing a one-time performance bump.

Buying time for software to develop

Ultimately, the most effective use of very large die-stacked memories could come from explicit, software-managed approaches. Caches have worked well for decades; however, caching becomes significantly more difficult as we dive more deeply down the cache hierarchy, where the upper-level caches have already filtered away all of the easy locality. A conventional hardware-managed cache is limited to what little behavior it can observe in relatively small windows of time and space.

Software (or a hardware–software cooperative approach) has the potential to do much more. The operating system (OS) has system-wide knowledge about memory usage within and among programs, and can use this knowledge to better assign memory allocations to the stacked DRAM. The OS can also track information across longer epochs of time, reusing long-past information (for example, from the last time a process was scheduled to execute) to optimize stacked-DRAM allocations. Compilers can take the time to perform increasingly sophisticated analyses of memory-access patterns, exploiting semantic knowledge of programming language constructs to help inform the OS about the best candidate memory regions for pinning into the

stacked DRAM. Just-in-time compilation and dynamic runtime binary optimization, which modern managed codes increasingly use, provide even more static and dynamic information to better use the stacked memory. Finally, application programmers often have a much better understanding of the inherent locality and memory characteristics of the algorithms they're implementing. New programming language constructs, pragmas, and other mechanisms let programmers provide guidance directly to the entire hardware–software stack. The collection of all these rich sources of information and analysis hold much more potential than simple, reactive, hardware-only caching approaches.

The problem is that all these opportunities require incredibly significant changes throughout the software stack. These changes will not (cannot?) happen very quickly, because major OS updates occur on multiyear timescales. Furthermore, many interdependencies exist among the layers; for example, advanced compiler analysis to determine memory regions that are likely (or unlikely) to benefit from being mapped to stacked DRAM will require a software interface to the OS to communicate the results of the analysis. All of these will take time to research effective techniques, and then more time to cooperatively implement across hardware and software. Before all of this can be accomplished, we will want other options for making use of stacked-DRAM technology. The techniques we propose here provide one such approach and buy more time for the other software-based technologies to develop.

Coping with GPU and GPGPU memory demands

AMD Fusion architectures feature combined CPU and GPU functionality. A significant challenge associated with this direction is that it also comes with the combined bandwidth demands of CPUs and GPUs. In particular, traditional graphics workloads for discrete GPUs can consume hundreds of Gbytes/s of bandwidth, whereas high-performance CPUs today provide only a few tens of Gbytes/s of bandwidth. Effectively exploiting stacked-DRAM technologies will play a critical role in dealing with the combined CPU/GPU bandwidth demands in future fused architectures. As programming of general-purpose computing on graphics processing units (GPGPU) expands into more application areas, the strain on the off-chip memory bandwidth will only increase. The techniques we propose here provide a practical method to implement a large on-chip cache that can provide substantial relief for current and future bandwidth-intensive GPU-based applications.

Power scaling

Finding ways to effectively use large die-stacked DRAMs will be critical to addressing the power wall, and in particular the problem that a system's main memory is consuming an increasingly large fraction of the total system power budget. Die-stacked DRAM can provide a significant reduction in energy per bit compared to conventional off-chip DDR3 dual inline memory modules (DIMMs), but we can realize this energy benefit only if we can find effective stacked-DRAM organizations (such as that presented in this article) that are practical to implement. Although this work did not explicitly quantify stacked-DRAM caches' total power-reduction potential, it did provide results showing that a 1-Gbyte stacked-DRAM cache can reduce the number of row activations (one of the most power-expensive DRAM operations) in the off-chip DRAM by 80 percent.

Supporting NVRAM hierarchies

Industry projections indicate that scaling conventional DRAM device technology will soon face significant challenges, potentially bringing traditional Moore's law scaling of DRAM to an end. These challenges are among the key reasons that have driven memory vendors to be early adopters of die-stacking technology, because it's becoming cheaper to achieve memory density by stacking as opposed to lithographic scaling. Various nonvolatile memory technologies (such as phase-change memory [PCM] and memristors) appear to be strong candidates as eventual DRAM replacements, but because of their write-endurance and write-latency problems, large multigigabyte DRAM write buffers will likely still be needed. Management of what are effectively very large DRAM caches can benefit from our MissMap approach. Although the DRAM buffer's

exact organization could vary to support differences in nonvolatile RAM (NVRAM) characteristics and properties, we could leverage the observation that tracking cache contents can be decoupled from the tracking of exact data location to provide more efficient NVRAM buffer designs.

Die-stacked DRAM has the potential to play a critical role in future computing systems, from small mobile devices to commercial mega-datacenters and exascale supercomputers. There are many open research problems directly related to memory stacking, as described in the previous section. However, many research directions beyond memory stacking remain to be explored to fully take advantage of die-stacking technologies. Some of these include exploring new architectures to better exploit the integration of heterogeneous silicon technologies, designing effective architectures that simultaneously exploit both horizontal interposer-based stacking and vertical 3D stacking, communication and on-chip network organizations to efficiently connect multiple chips, 3D circuits and pipelines partitioned across multiple silicon layers, integration of analog components, and overall system-on-a-chip (or "system in a stack") design and test methodologies. This is an exciting time for research in processor architectures and computer systems. MICRO

References

1. G.H. Loh and M.D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 454-464.
2. X. Dong et al., "Simple But Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," *Proc. ACM/IEEE Int'l Conf. High-Performance Computing, Networking, Storage, and Analysis* (SC 10), IEEE CS, 2010, doi:10.1109/SC.2010.50.
3. X. Jiang et al., "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," *Proc. 16th Int'l Symp. High-Performance Computer Architecture* (HPCA 10), IEEE CS, 2010, doi:10.1109/HPCA.2010.5416642.
4. L. Zhao et al., "Exploring DRAM Cache Architectures for CMP Server Platforms," *Proc. 25th Int'l Conf. Computer Design,* IEEE CS, 2007, pp. 55-62.
5. J. Torrellas, M.S. Lam, and J.L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Computers,* vol. 43, no. 6, 1994, pp. 651-663.
6. J. Jaminger and P. Stenström, "Improvement of Energy-Efficiency in Off-Chip Caches by Selective Prefetching," *Microprocessor and Microsystems,* vol. 26, no. 3, 2002, pp. 107-121.
7. J.S. Liptay, "Structural Aspects of the System/360 Model 85, Part II: The Cache," *IBM Systems J.,* vol. 7, no. 1, 1968, pp. 15-21.
8. N. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News,* vol. 39, no. 2, 2011, doi:10.1145/2024716.2024718.
9. W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News,* vol. 23, no. 1, 1995, pp. 20-24.

Gabriel H. Loh is a principal researcher at Advanced Micro Devices. His research interests include computer architecture, processor microarchitecture, emerging technologies, and 3D die stacking. Loh has a PhD in computer science from Yale University. He is a senior member of IEEE and the ACM.

Mark D. Hill is a professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin—Madison. This work was performed largely while Hill was on sabbatical leave at Advanced Micro Devices. His research interests include parallel-computer system design, memory system design, computer simulation, deterministic replay, and transactional memory. Hill has a PhD in computer science from the University of California, Berkeley. He is a fellow of IEEE and the ACM.

Direct questions and comments about this article to Gabriel H. Loh, Advanced Micro Devices, 2002 156th Ave. NE, Suite 300, Bellevue, WA 98007; gabriel.loh@amd.com.