2017 IEEE International Symposium on High Performance Computer Architecture

Near-Optimal Access Partitioning for Memory Hierarchies with Multiple Heterogeneous Bandwidth Sources

Jayesh Gaur[†] Mainak Chaudhuri[‡] Pradeep Ramachandran[$⋆] Sreenivas Subramoney[†]
[†]Microarchitecture Research Lab, Intel Corporation
[‡]Department of Computer Science and Engineering, Indian Institute of Technology Kanpur
[$]MulticoreWare Incorporated
jayesh.gaur@intel.com, mainakc@cse.iitk.ac.in, pradeep@multicorewareinc.com, sreenivas.subramoney@intel.com

***Abstract*—The memory wall continues to be a major performance bottleneck. While small on-die caches have been effective so far in hiding this bottleneck, the ever-increasing footprint of modern applications renders such caches ineffective. Recent advances in memory technologies like embedded DRAM (eDRAM) and High Bandwidth Memory (HBM) have enabled the integration of large memories on the CPU package as an additional source of bandwidth other than the DDR main memory. Because of limited capacity, these memories are typically implemented as a memory-side cache. Driven by traditional wisdom, many of the optimizations that target improving system performance have been tried to maximize the hit rate of the memory-side cache. A higher hit rate enables better utilization of the cache, and is therefore believed to result in higher performance.**

In this paper, we challenge this traditional wisdom and present DAP, a Dynamic Access Partitioning algorithm that sacrifices cache hit rates to exploit under-utilized bandwidth available at main memory. DAP achieves a near-optimal bandwidth partitioning between the memory-side cache and main memory by using a light-weight learning mechanism that needs just sixteen bytes of additional hardware. Simulation results show a 13% average performance gain when DAP is implemented on top of a die-stacked memory-side DRAM cache. We also show that DAP delivers large performance benefits across different implementations, bandwidth points, and capacity points of the memory-side cache, making it a valuable addition to any current or future systems based on multiple heterogeneous bandwidth sources beyond the on-chip SRAM cache hierarchy.

***Index Terms*—DRAM cache; memory system bandwidth; access partitioning;**

I. Introduction

Despite advances in CPU architecture and memory technology, the memory wall continues to remain a major bottleneck to application performance. The problem is further exacerbated by the ever-increasing bandwidth demand and working set of applications. Large eDRAM [1], [17], [21], [22], [31], [42], [45] and HBM [24], [46] memory-side caches located between the DDR main memory and on-chip SRAM cache hierarchy have been proposed as a solution to address this problem. These caches are much larger than traditional SRAM caches and can deliver $4\times$-$8\times$ higher bandwidth than the DDR main memory [4], [25]. Like traditional caches, most of the architecture optimizations for these memory-side caches have been focused on maximizing hit rates [26], minimizing hit latency [25], [39], or reducing additional overheads [4].

In this paper, we make the key observation that increasing the cache hit rate above a certain threshold no longer improves delivered bandwidth as the system bandwidth gets saturated to the cache bandwidth. At this point, however, the bandwidth from the main memory remains unused. Augmenting the cache bandwidth with additional bandwidth from the main memory may improve system performance. This is more useful for memory-side caches because their bandwidth is only $4\times$-$8\times$ of that of DDR main memory and latency is just slightly better [4], [26]; SRAM caches, on the other hand, have orders of magnitude better latency and bandwidth characteristics than DDR main memory. However, this augmentation needs to be carefully managed as the main memory is tightly coupled with the regular maintenance operations of the memory-side cache.

This paper, hence, presents DAP, a Dynamic Access Partitioning system, that improves delivered bandwidth by utilizing all available bandwidth sources effectively. While attempting to maximize delivered bandwidth, DAP may even sacrifice the hit rate of the memory-side cache. Specifically, we make the following contributions.

1) We present a simple, yet rigorous, analytical model to understand the optimal bandwidth partitioning in a system with multiple bandwidth sources.
2) Guided by this analytical model, we present DAP, a light-weight learning mechanism, that intelligently recruits under-utilized bandwidth at main memory to improve system performance. DAP is the first holistic and analytically complete solution to maximizing system bandwidth from multiple bandwidth sources.
3) We demonstrate DAP's natural scaling to different memory-side cache architectures by examining its operations when the memory-side cache is based on die-stacked HBM DRAM like the sectored DRAM cache [25], or the Alloy cache [39], both of which have a single set of bandwidth channels for serving reads and writes. We also explore DAP on sectored eDRAM caches [42] that have two independent sets of bandwidth channels for serving reads and writes.

Our results show that on 44 multi-programmed workloads, DAP improves the performance of an eight-core system with 4 GB of

⋆ The author contributed to the work while at Intel.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.46

die-stacked HBM DRAM cache having 102.4 GB/s bandwidth and a dual-channel DDR4-2400 main memory by 13%, while adding only sixteen bytes of hardware. We also show that DAP elegantly overcomes several fundamental flaws in the current memory-side cache optimizations and scales seamlessly to future memory-side caches with higher bandwidth and capacity.

II. Background and Motivation

Integrating stacked DRAM or eDRAM caches on the CPU package can help scale the memory wall. However, cost and technology constraints limit the capacity of these memories to a few hundred megabytes in case of eDRAM [31] and a few GB for DRAM cache [4]. Hence, such memories are always used in conjunction with a commodity DRAM of higher capacity, but much lower bandwidth. In such a capacity-limited scenario, using these memories as a memory-side cache is more attractive than using them as OS-visible memory. For brevity, we assume that the stacked or in-package memory is a memory-side cache, although the algorithms described can easily be extended to OS-visible implementations. For this work, we focus on three different implementations of the memory-side cache.

Die-stacked Sectored DRAM Cache. Sectored or sub-blocked caches use an allocation unit (referred to as a sector) ranging in size from 512 bytes to 4 KB. Each sector is composed of a number of contiguous conventionally-sized cache blocks (64 bytes). The data fetched from main memory on a demand miss is only a cache block. Therefore, the sectored cache proposals can simultaneously optimize the main memory bandwidth requirement and the tag store size [13], [18], [23], [25], [26], [33], [37], [38], [40], [47], [51], [52]. The metadata for each sector is stored in the DRAM cache. Hence, additional metadata reads and updates are needed. A small set-associative SRAM tag cache [4], [19], [36], [50] can be used to reduce these bandwidth overheads.

Die-stacked Non-sectored DRAM Cache. Alloy Cache [39] proposes to organize the DRAM cache as a direct-mapped cache with tag and data (TAD) of a cache block fused and maintained together in the DRAM array. On a lookup, one TAD unit is read from DRAM and used to determine a hit or a miss. This removes the serialization latency of the tag and data, thereby reducing the overall latency of hits. However, accessing a TAD (72B) instead of just data (64B) reduces the useful data bandwidth available in the alloy cache. Hence, alloy cache sacrifices bandwidth for improved latency. A recent proposal (BEAR) addresses a few bandwidth inefficiencies of the Alloy cache [4].

Sectored eDRAM Cache. Intel Crystalwell and Skylake architectures implement up to 128 MB sectored embedded DRAM caches with all tags on die [17], [22], [31], [42]. Unlike DRAM caches that have a single set of channels for serving reads and writes, the eDRAM caches have two separate sets of channels for serving reads and writes. Its operation is otherwise similar to a sectored DRAM cache.

A. Bandwidth and Hit Rate

Memory-side cache architectures are typically optimized for average access latency. Predictors and prefetchers have been proposed to further hide memory latency and improve hit rates [26], [39]. However, we should note that the primary benefit of a memory-side cache is its bandwidth and not the latency [4], [42]. To understand the bandwidth delivery, we experiment with a simple read bandwidth kernel that streams through read-only arrays at different target hit rates of the memory-side cache. We consider two different architectures for the memory-side cache. The first one, based on HBM DRAM, has a bi-directional 102.4 GB/s bus to handle both reads and writes. The second one, based on an eDRAM cache, has separate 51.2 GB/s channels for reads and writes. In both these architectures, 38.4 GB/s two-channel DDR4 is used as main memory.[1] For simplicity, we assume all memory-side cache tags on-die and no metadata maintenance overheads for the memory-side caches. Figure 1 shows the bandwidth delivered as the hit rate in the cache is increased.

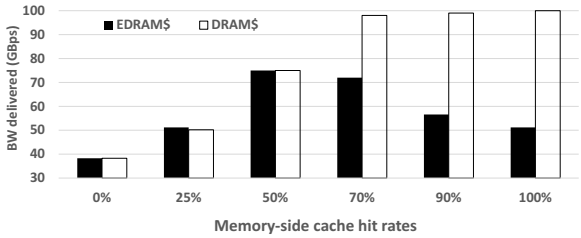

Fig. 1. Delivered bandwidth against memory-side cache hit ratio.

