CHOP: INTEGRATING DRAM CACHES FOR CMP SERVER PLATFORMS

INTEGRATING LARGE DRAM CACHES IS A PROMISING WAY TO ADDRESS THE MEMORY BANDWIDTH WALL ISSUE IN THE MANY-CORE ERA. HOWEVER, ORGANIZING AND IMPLEMENTING A LARGE DRAM CACHE IMPOSES A TRADE-OFF BETWEEN TAG SPACE OVERHEAD AND MEMORY BANDWIDTH CONSUMPTION. CHOP (CACHING HOT PAGES) ADDRESSES THIS TRADE-OFF THROUGH THREE FILTER-BASED DRAM-CACHING TECHNIQUES.

Xiaowei Jiang
Intel

Niti Madan
IBM Thomas J. Watson Research Center

Li Zhao
Mike Upton
Ravi Iyer
Srihari Makineni
Donald Newell
Intel

Yan Solihin
North Carolina State University

Rajeev Balasubramonian
University of Utah

Today's multicore processors already integrate multiple cores on a die. Many-core architectures enable far more small cores for throughput computing. The key challenge in many-core architectures is the *memory bandwidth wall*:[1-3] the required memory bandwidth to keep all cores running smoothly is a significant challenge. In the past, researchers have proposed large dynamic RAM (DRAM) caches to address this challenge.[4-7] Such solutions fall into two main categories: DRAM caches with small allocation granularity (64-byte cache lines) and DRAM caches with large allocation granularity (page sizes such as 4-Kbyte or 8-Kbyte cache lines). However, fine-grained DRAM cache allocation comes at the cost of tag array storage overhead. This overhead implies that it's necessary to reduce the last-level cache (LLC) to accommodate the DRAM cache tag because the DRAM tag must be stored on die for fast lookup.[8] The alternative coarse-grained allocation comes at the cost of significant memory bandwidth consumption. It tends to limit performance benefits for applications that don't have high spatial locality across all its pages.

To deal with this dilemma, we propose CHOP (Caching Hot Pages), a methodology comprising three filter-based DRAM caching techniques that cache only hot (frequently used) pages:

- a filter cache that determines the hot subset of pages to allocate into the DRAM cache,
- a memory-based filter cache that spills and fills the filter state to improve the filter cache's accuracy and reduce its size, and
- an adaptive DRAM-caching technique that switches DRAM-caching policy on the fly.

By employing coarse-grained allocation, we considerably reduce the on-die tag space needed for the DRAM cache. By allocating only hot pages into the DRAM cache, we avoid the memory bandwidth problem because we don't waste bandwidth on pages with low spatial locality.

