HAShCache: Heterogeneity-Aware Shared DRAMCache for Integrated Heterogeneous Systems

ADARSH PATIL and RAMASWAMY GOVINDARAJAN, Indian Institute of Science

Integrated Heterogeneous System (IHS) processors pack throughput-oriented General-Purpose Graphics Processing Units (GPGPUs) alongside latency-oriented Central Processing Units (CPUs) on the same die sharing certain resources, e.g., shared last-level cache, Network-on-Chip (NoC), and the main memory. The demands for memory accesses and other shared resources from GPU cores can exceed that of CPU cores by two to three orders of magnitude. This disparity poses significant problems in exploiting the full potential of these architectures.

In this article, we propose adding a large-capacity stacked DRAM, used as a shared last-level cache, for the IHS processors. However, adding the DRAMCache naively, leaves significant performance on the table due to the disparate demands from CPU and GPU cores for DRAMCache and memory accesses. In particular, the imbalance can significantly reduce the performance benefits that the CPU cores would have otherwise enjoyed with the introduction of the DRAMCache, necessitating a heterogeneity-aware management of this shared resource for improved performance.

In this article, we propose three simple techniques to enhance the performance of CPU application while ensuring very little to no performance impact to the GPU. Specifically, we propose (i) *PrIS*, a prioritization scheme for scheduling CPU requests at the DRAMCache controller; (ii) *ByE*, a selective and temporal bypassing scheme for CPU requests at the DRAMCache; and (iii) *Chaining*, an occupancy controlling mechanism for GPU lines in the DRAMCache through pseudo-associativity. The resulting cache, Heterogeneity-Aware Shared DRAMCache (HAShCache), is heterogeneity-aware and can adapt dynamically to address the inherent disparity of demands in an IHS architecture. Experimental evaluation of the proposed HAShCache results in an average system performance improvement of 41% over a naive DRAMCache and over 200% improvement over a baseline system with no stacked DRAMCache.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Hardware** → **3D integrated circuits**; **Dynamic memory**; • **Theory of computation** → *Caching and paging algorithms*;

Additional Key Words and Phrases: DRAM cache, 3D-stacked memory, Integrated CPU-GPU processors, cache sharing

ACM Reference format:

Adarsh Patil and Ramaswamy Govindarajan. 2017. HAShCache: Heterogeneity-Aware Shared DRAMCache for Integrated Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* 14, 4, Article 51 (December 2017), 26 pages.

<https://doi.org/10.1145/3158641>

Authors' addresses: A. Patil and R. Govindarajan, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, Karnataka 560012, India; emails: {adarsh, govind}@iisc.ac.in.

Current address: A. Patil is currently a Research Scientist at Intel Corporation, India; email: adarsh.patil@intel.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/12-ART51 \$15.00

<https://doi.org/10.1145/3158641>

1 INTRODUCTION

The remarkable advances in computing power of the modern microprocessor over the last few decades can predominantly be attributed to shrinking of feature sizes. Miniaturization of transistors has allowed the addition of specialized on-chip hardware circuitry for acceleration units. Graphics Processing Units (GPUs) have evolved from being fixed-function pipelines and are being used to accelerate data parallel code for General Purpose (GP) applications. Compared with multi-core CPUs, GPGPUs offer the potential for better performance at lower energy. Traditionally, these discrete processors have had their own independent memory systems. To take advantage of the discrete GPUs, the CPU must copy data to the GPUs memory and back. This data movement is wasteful and expends energy while also adding latency as the transfer happens over a slower PCIe bus. The separate address spaces and complex programming models that need to manage two sets of data further impede expansion of the workloads that benefit from GPGPU computing.

Modern processor chips have heterogeneous processors which integrate a lower capacity GPU on-die allowing for graphics rendering only. In view of the widespread use of the GPU for general purpose applications, processor manufacturers including AMD [4], Intel [6], and NVIDIA [3] are beginning to allow general purpose OpenCL [8]/CUDA [2] programs to run on their Integrated Heterogeneous System (IHS) platform which were so far restricted only to graphics rendering. The HSA Foundation [5] was set up to develop and define cross-vendor hardware specifications and software development tools needed to allow application software to better use this IHS architecture. The IHS architecture, as illustrated in Figure 1(a), consists of multiple CPU cores with private L1 and shared L2 caches along with multiple GPU cores or Compute Units (CUs) with private L1 and shared L2 cache. These caches (except GPU L1 caches) are kept coherent with a heterogeneous MOESI-like coherence protocol. The Network-on-Chip (NoC) connects the cores with the caches and the memory controllers, while the memory itself is off-chip and connected via memory channels.

The IHS processors have a cache coherent interconnect and support shared virtual address space making pointer sharing semantics possible between CPU and GPU which simplifies programming and allows fine-grain parallel processing.