In the case of HBM DRAM cache, Figure 1 shows that the delivered bandwidth increases initially as the hit rates increase. In this region, the main memory limits the bandwidth and as the hit rates increase, more traffic is served by the higher bandwidth memory-side cache, thereby increasing the delivered bandwidth. At around 70% hit rate, the bandwidth delivered is close to the cache bandwidth. At this cache hit rate, 70% of the reads are served by cache bandwidth, and 30% by memory bandwidth. However, the cache fills arising from the 30% read misses reduce the useful cache bandwidth. After this point, more requests are served by the memory-side cache and read miss fills reduce. At 100% hit rate, all requests are served by the memory-side cache and the main memory bandwidth is completely unused. The overall delivered bandwidth is almost constant from 70% to 100% hit rate because the bandwidth contribution from the main memory reduces as hit rates increase.

The case of eDRAM cache is even more interesting. Figure 1 shows that once the hit rate increases beyond 50%, there is actually a loss in delivered bandwidth. At 50% hit rate, half of the requests are satisfied by the main memory and half by the eDRAM cache's read channels. We should note that the eDRAM cache has separate write channels to handle the fills. Hence, unlike the DRAM cache, the fills arising from the read misses are served by the eDRAM cache's write channels and do not reduce the read bandwidth of the cache. As a result, the overall delivered read bandwidth is the sum of the delivered bandwidth from the main memory and the eDRAM read channels. However, at 100% hit rate all requests have to be served by the eDRAM cache, and hence, the bandwidth saturates to the eDRAM cache's read bandwidth. This phenomenon is also seen on real benchmarks. The top panel in Figure 2 shows the performance impact of doubling the eDRAM memory-side cache from 256 MB to 512 MB in an eight-core system for twelve bandwidth-sensitive benchmarks that were run in rate-8 mode (eight copies of each

[1] Simulation framework is discussed in Section V.

benchmark run on eight cores). Also shown is the drop in the miss rate with increased eDRAM cache capacity (bottom panel).

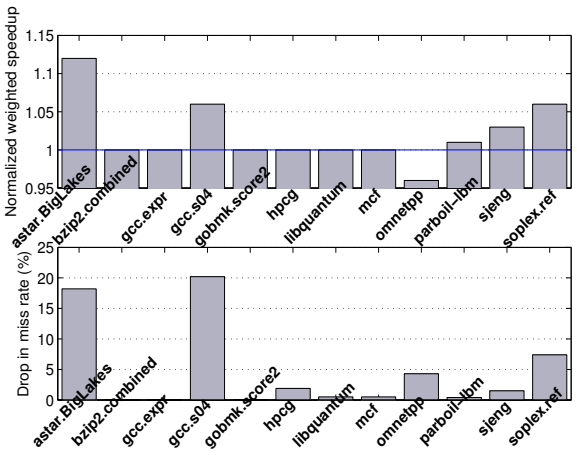

Fig. 2. Top panel: Weighted speedup with a 512 MB eDRAM cache normalized to a 256 MB eDRAM cache. Bottom panel: Drop in miss rate while going from a 256 MB eDRAM cache to a 512 MB eDRAM cache.

As expected, the performance of most applications improves significantly with significant drop in miss rates. However, `gcc.s04` gains only 5% when its miss rate drops by almost 20%, and `omnetpp` suffers a 4% drop in performance even though there is a 5% drop in miss rate. These results clearly indicate that merely improving the hit rate or latency of the memory-side caches may not yield the optimal performance. For the best outcome, we need to utilize all available sources of bandwidth effectively. To solve this optimization problem, we first develop an analytical model to understand how bandwidth is delivered in systems with multiple sources of bandwidth.

III. Bandwidth Equation

Consider a system with n distinct, non-blocking, parallel sources of bandwidth. Let the number of accesses served by these n sources be $A_1, A_2, \ldots, A_n$. Then $A = \sum_{i=1}^{n} A_i$, is the total number of accesses that have to be satisfied by these sources. Let the bandwidth of the sources be $B_1, B_2, \ldots, B_n$ expressed as number of accesses per unit time, where each access transfers a fixed amount of data (64 bytes in this study) over the channels. In this section, we answer the following question: given the bandwidth of each bandwidth source and the total number of accesses that the bandwidth sources serve, how should the accesses be distributed across the bandwidth sources so that the overall bandwidth delivered by the system is maximized?

We know that bandwidth is work done divided by the time taken to complete the work. The time taken to serve A accesses together by all sources is given by $T = \max(A_1/B_1, A_2/B_2, \ldots, A_n/B_n)$. If we denote A_i/A by f_i (the fraction of work done by source i), the overall delivered bandwidth is:

$$B = \frac{A}{T} = \frac{1}{\max(f_1/B_1, f_2/B_2, \ldots, f_n/B_n)}. \quad (1)$$

Equation 1 can be simplified to

$$B = \min(B_1/f_1, B_2/f_2, \ldots, B_n/f_n). \quad (2)$$

To understand Equation 2, let us consider a simple example of a system comprising of two different memory modules, M_1 and M_2, one delivering 102.4 GB/s and the other delivering 51.2 GB/s. If all accesses go to M_1 ($f_1 = 1, f_2 = 0$), then the delivered bandwidth will be 102.4 GB/s. If exactly half of the accesses go to each memory ($f_1 = 0.5, f_2 = 0.5$), then Equation 2 shows that the delivered bandwidth of the system will only be 102.4 GB/s, bottlenecked by M_2. It also shows that M_1 is underutilized and the system is imbalanced. This simple observation leads to a powerful conclusion that we derive in the following.

We note that maximizing B would automatically minimize $T = A/B$, given a constant A. Therefore, we can succinctly state the optimization problem that we strive to solve: maximize $\min(B_1/f_1, B_2/f_2, \ldots, B_n/f_n)$ subject to $\sum_{i=1}^{n} f_i = 1$. Let $\min(B_1/f_1, B_2/f_2, \ldots, B_n/f_n) = \lambda$. Therefore, for all i, $B_i/f_i \geq \lambda$ or $\lambda f_i \leq B_i$. Summing up on both sides of this inequality yields the upper bound for λ: $\lambda \leq \sum_{i=0}^{n} B_i$. Therefore, the maximum possible value of λ is $\sum_{i=0}^{n} B_i$ leading to

$$\max \min(B_1/f_1, B_2/f_2, \ldots, B_n/f_n) = \sum_{i=0}^{n} B_i. \quad (3)$$

This is attained when $f_i = B_i / \sum_{i=0}^{n} B_i$ and $B_1/f_1 = B_2/f_2 = \cdots = B_n/f_n = \sum_{i=0}^{n} B_i$. This solution also matches with the intuition that the maximum delivered bandwidth of the system is the sum total of the bandwidths available from different sources. Therefore, we conclude that to maximize the delivered bandwidth, we need to make sure that $B_1/f_1 = B_2/f_2 = \cdots = B_n/f_n$. In other words, the accesses should be distributed to the different bandwidth sources in the proportion of their bandwidths. In our simple example discussed above, this means sending $\frac{2}{3}$rd accesses to M_1 and $\frac{1}{3}$rd accesses to M_2. M_1 should do twice the work as compared to M_2, since M_1 has double the bandwidth.

The calculation for bandwidth is slightly more involved for our target systems with memory-side caches backed up by main memory. This is because apart from serving accesses from the CPU, some part of the cache and main memory bandwidth is taken away by maintenance operations like read miss fills, dirty evictions, etc..

Let us denote the actual volume of accesses served by the bandwidth sources (including maintenance accesses) by $\tilde{A}_1, \tilde{A}_2, \ldots, \tilde{A}_n$, where $\tilde{A}_i \geq A_i \; \forall \; i$. We need to maximize $\min(B_1/\tilde{f}_1, B_2/\tilde{f}_2, \ldots, B_n/\tilde{f}_n)$ subject to $\sum_{i=1}^{n} \tilde{f}_i = C \geq 1$, where $\tilde{f}_i = \tilde{A}_i/A$ and $\sum_{i=0}^{n} \tilde{A}_i = CA$. In other words, C is the overall access volume inflation factor due to additional maintenance accesses. In this more general case also, it can be shown that the access partitioning algorithm that decides $\tilde{f}_i$ dynamically $\forall \; i$ must attempt to converge on

$$B_1/\tilde{f}_1 = B_2/\tilde{f}_2 = \cdots = B_n/\tilde{f}_n. \quad (4)$$

The maximum delivered bandwidth in this case is $\sum_{i=0}^{n} B_i/C$. Therefore, to maximize system bandwidth, we not only need to partition accesses appropriately according to Equation 4, but also need to reduce maintenance overhead (C).

To summarize, in order to improve the performance of memory-side caches, we need to 1) partition total accesses (including maintenance accesses) between the memory-side cache

and the main memory in the ratio of their bandwidths and 2) reduce maintenance overheads. Guided by Equation 4, we now explore dynamic access partitioning (DAP) to achieve optimal system bandwidth.

IV. Dynamic Access Partitioning

The goal of DAP is to detect execution phases where the cache bandwidth is a bottleneck and direct some of the traffic from the cache to the main memory while satisfying Equation 4. To achieve this, DAP first needs to learn the system bandwidth profile and find out whether partitioning is needed. In phases where the main memory is a bottleneck (low memory-side cache hit rate) or the bandwidth demand from the cores is low, DAP should not partition any accesses. The second case is important because needless partitioning when the demand is lower than the cache bandwidth would incur the higher latency of the main memory although the delivered bandwidth remains the same. This may result in performance degradation.