DRAM-caching benefits and challenges

Researchers have investigated DRAM caches as a solution to the memory

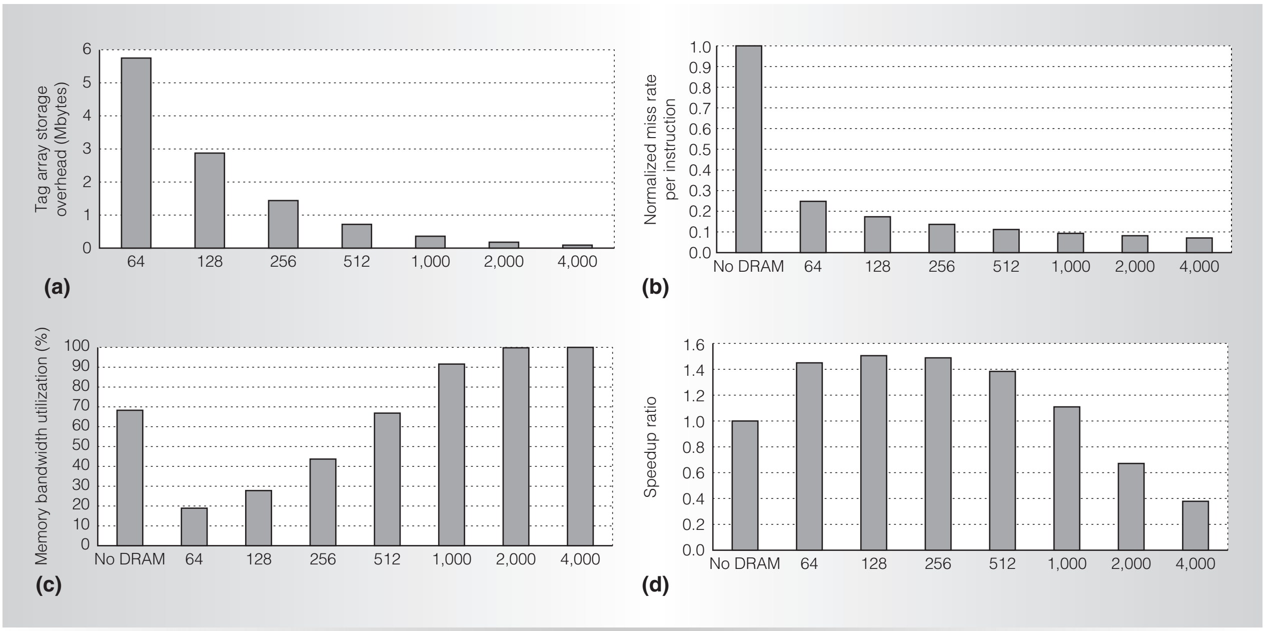

Figure 1. Overview of dynamic RAM (DRAM) cache benefits, challenges, and trade-offs: tag array storage overhead (a), misses per instruction (b), memory bandwidth utilization (c), and performance improvement (d).

bandwidth wall problem.[3,6] Such a solution enables 8 to 16 times higher density than traditional static RAM (SRAM) caches,[4] and consequently provides far higher cache capacity. The basic DRAM-caching approach is to integrate the tag arrays on die for fast lookup, whereas the DRAM cache data arrays can be either on chip (embedded DRAM, or eDRAM) or off chip (3D stacked or misses per instruction [MPI]).

To understand the benefits and challenges of DRAM caching, we experimented with the Transaction Processing Performance Council C benchmark (TPC-C). Figure 1 shows the DRAM cache benefits (in terms of MPI and performance) along with the trade-offs (in terms of tag space overhead and memory bandwidth utilization). The baseline configuration has no DRAM cache. At smaller cache lines, tag space overhead becomes significant.

Figure 1a shows that a 128-Mbyte DRAM cache with 64-byte cache lines has a tag space overhead of 6 Mbytes. If we implement this cache, then we must either increase the die size by 6 Mbytes or reduce the LLC space to accommodate the tag space. Either alternative significantly offsets the performance benefits, so we don't consider this a viable approach. The significant tag space overhead at small cache lines inherently promotes the use of large cache lines, which alleviates the tag space problem by more than an order of magnitude (as Figure 1a shows, the 4-Kbyte line size enables a tag space of only a few kilobytes). However, Figure 1b shows that using larger cache lines doesn't significantly reduce MPI. This is because larger cache line sizes alone aren't sufficient for providing spatial locality. More importantly, the memory bandwidth utilization increases significantly (Figure 1c) due to the decrease in DRAM cache miss rate (or MPI); it isn't directly proportional to the increase in line size. Such an increase in main memory bandwidth requirements saturates the main memory subsystem and therefore immediately affects this DRAM cache configuration's performance significantly (as shown by the 2-Kbyte and 4-Kbyte line size results in the last two bars of Figure 1d).

Server workload DRAM-caching characterization

Using large cache line sizes (such as 4-Kbyte page sizes) considerably alleviates the tag space overhead. However, the main memory bandwidth requirement increases accordingly and becomes the primary bottleneck. Therefore, at large cache line sizes, the DRAM-caching challenge changes into a memory bandwidth efficiency issue. Saving memory bandwidth requires strictly controlling the amount of cache lines that enter the DRAM cache. However, even when caching only a subset of the working set, we must retain the amount of useful DRAM cache to provide performance benefits. So, we must carefully choose this subset of the working set.

To understand how we should choose this subset, we performed a detailed offline profiling of workloads to obtain their DRAM cache access information (assuming 4-Kbyte page-size cache lines). Table 1 shows the results for four server workloads. For this profiling, we define *hot pages* as the topmost accessed pages that contribute to 80 percent of the total access number. We calculate the hot-page percentage as the number of hot pages divided by the total number of pages for each workload. We consider about 25 percent of the pages to be hot pages. The last column in Table 1 also shows the minimum number of accesses to these hot pages. On average, a hot page is accessed at least 79 times. If we can capture such hot pages on the fly and allocate them into the DRAM cache, we can reduce memory traffic by about 75 percent and retain about 80 percent of the DRAM cache hits.

Table 1. Offline hot-page profiling statistics for server workloads.

Workload	Hot-page percentage	Hot-page minimum no. of accesses
SAP	24.8	95
Sjas	38.4	65
Sjbb	30.6	93
TPC-C	7.2	64
Average	25.2	79

*SAP: SAP Standard Application; Sjas (SPECjApps2004): SPEC Java Application Server 2004; Sjbb (SPECjbb2005): SPEC Java Server 2005; TPC-C: Transaction Processing Performance Council C.