The advent of die-stacking technology [11] provides a way to integrate disparate silicon die of NMOS DRAM chips and CMOS logic chips with better interconnects. The implementation is accomplished either by 3D vertical stacking of DRAM chips using Through-Silicon Vias (TSV) interconnects or horizontally/2.5D stacking on an interposer chip as depicted in Figure 1(b). This allows the addition of a sizable DRAM chip of capacities ranging from a couple of hundreds of megabytes to a few gigabytes close to the processing cores. These stacked DRAM devices provide high bandwidths of close to 400GB/s compared to the 90GB/s of DDR4 bandwidth [7]. The better interconnect also lowers the latency of access compared to off-chip memory [39].

SRAM has traditionally been used for smaller caches closer to the processing elements because of the need for fast access times. However, modern last-level SRAM caches typically have capacities between 2 and 8MB [1] and this is often not large enough to contain the working set of IHS workloads. Moreover, GPUs have a large number of concurrent threads and their access patterns thrash the Last-Level Cache (LLC), removing useful information from the cache due to its small size. Non-volatile STT-RAM caches have been proposed in recent literature as replacement for on-chip SRAM caches [47]. STT-RAM caches provide about 4× the capacity of SRAM [47] (in the order of 32MB) for the same area due to its higher cell density. Stacked DRAMs offer much higher capacities than its SRAM and STT-RAM counterparts by one to two orders of magnitude [33].

Further, stacked DRAMCaches are attractive because they provide much larger bandwidth than that of an on-chip SRAM and STT RAM caches. This is especially compelling as IHS architecture

Fig. 1. (a) Architecture of an IHS. (b) Proposed DRAM stacking for an IHS.

pack multiple CPU and GPU cores have higher memory bandwidth demands and these inter-die vias and interconnects in the 3D chips can supply these data hungry processors. Stacked DRAMs are not constrained by the bandwidth wall problem as the TSV interface between the DRAM stacks and the logic chip scales with the surface area of the chip rather than with the number of pins at its perimeter. This allows more direct, parallel access to the DRAM modules than accessing the memory external to the chip.

Stacked DRAM can also be used in conjunction with the on-chip SRAM/STT-RAM caches to augment the memory with higher capacities and bandwidth. We expect the key observations of the heterogeneous nature of the requests and the inferences thereof to be similar even with the shared on-chip caches in IHS architectures.

Several recent proposals advocate the use of on-chip DRAM capacity as a hardware-managed last-level cache for improving the performance of multi-core Chip Multi-Processors (CMPs) [21, 25, 34, 39]. In the context of IHS architecture, the stacked DRAM can cater to the large bandwidth requirements of throughput-oriented GPUs while the latency-sensitive CPU applications can benefit from reduced latency of data access.

Managing contention for shared DRAMCache in the memory hierarchy of the two heterogeneous processor architectures which have asymmetric sensitivity and demands, however, introduces novel challenges. When a GPU kernel is launched it creates a large number of concurrent threads which run in a lock-step Single Instruction Multiple Data (SIMD) execution model and sends a large number of requests into the memory hierarchy causing congestion. This causes bottlenecks in request queues at the DRAMCache thus severely hampering CPU performance. Further, GPUs are designed to tolerate longer memory latencies while the memory hierarchy of typical CPUs have been designed to optimize memory access latencies for CPUs. Thus, latency-sensitive CPU applications would benefit from a DRAMCache design that offers lower hit-time and hence a lower miss penalty for the previous level caches (L2 cache). On the other hand, the GPU cores would benefit from higher bandwidth and higher hit-rates in DRAMCache even at the cost of higher hit-time. This necessitates the need for a careful design of memory system for IHS processors that can handle the GPGPU bursty behavior as well as service requests for the CPU with a consistent latency without idling resources while delivering improved system throughput at lower energy.

To improve the performance of IHS systems we introduce a large-capacity stacked DRAMCache, used as a hardware-managed cache, as the first-level shared resource in the memory hierarchies of CPU and GPU cores. Our organization, Heterogeneity-Aware Shared DRAMCache (HASHCache), is aware of the asymmetric and contrasting requirements from the heterogeneous processors. HASHCache attempts to meet CPUs requirement of reduced hit times and lower miss penalties while at the same time improving hit rates for GPU to allow it to better use the DRAMCache bandwidth. In the common case when CPU is running alone, HASHCache is an aggressive direct