To learn the temporal behavior of bandwidth demand, the execution is divided into windows of length W CPU cycles. DAP observes the bandwidth demand to the memory-side cache and the main memory in window N. If the number of accesses to the memory-side cache is higher than what the cache bandwidth can serve, partitioning is invoked in window $N+1$ based on the estimates of bandwidth demand collected in the previous window. The length of a window cannot be too large as the bandwidth profile may change from window N to window $N+1$. Likewise if W is too small, spurious bursts in bandwidth may be learned as phases of high bandwidth demand.

Once DAP has learned that partitioning is needed, it uses four different techniques to maximize the impact.

(i) **Fill Write Bypass (FWB)** drops incoming read miss fills to the memory-side cache to reduce the bandwidth demand. While these drops have no immediate impact on the main memory bandwidth, they may reduce the hit rate of the cache in future. Note that fill write bypass is different from general cache bypassing that targets potentially dead blocks in order to improve hit rates [2], [5], [10], [11], [12], [28], [29], [30], [48].

(ii) **Write Bypass (WB)** steers a fraction of incoming L3 cache dirty evictions to the main memory instead of writing to the memory-side cache. In the baseline, these would have been written to the memory-side cache. If the cache block is present in the memory-side cache, it needs to be invalidated when write bypassing is enabled for the block. Unlike fill write bypass, write bypass needs main memory bandwidth as the bypassed blocks have to be written to the main memory.

(iii) **Informed Forced Read Miss (IFRM)** forces clean hits in the cache to be served out of the main memory. This alleviates pressure at the memory-side cache but forces reads to be served at a higher memory latency. However, as our results indicate in the later sections, the overall bandwidth improvement significantly reduces the queuing latency for hits and improves performance. Like write bypass, IFRM also contributes to additional traffic to the main memory.

(iv) **Speculative Forced Read Miss (SFRM)** is similar to IFRM, except that this is done speculatively even before the state of the read hit is known. SFRM is applicable to architectures where the metadata is fetched from the memory-side cache itself. SFRM may end up bypassing hits to modified cache lines and hence, may need to be re-issued, thereby wasting bandwidth.

At the beginning of a window, DAP calculates the number of FWB, WB, IFRM, and SFRM that it needs to do in the current window to achieve optimal partitioning. DAP stores these estimates in four separate credit counters. A credit counter is incremented by the computed solution of the corresponding technique at the beginning of a window. During the rest of the window, each application of a technique decrements the corresponding credit counter. A technique can be applied as long as it has non-zero credits. All credit counters are saturating.

We now show how the credit counter values in a given window are calculated for three different implementations of the memory-side cache.

A. Algorithm for Sectored DRAM Cache

The systems with die-stacked HBM DRAM caches have two bandwidth sources beyond the on-chip SRAM cache hierarchy. These are the DRAM cache (memory-side L4 cache) and the DDR main memory.

For a given window N, DAP observes the number of accesses that need to be sent to the memory-side cache (denoted by $A_{MS\$}$) and those that need to be sent to main memory (A_{MM}). Read hits, dirty L3 cache evictions (L4 cache writes), reads for dirty evictions from L4 cache, fill writes, and maintenance operations like metadata fetch and update contribute to the memory-side cache accesses ($A_{MS\$}$). The main memory accesses (A_{MM}) are composed of read misses and dirty evictions from the memory-side cache. If $B_{MS\$}$ is the effective bandwidth of the memory-side cache (expressed in accesses per cycle), then the number of accesses that it is capable of serving in a window of W CPU cycles is equal to $B_{MS\$}.W$. Similarly, the main memory can serve $B_{MM}.W$ accesses. If the number of accesses demanded from the cache in a window is greater than what it can supply ($A_{MS\$} > B_{MS\$}.W$), then DAP invokes access partitioning. DAP first invokes fill write bypass because fill bypasses do not need any immediate main memory bandwidth. After that it invokes write bypass as the higher latency of main memory for bypassed writes does not affect performance. Forced read misses suffer from higher memory latency for reads, and hence, IFRM and SFRM are attempted last.

Fill Write Bypass (FWB). If the number of fill write bypasses is denoted by N_{FWB}, using Equation 4 we get

$$\frac{B_{MS\$}}{A_{MS\$} - N_{FWB}} = \frac{B_{MM}}{A_{MM}}. \tag{5}$$

If we denote the constant $B_{MS\$}/B_{MM}$ by K, the formula can be simplified to

$$N_{FWB} = A_{MS\$} - K.A_{MM}. \tag{6}$$

Ideally, the denominator on the right-hand side of Equation 5 would be $(A_{MM} + \delta.N_{FWB})$ where δ represents the fraction of N_{FWB} that causes additional memory-side cache misses at a future time. We do not include this term for two reasons. First, we are interested in computing the optimal bandwidth partition for the present time-window, which is too short to experience any additional memory-side cache misses due to fill write bypass. Including the $\delta.N_{FWB}$ term would lead to a sub-optimal partition of the accesses in the current window. Second,

in a future window if the memory-side cache misses increase in volume, A_{MM} would automatically increase leading to a drop in N_{FWB}, which is computed as $A_{MS\$} - K.A_{MM}$.

We note that the maximum partitioning that is needed is $A_{MS\$} - B_{MS\$}.W$, which is basically the extent by which the request demand on the memory-side cache exceeds what it can supply in a window of length W. If the value of N_{FWB} is greater than it, we cap its value to this maximum value. Also, N_{FWB} cannot be higher than the actual number of read miss fills. If the number of read misses observed is lower than N_{FWB}, we cap N_{FWB} to the number of read misses. This means that fill bypass is insufficient to reduce all the pressure on the memory-side cache and other steps are needed. We, hence, attempt write bypasses next. If N_{FWB} is negative, it means that the main memory is a bottleneck and we need to exit partitioning. This is done for WB, IFRM and SFRM also.

Write Bypass (WB). We reduce the number of accesses to the memory-side cache by N_{FWB} i.e., $A_{MS\$} = A_{MS\$} - N_{FWB}$, and then calculate additionally how many write bypasses (denoted by N_{WB}) are needed. If we reduce the number of writes to the cache by N_{WB}, we need to increase the number of writes to the main memory by the same amount.

$$\frac{B_{MS\$}}{A_{MS\$} - N_{WB}} = \frac{B_{MM}}{A_{MM} + N_{WB}} \quad (7)$$
$$\text{or}, (K+1)N_{WB} = A_{MS\$} - K.A_{MM}$$

N_{WB} cannot exceed the number of writes. If it does, it means that the memory-side cache is still a bottleneck and, hence, we will attempt to do forced read misses. The value of N_{WB} will be capped to the number of writes. Equation 7 avoids a costly division by storing $(K+1)N_{WB}$. This makes the implementation of this counter simple in hardware.

Informed Forced Read Miss (IFRM). The IFRM policy bypasses read hits to clean lines. We reduce the number of accesses to the memory-side cache further by the number of write bypasses ($A_{MS\$} = A_{MS\$} - N_{WB}$) and increase the number of accesses to the main memory by the number of write bypasses ($A_{MM} = A_{MM} + N_{WB}$). Next, we calculate the target number of IFRM, N_{IFRM}.

$$\frac{B_{MS\$}}{A_{MS\$} - N_{IFRM}} = \frac{B_{MM}}{A_{MM} + N_{IFRM}} \quad (8)$$
$$\text{or}, (K+1)N_{IFRM} = A_{MS\$} - K.A_{MM}$$

The value of N_{IFRM} can be at most the number of clean hits observed. Our implementation of the IFRM policy does not distinguish between the latency-sensitive and the latency-insensitive threads. A thread-aware IFRM policy would prioritize the clean hits of the latency-insensitive threads before the latency-sensitive ones for bypassing to the main memory.

Speculative Forced Read Miss (SFRM). The SFRM policy is invoked for a read access to the memory-side cache even before it is known whether the read will hit the cache, provided there is excess main memory bandwidth available ($A_{MM} < B_{MM}.W$). This policy sends the read access to the main memory while a traditional metadata fetch to the memory-side cache is happening. If the metadata lookup reveals that the block is absent in the memory-side cache or is resident in clean state, the response from the main memory is forwarded to the CPU. If the block is found in the memory-side cache in dirty state, a data read access is enqueued to the memory-side cache and the response from the main memory is dropped. The SFRM policy saves lookup latency for architectures with off-die tags by trading the main memory bandwidth. The SFRM policy is also useful in memory-side caches with on-die SRAM tag caches because it can significantly reduce the lookup latency in the case of a tag cache miss. DAP sets the number of speculative forced read misses, N_{SFRM}, to $0.8|B_{MM}.W - A_{MM}|$ and leaves the remaining 20% main memory bandwidth for handling any bandwidth emergency. A_{MM} used here is assumed to be adjusted for additional traffic arising from WB and IFRM. The complete flow of the algorithm is shown in Figure 3.