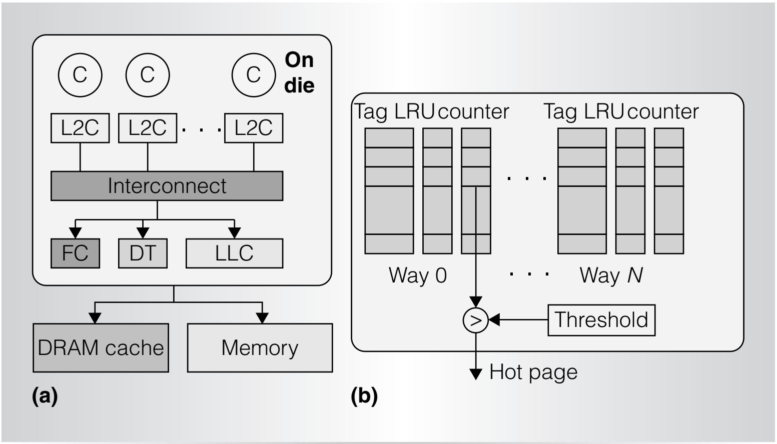

Figure 2. The filter cache for DRAM caching: architecture overview (a) and filter cache structure (b). (DT: DRAM cache tag array; FC: filter cache; L2C: level-two cache; LLC: last-level cache; LRU: least recently used; Way: cache way of the filter cache.)

Filter-based DRAM caching

We introduce a small filter cache that profiles the memory access pattern and identifies hot pages: pages that are heavily accessed because of temporal or spatial locality. By enabling a filter cache that identifies hot pages, we can introduce a filter-based DRAM cache that only allocates cache lines for the hot pages. By eliminating allocation of cold pages in the DRAM cache, we can reduce the wasted bandwidth due to allocating cache lines that never get touched later.

Filter cache (CHOP-FC)

Figure 2a shows our basic filter-based DRAM-caching architecture (CHOP-FC), which incorporates a filter cache on die with the DRAM cache tag array (the data array is off die via 3D stacking or MCP). Both the filter cache and DRAM cache use page-size lines. Each filter cache entry (Figure 2b) includes a tag, LRU (least recently used) bits, and a counter indicating how often the line is touched. The filter cache allocates a line into its tag array the first time an LLC miss occurs on this line. The counter returns to zero for a newly allocated line and increases incrementally for subsequent LLC misses on that line. Once the counter reaches a threshold, the filter cache considers the line to be a hot page and puts it into the DRAM cache.

To allocate a hot page into the DRAM cache, the filter cache sends a page request to the memory to fetch the page, and the

DRAM cache chooses a victim. Our measurements indicated that the LFU (least frequently used) replacement policy performs better than the LRU policy for the DRAM cache. To employ the LFU policy, the DRAM cache also maintains counters similarly to the way the filter cache does. The counter is incremented when a hit to the line occurs.

When the DRAM cache replaces a line, it chooses the line with the smallest counter value as the victim. The DRAM cache then returns the victim to the filter cache as a cold page, while writing its data back to memory. The victim's counter is then set to half its current value so that the victim has a better chance to become hotter than other new lines in the filter cache. The processor considers as cold pages any LLC misses that also miss in the filter cache and DRAM cache (or hit in the filter cache with the counter less than the threshold). For such pages, the LLC sends a regular request (64 bytes) to the memory. So, the processor doesn't waste memory bandwidth on fetching cold pages into the DRAM cache.

Memory-based filter cache (CHOP-MFC)

In the baseline filter cache scheme, the filter cache size directly impacts how long a line can stay in it. The counter history is completely lost when the line is evicted. This could reduce the likelihood of a hot page being identified. To deal with this problem, we propose a memory-based filter cache (CHOP-MFC), which spills a counter into memory when the line is replaced and restores it when the line is fetched back again. With a memory backup for counters, we can expect the filter cache to provide more accurate hot-page identification. In addition, this scheme lets us safely reduce the filter cache size significantly.

Storing counters in memory requires allocating extra space in the memory. A 16-Gbyte memory with a 4-Kbyte page size and a counter threshold of 256 (8-bit counters) requires 4 Mbytes of memory space. We propose three options for this memory backup.

In the first option, either the operating system allocates this counter memory space in the main memory or the firmware or BIOS reserves it without exposing it to the operating system. However, this option requires off-chip memory accesses to look up the counters in the allocated memory region. For smaller filters with potentially higher miss rates, the performance benefit of using a filter cache might not be able to amortize the cost of off-chip counter lookups.

To deal with these problems, the second option for memory backup is to pin this counter memory space in the DRAM cache. If the DRAM cache size is 128 Mbytes, and only 4 Mbytes of memory space are required for counter backup, then the loss in DRAM cache space is minimal (about 3 percent). To generate the DRAM cache address for retrieving a counter, we simply use the page frame number of a page to reference the counter memory space in the DRAM cache.