mapped DRAMCache optimized for hit times and lower miss penalty through hit/miss prediction as in [39]. However, when GPU is active we propose (i) *PrIS*—a DRAMCache controller scheduling algorithm to prioritize scheduling of CPU requests in the queues to reduce the large waiting time caused due to burst of GPU requests; (ii) *ByE*—a mechanism to allow some of the CPU requests (access requests that are clean data in DRAMCache or data that is currently not in DRAMCache) to be temporally bypassed to utilize the under-utilized off-chip memory bandwidth; and (iii) *Chaining*—a technique to force spatial occupancy control by providing a pseudo-associativity for GPUs to improve hit rates while still allowing certain guaranteed occupancy for CPU lines. HASHCache achieves these goals using a lightweight and dynamic scheme which does not impose hard partitions on the shared DRAMCache. Using detailed simulations, we show that the proposed optimizations improve overall system performance, on an average by 41% over a carefully designed, but heterogeneity-unaware DRAMCache. Although the techniques proposed in the article are simple and proposed in other contexts, deploying them in the DRAMCache helps to effectively address the disparate demands of IHS processors and improve the performance of both CPU and GPU cores by 211% and 20%, respectively, over an IHS processor with no DRAMCache.

Although there have been studies and proposals to share the on-chip last-level SRAM caches [31, 35] and non-volatile STT-RAM caches [47] in IHS architectures previously, to the best of our knowledge this is the first study on the interactions of such an architecture with a stacked DRAMCache focusing on shared cache management issues. The rest of the article is organized as follows: Section 2 demonstrates the performance that can be gained by adding a DRAMCache to an IHS processor chip and motivates the need for architecting a heterogeneity-aware DRAMCache organization. We present the design principles, organization, and the working of HASHCache in Section 3. Next, in Section 4, we describe the experimental setup and methodology. This is followed by evaluation and results in Section 5. Section 6 presents related work in this area and Section 7 concludes the article.

2 MOTIVATION

As mentioned in Section 1, the large capacity provided by the stacked DRAM is in line with the working set requirements of IHS processors. Using this capacity as a hardware-managed cache could provide performance gains without any application modifications. The bandwidth benefits and modestly improved latency provided by the DRAMCache help improve the performance of IHS processors over on-chip SRAM caches of reasonable sizes [46].

In this article, we assume a cache organization similar to Alloy cache [39] with a block size of 128 bytes and study the problems and challenges in designing the DRAMCache for IHS architectures. We justify the design decisions in the following section. Each CPU core has a private L1 cache and a shared split L2 cache across the CPU cores. The GPU cores have private L1 and shared L2 cache among themselves. The stacked DRAMCache (of size 64MB) is the first-level shared cache across CPU cores and GPU cores. In our experiments, we consider a multi-programmed workload on the CPU cores and a GPGPU application on a single CPU core and multiple CUs.¹ We refer to this workload as Heterogeneous Application (HeA), as the CPU and GPU applications are co-run. We use the terms Homogeneous Application (HoA-CPU and HoA-GPU) to refer to the cases when (multiple) CPU applications are run alone and GPU application is run alone. Additional details relating to the experimental methodology and workloads are described in Section 4.

First, we evaluate the benefits of having a large DRAMCache for the CPU in an IHS architecture. Figure 2(a) presents the performance improvement for CPU cores due to the addition of a stacked

¹It should be noted here that the GPU application has a CPU component and alternates execution on CPU core and the CUs (kernel execution).

Fig. 2. Performance comparison of (a) CPU and (b) GPU when running alone, co-run, with and w/o DRAMCache.

Fig. 3. DRAMCache CPU access latency and hit rate.

DRAMCache (in terms of harmonic mean of IPCs, normalized to HoA without a DRAMCache) when co-run with a GPU and when run alone. As can be seen from the figure, the addition of a stacked DRAMCache improves the performance of CPU applications in an IHS architecture, on average by just 42%. However, when the CPU runs alone with a stacked DRAMCache it achieves a $3.72\times$ better performance than that achieved in IHS without a DRAMCache. We observe that this huge performance gap is primarily due to the interference of GPU applications. Although this HoA-CPU performance cannot be achieved in IHS, there still exists a significant opportunity for improvement. This gap in performance can be attributed to the unmanaged heterogeneity and interference in the DRAMCache organization.

We further investigate the cause of this performance gap. Figure 3 presents the DRAMCache access latency experienced by a CPU request (in terms of CPU cycles) on the primary y -axis and the CPU hit rates on the secondary y -axis, while running alone and co-run with the GPU application. We find that the presence of GPU application increases the average access latency of the DRAMCache by 213% while hit rates of the CPU are marginally impacted (only by about 4%) when co-running. This increase in average access latency is primarily attributed to the large number of GPU requests flooding the DRAMCache controller when the GPU kernels are co-running with the CPU applications. It should be noted here that, even though we use the terms CPU and GPU requests separately, they may refer to the same data. This terminology only indicates the source of the request.