Hardware Overheads: As is evident from Figure 3, we need counters for number of read misses, writes, clean hits, N_{FWB}, $(K+1)N_{WB}$, $(K+1)N_{IFRM}$, and N_{SFRM}. We also need four credit counters to execute each of the schemes. The ratio of memory-side cache to main memory bandwidth, K, can be a fraction. For example, for $B_{MS\$} = 102.4$, and $B_{MM} = 38.4$, we have $K = 8/3$. We approximate K as 11/4 in this case so that multiplication by K can be easily done in hardware. If we restrict the maximum value of N_{WB} to 63 and assume K to be 11/4, the maximum value of $(K+1)N_{WB}$ can be contained within an eight-bit counter. Therefore, the credit counters are of eight-bit width. Based on these estimates, we can calculate the total storage overhead to be only about sixteen bytes. Also, all the calculations are off the critical path of the memory-side cache controller and can be easily implemented.

B. Algorithm for Alloy Cache

The Alloy cache fetches a TAD from the DRAM cache before it can do a read or a write. This leads to a significant bandwidth bloat [4]. Since most of the writes coming from the last-level SRAM cache are expected to hit in the much larger DRAM cache, it is helpful to maintain a bit with each block resident in the last-level SRAM cache indicating its presence/absence in the Alloy cache. This bit obviates the need to fetch the TAD for a dirty block. This optimization was proposed in BEAR [4] and we also use it in our design.

Since the tag and state are stored along with the data in Alloy cache, we cannot do write bypasses on hits as that would take away Alloy cache bandwidth to invalidate the cache-line. Similarly, fill bypasses need Alloy cache bandwidth to fetch the TAD in order to determine if a fill would be needed. However, to do a forced miss (IFRM), we only need to make sure that the location of the accessed block is not dirty. To be able to know this without fetching the corresponding TAD, we maintain a dirty bit cache (DBC) in SRAM. Each entry of the DBC maintains the dirty bits of a stretch of 64 consecutive Alloy cache sets (recall that Alloy cache is direct-mapped). The entry also maintains the group id of the stretch (the Alloy cache sets are divided into groups of 64 consecutive sets). We organize the DBC to have 32K entries (of size twelve bytes each) and four ways. We borrow one way of the L3 cache to accommodate the DBC (the baseline Alloy cache has 16 ways in the L3 cache). The DBC lookup latency is five cycles. We note that a previous study has explored a decoupled organization of the dirty bits for the SRAM last-level cache where each dirty bit array entry maintains a vector of dirty bits corresponding to the blocks in some main memory DRAM row [41]. In contrast, our DBC organization

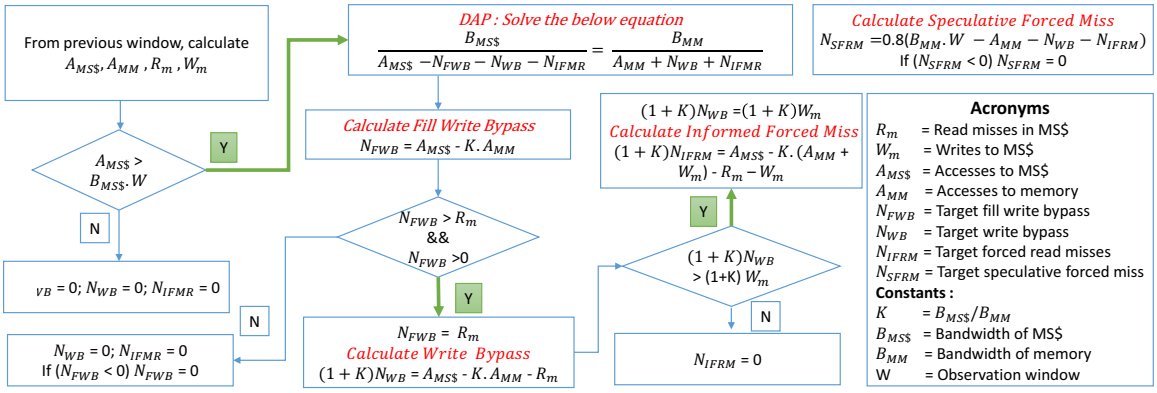

Fig. 3. Flow of the proposed DAP algorithm for sectored DRAM caches.

is designed to augment the DRAM caches exercising fine-grain allocation units (such as the Alloy cache) and helps improve bandwidth utilization in such architectures as discussed below.

A read access looks up the DBC and on a hit, we get the dirty bit of the accessed set. If the dirty bit is reset, the read access can be considered for IFRM provided the IFRM credits are not exhausted in the current window. It is important to note that a DBC hit/miss does not correspond to an Alloy cache hit/miss. To carry out IFRM, it is enough to know if the block resident in the target set is dirty or not. We should also note that if the IFRM line was actually not present in the Alloy cache, the corresponding fill would also not happen (fill bypass). N_{IFRM} is determined using Equation 8. To maintain enough number of clean blocks in the Alloy cache so that IFRM can be invoked sufficiently often, we opportunistically use the residual main memory bandwidth to do write-through for a fraction of writes coming to the Alloy cache. After adjusting for IFRM, if $B_{MM}.W - A_{MM}$ is positive, our algorithm uses $0.8|B_{MM}.W - A_{MM}|$ as the write-through count in a window.

C. Algorithm for eDRAM Cache

The systems with sectored eDRAM caches have three bandwidth sources beyond the on-chip SRAM cache hierarchy. These are the independent read and write channels of the memory-side cache (eDRAM cache) and the DDR main memory channels. Let the peak bandwidth of these three sources be $B_{MS\$-R}$, $B_{MS\$-W}$, and B_{MM}, respectively. We will assume that $B_{MS\$-R} = B_{MS\$-W} = B_{MS\$}$ and $K = B_{MS\$}/B_{MM}$. Since the metadata of the eDRAM cache is maintained on die, there is no need of SFRM. To apply the FWB, WB, and IFRM techniques, we consider three relevant scenarios discussed below.

(i) $A_{MS\$-R} > B_{MS\$-R}.W$ but $A_{MS\$-W} < B_{MS\$-W}.W$ (only read bandwidth shortage). DAP applies the IFRM technique only by solving

$$\frac{B_{MS\$-R}}{A_{MS\$-R} - N_{IFRM}} = \frac{B_{MM}}{A_{MM} + N_{IFRM}}. \quad (9)$$

(ii) $A_{MS\$-W} > B_{MS\$-W}.W$ but $A_{MS\$-R} < B_{MS\$-R}.W$ (only write bandwidth shortage). DAP applies the FWB and WB techniques. To apply fill write bypass, it solves

$$\frac{B_{MS\$-W}}{A_{MS\$-W} - N_{FWB}} = \frac{B_{MM}}{A_{MM}}. \quad (10)$$

After adjusting the accesses, it then solves for write bypass:

$$\frac{B_{MS\$-W}}{A_{MS\$-W} - N_{FWB} - N_{WB}} = \frac{B_{MM}}{A_{MM} + N_{WB}}. \quad (11)$$

(iii) $A_{MS\$-R} > B_{MS\$-R}.W$ and $A_{MS\$-W} > B_{MS\$-W}.W$ (both read and write bandwidth shortage). DAP first computes N_{FWB} using Equation 10. Next, it simultaneously solves for N_{WB} and N_{IFRM} by considering the following equations.

$$\frac{B_{MS\$-W}}{A_{MS\$-W} - N_{FWB} - N_{WB}} = \frac{B_{MS\$-R}}{A_{MS\$-R} - N_{IFRM}} = \frac{B_{MM}}{A_{MM} + N_{WB} + N_{IFRM}} \quad (12)$$

These lead to $(2K+1)N_{WB} = ((K+1)(A_{MS\$-W} - N_{FWB}) - K.A_{MS\$-R} - K.A_{MM})$ and $(2K+1)N_{IFRM} = ((K+1)A_{MS\$-R} - K.(A_{MS\$-W} - N_{FWB}) - K.A_{MM})$. The implementation of the techniques makes use of the credit counters, as already discussed.

V. Evaluation Methodology

For our simulations, we model eight dynamically scheduled x86 cores with an in-house modified version of the Multi2Sim simulator [49]. Each core is four-wide with 224 ROB entries and clocked at 4 GHz. The core microarchitecture parameters are taken from the Intel Skylake processor [7]. The load/store queues and the memory request buffers inside the core are scaled up appropriately so that they can operate with a 224-entry ROB without becoming a bottleneck to core bandwidth demand. Our preliminary experiments with streaming microbenchmarks confirm that our core configuration can demand the peak total bandwidth of the memory-side cache and the main memory. Each core has 32 KB, 8-way L1 instruction and data caches with a latency of three cycles and a private 256 KB 8-way L2 cache with a round-trip latency of eleven cycles. The cores share an 8 MB 16-way inclusive L3 cache with round-trip latency of twenty cycles. Each core is equipped with an aggressive multi-stream stride prefetcher that prefetches into the L2 and L3 caches. The main memory DRAM model includes two DDR4-2400 channels (total bandwidth 38.4 GB/s), two ranks per channel, eight banks per rank, and burst length of eight. Each bank has a 2 KB row buffer and 15-15-15-39 (tCAS-tRCD-tRP-tRAS) timing parameters. An additional ten-cycle I/O delay (at 1.2 GHz) is charged for each access to account for floorplan,

board delays, etc.. Writes are scheduled in batches to reduce channel turn-arounds.

