CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms

Xiaowei Jiang†, Niti Madan⋆, Li Zhao‡, Mike Upton‡, Ravishankar Iyer‡, Srihari Makineni‡, Donald Newell‡

Yan Solihin†, Rajeev Balasubramonian⋆

†Dept. of Electrical and Computer Engineering
North Carolina State University
{xjiang, solihin}@ncsu.edu

‡Intel Labs
Intel Corporation
{li.zhao, ravishankar.iyer}@intel.com

⋆School of Computing
University of Utah

***Abstract*—As manycore architectures enable a large number of cores on the die, a key challenge that emerges is the availability of memory bandwidth with conventional DRAM solutions. To address this challenge, integration of large DRAM caches that provide as much as 5× higher bandwidth and as low as 1/3rd of the latency (as compared to conventional DRAM) is very promising. However, organizing and implementing a large DRAM cache is challenging because of two primary tradeoffs: (a) DRAM caches at cache line granularity require too large an on-chip tag area that makes it undesirable and (b) DRAM caches with larger page granularity require too much bandwidth because the miss rate does not reduce enough to overcome the bandwidth increase. In this paper, we propose CHOP (Caching HOt Pages) in DRAM caches to address these challenges. We study several filter-based DRAM caching techniques: (a) a filter cache (CHOP-FC) that profiles pages and determines the hot subset of pages to allocate into the DRAM cache, (b) a memory-based filter cache (CHOP-MFC) that spills and fills filter state to improve the accuracy and reduce the size of the filter cache and (c) an adaptive DRAM caching technique (CHOP-AFC) to determine when the filter cache should be enabled and disabled for DRAM caching. We conduct detailed simulations with server workloads to show that our filter-based DRAM caching techniques achieve the following: (a) on average over 30% performance improvement over previous solutions, (b) several magnitudes lower area overhead in tag space required for cache-line based DRAM caches, (c) significantly lower memory bandwidth consumption as compared to page-granular DRAM caches.**

***Index Terms*—DRAM cache; CHOP; adaptive filter; hot page; filter cache**

I. INTRODUCTION

Today' multi-core processors [9], [10], [12], [15] are already integrating four to eight large cores on the die for client as well as server platforms. Manycore architectures enable many more small cores for throughput computing. Niagara [14] and the Intel Terascale effort [11] are examples of this trend. Server workloads can take advantage of the abundant compute parallelism enabled by such processors. The key challenge in manycore server architectures is the memory bandwidth wall: the amount of memory bandwidth required to keep multiple threads and cores running smoothly is a significant challenge. The brute force solution of adding more memory channels is infeasible and inefficient because of the pin limitations of processors. In this paper, we address this memory bandwidth challenge for manycore processors by investigating bandwidth and area efficient DRAM caching techniques.

Large DRAM caches have been proposed in the past to address the memory bandwidth challenge of manycore processors. For example, 3D stacked DRAM cache [16], [24], [40] has been proposed to enable 3×-5× more memory bandwidth and 1/3rd of the latency as compared to the conventional DDR-based memory subsystem. Similarly, Zhao et al. [41] evaluated the benefits and trade-offs of an embedded DRAM cache that provides similar bandwidth and latency benefits for improved server application performance. Such solutions can be classified into two primary buckets: (a) DRAM caches with small allocation granularity (i.e. 64 bytes) and (b) DRAM caches with large allocation granularity (i.e. page sizes such as 4KB or 8KB). Fine-grain DRAM cache allocation comes at the high cost of tag space which is stored on-die for fast lookup [41]. This overhead implies that the last-level cache has to be reduced in order to accommodate the DRAM cache tag, thereby incurring significant inefficiency. The alternative coarse-grain DRAM cache allocation comes at the cost of significant memory bandwidth consumption and tends to limit performance benefit significantly for memory-intensive applications that do not have significant spatial locality across all of its pages. In this paper, we address these two major limitations by proposing filter-based coarse-grain DRAM cache allocation techniques to only Cache HOt Pages (CHOP).

Our first filter-based coarse-grain DRAM caching technique employs a filter cache (CHOP-FC) to profile the pages being accessed and select only hot pages to allocate into the DRAM cache. By employing a coarse-grain DRAM cache, we considerably reduce the on-die tag space needed for the DRAM cache. By selecting only hot pages to be allocated into the DRAM cache, we avoid the memory bandwidth problem because memory bandwidth is not wasted on pages with low spatial locality. Our second DRAM caching technique employs an even smaller memory-based filter-cache (CHOP-MFC). By storing the replaced filter states of the filter cache into memory and putting them back into the filter cache when needed, we can improve the accuracy of the hot page detection as

well as reduce the size of the on-die filter cache. The third filter-based coarse-grain DRAM caching technique (CHOP-AFC) adapts between a filter-based policy and a full DRAM cache allocation policy. For workloads that do not significantly saturate the memory bandwidth with a coarse-grain DRAM cache, full allocation is employed. For workloads that saturate the memory bandwidth with a coarse-grain DRAM cache, filter-based allocation is employed.

We perform detailed simulations comparing the filter-based DRAM caching techniques to traditional DRAM caching techniques as well as platforms with no DRAM caches. For several commercial server workloads (TPC-C [4], SPECjbb [3], SAP [1], SpecjAppserver [2]), we show that the filter-based DRAM caching provides over 30% performance improvement on average, while significantly reducing the bandwidth and area consumption as compared to previous DRAM caching solutions.

The contributions of the paper are as follows:

- We introduce a new class of coarse-grain DRAM caching techniques based on filter caches that determine what to allocate into the DRAM cache.
- We present two filter caching techniques (with and without memory-backup) to show that extremely small filter caches are capable of identifying the hot pages for DRAM cache allocation.
- We present an adaptive technique that determines when to employ filters and when to perform full allocation.

The rest of this paper is organized as follows. Section II presents an overview of DRAM caching and motivates the need to improve DRAM caching performance in terms of memory bandwidth and area consumption. Section III presents the filter-based coarse-grain DRAM caching techniques. Section IV presents the evaluation methodology and analyzes the benefits of the proposed filter-based DRAM caching techniques. Section V discusses related works and Section VI summarizes the paper with the conclusions and a direction towards future work in this area.

II. DRAM CACHING OVERVIEW

In this section, we provide an overview of DRAM caches and highlight the challenges that need to be addressed to improve the efficiency and adoption of DRAM cache.

A. Background on DRAM Caching

DRAM caches [16], [40] have been investigated in the recent past to provide much higher bandwidth and much lower latency as compared to conventional DRAM memory. Figure 1 shows a typical CMP architecture with a cache/memory hierarchy. The cache hierarchy introduces an additional DRAM cache which can be either implemented with 3D stacking or as a multi-chippackage (MCP) or embedded on the same chip. Employing DRAM caches enables significant cache capacity as opposed to SRAM caches (~8× [6]). DRAM cache integration also shows the promise for more than 3× memory bandwidth and at least 1/3rd the latency of conventional DRAM memory subsystem [16], [40]. Researchers [6], [25] recently evaluated a 3D stacked approach for DRAM caches that provides significant bandwidth and latency benefits for improved overall performance for server and RMS workloads respectively. In the past, Yamauchi et al. [37] have studied the use of on-chip DRAM as memory or as cache. Zhao et al. [41] also evaluated the benefits of DRAM caching and showed that tag space is a significant challenge to adopting DRAM caches in CMP processors since it requires additional die area. These studies motivate the need to consider DRAM caches in future large-scale CMP platforms, but also highlight the need to address issues described in the next section.

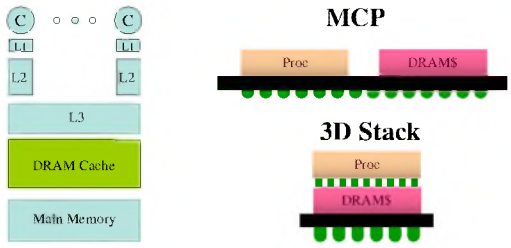

Fig. 1. Incorporating DRAM cache in large-scale CMP platforms.

B. DRAM Caching Benefits and Challenges

The basic approach to DRAM caching 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 off-chip (3D stacked or MCP). However, the tag space can be a significant challenge since they are large (e.g. 6MB for a 128MB cache) and require displacement of either cores or SRAM cache space on the die. To understand this problem better, we did an experiment with the TPC-C benchmark [4] running on an 8-threaded 4GHz processor with an 8MB last-level cache, a 128MB DRAM cache (with 64GB/Sec bandwidth and less than 30ns of latency) and a 12.8GB/Sec DRAM memory subsystem with 100ns of latency.