Similar to CHOP-FC, whenever an LLC miss occurs, the processor checks the filter cache and increments the corresponding counter. To further reduce the counter lookups' latency, the processor can send prefetching requests of counters upon a lower-level cache miss. If a replacement in the filter cache occurs, the processor updates the corresponding counter in the reserved memory space. This counter writeback can occur in the background. Once the counter in the filter cache reaches the threshold, the filter cache identifies a hot page and installs it into the DRAM cache. (More details on this option are available elsewhere.[9])

We further propose a third option for the MFC, in which we distribute the counter memory space and augment it into the page table entry (PTE) of each page. The processor allocates the unused bits in the PTEs (18 bits in a 64-bit machine[10]) for the denoted page's counter. Upon a translation look-aside buffer (TLB) miss, the processor's TLB miss handler or the page table walk routine brings the counter along with other PTE attributes into the TLB. The processor performs virtual to physical page translation at the beginning of the instruction fetch stage (for instruction pages) and the memory access stage (for data pages). Therefore, by the time a filter cache access fires for a load or store instruction, the TLB has already obtained the desired page's counter and could supply it to the filter cache. So, we can significantly reduce the counter fetch operation's overhead because this operation doesn't need to go off chip.

Upon a filter cache eviction, the filter cache must write the replaced counter back to the PTEs. To assist in PTE lookup, the filter cache must store both a page's virtual and physical tag bits. This approach is transparent to the operating system in that the latter doesn't allocate or update the counters in PTEs. Rather, the processor TLB and the filter cache alone operate these counters.

Adaptive filter cache schemes (CHOP-AFC and CHOP-AMFC)

Because different workloads can have distinct behaviors, and even the same workload can have different phases, memory subsystems could be affected differently. To address such effects, we propose two adaptive filter cache schemes, CHOP-AFC and CHOP-AMFC, in which the processor turns the filter cache on and off dynamically on the basis of the memory utilization status. CHOP-AMFC differs from CHOP-AFC in that the former is based on CHOP-MFC, whereas the latter is based on CHOP-FC. We add monitors to track memory traffic. When memory utilization exceeds a certain threshold, the processor turns on the filter cache so that only hot pages are fetched into the DRAM cache; otherwise, the processor turns off the filter cache and brings all pages into the DRAM cache on demand. With the capability of adjusting the DRAM cache allocation policy on the fly, we expect more performance benefits by using the entire available memory bandwidth.

Evaluation methodology

For the simulation environment, we used the ManySim simulator to evaluate our filter cache schemes.[11] The simulated chip multiprocessor (CMP) architecture consists of 8 cores (each operating at 4 GHz). Each core has an eight-way, 512-Kbyte private level-two (L2) cache, and all cores share a 16-way, 8-Mbyte L3 cache. An on-die interconnect with bidirectional ring topology connects the L2 and L3 caches.

The DRAM cache is 128 Mbytes with an on-die tag array. The filter cache has a coverage of 128 Mbytes. The MFC contains 64 entries with four-way associativity. The memory access latency is 400 cycles with a bandwidth of 12.8 Gbytes per second. For CHOP-MFC, we evaluated the second option, which pins down counter memory space in the DRAM cache, because this option involves fewer hardware changes.

We chose four key commercial server workloads: TPC-C, SAP Standard Application (SAP), SPEC Java Server 2005 (Sjbb), and SPEC Java Application Server 2004 (Sjas). We collected long bus traces (data and instruction) for workloads on an Intel Xeon Multiprocessor platform with the LLC disabled.

Evaluation results

We evaluated the performance efficacy of the three filter-cache-based DRAM caching techniques individually, and then we compared their results collectively.

CHOP-FC evaluation

To demonstrate our basic filter cache's effectiveness, we compared CHOP-FC's performance results to a regular 128-Mbyte DRAM cache and a naive scheme that caches a random portion of the working set.

Figure 3a shows the speedup ratios of the basic 128-Mbyte DRAM cache (DRAM), a scheme that caches a random subset of the LLC misses (RAND) into the DRAM cache, and our CHOP-FC scheme, which captures the hot subset of pages. We normalized all results to the base case in which no DRAM cache was applied. For the RAND scheme, we generated a random number for each new LLC miss and compared it to a probability threshold to determine whether the block should be cached into the DRAM cache. We adjusted the probability threshold, and Figure 3 shows the one that leads to the best performance.