We select seventeen applications from the SPEC CPU 2006, HPCG [20], and Parboil [44] benchmark suites based on their L3 cache MPKI. For each application, we select a snippet of one billion dynamic instructions. To understand the bandwidth sensitivity of these application snippets, we simulate them in rate-8 mode (eight copies of an application snippet run on eight cores) and check the performance impact of doubling the bandwidth of a 4 GB die-stacked sectored DRAM cache from 102.4 GB/s to 204.8 GB/s (Figure 4). The top panel of Figure 4 shows that twelve of these seventeen snippets are bandwidth-sensitive. The bottom panel shows that the average L3 cache MPKI of the bandwidth-sensitive workloads is 20.4, while that of the bandwidth-insensitive workloads is 11.6.

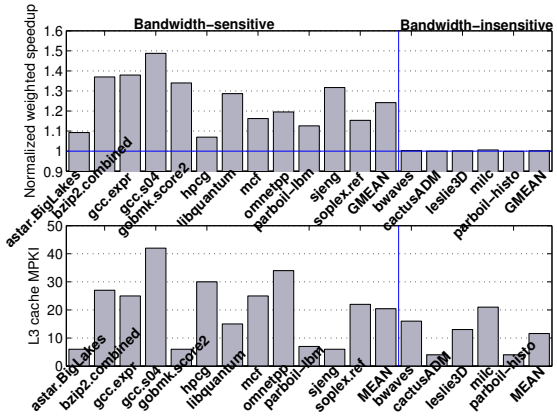

Fig. 4. Top panel: Weighted speedup achieved when DRAM cache bandwidth is doubled. Bottom panel: L3 cache MPKI.

In addition to these seventeen homogeneous multi-programmed mixes, we prepare 27 eight-way heterogeneous multi-programmed mixes by combining the seventeen application snippets. These mixes are prepared carefully so that roughly half of these mixes have application snippets with similar bandwidth-sensitivity, while the rest have application snippets with dissimilar bandwidth-sensitivity. In all these 44 multi-programmed mixes, each thread is simulated for one billion dynamic instructions. Threads that finish early continue to run.

VI. Simulation Results

In this section, we evaluate our proposal DAP on sectored DRAM cache (Section VI-A), Alloy cache (Section VI-B), and sectored eDRAM cache (Section VI-C).

A. Sectored DRAM Cache

We model a four-way set-associative sectored DRAM cache with 4 KB sectors. The DRAM cache uses the single-bit not-recently-used (NRU) policy for replacement and stores the NRU replacement states in on-die SRAM. The cache is equipped with a footprint prefetcher [26]. The DRAM cache array parameters are drawn from the JEDEC HBM standard [24]. Our default set of parameters corresponds to a 4 GB cache clocked at 800 MHz having four 128-bit DDR channels with a burst length of four (aggregate bandwidth of 102.4 GB/s). Each channel has a single rank and 16 banks per rank. Each bank has a 2 KB row buffer and 10-10-10-26 timing parameters. Writes are scheduled in batches to reduce channel turn-around. We also evaluate 2 GB and 8 GB capacity points as well as 128 GB/s and 204.8 GB/s bandwidth points. For simulating 128 GB/s bandwidth, we increase the frequency to 1 GHz and the latency parameters are scaled up to 12-12-12-32. The 204.8 GB/s bandwidth point is simulated by modeling eight channels and frequency of 800 MHz. We note that the main memory latency is almost same as the memory-side cache latency if the additional ten-cycle I/O delay of the main memory is excluded.

1) An Optimized Baseline: We improve the baseline by modeling a 32K-entry four-way set-associative SRAM tag cache [4], [19], [36], [50]. While addition of a tag cache is not essential for our proposal, it significantly reduces the bandwidth overhead arising from the metadata accesses in the baseline. Without a tag cache, the baseline is sub-optimal in bandwidth delivery and access partitioning algorithms will show much higher performance sensitivity. Overall, the tag cache size is 32K$\times$156 bits or 624 KB. We take one way of the L3 cache to accommodate the tag cache. We conservatively charge five CPU cycles to model the part of the tag cache lookup latency that cannot be overlapped with the L3 cache lookup.

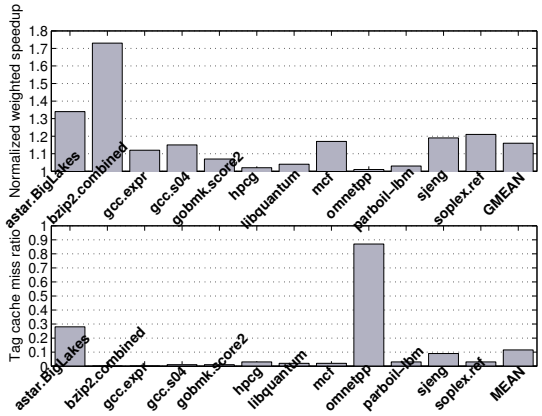

Fig. 5. Top panel: Weighted speedup achieved with a 32K-entry tag cache. Bottom panel: Miss rate of a 32K-entry tag cache.

Figure 5 quantifies the weighted speedup achieved due to inclusion of the tag cache for the twelve bandwidth sensitive applications run in rate-8 mode (top panel). The bottom panel of Figure 5 quantifies the tag cache miss rate. The top panel shows that a vast majority of the workloads benefit from the tag cache leading to an average 16% improvement in performance. The relatively high tag cache miss rates of astar.BigLakes and omnetpp result from poor sector utilization in these two applications leading to low temporal utility of the tag cache entries. For all subsequent results on sectored DRAM caches, we will assume an optimized baseline that has a tag cache.

2) Evaluation of DAP: We first analyze the results for the twelve bandwidth-sensitive workloads in the context of an eight-core system having a 4 GB die-stacked HBM DRAM cache with 4 KB sectors and 102.4 GB/s bandwidth. The default value of window size, W, is 64. We assume the bandwidth efficiency of all sources to be 0.75 of the peak. The effective bandwidth of a source is always less than the peak bandwidth due to several inefficiencies such as non-zero row buffer miss rates, less than ideal access schedulers, and write-induced interference

and channel turn-around (this inefficiency is not applicable to eDRAM caches that have separate read and write channels).

The top panel of Figure 6 quantifies the weighted speedup achieved by our dynamic access partitioning (DAP) proposal normalized to the baseline. The performance profile of DAP varies from a 1% loss (parboil-lbm) to a 2× gain (omnetpp). Several workloads gain at least 10%. The average speedup achieved by DAP for the twelve bandwidth-sensitive workloads is 15.2%. The bottom panel of Figure 6 shows the average L3 cache read miss latency of DAP normalized to baseline. The savings in the average L3 cache read miss latency vary from 0% (parboil-lbm) to 49% (omnetpp); the average saving is 18%. The speedup figures correlate well with the savings in the average L3 cache read miss latency.

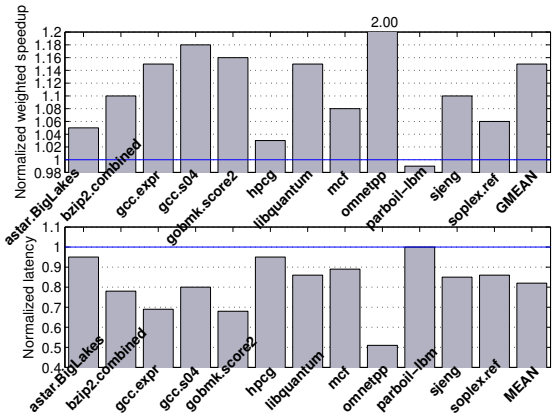

Fig. 6. Top panel: Weighted speedup achieved by DAP. Bottom panel: Normalized L3 cache read miss latency for DAP.

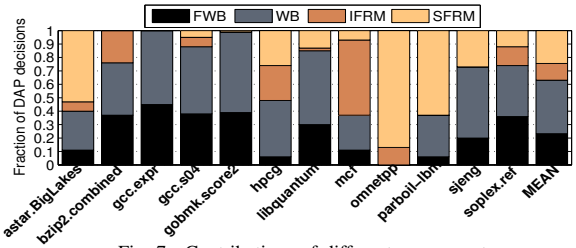

Fig. 7. Contributions of different components.

Figure 7 shows the contributions of the four different components (FWB, WB, IFRM, SFRM) to all the decisions made by DAP. The FWB and WB techniques are effective across the board except in omnetpp. The gcc.expr and gobmk.score2 workloads employ only FWB and WB. The IFRM and SFRM techniques are beneficial to several workloads. Out of all DAP decisions employed in omnetpp, 87% are SFRM and the rest are IFRM. The high contribution of SFRM in omnetpp is due to a very high tag cache miss rate (see Figure 5). As a result, a major portion of the performance gain enjoyed by omnetpp arises from hiding the tag cache miss latency achieved by SFRM. On average, the contributions of FWB, WB, IFRM, and SFRM to all DAP decisions for these twelve workloads are 23%, 40%, 12%, and 25%.