Figure 2 shows the DRAM cache benefits (in terms of misses per instruction and performance) along with the trade-offs (in terms of tag space required and main memory bandwidth utilization). The baseline is a platform configuration without DRAM cache and the first DRAM cache configuration assumes a cache line of 64 bytes. One approach to addressing tag space overhead is to increase the line size, thereby reducing the tag bits required. As shown in Figure 2(a), enabling larger cache line sizes provides a good improvement in the misses per instruction (MPI) due to the good spatial locality properties of server benchmarks such as TPC-C. However, the overall performance benefits are not significant beyond 64-byte cache lines due to (a) diminishing returns in memory stall time beyond the first jump from no DRAM cache to a DRAM cache with 64-byte lines and (b) the memory subsystem pressure in terms of significant increase in main memory bandwidth utilization. For example, the utilization is close to 100% when running with 2KB and 4KB cache lines. While this promotes the use of small cache lines, the significant challenge of tag space overhead remains unsolved. Figure 2(c) shows that a

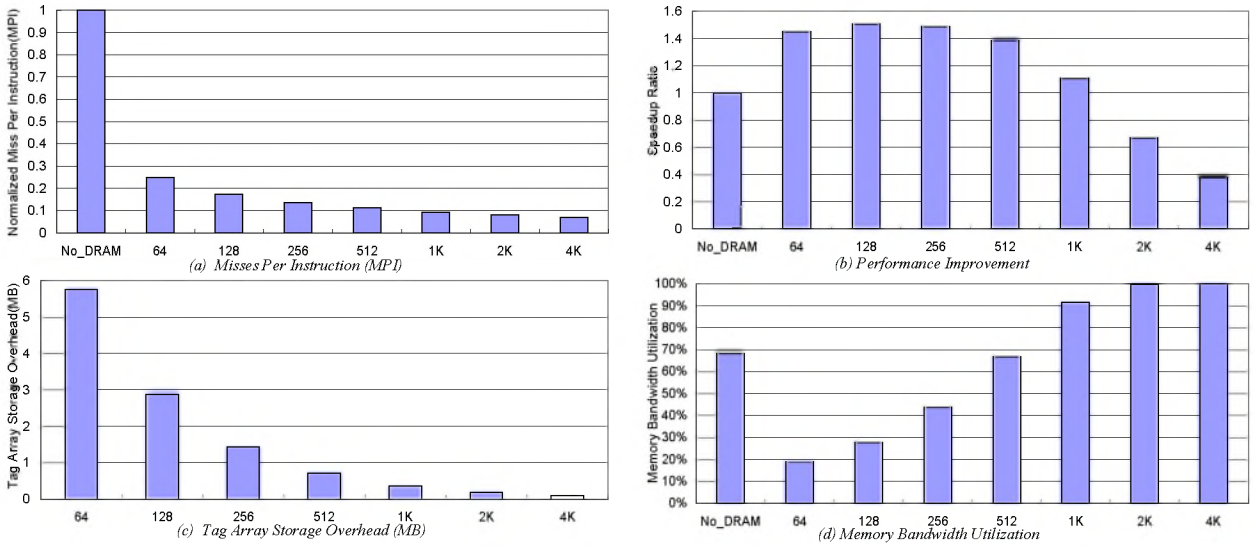

Fig. 2. Illustrative Overview of DRAM Cache Benefits, Challenges and Trade-offs.

128MB DRAM cache with 64 byte cache lines has a tag space overhead of 6MB. If we implement this, the die size has to either increase by 6MB (not attractive) or the last-level cache space has to be reduced to accommodate the tag space (i.e. the baseline of 8MB last-level cache requires to be shrinked to 2MB). We find that this offsets the performance benefits significantly and therefore is not a viable approach.

The goal of our work in this paper is to find approaches that enable inclusion of DRAM cache with minimal additional in die area and complexity. In order to do so, we adopt the DRAM cache with large line sizes as the basis for exploration since this solves the tag space problem by more than an order of magnitude (as shown in Figure 2(c), the 4KB line size enables a tag space that is only a few KBs). The problem then gets converted to addressing the memory bandwidth issue (as shown in Figure 2(d)). The data in Figure 2(d) shows how main memory bandwidth utilization increases with increase in line size. The memory bandwidth increases because the decrease in DRAM cache miss rate (or MPI) is not directly proportional to the increase in line size. For example, the reduction in MPI when going from line size of 64 bytes to 4KB is approximately 70% (resulting in a 1/3rd of MPI), whereas the increase in line size is 64×. In other words, if we had 100 misses with a DRAM cache of 64 bytes and it reduces to 30 misses with a DRAM cache of 4KB, then the memory bandwidth utilization will increase by ∼20× ((30*4KB)/(100*64B)). Such an increase in main memory bandwidth requirements results in saturation of the main memory subsystem and therefore immediately affects the performance of this DRAM cache configuration significantly (as shown by the 2K and 4K line size results in the last two bars of Figure 2(d) and Figure 2(b)).

The rest of this paper attempts to address this memory bandwidth increase by introducing filter-based DRAM caching techniques to reduce DRAM cache allocation while still retaining the majority of the performance benefits. In the next section, we begin by introducing filter-based DRAM caching and describe how they will satisfy these goals and requirements.

III. FILTER-BASED DRAM CACHING

In this section, we introduce the filter-based DRAM caching approach and then present several filter-based DRAM caching implementation options along with the detailed hardware support required for them.

The proposed filter-based DRAM caching approach is illustrated in Figure 3. Figure 3(a) shows the baseline DRAM caching approach where the DRAM cache is viewed as typical next level cache which allocates a cache line on every cache miss. As described in the previous section, with large cache lines (such as page-size), the main memory bandwidth requirement increases significantly and becomes the primary performance bottleneck. In order to deal with this, we introduce a very small filter cache (Figure 3b) that profiles the memory access pattern and identifies *hot pages*: pages that are heavily accessed due to temporal or spatial locality. By enabling a filter cache that identifies hot pages, we can then introduce a filter-based DRAM cache that only allocates for the hot pages. By eliminating allocation of cold pages in the DRAM cache, we expect to reduce the wasted bandwidth on allocating cache lines that never get touched later.

TABLE I
OFFLINE HOT PAGE PROFILING STATISTICS FOR SERVER WORKLOADS.

Workload	Hot Page Percentage	Hot Page Min #Access
SAP	24.8%	95
SPECjApps	38.4%	65
SPECjbb	30.6%	93
TPCC	7.2%	64
Avg	25.2%	79

Before we discuss the filter-based caching schemes, we performed a detailed offline profiling of workloads to obtain

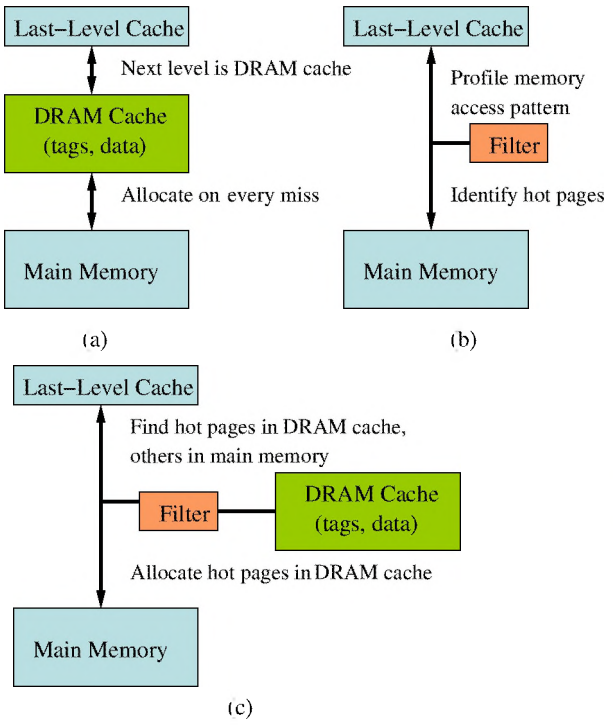

Fig. 3. Filter-based DRAM caching approach: (a) baseline DRAM cache without filter cache, (b) filter cache to profile access patterns for identifying hot pages, (c) filter-based DRAM cache allocation.

hot page (assuming a page is 4KB in size) information. Table I shows the results for four server workloads (each with 8 threads running in a platform with 16GB memory). The profiling is based on misses from an 8MB Last-level Cache (LLC) to remove the impact of lower-level cache accesses (detailed simulation information is presented in Section IV-A). For this profiling, *we define hot pages as the topmost accessed pages that contribute to 80% of the total access number.* The hot page percentage is calculated as the number of hot pages divided by the total number of pages for each workload. We can see that about 25% of the pages can be considered as hot pages. The last column also shows the minimum number of accesses to these hot pages. On average, a hot page is accessed at least 79 times.