The DRAM bar shows that directly adding a 128-Mbyte DRAM cache doesn't improve performance as much as expected. Instead, this addition incurs a slowdown in many cases. With a 4-Kbyte cache line size, the DRAM resulted in an average slowdown of 21.9 percent, with a worst case of

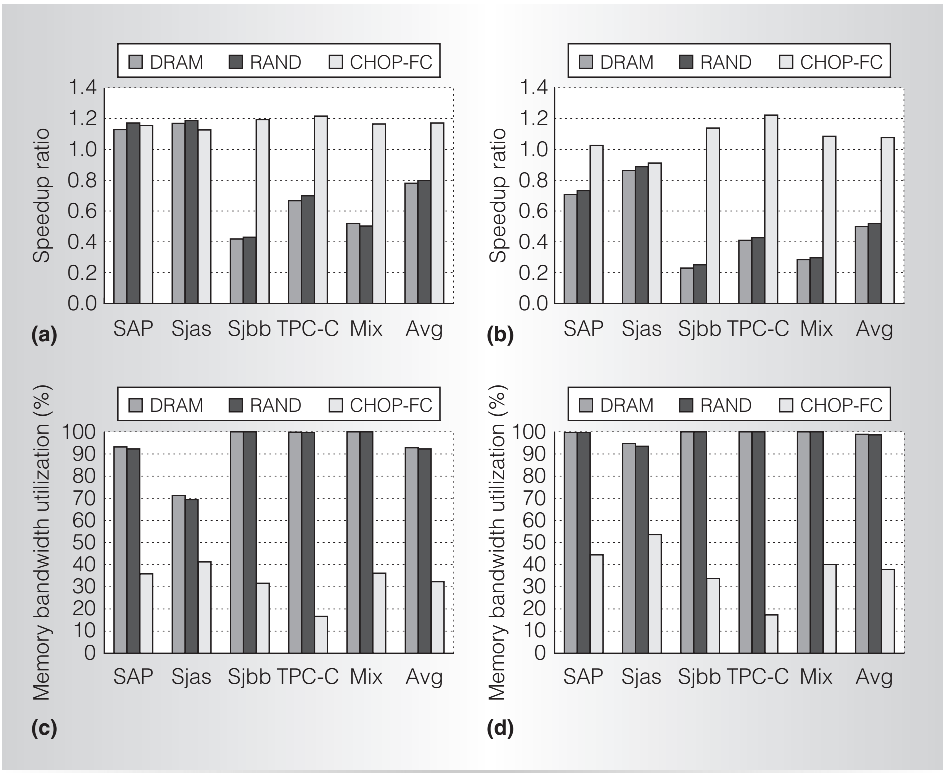

Figure 3. Speedup ratios for a 4-Kbyte cache line (a) and an 8-Kbyte cache line (b), and memory bandwidth utilization for a 4-Kbyte cache line (c) and an 8-Kbyte cache line (d), of our CHOP (Caching Hot Pages) filter-cache-based schemes and two other schemes. (CHOP-FC: CHOP filter-cache-based DRAM-caching architecture; DRAM: regular DRAM caching; RAND: caching a random subset of LLC misses; Mix: a mixture of all the benchmarks shown (SAP, Sjas, Sjbb, and TPC-C.)

58.2 percent for Sjbb. With the 8-Kbyte cache line size, it resulted in an average slowdown of 50.1 percent, with a worst case of 77.1 percent for Sjbb. To understand why this occurred, Figure 3b presents each scheme's memory bandwidth utilization. Using large cache line sizes easily saturates the available memory bandwidth, with an average of 92.8 percent and 98.9 percent for 4-Kbyte and 8-Kbyte line sizes, respectively.

Because allocating cache lines for each LLC miss in the DRAM cache saturates the memory bandwidth, one possible solution is to cache only a subset of it. However, the RAND bar proves that this subset must be carefully chosen. On average, RAND shows 20.2 percent and 48.1 percent slowdowns for the 4-Kbyte and 8-Kbyte line sizes, respectively.

Figure 3a also demonstrates that our CHOP-FC scheme generally outperforms DRAM and RAND. With the 4-Kbyte and 8-Kbyte line sizes, CHOP-FC achieved an average speedup of 17.7 percent and 12.8 percent using counter threshold 32. The reason is that although hot pages were only 25.24 percent of the LLC-missed portion of the working set, they contributed to 80 percent of the entire LLC misses. Caching those hot pages significantly reduces memory bandwidth utilization and thus reduces the associated queueing delays. As compared to RAND, caching hot pages also provides far smaller MPI.