To understand how well our proposal is able to partition the accesses, the top panel of Figure 8 shows the number of CAS operations done by the main memory as a fraction of the

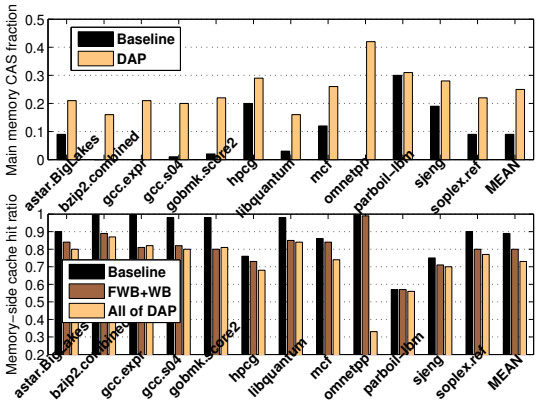

Fig. 8. Top panel: The fraction of main memory CAS operations out of all CAS operations. Bottom panel: DRAM cache hit rate.

total number of CAS operations done in the main memory and the memory-side cache. This fraction, according to Equation 4, should be $B_{MM}/(B_{MM} + B_{MS\$})$ i.e., 0.27 under optimal access partitioning. On average, in the baseline system, 9% of all CAS operations are served by the main memory, while with our DAP proposal this fraction increases to 25%. It is encouraging to see that this is close to the optimal fraction of 0.27. In fact, except for omnetpp, this fraction for DAP is between 16% and 31% for all the workloads. For omnetpp, this fraction is 42% due to high contribution of SFRM. We also note that for parboil-lbm the baseline system has a main memory CAS fraction of 30%, which is already close to the optimal main memory CAS fraction. As a result, our proposal is unable to improve the performance of this workload any further.

The bottom panel of Figure 8 quantifies the memory-side cache hit rate (read and write hits combined). For baseline, the average hit rate is 89%. When the FWB and WB techniques are incorporated, the average hit rate drops to 80%. Further introduction of IFRM and SFRM causes the average hit rate to drop to 73%. In omnetpp, the IFRM and SFRM techniques lower the memory-side cache hit rate from 99% to 33%. Another application that experiences 10% drop in memory-side cache hit rate due to IFRM and SFRM is mcf (84% to 74% hit rate). Referring back to Figure 7, we see that for mcf out of all DAP decisions, 56% are IFRM. Now, we turn to analyze our proposal in more detail.

3) Sensitivity Studies: In the following, we evaluate the sensitivity of DAP performance to the algorithm parameters, main memory technology, and DRAM cache capacity and bandwidth.

Sensitivity to Algorithm Parameters. By default, our evaluation uses a window size (W) of 64 and a bandwidth efficiency (E) of 0.75 for all the sources. Table I shows the normalized weighted speedup achieved by our DAP proposal as these two parameters are varied for the twelve bandwidth-sensitive workloads. We observe that the default parameter set, $W = 64$ and $E = 0.75$, offers the best performance among the parameter values explored. It is important to note that the 100% bandwidth efficiency point delivers the worst performance among the three efficiency points considered. This is because at 100% bandwidth efficiency, the DRAM cache delivers the highest bandwidth resulting in less partitioning from DAP. This lowers the benefit that can be achieved by DAP.

TABLE I
NORMALIZED WEIGHTED SPEEDUP

Attribute	Variation in W ($E = 0.75$)			Variation in E ($W = 64$)		
Param. value	32	64	128	0.50	0.75	1.00
Speedup	1.13	1.15	1.14	1.14	1.15	1.12

Sensitivity to Main Memory Technology. Figure 9 explores how the normalized weighted speedup achieved by our proposal gets affected by the main memory latency and bandwidth. The first three bars in each group show the sensitivity to latency. The leftmost bar corresponds to the default main memory configuration (Section V). The second bar shows the impact when all additional board and I/O latencies are removed from the main memory model. The third bar shows the effect of using a higher-latency quad-channel (32-bit channels with burst length 16) LPDDR4-2400 24-24-24-53 main memory model (same bandwidth as default, but nearly 70% higher row hit latency). From these results we observe that the benefit of DAP typically decreases with higher main memory latency. On average, removing I/O latency improves the benefit from 15.2% to 16%, while introduction of a slower LPDDR4 module lowers the benefit to 8%. Such a trend is expected because the accesses steered to the main memory will incur longer latency, thereby lowering the overall benefit. Only `bzip2.combined` benefits from the higher cross-channel parallelism in the quad-channel LPDDR4 system.

The rightmost bar in each group of Figure 9 shows the effect of using a higher-bandwidth (51.2 GB/s) dual-channel DDR4-3200 module with the same latency as the default DDR4-2400 module. A comparison of the leftmost and the rightmost bars shows that higher memory bandwidth helps improve the benefit of dynamic access partitioning across the board. This is an expected trend because a higher main memory bandwidth shifts the optimal bandwidth partitioning point more toward main memory leading to a higher fraction of accesses being steered to the main memory (see Equation 4).

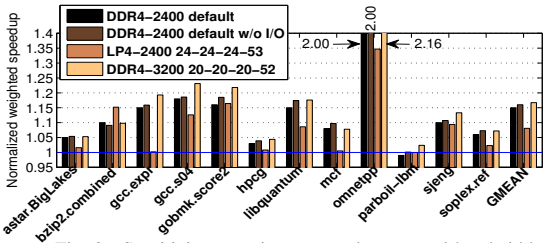

Fig. 9. Sensitivity to main memory latency and bandwidth.

Sensitivity to DRAM Cache Capacity and Bandwidth. The top panel of Figure 10 shows the normalized weighted speedup achieved by our proposal as the memory-side cache capacity is varied from 2 GB to 8 GB while the memory-side cache bandwidth is held constant at the default value of 102.4 GB/s. The main memory module uses the default DDR4-2400 parts. With increasing memory-side cache capacity, our dynamic access partitioning proposal gains in importance. This is expected because as the memory-side cache grows in size, it can serve more accesses in the baseline system leading to a larger departure from the optimal operating point. The bottom panel of Figure 10 shows the normalized weighted speedup achieved

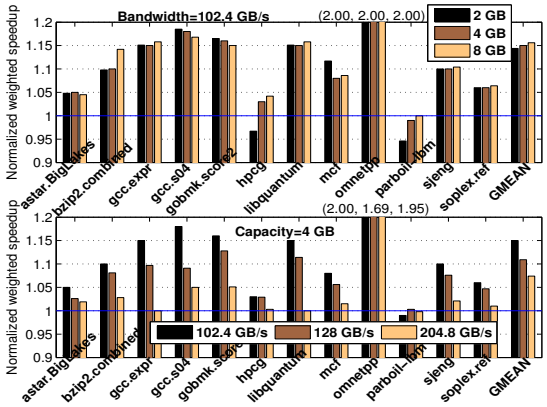

Fig. 10. Top panel: Impact of the memory-side cache capacity. Bottom panel: Impact of the memory-side cache bandwidth.

by our proposal as the memory-side cache bandwidth is varied from 102.4 GB/s to 204.8 GB/s holding the memory-side cache capacity constant at 4 GB. With increasing cache bandwidth, the speedup drops significantly (from average 15.2% with 102.4 GB/s to 7% with 204.8 GB/s). With increasing memory-side cache bandwidth, the optimal bandwidth partitioning decision steers a bigger fraction of accesses to the memory-side cache (see Equation 4) making the baseline closer to the optimal point.

4) Comparison to Related Proposals: Two existing proposals have explored access partitioning for systems with die-stacked DRAM caches. The first proposal, named self-balancing dispatch (SBD), attempts to steer accesses to the cache or main memory that has the lowest expected latency [43]. The expected latency is estimated by taking into consideration the waiting queue length and the average latency to serve an access at that source. To avoid steering accesses to the main memory for blocks that are dirty in the DRAM cache, the proposal identifies the highly written to pages with the help of a bank of counting Bloom filters and stores them in a Dirty List. The remaining pages are operated in write-through mode. An access that finds its page in the Dirty List is always steered to the DRAM cache. For the remaining accesses, if a predictor predicts an access to be a DRAM cache hit, the access is steered to the bandwidth source with the lowest expected service latency.

The second related proposal, named bandwidth-aware tiered-memory management (BATMAN), steers accesses to the main memory so that the die-stacked DRAM cache operates at a target hit rate dictated by the ratio of the bandwidths at the sources [3]. When the DRAM cache operates at a hit rate higher than the target hit rate, the proposal starts disabling a fraction of the DRAM cache sets to achieve the target hit rate. Thus by modulating hit rates, this proposal tries to steer a fraction of the requests to the main memory.

Both these proposals attempt to utilize the spare bandwidth of the main memory. However, unlike DAP that dynamically calculates and tries to achieve optimal partitioning on small windows of time, these proposals attempt to achieve only some amount of partitioning, which is generally far from optimal and can sometimes lead to losses. This is evident from Figure 11 which compares the performance achieved by SBD, BATMAN,

and our proposal (DAP) normalized to the baseline for the twelve bandwidth-sensitive workloads.