A. Filter Cache (CHOP-FC)

Figure 4(a) shows our first filter-based DRAM caching architecture, where a Filter Cache (CHOP-FC) is incorporated on die along with the DRAM cache tag arrays (labeled as DT). The DRAM cache data arrays are assumed to be off die (either via 3D stacking or MCP). The filter cache stores its information at the same page granularity as the DRAM cache. In this baseline filter cache scheme, each entry in the filter cache includes a tag, LRU bits, and a counter to indicate how often the cache line (or page) is touched (Figure 4(b)). Using the counter information, the topmost referenced lines in the filter cache are recognized as hot pages that should be allocated into the DRAM cache, while the rest lines are filtered out as cold pages. When an LLC miss occurs, the filter cache is accessed based on page granularity. The counter value is initialized to zero for any newly allocated lines and incremented for every hit to the line. Once the counter value becomes greater than a certain threshold, this line is considered as a hot page and needs to be put into the DRAM cache. For simplicity pages in the filter cache are maintained exclusive to the ones in the DRAM cache.

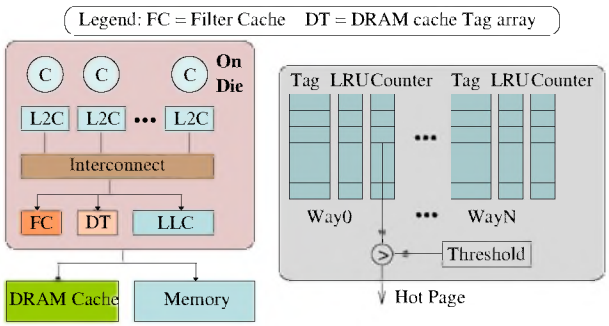

Fig. 4. Filter cache (CHOP-FC) for DRAM caching.

Once a hot page is identified by the filter cache, it needs to be allocated into the DRAM cache. To allocate a new line in the DRAM cache, a request for the new hot page is sent to the memory and a victim line needs to be selected if all cache ways in the corresponding cache set are occupied. Experimentally, we have measured that applying Least Frequently Used (LFU) replacement policy for picking victims in the DRAM cache achieves better performance than using regular LRU replacement policy. To employ LFU policy, the DRAM cache also maintains counters in the similar fashion as the filter cache. The counter value is incremented whenever a hit to the cache line occurs. Hence, the line that has the minimum counter value is selected as the victim to be replaced. Once a cache line is to be replaced from the DRAM cache, the victim along with its current counter value is put back into the filter cache with an initial value, while its data is written back to memory if needed. The initial value determines how fast the victim (which now becomes a cold page in the filter cache) can become hot again. We set it to be one half of the filter cache counter threshold so that the victim pages have a better chance to become hot again than other newer pages. We employ LRU as the default replacement policy for the filter cache. However, we also evaluated other alternative replacement policies (such as LFU), but our simulation results indicate that LRU works most efficiently for the filter cache. To compensate for program phase changes and process context switching effects that may vary the accuracy of the filter cache, a periodical reset to all counters in the filter cache and DRAM cache is simply applied.

For LLC misses that also miss in both the filter cache and DRAM cache, or hit in the filter cache but have their counter values less than the threshold, the requested pages are considered to be cold pages. For those pages, a regular request (64 bytes) is sent to the memory. If the LLC miss hits in the DRAM cache, which indicates that a hot page hit occurs, its counter is incremented and the request is sent to the DRAM cache instead of the memory.

Because the filter cache is accessed after an LLC miss

occurs, it adds extra latency to the system. In order to reduce/remove the extra latency, both the filter cache and the DRAM cache tag accesses can be performed in parallel with the LLC accesses. However, the counter update in the filter cache is delayed until LLC hit/miss is resolved. In this way, when the LLC miss occurs, the hot/cold page is already identified, so that a request to either the memory or the DRAM cache can be sent out immediately. However, if the LLC request turns out to be a hit, the result from the filter cache is simply discarded.

The filter cache and DT can be combined into one structure, where one more bit is required to indicate whether it is a filter cache line or DRAM cache line. We choose to separate the filter cache and DT so that they can have different cache organizations (associativity, number of entries, etc.) as well as different replacement policies.

B. Memory-based Filter Cache (CHOP-MFC)

In the baseline filter cache scheme, whenever a new line is brought into the filter cache, its counter value is set to zero. Therefore if a victim line is kicked out of the filter cache and later brought back again, all its history information is lost and it will be treated as a new line. If the filter cache is not big enough in size, many potential hot pages will be replaced before they reach the threshold so that very few hot pages can be identified. To deal with this, we propose a Memory-based Filter Cache (CHOP-MFC), where the counter is stored into memory (spill) when the line is replaced from the filter cache and restored (fill) when the line is fetched back. With a backup in the memory for counters, we can expect more accurate hot page identification being provided by the filter cache. In addition, it also allows us to safely reduce the number of entries in the filter cache significantly.

Storing the counters into memory requires allocating extra space in the memory. For a 16GB memory with 4KB page size and counter threshold of 256 (8-bit counter), 4MB memory space is required. We propose two options for this memory backup. In the first option, this memory space can be allocated in the main memory either by the Operating System (OS) or be reserved by the firmware/BIOS for the filter cache without being exposed to the OS. However, one performance issue still remains in that upon filter misses, off-chip memory accesses are required to look up the counters in the allocated memory region. For smaller filters that potentially have higher miss rates, the performance benefit of using filter cache may 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 128MB and only 4MB of memory space is required for counter backup, then the loss in DRAM cache space is minimal ($\sim$3%). This essentially requires a 32-way DRAM cache to lose one way in each set for data (this cache way is reserved for counter backup).While we can still implement a full tag array of 32 ways, one way in each set will be reserved for this configuration. For other configurations with different page sizes or DRAM cache sizes, the appropriate number of ways will need to be reserved based on the counter space reservation required.

To generate the DRAM cache address for retrieving the counter for a main memory page, we simply use the page address of a page to reference the counter memory space in the DRAM cache. In other words, the counter for page 0 (of main memory) is located at the first byte of the counter memory space, the counter for page 1 is located at the second byte, etc. We prefer this storing the counter in DRAM cache as opposed to storing the counter in main memory because the DRAM cache has much higher bandwidth than the main memory. But it should be noted that this DRAM cache backup option will only work as long as it continues to scale in size proportional to the main memory size.

Similar to CHOP-FC, whenever an LLC miss occurs, the filter cache is checked and the corresponding counter of the line is incremented. To further reduce the latency of counter lookups, prefetch requests of counters can be sent upon a lower-level cache miss. If a replacement in the filter cache occurs, the corresponding counter in the reserved memory space is updated (either in DRAM cache or in main memory). This counter write-back can be performed in the background. Once the counter in the filter cache reaches the threshold, a hot page is identified and installed into the DRAM cache.

One difference between CHOP-FC and CHOP-MFC is that in CHOP-MFC, victim lines from the DRAM cache are not put back into the filter cache again. The reason is that we now have the counter values backed up in memory, so prioritization over newer lines for DRAM cache victims is no longer needed. Since we keep the counter history for pages, once a page's accumulative access reaches the threshold, this page will stay hot forever. One problem arises in that pages that were identified as hot pages in a earlier period may become cold later on. To incorporate the timeliness information for hot pages, we clear the in-memory counter history information periodically by resetting the counter value to zero. Instead of walking through the entire counter backups and resetting the counters, we apply an approach that is much simpler in hardware cost. A time interval is set periodically, during which all counters are set to zero at the time when they are fetched into the filter cache. An alternative approach is to keep track of previous hot page utilization once it is brought back from the memory and downgrade it to cold page if its access number is less than a certain threshold. However this needs more hardware support in the DRAM tag array, so we do not consider this approach. Our simulation results indicate that the simple resetting approach works well.