CHOP-MFC evaluation

We compared CHOP-MFC's performance and memory bandwidth utilization to that of a regular DRAM cache. Figure 4 compares the speedup ratios (Figure 4a) and memory bandwidth utilization (Figure 4b) of a basic DRAM cache

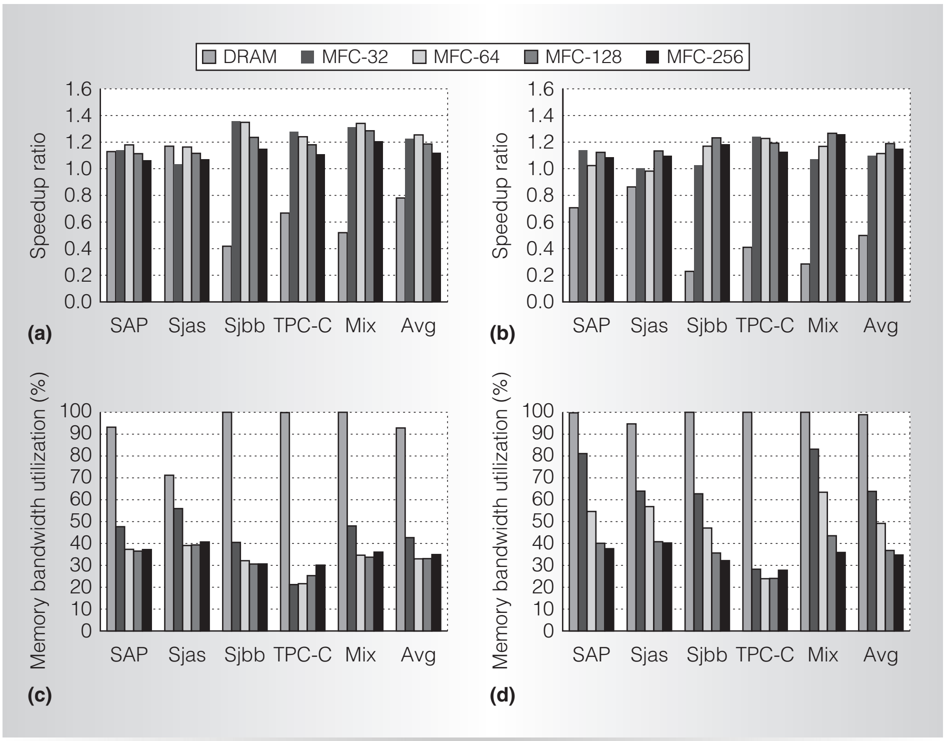

Figure 4. Speedup ratios for a 4-Kbyte cache line (a) and an 8-Kbyte cache line (b), and memory bandwidth utilization for a 4-Kbyte cache line (c) and an 8-Kbyte cache line (d), of the CHOP memory-based filter cache (CHOP-MFC) with various thresholds.

(DRAM) with CHOP-MFC under various counter thresholds (MFC-32 for threshold 32, and so on).

Figure 4a shows that CHOP-MFC outperforms DRAM in most cases. For the 4-Kbyte line size, MFC-64 achieved the highest speedup (25.4 percent), followed by MFC-32 (22.5 percent), MFC-128 (18.6 percent), and MFC-256 (12.0 percent). For the 8-Kbyte line size, MFC-128 performed best, with an average speedup of 18.9 percent. The performance improvements are due to the significant reduction in memory bandwidth utilization, as Figure 4b shows. These results demonstrate that having a small MFC is sufficient for keeping the fresh hot-page candidates. Unlike CHOP-FC, replacements in the MFC incur counter writebacks to memory rather than a total loss of counters. Therefore, CHOP-MFC achieves higher hot-page identification accuracy. CHOP-MFC with 64 entries had an average miss rate of 4.29 percent compared to 10.34 percent in CHOP-FC, even with 32,000 entries for the 4-Kbyte line size. Thanks to the performance robustness (speedups in all cases) that counter threshold 128 provided, we chose to use it as the default value.

CHOP-AFC evaluation

Figure 5 exhibits the results for CHOP-AFC with memory utilization thresholds varying from 0 to 1. (For this evaluation, we show CHOP-AFC's results only because CHOP-AMFC exhibited similar performance.) For memory bandwidth utilization thresholds less than 0.3, the speedups achieved remained the same (except for TPC-C). For threshold values between 0.3 and 1, the speedups show an increasing trend followed by an immediate decrease (see Figure 5a). On the other hand, Figure 5b shows that for threshold values less than 0.3, the memory bandwidth utilization remained approximately the same (except for an increase in TPC-C). For threshold values greater than 0.3, memory