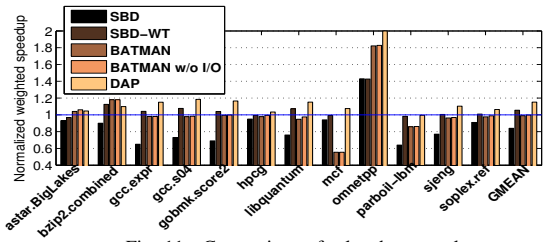

Fig. 11. Comparison of related proposals.

SBD degrades baseline performance by 16% on average, while BATMAN performs within 1% of the baseline. We find that both these proposals suffer from the congestion arising from the need to write out dirty blocks from the memory-side cache. In SBD, when a 4 KB page falls out of the Dirty List, it must be cleaned. This requires reading out the dirty blocks and writing them back to main memory. The original SBD proposal was evaluated on smaller memory-side caches where a large volume of cache evictions would reduce the necessity of such forced write-outs. In BATMAN, when a memory-side cache set is disabled, the dirty blocks from the set must be read out and written to main memory. We consider a variant of SBD, named SBD-WT, which disables forced writing out of dirty blocks and relies on write-through for pages with low volume of writes for creating enough clean blocks. SBD-WT improves performance by 5.5% on average compared to the baseline, but still falls significantly short of DAP.

BATMAN suffers from three shortcomings. First, in a large cache, it is often the case that the disabled sets do not intersect with the cache region in use in the current phase. Hence, a lot of unrelated sets may have to be disabled to reach the cache region of interest. This leads to significant overheads. Further, for such a scheme to be effective over small time windows, it is necessary that the access distribution be homogeneous over the cache sets. For large caches, this is true only over large time windows. In contrast, our proposal can affect optimal access partitioning within time windows as small as 64 CPU cycles. Second, if the hit rate of a workload fluctuates frequently, it may not be possible to recover the lost hits quickly by opening the disabled sets; it takes time to warm up the cold sets. We observe this phenomenon in mcf, which experiences very poor performance with BATMAN. Third, BATMAN would end up doing access partitioning based on the memory-side cache hit rate even when there is no shortage of bandwidth in the memory-side cache. To understand this problem, we run BATMAN with no I/O latency added to main memory. As can be seen, several workloads improve by up to 5% (1% on average). This problem would be worse if the main memory has much higher latency (like LPDDR4) or the workloads are latency-sensitive. Both SBD and BATMAN deliver excellent performance for omnetpp. Since this workload suffers from a high tag cache miss rate, steering a subset of accesses to the main memory always saves latency for this workload. Overall, the performance of BATMAN that we observe differs from what was reported originally [3] because the original proposal was evaluated on top of Alloy cache which has significant bandwidth inefficiencies arising from metadata traffic [4].

5) Performance Scaling: In the following, we evaluate DAP on a larger set of workloads and larger core counts.

Performance on Larger Set of Workloads. Figure 12 evaluates our proposal on the entire set of 44 multi-programmed workloads running on an eight-core system. The workloads are classified into bandwidth-sensitive (twelve homogeneous rate mixes), bandwidth-insensitive (five homogeneous rate mixes), and heterogeneous (27 mixes). Within each category, the workloads are sorted in the increasing order of speedup. None of the bandwidth-insensitive workloads suffer from any performance loss, as DAP seldom invokes partitioning for these workloads. The performance improvement of the heterogeneous workloads varies from 4% to 72%. Overall, our dynamic access partitioning proposal achieves a 13% speedup averaged over all 44 workloads.

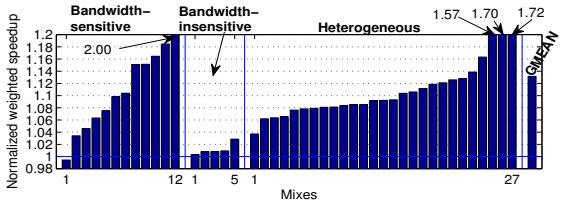

Fig. 12. Normalized weighted speedup.

Scaling to Larger Core Counts. We scale up our simulated system to support 16 cores. The shared L3 cache is scaled to 16 MB keeping the associativity unchanged at sixteen. The memory-side cache considered in such a system has an 8 GB capacity and 204.8 GB/s bandwidth. The main memory is dual-channel DDR4-3200 20-20-20-52, offering an aggregate bandwidth of 51.2 GB/s. Figure 13 quantifies the speedup achieved by DAP for the twelve bandwidth-sensitive workloads, run in sixteen-way rate mode. Our proposal improves performance by average 14.6% in a sixteen-core system.

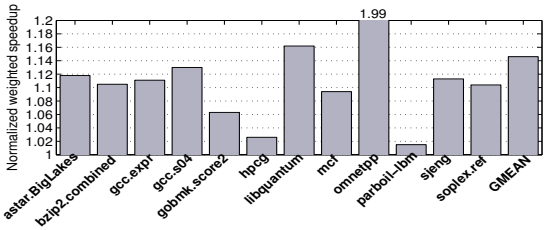

Fig. 13. Normalized weighted speedup achieved by DAP on a 16-core system.

B. Alloy Cache

The DRAM array of the Alloy cache is modeled following the JEDEC HBM standard and is identical to the sectored cache model, as discussed in Section VI-A. The Alloy cache organizes the 2 KB rows within each bank to hold the tag and data (TAD) for a stretch of consecutive sets. We model a burst length of six spread over three cycles in our simulations. The Alloy cache model employs a program counter-based DRAM cache hit/miss predictor to initiate miss handling early.

The top panel of Figure 14 presents the weighted speedup achieved by DAP for the twelve bandwidth-sensitive workloads. Also shown is the performance of BEAR [4], which attempts to reduce the bandwidth bloat of the Alloy cache. BEAR bypasses those fills that will potentially not give hits in the future. In doing so it makes sure that the hit rate of the cache does not

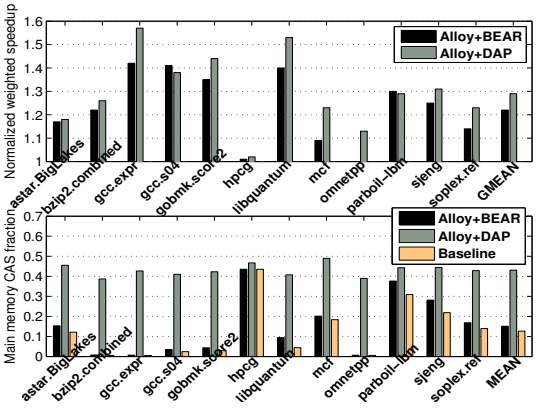

Fig. 14. Top panel: Normalized weighted speedup in a system with an Alloy cache. Bottom panel: Fraction of CAS operations served by main memory.

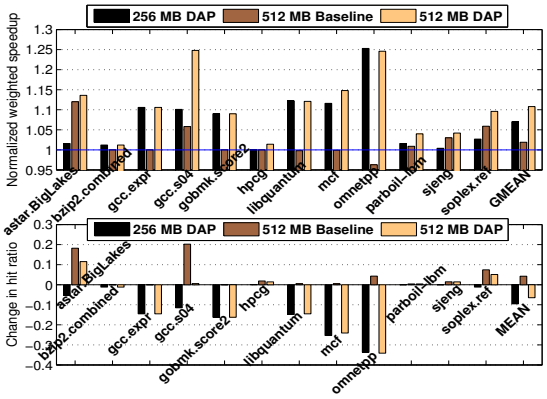

Fig. 15. Top panel: Normalized weighted speedup in a system with an eDRAM cache. Bottom panel: Change in memory-side cache hit rate relative to baseline 256 MB.

suffer. In contrast, DAP bypasses fills to balance the accesses between the Alloy cache and the main memory, in accordance with Equation 4.

On average, BEAR improves the baseline Alloy cache performance by 22%, while our proposal achieves a 29% speedup. The bottom panel of Figure 14 quantifies the number of CAS operations served by the main memory as a fraction of the total number of CAS operations served by the main memory and the Alloy cache. The average value of this fraction is 13% for the baseline, 15% for BEAR, and 43% for our proposal. The Alloy cache requires three channel cycles to transfer a TAD. Out of these three cycles, only two cycles are used to transfer the data. Therefore, $B_{MS\$}$ for Alloy cache is $\frac{2}{3} \times 102.4$ GB/s. Equation 4 dictates that at the optimal partition point, the main memory should serve 36% of the accesses. Our proposal comes close to it. Across the board, the main memory CAS fraction for our proposal varies between 39% and 49%.

C. Sectored eDRAM Cache

We model sectored eDRAM caches up to 512 MB capacity. The sector size is 1 KB and the associativity of the cache is sixteen. The metadata is maintained in on-die SRAM with an eight-cycle (at 4 GHz) lookup latency. The cache access latency is about two-third of the page hit latency of the main memory [6]. The separate read and write channels have 51.2 GB/s bandwidth each.