C. Adaptive Filter Cache (CHOP-AFC)

As we already discussed, the CHOP-FC and CHOP-MFC schemes identify hot pages so that the DRAM cache allocates for hot pages only. In this way, the memory bandwidth utilization can be significantly reduced. However various workloads have distinct behaviors and even the same workload can have different phases, which all have various impacts on the memory subsystem. To incorporate the effect of such impacts,

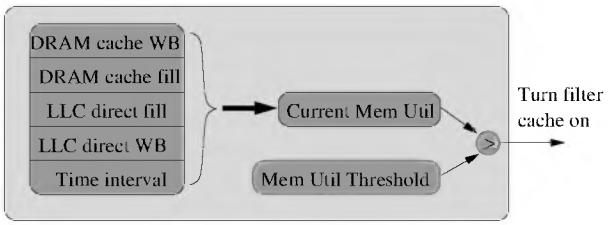

Fig. 5. Adaptive filter cache (CHOP-AFC).

we propose an Adaptive Filter Cache scheme (CHOP-AFC as shown in Figure 5), where the filter cache is turned on and off dynamically based on the up-to-date memory utilization status. We add a monitor to keep track of memory traffic and a register to contain an initial memory utilization threshold. Figure 5 shows that the monitor computes the memory utilization based on DRAM cache fills, write-backs, LLC fills from main memory and LLC write-backs to main memory. Alternatively, the memory bandwidth utilization can also be captured by using the performance monitoring registers in the memory controller that keep track of per DRAM channel utilization.

When memory utilization is greater than the threshold, the filter cache is turned on so that only hot pages will be fetched and allocated into the DRAM cache. If the memory utilization is less than this threshold, the filter cache is turned off, which means all pages are considered as hot pages so that they are brought into the DRAM cache on demand. This adaptive scheme can be combined with the baseline filter cache as well as the memory-based filter cache.

IV. FILTER-BASED DRAM CACHING EVALUATION

In this section, we first describe the simulation framework and the workloads that we use to evaluate the filter-based DRAM caching techniques, and then analyze the experiment results. The major metrics that we focus on are Speedup Ratios for performance, and Memory Bandwidth Utilization for cache memory subsystem behavior.

A. Evaluation Methodology

Simulation Environment. We use a trace-driven platform simulator called ManySim [42] to evaluate various filter cache schemes for CMP platforms. ManySim simulates the platform resources with high accuracy, while abstracting the core to optimize for speed. It contains a detailed cache hierarchy model, a detailed coherence protocol implementation (MESI), an on-die interconnect model and a memory model that simulates the maximum sustainable bandwidth specified in the configuration. ManySim was extended to support all filter cache schemes.

Architecture Configuration. The simulated CMP architecture is the same as the one illustrated in Figure 4(a). It consists of 8 cores operating at a frequency of 4GHz. Each core has its private 512KB L2 cache, and all cores share an 8MB Last level L3 cache. An on-die interconnect with bi-directional ring topology is used to connect L2 caches and the L3 cache. The off-die DRAM cache has a fixed size of 128MB, with an on-die tag array. L2 and L3 caches are inclusive whereas L3 and DRAM cache are non-inclusive. Filter cache has 32K entries, which is equivalent to the coverage of 128M bytes address space. Memory-based filter cache contains 64 entries with 4-way associativity. The detailed experiment parameters are listed in Table II.

TABLE II
MACHINE CONFIGURATIONS AND PARAMETERS.

Parameters	Values
Core	8 cores, 4GHz, in-order
L2 cache	512KB, 8-way, 64-byte block, 18-cycle hit latency, private
L3 cache	8MB, 16-way, 64-byte block, 30-cycle hit latency, shared
Interconnect BW	512GB/Sec
DRAM cache	128MB, 32-way, 64-byte block, 110-cycle access time, maximum sustainable bandwidth at 64GB/Sec
Filter Cache	128MB coverage
Memory	400-cycle access time, maximum sustainable bandwidth at 12.8GB/Sec

Workloads and Traces. We chose four key commercial server workloads: TPCC [4], SAP [1], SPECjbb2005 [3] and SPECjappserver2004 [2]. TPC-C is an online-transaction processing benchmark that simulates a complete computing environment where a population of users execute transactions against a database. The SAP SD 2-tier benchmark is a sales and distribution benchmark to represent enterprise resource planning (ERP) transactions. SPECjbb2005 is a Java-based server benchmark that models a warehouse company with warehouses that serve a number of districts (much like TPC-C). SPECjappserver2004 is a J2EE 1.3 application server. It is a multi-tier e-commerce benchmark that emulates an automobile manufacturing company and its associated dealerships. We also simulate the consolidated workloads by combining the four benchmarks together. For the consolidated workload, we quadruple the number of cores and the maximum sustainable memory bandwidth shown in Table II.

For all of these workloads, we collected long bus traces on Intel Xeon MP platform with 8 hardware threads running simultaneously and the last-level cache disabled. The traces include both instruction and data accesses, synchronization and inter-thread dependencies if there are any. They were replayed in the 8-core and 32-core simulator with different cache hierarchies and memory configurations as shown in Table II.

B. Filter Cache Evaluation

We now present the evaluation results of our baseline filter cache scheme (CHOP-FC). We first compare CHOP-FC with a baseline 128MB DRAM cache. To demonstrate the effectiveness of caching hot pages, we present the results of a naive scheme that caches a random portion of the working set. Ideally even this approach should be able to reduce the memory utilization. We also provide sensitivity studies of the filter cache with various counter thresholds, different coverages and an alternative replacement policy.

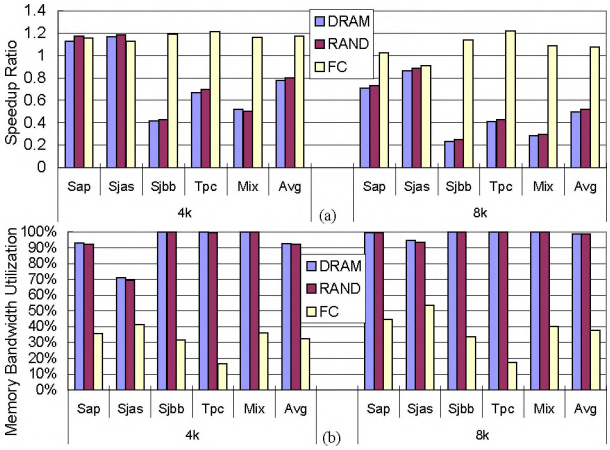

Fig. 6. Speedup ratios (a) and memory bandwidth utilization (b) of various schemes.

Effectiveness of Filter Cache. Figure 6(a) shows the speedup ratios of a basic 128MB DRAM cache (*DRAM*), a scheme that caches a random subset of the LLC misses (*RAND*) into the DRAM cache and our CHOP-FC scheme (*FC*) that captures the hot subset of pages. All results are normalized to the base case where no DRAM cache is applied. For the *RAND* scheme, a random number is generated for each new LLC miss and compared to a probability threshold to determine whether the block should be cached into the DRAM cache or not. We adjust the probability threshold and present the one that leads to the best performance in Figure 6.

The *DRAM* bar shows that directly adding a 128MB DRAM cache does not achieve as much performance benefit as expected, compared to the baseline no-DRAM cache case. Instead, it incurs slowdown in many cases. With 4KB cache line size, *DRAM* results in 21.9% slowdown on average with the worst case of 58.2% for Sjbb; with 8KB cache line size, it results in 50.1% slowdown on average with the worst case of 77.1% for Sjbb. To understand why this is the case, Figure 6(b) presents the memory bandwidth utilization of each scheme. Using large cache line sizes can easily saturate the maximum sustainable memory bandwidth, with an average of 92.8% and 98.9% for 4KB and 8KB line sizes, respectively.