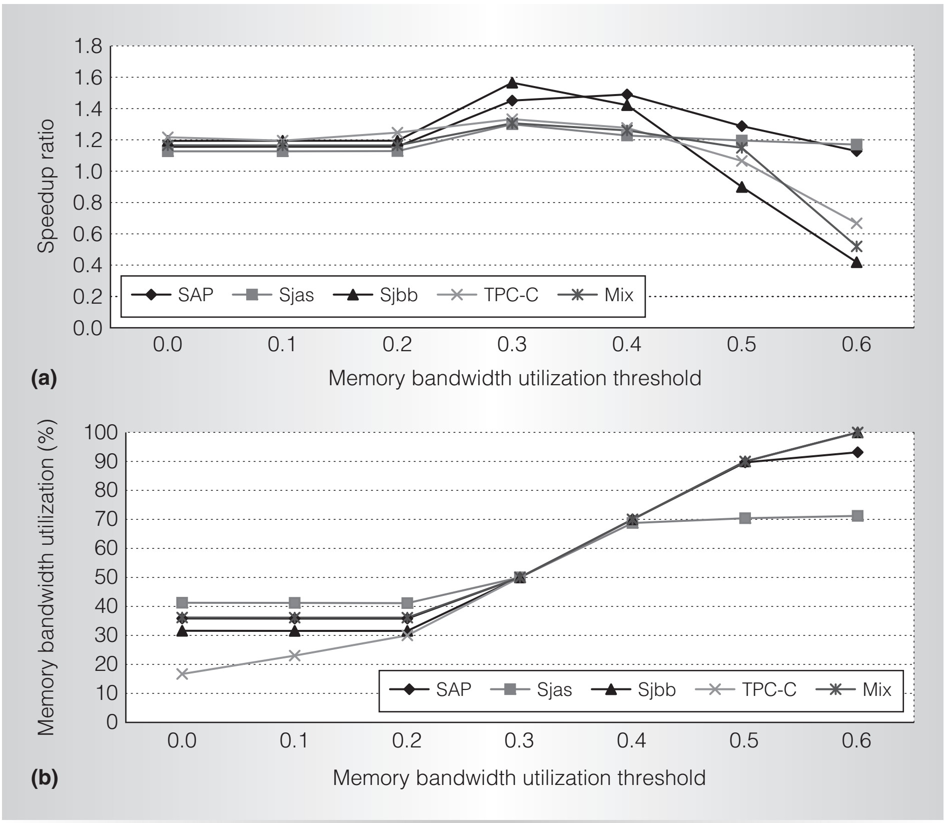

Figure 5. Speedup ratios (a) and memory bandwidth utilization (b) of the CHOP adaptive filter cache (CHOP-AFC) with various memory bandwidth utilization thresholds.

bandwidth utilization tended to increase as the threshold increased.

To understand why this is the case, recall that the AFC scheme adjusts its DRAM cache allocation policy with respect to the available memory bandwidth. Moreover, the average memory bandwidth utilization achieved was 93 percent for the DRAM and 32 percent for CHOP-FC (Figure 3b), which essentially denotes the upper and lower bounds of the memory bandwidth utilization that the CHOP-AFC scheme can reach. When the threshold was less than 0.3, the measured memory utilization with the filter cache turned on was always larger than the threshold, so the filter cache was always turned on, resulting in an average of 32 percent memory bandwidth utilization. Therefore, with the threshold between 0 and 0.3, CHOP-AFC behaved the same as CHOP-FC. This explains why CHOP-AFC shows a constant speedup and memory bandwidth utilization under those thresholds.

When the threshold was greater than 0.3, owing to the abundance of available memory bandwidth, the filter cache could be turned off for some time, and therefore more pages could go into the DRAM cache. As a result, the useful hits provided by the DRAM cache tended to increase as well. This explains why memory bandwidth utilization kept increasing and performance kept increasing in the beginning for threshold values between 0.3 and 1. However, after some point, because of the high memory bandwidth utilization, queueing delay began to dominate and consequently performance decreased. The reason the TPC-C workload shows a different trend for thresholds between 0 and 0.3 is its smaller lower-bound memory bandwidth utilization of 15 percent (the DRAM bar in Figure 5b).

Comparison of the three schemes

Compared to the traditional DRAM-caching approach, all our filter-based techniques improve performance by caching only the hot subset of pages. For a 4-Kbyte block size, CHOP-MFC achieved on average speedup of 18.6 percent with only 1 Kbyte of extra on-die storage overhead, whereas CHOP-FC achieved a 17.2 percent speedup with 132 Kbytes of extra on-die storage overhead. Comparing these two schemes, CHOP-MFC uniquely offers larger address space coverage with very small storage overhead, whereas CHOP-FC naturally offers both hotness and timeliness information at a moderate storage overhead and minimal hardware modifications.