The top panel of Figure 15 quantifies the weighted speedup of three systems relative to the baseline system with a 256 MB eDRAM cache for the twelve bandwidth-sensitive workloads. Within each group of bars, the leftmost bar corresponds to a system that employs our proposal on a 256 MB eDRAM cache. The middle bar corresponds to the baseline with a 512 MB eDRAM cache. The rightmost bar corresponds to a system that employs our proposal on a 512 MB eDRAM cache. The bottom panel shows the change in the eDRAM cache hit rate relative to the baseline 256 MB eDRAM cache.

DAP, when applied to a 256 MB eDRAM cache, lowers the average hit rate by 9.5% relative to the baseline 256 MB eDRAM cache while improving performance by 7%. With 512 MB as baseline, average hit rate increases by 4%, but performance improves by only 2%. DAP, working with a 512 MB eDRAM cache, lowers the hit rate by 6.5% relative to the 256 MB baseline while improving performance by 11%, on average.

VII. Related Work

Recent research studies exploring the architecture of DRAM caches have focused on traditional cache organizations with fine-grain (64B/128B) [8], [9], [16], [34], [36], [39], [43], coarse-grain (512B to 4KB) [23], [25], [26], [27], [32], or mixed-grain [14] allocation units (referred to as the DRAM cache block size). There are other studies that explore a range of allocation units [52] and configurable block sizes ranging from 64 bytes to 512 bytes [35]. The studies assuming fine-grain or conventional allocation units focus their efforts on managing the large tag store and minimizing the impact of the serialization latency introduced by tag lookup. Some studies have also looked at optimizations for bandwidth delivery. BEAR is proposed to reduce some of the bandwidth overheads found in Alloy cache [4]. SBD [43] and BATMAN [3] propose heuristics to steer a fraction of requests to be served out of the main memory, in order to reduce hit latency. We discussed these schemes along with a quantitative evaluation in Section VI. The MicroRefresh proposal also observes the under-utilization of the main memory bandwidth in systems having DRAM caches [15]. This proposal explores the possibility of not refreshing a fraction of DRAM cache pages that have reuse distances larger than a dynamically determined threshold exceeding the refresh interval. The accesses to such pages are sent to the main memory. Overall, the idle main memory bandwidth is utilized to save refresh energy in the DRAM cache.

Unlike most of the recent work on memory-side caches that focus on cache organization and architecture, the focus of our proposal is on increasing the delivered bandwidth of a system with multiple bandwidth sources. The algorithms described in this paper will work on top of any memory-side cache architecture and are additive to many of the other proposed optimizations.

VIII. Summary

In this paper we have presented DAP, a holistic and analytically complete solution to the problem of maximizing delivered bandwidth in a system with multiple bandwidth sources. We

demonstrate the effectiveness of this algorithm on systems with memory-side cache and DDR main memory. We show that DAP applied to an eight-core system with DDR4-2400 main memory and 4 GB of die-stacked DRAM cache capable of delivering 102.4 GB/s improves the baseline performance by 13% on average for 44 multi-programmed workloads, while requiring only sixteen additional bytes of storage. We also show that the algorithm is robust and scales seamlessly to future memory-side caches with higher bandwidth and capacity.

REFERENCES

[1] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. IBM POWER7 Systems. In *IBM Journal of Research and Development*, May/June 2011.

[2] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *PACT* 2012.

[3] C-C. Chou, A. Jaleel, M. K. Qureshi. BATMAN: Maximizing Bandwidth Utilization for Hybrid Memory Systems. *Technical Report, TR-CARET-2015-01*, March 2015.

[4] C-C. Chou, A. Jaleel, M. K. Qureshi. BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches. In *ISCA* 2015.

[5] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *MICRO* 1999.

[6] I. Cutress. The Intel 6th Gen Skylake Review: Core i7-6700K and i5-6600K Tested. August 2015. Available at http://www.anandtech.com/show/9483/intel-skylake-review-6700k-6600k-ddr4-ddr3-ipc-6th-generation/9.

[7] I. Cutress. The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis. September 2015. Available at http://www.anandtech.com/show/9582/intel-skylake-mobile-desktop-launch-architecture-analysis/5.

[8] M. El-Nacouzi, I. Atta, M. Papadopoulou, J. Zebchuk, N. Enright-Jerger, and A. Moshovos. A Dual Grain Hit-miss Detector for Large Die-stacked DRAM Caches. In *DATE* 2013.

[9] S. Franey and M. Lipasti. Tag Tables. In *HPCA* 2015.

[10] H. Gao and C. Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[11] J. Gaur, M. Chaudhuri and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ISCA* 2011.

[12] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *ICS* 1995.

[13] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *ISCA* 1983.

[14] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *MICRO* 2014.

[15] N. D. Gulur, R. Govindarajan, and M. Mehendale. MicroRefresh: Minimizing Refresh Overhead in DRAM Caches. In *MEMSYS* 2016.

[16] F. Hameed, L. Bauer, and J. Henkel. Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies. In *CASES* 2013.

[17] F. Hamzaoglu, U. Arslan, N. Bisnik, S. Ghosh, M. B. Lal, N. Lindert, M. Meterelliyoz, R. B. Osborne, J. Park, S. Tomishima, Y. Wang, and K. Zhang. A 1 Gb 2 GHz 128 GB/s Bandwidth Embedded DRAM in 22 nm Tri-Gate CMOS Technology. In *IEEE Journal of Solid-State Circuits*, January 2015.

[18] M. D. Hill and A. J. Smith. Experimental Evaluation of On-chip Microprocessor Cache Memories. In *ISCA* 1984.

[19] C-C. Huang and V. Nagarajan. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. In *PACT* 2014.

[20] HPCG Benchmark. Available at http://hpcg-benchmark.org/.

[21] IBM Corporation. IBM POWER Systems. Available at http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf.

[22] Intel Corporation. Crystalwell products. Available at http://ark.intel.com/products/codename/51802/Crystal-Well.

[23] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee. Efficient Footprint Caching for Tagless DRAM Caches. In *HPCA* 2016.

[24] JEDEC. High Bandwidth Memory (HBM) DRAM. *Standard Documents JESD235A*, November 2015. Available at https://www.jedec.org/standards-documents/docs/jesd235.

[25] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *MICRO* 2014.

[26] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *ISCA* 2013.

[27] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms. In *HPCA* 2010.

[28] T. L. Johnson. Run-time Adaptive Cache Management. *PhD thesis*, University of Illinois, Urbana, May 1998.

[29] S. Khan, Y. Tian, and D. A. Jimènez. Dead Block Replacement and Bypass with a Sampling Predictor. In *MICRO* 2010.

[30] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE TC*, April 2008.

[31] N. Kurd, M. Chowdhury, E. Burton, T. P. Thomas, C. Mozak, B. Boswell, M. Lal, A. Deval, J. Douglas, M. Elassal, A. Nalamalpu, T. M. Wilson, M. Merten, S. Chennupaty, W. Gomes, and R. Kumar. Haswell: A Family of IA 22 nm Processors. In *ISSCC* 2014.

[32] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, J. W. Lee. A Fully Associative, Tagless DRAM Cache. In *ISCA* 2015.

[33] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. In *IBM Systems Journal* 1968.

[34] G. H. Loh and M. D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *MICRO* 2011.

[35] N. Madan, L. Zhao, N. Muralimanohar, A. N. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *HPCA* 2009.

[36] J. Meza, J. Chang, H-B. Yoon, O. Mutlu, and P. Ranganathan. Enabling Efficient and Scalable Hybrid Memories using Fine-Granularity DRAM Cache Management. In *IEEE CAL*, July-December 2012.

[37] C. R. Moore. The PowerPC 601 Microprocessor. In *IEEE COMPCON* 1993.

[38] S. A. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *ISCA* 1990.

[39] M. K. Qureshi and G. H. Loh. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *MICRO* 2012.

[40] J. B. Rothman and A. J. Smith. Sector Cache Design and Performance. In *MASCOTS* 2000.

[41] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. The Dirty-Block Index. In *ISCA* 2014.

[42] A. L. Shimpi. Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested. June 2013. Available at http://www.anandtech.com/show/6993/intel-iris-pro-5200-graphics-review-core-i74950hq-tested/3.

[43] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *MICRO* 2012.

[44] J. A. Stratton, C. Rodrigues, I-J. Sung, N. Obeid, L-W. Chang, N. Anssari, G. D. Liu, and W-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *IMPACT Technical Report IMPACT-12-01*, March 2012.

[45] J. Stuecheli. Next Generation POWER microprocessor. In *Hot Chips* 2013.

[46] K. Tran and J. Ahn. HBM: Memory Solution for High Performance Processors. In *MemCon* 2014.

[47] M. Tremblay and J. M. O'Connor. UltraSparc I: A Four-issue Processor Supporting Multimedia. In *IEEE Micro*, April 1996.

[48] G. Tyson et al. A Modified Approach to Data Cache Management. In *MICRO*, 1995.

[49] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT* 2012.

[50] H. Wang, T. Sun, and Q. Yang. CAT - Caching Address Tags: A Technique for Reducing Area Cost of On-Chip Caches. In *ISCA* 1995.

[51] D. Windheiser, E. L. Boyd, E. Hao, S. G. Abraham, and E. S. Davidson. KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver. In *IPPS* 1993.

[52] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM Cache Architectures for CMP Server Platforms. In *ICCD* 2007.