Since allocating for each LLC miss in the DRAM cache saturates the memory bandwidth, one may suggest that why not caching just a subset of it. However, the *RAND* bar proves that this subset has to be carefully chosen. On average, *RAND* shows 20.2% (48.1%) slowdown for 4KB (8KB) line size. Figure 6(a) also demonstrates that our CHOP-FC scheme outperforms *DRAM* and *RAND* in general. With 4KB and 8KB line sizes, CHOP-FC achieves on average 17.7% and 12.8% speedup using counter threshold 32. The reason is that while hot pages are only 25.24% (Table I) of the LLC-missed portion of working set, it contributes to 80% of the entire LLC misses. Caching those hot pages significantly reduces memory bandwidth utilization and hence reduces the associated queuing delays. As compared to *RAND*, caching hot pages also provides much smaller MPI. It is observed that *DRAM* and *RAND* for SAP and SJAS with 4KB line size perform slightly better than CHOP-FC. This is because the memory bandwidth is not saturated even with 4KB line size for those two workloads. However, for the rest massive cases, DRAM and RAND incur slowdowns due to the saturated memory bandwidth.

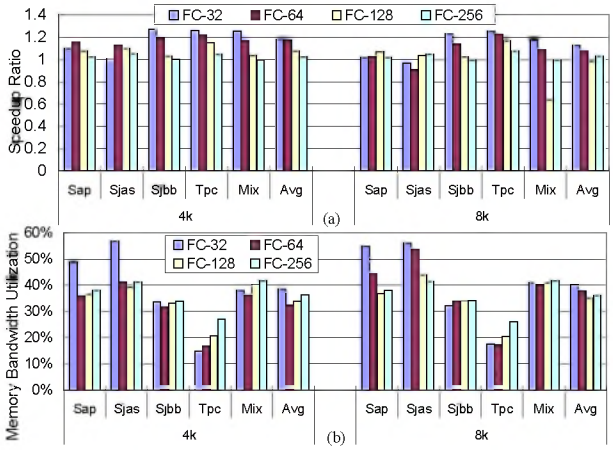

Fig. 7. Speedup ratios (a) and memory bandwidth utilization (b) of CHOP-FC with various thresholds.

Sensitivity to Counter Threshold. Figure 7(a) shows the speedup ratios of the CHOP-FC scheme with various counter thresholds (*FC-32* for threshold 32, *FC-64* for threshold 64 and so on) normalized to the no DRAM cache case, while Figure 7(b) shows the memory bandwidth utilizations.

Figure 7(a) depicts that increasing the counter threshold tends to reduce the performance benefits. With 4KB line size, *FC-32* achieves an average of 17.7% speedup. As we keep increasing the threshold to 256, the speedup is reduced down to 2.4%. CHOP-FC with line size of 8KB has similar trend. Not to our surprise, as illustrated in Figure 7(b), the memory bandwidth utilization reduces in most cases while the counter threshold increases. The first reason behind this is that increasing the counter threshold reduces the number of hot pages and hence the amount of DRAM fetches are also reduced. Secondly, with a higher counter threshold, the likelihood for a block to be evicted from the filter cache before it is identified as a hot page also increases. It essentially reduces the the chances for a block to be put into the DRAM cache and hence reduces the effectiveness of the filter cache. Since counter threshold 64 achieves the best (or the second best) performance in all workloads we evaluated, we use 64 as the default counter threshold for the rest of this paper.

Sensitivity to Filter Cache Sizes. Figure 8(a) presents the speedup ratios of our CHOP-FC with various address space coverages from 256KB (*FC_256K*) up to 128MB (*FC_128M*), and Figure 8(b) presents the memory bandwidth utilizations.

Figure 8 shows that with a larger coverage, CHOP-FC tends to provide better performance and to be more memory bandwidth efficient. For example, for 4KB line size, *FC_128M* outperforms all other cases with an average speedup of 17.2%. When the coverage is less than 64MB, CHOP-FC loses its functionality with the worst case slowdown of 0.1% for *FC_256K*. This is because having a smaller filter cache (or

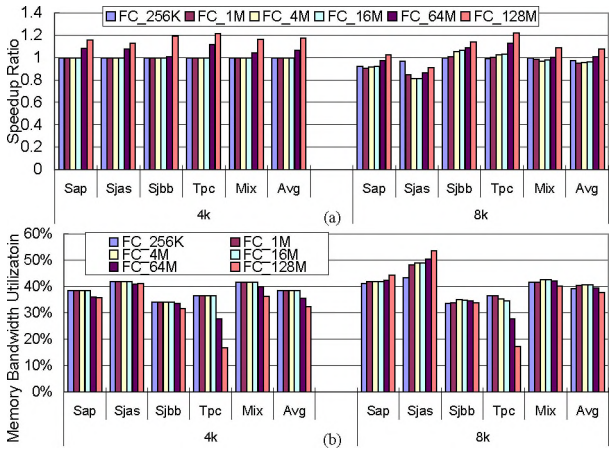

Fig. 8. Speedup ratios (a) and memory bandwidth utilization (b) of CHOP-FC with various filter cache sizes.

smaller coverage) tends to produce higher filter miss rate. Table III shows the filter cache miss rates as a function of cache sizes. With the worst case of 256KB coverage, filter cache leads to an average of 38.64% and 43.62% miss rate for 4KB and 8KB line sizes, respectively. Most blocks are evicted from the filter cache before they have a chance to become hot pages. Considering the fact that a block needs to be hit for 64 times in the filter cache in order to be allocated into the DRAM cache, such likelihood is quite low. Increasing the filter cache coverage increases performance. However, the performance improvement is achieved at the cost of increasing tag array storage overhead for the filter cache. Since *FC_128M* provides a reasonable speedup ratio and requires an acceptable (i.e. negligible impact to the LLC) storage overhead (0.8MB for 4KB line size), we choose to use 128MB coverage for the rest of the paper.

TABLE III
FILTER CACHE MISS RATES FOR VARIOUS FILTER CACHE SIZES.

Lsize	Workload	256K	1M	4M	16M	64M	128M
4K	Sap	30%	24%	18%	12%	7%	5%
	Sjas	28%	23%	18%	11%	5%	3%
	Sjbb	42%	36%	30%	26%	21%	11%
	Tpc	38%	31%	26%	20%	16%	21%
	Mix	54%	38%	29%	22%	15%	12%
	Avg	39%	30%	24%	18%	13%	10%
8K	Sap	35%	27%	21%	14%	8%	5%
	Sap	35%	27%	21%	14%	8%	5%
	Sjas	32%	25%	21%	13%	6%	4%
	Sjbb	44%	35%	30%	25%	20%	16%
	Tpc	38%	31%	26%	20%	14%	18%
	Mix	69%	42%	31%	24%	17%	14%
	Avg	44%	32%	26%	19%	13%	11%

Impact of Replacement Policy. In addition to the default LRU replacement policy, we also evaluated an alternative policy, namely Least Frequently Used (LFU) policy for the filter cache. Rather than replacing the least recently used block, a block with the minimal counter value is chosen to be the victim block. Although space limitation prevents us showing the detailed results here, we find that LFU is outperformed by LRU replacement policy in all cases. Using LFU replacement policy results in higher miss rate than LRU (on average 17.07% vs. 9.87% for 4KB line size). The reason is that using LRU for filter cache combines both timeliness and hotness information, while using LFU provides only hotness information and loses the equivalently important timeliness information.

C. Memory-based Filter Cache Evaluation

In this subsection, we present the experiment results for our memory-based filter cache scheme (CHOP-MFC). We first compare the performance and memory bandwidth utilization of CHOP-MFC against a regular DRAM cache using various counter thresholds, and then show its sensitivity to filter sizes.

Effectiveness of Memory-based Filter Cache. Figure 9 shows the speedup ratios (Figure 9(a)) and memory bandwidth utilization (Figure 9(b)) of a basic DRAM cache (*DRAM*) and CHOP-MFC with various counter thresholds (*MFC-32* for threshold 32 and so on).

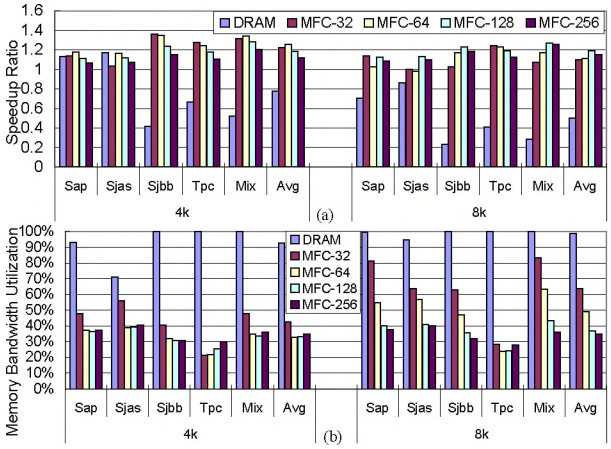

Fig. 9. Speedup ratios (a) and memory bandwidth utilization (b) of CHOP-MFC with various thresholds.