The adaptive filter cache schemes (CHOP-AFC and CHOP-AMFC) typically outperformed CHOP-FC and CHOP-MFC. These schemes intelligently detect available memory bandwidth to dynamically adjust DRAM-caching coverage policy and thereby improve performance. Dynamically turning the filter cache on and off can adapt to the memory bandwidth utilization on the fly. When memory bandwidth is abundant, the adaptive filter cache enlarges coverage, and more blocks are cached into the DRAM cache to produce more useful cache hits. When memory bandwidth is scarce, the adaptive filter cache reduces coverage, and only hot pages are cached to produce a reasonable amount of useful cache hits as often as possible. By setting a proper memory utilization threshold, CHOP-AFC and CHOP-AMFC can use memory bandwidth wisely.

The adaptive filter cache schemes show their performance robustness and guarantee performance improvement by quickly adapting to the bandwidth utilization situation. Possible extensions to these schemes include application to other forms of two-level memory hierarchies, including phase-change memory (PCM) and flash-based memory, and architectural support for exposing hot-page information back to the operating system to schedule optimizations. MICRO

Acknowledgments

We thank Zhen Fang for providing valuable comments and feedback on this article.

References

1. D. Burger, J.R. Goodman, and A. Kägi, ''Memory Bandwidth Limitations of Future Microprocessors,'' *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), ACM Press, 1996, pp. 78-79.
2. F. Liu et al., ''Understanding How Off-Chip Memory Bandwidth Partitioning in Chip-Multiprocessors Affects System Performance,'' *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture* (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416655.
3. B.M. Rogers et al., ''Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture* (ISCA 09), ACM Press, 2009, pp. 371-382.
4. B. Black et al., ''Die Stacking (3D) Microarchitecture,'' *Proc. 39th Int'l Symp. Microarchitecture,* IEEE CS Press, 2006, pp. 469-479.
5. R. Iyer, ''Performance Implications of Chipset Caches in Web Servers,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 03), IEEE CS Press, 2003, pp. 176-185.
6. N. Madan et al., ''Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy,'' *Proc. IEEE 15th Int'l Symp. High Performance Computer Architecture* (HPCA 09), IEEE Press, 2009, pp. 262-274.
7. Z. Zhang, Z. Zhu, and X. Zhang, ''Design and Optimization of Large Size and Low Overhead Off-Chip Caches,'' *IEEE Trans. Computers,* vol. 53, no. 7, 2004, pp. 843-855.
8. L. Zhao et al., ''Exploring DRAM Cache Architectures for CMP Server Platforms,'' *Proc. 25th Int'l Conf. Computer Design* (ICCD 07), IEEE Press, 2007, pp. 55-62.
9. X. Jiang et al., ''CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms,'' *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture* (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416642.
10. *Intel Architecture Software Developer's Manual,* vol. 1, Intel, 2008; http://www.intel.com/design/pentiumii/manuals/243190.htm.
11. L. Zhao et al., ''Exploring Large-Scale CMP Architectures Using ManySim,'' *IEEE Micro,* vol. 27, no. 4, 2007, pp. 21-33.

Xiaowei Jiang is a research scientist at Intel. His research interests include system-on-chip (SoC) and heterogeneous-core architectures. He has a PhD in computer engineering from North Carolina State University.

Niti Madan is a National Science Foundation and Computing Research Association Computing Innovation Fellow at the IBM Thomas J. Watson Research Center. Her research interests include power and reliability-aware architectures and the use of emerging technologies such as 3D die stacking. She has a PhD in computer science from the University of Utah.

Li Zhao is a research scientist at Intel. Her research interests include SoC architectures and cache memory systems. She has a PhD in computer science from the University of California, Riverside.

Mike Upton is a principal engineer at Intel. His research interests include graphics processor architectures and many-core architectures. He has a PhD in computer architecture from the University of Michigan, Ann Arbor.

Ravi Iyer is a principal engineer at Intel. His research interests include SoC and heterogeneous-core architectures, cache memory systems, and accelerator architectures. He has a PhD in computer science from Texas A&M University.

Srihari Makineni is a senior researcher at Intel. His research interests include cache and memory subsystems, interconnects, networking, and large-scale chip multiprocessor (CMP) architectures. He has an MS in electrical and computer engineering from Lamar University, Texas.

Donald Newell is the CTO of the Server Product Group at AMD. His research interests include server architectures, cache memory systems, and networking. He has a BS in computer science from the University of Oregon.

Yan Solihin is an associate professor in the Department of Electrical and Computer Engineering at North Carolina State University. His research interests include cache memory systems, performance modeling, security, and reliability. He has a PhD in computer science from the University of Illinois at Urbana-Champaign.

Rajeev Balasubramonian is an associate professor in the School of Computing at the University of Utah. His research interests include memory systems, interconnect design, large-cache design, transactional memory, and reliability. He has a PhD in computer science from the University of Rochester.

Direct questions and comments about this article to Xiaowei Jiang, JF2, Intel, 2111 NE 25th Ave., Hillsboro, OR 97124; xiaowei.jiang@intel.com.