Figure 9(a) shows that CHOP-MFC with various counter thresholds outperforms *DRAM* in most cases. For 4KB line size, *MFC-64* achieves the highest speedup of 25.4%, followed by *MFC-32* of 22.5%, *MFC-128* of 18.6% and *MFC-256* of 12.0%, respectively. For 8KB line size, *MFC-128* leads the performance with an average speedup of 18.9%. The performance improvements come from the significant reduction in memory bandwidth utilization as shown in Figure 9(b). The results demonstrate that having a small-size memory-based filter cache is sufficient to keep the fresh hot page candidates. Unlike the replacements in CHOP-FC that result in the loss of hot page candidates, candidates have their counters backed up in the memory. Replacements in memory-based filter cache incur counter write-backs to memory rather than total loss of counters. Therefore higher hot page identification accuracy can be obtained by CHOP-MFC. We observe that CHOP-MFC with 64 entries has an average miss rate of 4.29% compared to 10.34% in CHOP-FC even with 32K entries for 4KB line size. Due to the performance robustness (i.e. speedups in all cases) provided by counter threshold 128, we choose to use it as the default value for the rest of this paper.

Sensitivity to Memory-based Filter Cache Sizes. We also vary the number of filter cache entries in CHOP-MFC from 16 to 1024 and measure the performance. We find that having only 16 entries for CHOP-MFC incurs much higher filter miss rate than having 64 entries (e.g. 20.36% vs. 4.29% for 4KB line size) for both 4KB and 8KB line sizes. As a result, more memory accesses are incurred for counter lookups that directly degrades the performance benefit and memory bandwidth efficiency achieved by CHOP-MFC. We also find that having more than 64 entries for the filter cache does not improve performance much because low miss rate is already obtained in the 64-entry filter cache, and filter cache with different sizes provides the same address space coverage since counters are backed up in memory. Therefore, we choose to use 64-entry CHOP-MFC as the default setup for the rest of this paper.

D. Adaptive Filter Cache Evaluation

We now apply the adaptive switching methodology for both CHOP-FC and CHOP-MFC. As mentioned before, we choose to use counter threshold 64 for CHOP-FC and 128 for CHOP-MFC. Due to space limitation, we only present the result of 4KB line size case. However, we observe the similar trends in 8KB line size case as well.

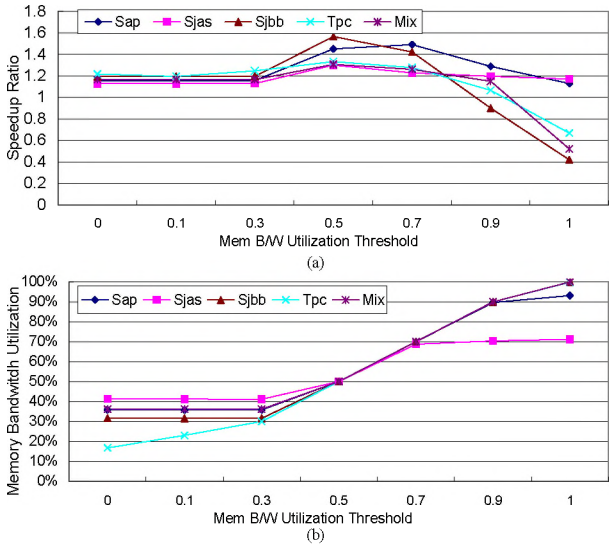

Fig. 10. Speedup ratios (a) and memory bandwidth utilization (b) of CHOP-AFC with various memory bandwidth utilization thresholds.

Adaptive Filter Cache. Figure 10 exhibits the speedup ratios (Figure 10(a)) and memory bandwidth utilization (Figure 10(b)) for Adaptive Filter Cache (CHOP-AFC) with memory utilization thresholds varying from 0 up to 1. We can see that for memory bandwidth utilization thresholds less than 0.3, the speedups achieved remain the same (except for TPCC); for threshold values between 0.3 and 1, the speedups show an increasing trend followed by an immediate decreasing. On the other hand, Figure 10(b) shows that for threshold less than 0.3, it achieves approximately the same memory bandwidth utilization (except for an increase in TPCC); for threshold values greater than 0.3, memory bandwidth utilization tends to increase along with the increase of the threshold.

To understand why this is the case, recall that for CHOP-AFC, when the measured memory bandwidth utilization is less than the threshold, all blocks are cached regardless of its hotness; while only hot pages are cached when measured memory bandwidth utilization is greater than the memory bandwidth utilization threshold (Section III-C). Moreover, the average memory bandwidth utilization achieved is 93% for *DRAM* case and 32% for CHOP-FC (Figure 6(b)), which essentially denotes the upper bound and lower bound of the memory bandwidth utilization that can be arrived by the CHOP-AFC scheme. When the threshold is less than 0.3, the measured memory utilization with filter cache turned on is always larger than the threshold so that the filter cache is always turned on, resulting in an average of 32% memory bandwidth utilization. Therefore with the threshold between 0 and 0.3, CHOP-AFC behaves the same as CHOP-FC. This explains why CHOP-AFC shows the constant speedups and memory bandwidth utilizations under those thresholds. When the threshold is greater than 0.3, due to the abundance of available memory bandwidth, the filter cache can be turned off for some time and therefore more pages are brought into the DRAM cache. As a result, the useful hits provided by the DRAM cache tend to increase as well. This explains why memory bandwidth utilization keeps increasing and performance keeps increasing in the beginning for threshold between 0.3 and 1. However, after some point, due to the high memory bandwidth utilization, queuing delay begins to dominate and consequently performance decreases. The reason that TPCC workload shows a different trend for thresholds between 0 and 0.3 is because of its smaller lower bound memory bandwidth utilization of 15% (*DRAM* bar in Figure 10(b)).

Adaptive Memory-based Filter Cache. Figure 11 shows the speedup ratios (Figure 11(a)) and memory bandwidth utilization (Figure 11(b)) for Adaptive Memory-based Filter Cache scheme (CHOP-AMFC). We can see that CHOP-AMFC has a similar trend as CHOP-AFC due to the same reason as explained above.

E. Comparison of Three Filter Cache Schemes

We now compare the effectiveness of all proposed filter schemes together. We first compare the extra on-die tag storage overhead incurred by CHOP-FC and CHOP-MFC, and then show the performance results for various schemes.

Tag Array Storage Overhead Comparison. Figure 12 shows the extra on-die storage overhead of using CHOP-FC (*FC*) and CHOP-MFC (*MFC*). The storage overhead incurred by CHOP-FC is roughly two orders of magnitude higher than CHOP-MFC. The reason is that CHOP-FC requires significantly more entries to keep the hot page candidates on die (e.g. 32K entries for 128MB coverage with 4KB line size), while CHOP-MFC only requires a small-size (e.g. 64 entries) storage to keep the fresh candidates. The reason that the storage overhead of CHOP-MFC remains constant while line size varies is because CHOP-MFC always stores counters

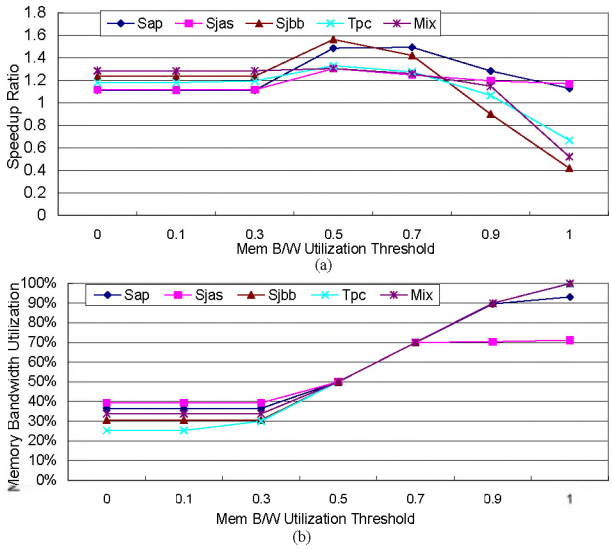

Fig. 11. Speedup ratios (a) and memory bandwidth utilization (b) of CHOP-AMFC with various memory bandwidth utilization thresholds.

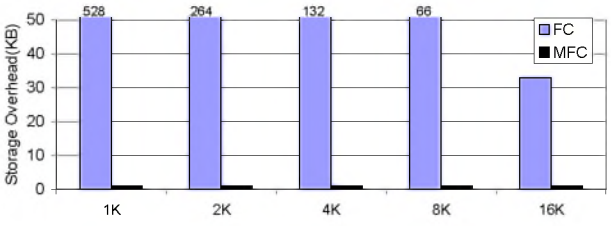

Fig. 12. Comparison of tag storage overhead for CHOP-FC and CHOP-MFC.

at a constant page-granularity rather than varying with the changes in line sizes.

Performance Comparison. Figure 13 exhibits the speedup ratios (Figure 13(a)) and memory bandwidth utilization (Figure 13(b)) obtained by the basic 128MB DRAM cache (*DRAM*), CHOP-FC (*FC*) , CHOP-MFC (*MFC*), CHOP-AFC (*AFC*) and CHOP-AMFC (*AMFC*). All results are normalized to the base case where no DRAM cache is used. We center our discussion around 4KB line size since results of 8KB line size show the similar trend.

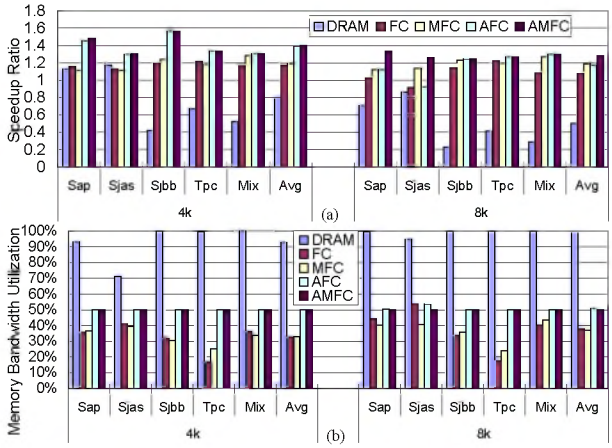

Fig. 13. Speedup ratios (a) and memory bandwidth utilization (b) of various filter cache schemes.

Figure 13(a) shows that while naively adding a DRAM cache with large line size does not improve performance due to the saturated memory bandwidth, using various filter cache schemes to cache only hot pages improve performance in general. For 4KB block size, CHOP-MFC achieves on average 18.6% speedup with only 1KB extra on-die storage overhead while CHOP-FC achieves 17.2% speedup with 132KB extra on-die storage overhead. Comparing these two schemes, CHOP-MFC uniquely offers larger address space coverage with very small storage overhead, while CHOP-FC naturally offers both hotness and timeliness information at a moderate storage overhead and minimal hardware modifications.

Figure 13(a) also shows that the adaptive filter cache schemes (CHOP-AFC and CHOP-AMFC) outperform CHOP-FC and CHOP-MFC schemes in all cases, with an average of 39.0% and 39.8% speedup, respectively (detailed analysis can be found in Section IV-D). CHOP-AFC and CHOP-AMFC intelligently detect the available memory bandwidth to dynamically adjust DRAM caching coverage policy to improve performance. Dynamically turning the filter cache on and off can adapt to the memory bandwidth utilization on the fly: (1) when memory bandwidth is abundant, coverage is enlarged and more blocks are cached into the DRAM cache to produce more useful cache hits, and (2) when memory bandwidth is scarce, coverage is reduced and only hot pages are cached to produce reasonable amount of useful cache hits as much as possible. By setting a proper memory utilization threshold, CHOP-AFC and CHOP-AMFC can use memory bandwidth wisely.

F. Sensitivity to Higher Maximum Sustainable Memory Bandwidth

Figure 14 shows the results of various schemes under higher maximum sustainable memory bandwidth. What is markedly different is that when memory bandwidth becomes abundant, the effectiveness of CHOP-FC and CHOP-MFC reduces. The reason is that when the maximum sustainable memory bandwidth is high enough, even a regular DRAM cache with large line sizes does not saturate the bandwidth (72.9% and 50.9% bandwidth utilization for 25.6GB/Sec and 51.2GB/Sec cases, respectively). CHOP-FC and CHOP-MFC reduce memory bandwidth utilization but provide a lower coverage and thus incur more DRAM cache misses. This demonstrates the fact that CHOP-FC and CHOP-MFC only perform well under scarce memory bandwidth conditions.

However, CHOP-AFC and CHOP-AMFC again intelligently adjust their coverage with respect to the available memory bandwidth and show a robust performance of higher speedup ratios (51.1% and 56.3% speedup ratios under 25.6GB/Sec and 51.2GB/Sec cases, respectively)

V. RELATED WORK

DRAM caches have been investigated to improve performance especially for CMP platforms. Madan et al. [25] and Black et al. [6] recently evaluated a 3D stacked approach for DRAM caches that provides significant bandwidth and latency

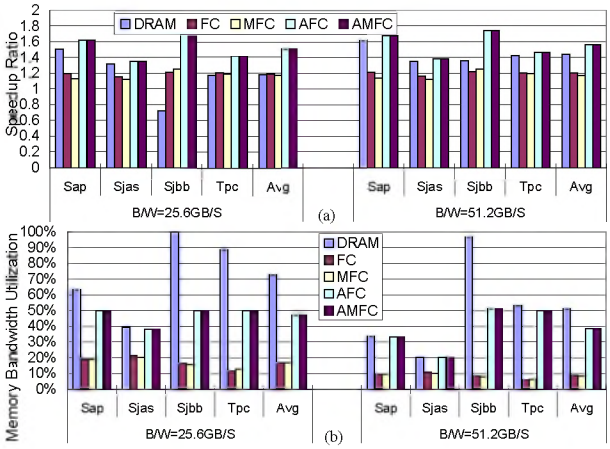

Fig. 14. Speedup ratios (a) and memory bandwidth utilization (b) of various filter cache schemes at 25.6GB/Sec and 51.2GB/Sec maximum sustainable memory bandwidth.

benefits for improved overall performance for server and RMS workloads respectively. Zhao et al. [41] observed that the overhead of tag area is a crucial impediment to adopting DRAM caches since the tags need to be placed on-die. They proposed the use of partial tags and sectoring as a potential approach to reducing DRAM cache tag size. Zhang et al. [39] studied a cached DRAM that integrates an SRAM cache in the DRAM memory to exploit the locality in memory accesses and thus reduces the miss penalty. Rogers et al. [34] pointed out using DRAM cache can achieve super-proportional core scaling for future generations of CMPs even under memory bandwidth wall constraints. In contrast to these works, our work uniquely tackles the tradeoff between memory bandwidth efficiency and tag storage overhead of DRAM caches.

There are also many prior proposals on managing off-chip memory bandwidth usages. Those proposals effectively use the available memory bandwidth by scheduling memory requests based on their characteristics [21], [33], or by partitioning the off-chip memory bandwidth across different cores [16], [27], [28], [29], [32]. However, while alleviating the average queuing delay for memory requests, none of them alleviates the memory bandwidth problem incurred by the DRAM cache with large allocation granularity. In contrast, our scheme saves memory bandwidth by identifying and caching only hot pages.

Many prior schemes have been proposed on efficiently managing the cache using filters, predictors and so on. Some of those schemes address power and thermal issues. For example, filters are used to filter out infrequently accessed cache blocks to reduce the power consumption of L1 caches [13], [19]; while other schemes strive to improve the cache performance. Qureshi et al. [31] propose line distillation to filter out the unused words in a cache line to increase effective cache capacity. Moshovos et al. [26] propose RegionScout to identify coarse-grain sharing patterns in shared memory multiprocessors to improve performance. Liu et al. [23] propose a new class of dead-block predictors to predict dead cache blocks and use them as prefetch positions to improve performance. Subramanian et al. [35] propose an adaptive scheme that switches between various cache replacement policies to improve performance. Those schemes help improve the performance of L1/L2 caches in which the design tradeoff significantly differs from DRAM caches. In contrast, our scheme deals with the memory bandwidth and tag space storage overhead tradeoff in DRAM caches. There are also proposals [18], [36] that dynamically adjust the cache block size to help cache performance. However, the proposed schemes have significant amount of hardware complexity and still do not solve the memory bandwidth and tag space overhead tradeoff. Cache affinity control mechanisms [5], [8], [17] have been proposed to control whether a certain block or a specific memory region should be fetched into or bypass the cache hierarchy. While these schemes rely on programmers to identify opportunities for performance improvement, our scheme is a hardware-based scheme that does not require any programmer's intervention. In addition, sectored caches [7], [20], [30] also save memory bandwidth by fetching on chip only words that are likely to be referenced. However, sectored caches cannot effectively utilize the entire cache capacity since unfilled sectors still occupy cache space. In contrast, our scheme intelligently fetches only hot pages and hence effectively uses the entire cache capacity while saving memory bandwidth.

Identifying hot subset of working sets to utilize them for optimizing performance has also been proposed before [22]. However, our work differs significantly from all prior works. Etsion et al. [13] use probabilistic filter to identify blocks that contribute to the most L1 cache accesses and put such blocks into a small direct-mapped lookup table to eliminate the vast majority of costly fully associative lookups. While they point out that 80% of the L1 accesses come from 20% of the working set, the big question of whether the LLC misses still obey this *20/80 principle* remains unclear. In contrast, in our work, we identify that 20/80 principle stays valid for LLC misses in server workloads. We also apply this rule to improve performance and save memory bandwidth for DRAM caches. Zhang et al. [38] propose augmenting TLB with counters to identify hot pages as page coloring candidates to reduce the cost of applying page coloring. In contrast, our scheme differs in three aspects. First, our filter cache scheme looks for hot pages targeting towards LLC misses while their scheme focuses on all memory accesses. Secondly, hot pages in our work are cached to uniquely offer a storage and memory bandwidth efficient DRAM cache organization, while hot pages in their work are used for page coloring purposes. Thirdly, counters in our memory-based filter cache are used without interfering the Operating System (OS) or changing the TLB structure, while counters in their scheme require OS interaction as well as TLB structural changes.

VI. Conclusions

In this paper, we proposed Caching HOt Pages (CHOP) for DRAM caching. We studied four schemes: the basic filter cache, memory-based filter cache, adaptive filter cache and adaptive memory-based filter cache. Our simulation results demonstrated that using a regular 128MB DRAM cache with large line size alone easily saturates the available memory

bandwidth. This phenomenon will become more significant as more and more cores will be integrated on die and generate more memory traffic. However, with our carefully designed various filter cache schemes, up to over 30% speedup can be obtained by having a 128MB DRAM cache with the filter cache. Only negligible amount of storage overhead is incurred for holding the filter cache on die (132KB for CHOP-FC and 1KB for CHOP-MFC). Our adaptive filter schemes show their performance robustness and guarantee performance improvement regardless of whether memory bandwidth is abundant or scarce.

As future work, we would like to apply the filtering techniques to other forms of two-level memory hierarchies including PCM or Flash-based memories. We expect that exploring hot page allocation for these will enable significant efficiency in the behavior of the large DRAM caches and therefore improve performance significantly. We also plan to explore architectural support for exposing hot page information back to the OS and applications so that dynamic optimizations can be achieved either in terms of scheduling or in terms of runtime binary optimizations.

ACKNOWLEDGMENT

We are very grateful to the anonymous reviewers and Zhen Fang for their comments and feedback on the paper.

REFERENCES

[1] SSAP Benchmarks. *http://www.sap.com/solutions/benchmark/index.epx*.
[2] SSPECjAppServer Java Application Server Benchmark. *http://www.spec.org/jAppServer/*.
[3] SSPECjbb2005. *http://www.spec.org/jbb2005/*.
[4] TPC-C Design Document. *www.tpc.org/tpcc/*.
[5] PowerPC User Instruction Set Architecture. *IBM Corporation*, page 161, 2003.
[6] B. Black el al. Die Stacking (3D) Microarchitecture. *In the Proc. of th 39th Int. Symposium on Microarchitecture(MICRO)*, 2006.
[7] C. Chen, S. Yang, B. Falsafi, and A. Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. *In the Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, 2004.
[8] Intel Corporation. Intel Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. *http://www.intel.com*, 2006.
[9] Intel Corporation. World's first quad-core processors for desktop and mainstream processors. *http://www.intel.com/quad-core/*, 2007.
[10] Intel Corporation. Intel Nehalem Processors. *http://www.intel.com/technology/architecture-silicon/next-gen/*, 2008.
[11] Intel Corporation. Tera-Scale Computing. *http://www.intel.com/research/platform/terascale/index.htm*, 2008.
[12] Intel Corporation. Intel Xeon 5500 Series. *http://www.intel.com/p/en_US/products/server/processor/xeon5000*, 2009.
[13] Y. Etsion and D. Feitelson. L1 Cache Filtering Through Random Selection of Memory References. *In the Proc. of the 16th Int. Conf on Parallel Architecture and Compilation Techniques(PACT)*, 2007.
[14] R. Golla. Niagara2 : A Highly Threaded Server-on-a-Chip. *http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf*, 2006.
[15] AMD Inc. AMD Operton Processor Family. *http://www.amd.com*, 2009.
[16] R. Iyer. Performance Implications of Chipset Caches in Web Servers. *In the Prof of the 2003 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2003.
[17] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. *In the Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 2009.
[18] T. Johnson. Run-time adaptive cache management. *PhD thesis, University of Illinois, Urbana Champion* , 1998.
[19] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *In the Proc. of the 30th Int. Symp. on Microarchitecture (MICRO)*, 1997.
[20] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. *In the Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, 1998.
[21] C. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controller. *In the Proc. of the 41st Int. Symp. on Microarchitecture(MICRO)*, 2008.
[22] C. Lin, C. Yang, and C. Lee. HotSpot Cache: Joint Temporal and Spatial Locality Exploitation for ICache Energy Reduction. *In the Proc. of Int. Symp. on Low Power Electronics and Design(ISLPED04)*, 2004.
[23] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. *In the Proceedings of the International Symposium on Microarchitecture(MICRO)*, 2008.
[24] G. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. *In the Proc. of the 35th Int. Symposium on Computer Architecture(ISCA)*, 2008.
[25] 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 the Proceedings of the International Symposium on High Performance Computer Architecture(HPCA)*, 2009.
[26] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *In the Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.
[27] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. *In the Proceedings of the International Symposium on Microarchitecture(MICRO)*, 2007.
[28] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. *In the Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.
[29] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair Queuing Memory System. *In the Proceedings of the International Symposium on Microarchitecture(MICRO)*, 2006.
[30] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. *In the Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, 2006.
[31] M. Qureshi, A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. *In the Proceedings of the International Symposium on High Performance Computer Architecture(HPCA)*, 2007.
[32] N. Rafique et al. Effective Management of DRAM Bandwidth in Multicore Processors. *In the Proc. of the 16th Int. Conf on Parallel Architectures and Compilation Techniques(PACT)*, 2007.
[33] S. Rixner. Memory Access Scheduling. *In the Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.
[34] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *In the Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009.
[35] R. Subramanian, Y. Smaragdakis, and G. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. *In the Proceedings of the International Symposium on Microarchitecture(MICRO)*, 2006.
[36] A. Veidenbaum et al. Adapting cache line size to application behavior. *In the Proc. of Int. Conference on Supercomputing(ICS)*, 1999.
[37] T. Yamauchi, L. Hammond, and K. Olukotun. A Single Chip Multiprocessor Integrated with High Density DRAM. *Tech. Rep., Stanford University*, 1997.
[38] X. Zhang et al. Towards Practical Page Coloring-based Multi-core Cache Management. *In EuroSys09*, 2009.
[39] Z. Zhang et al. Cached DRAM: A Simple and Effective Technique for Memory Access Latency Reduction on ILP Processors. *In IEEE Micro*, 2001.
[40] Z. Zhang et al. Design and Optimization of Large Size and Low Overhead Off-chip Caches. *In IEEE Transactions on Computer*, 2004.
[41] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM Cache Architectures for CMP Server Platforms. *In the Proceedings of the 25th International Conference on Computer Design (ICCD)*, 2007.
[42] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring Large-Scale CMP Architectures Using ManySim. *In IEEE Micro*, 2007.