Next, we study the impact of co-running on the GPU applications. Figure 2(b) presents GPU performance obtained by the addition of a stacked DRAMCache when co-run with the CPU and when run alone. We observe that with the introduction of DRAMCache in an IHS, the GPU performance improves on average by 24%. This performance is within 10% of its HoA GPU performance with DRAMCache. Thus, co-running with CPU applications has only a minor impact on the GPU performance. Although the above discussion indicates that the design decisions of heterogeneity-aware DRAMCache are influenced by the latency-sensitive CPU applications, we emphasize the need to effectively utilize the large capacity and the higher bandwidth of DRAMCache by GPU applications. Toward this goal, we experimented with a DRAMCache of 128MB and two-way associativity. We observe that the hit rate for GPU applications improves, on average, by 5%. While

Fig. 4. Performance comparison of (a) CPU and (b) GPU when with un-partitioned D\$ and partitioned D\$.

the improvement in hit-rate is not significant, as the number of GPU requests is very large (two or three orders of magnitude higher than the CPU requests), even such a small increase in the hit rate leads to large improvements in utilization of the DRAMCache bandwidth by the GPU application. This makes a case for improving the associativity for GPU requests by introducing some pseudo-associativity in the DRAMCache, without impacting the hit-time offered by a direct-mapped Tag-and-Data (TAD) cache.

Based on the above motivation study, we conclude that there are significant performance benefits that could be obtained with the introduction of the stacked DRAMCache for the latency-sensitive CPU applications. However, in a naive implementation of the DRAMCache, these benefits can be lost due to interference from the co-running GPU application. Hence, it is important to carefully architect the DRAMCache organization to ensure CPU applications are not hampered due to this interference. This requires the design to be aware of the heterogeneity of the applications (CPU vs GPU) and their demands on the memory hierarchy.

To quantify the effect of interference between CPU and GPU, we further experimented with a hard partitioned DRAMCache. We partition the request queues and allocate a different set of banks for CPU and GPU data. We divide the 32 banks in the DRAMCache into 20 banks (62.5% cache size) and 12 banks (37.5% cache size) to alleviate bank conflicts. We use a data management and coherence policy such that the data requested by the core is found in its corresponding partition of the DRAMCache. Cache misses within the partition incur memory access latency. Although such a partitioning scheme is not realistic, it gives us the ceiling for the performance that can be expected with a partitioning approach. With this setup, we experiment with allocating a larger cache partition for the CPU first. The second bar in Figure 4(a) and (b) shows the performance of the CPU and GPU with such a partitioning scheme (in terms of harmonic mean of IPCs, normalized to HeA with an un-partitioned DRAMCache). We observe that the performance of the CPU reduces by a mean of 5.2% while the performance of the GPU too reduces by a mean of 2.6%. Next, we allocate the larger DRAMCache partition to the GPU cores. The third bar in Figure 4(a) and (b) shows the performance of CPU and GPU for such a partitioning scheme, respectively. We observe that the performance of the CPU and GPU cores reduces by 7.7% and less than 0.1%, respectively. Some of the workloads, e.g., Qg1, Qg2, and Qg5 experience considerable fall in the H-mean of CPU cores (by more than 30%), and in none of the workloads the performance of CPU or GPU cores improves significantly. Thus, hard partitioning of DRAMCache is not an effective design as it leads to under-utilization of the large-capacity stacked DRAM, as the effects of interference due to co-running GPU application is felt only when the kernel is run on the GPU sporadically. Further, effectively utilizing the under-utilized main memory bandwidth [15, 18, 20] is also important to achieve higher performance. Lastly, the design should ensure that the CPU applications and the GPU application are able to utilize the large capacity of the stacked DRAM effectively to meet the working set requirements of the respective applications in the best possible way. In the next section, we present our HASHCachedesign which addresses these issues.

3 HASHCACHE FOR IHS

HAShCache uses HMC-like [10] stacked DRAM memory.² It adapts an aggressive direct mapped cache design with tags stored in DRAM as TAD units [39] with a cache line size of 128 bytes. In this section, we first rationalize these design decisions.

3.1 HAShCache Design

Metadata overhead: The metadata requirement for DRAMCaches, even for caches of size 64MB, is large and is in the order of few MBs. The large storage requirement along with the associated cost if it has to be stored in SRAM, has driven DRAMCache designs to either use larger block sizes (of size 2KB or 4KB [26, 28]) to reduce metadata overhead or co-locate metadata alongside data in the DRAMCache [25, 34, 39].

To understand the spatial locality characteristics of large blocks in an IHS configuration, we experimented with 512 byte block organization for the DRAMCache. The 512 byte block size provides a good tradeoff between increased hit rates and reduced metadata for multi-core CPU workloads [21]. While we increase the cache line size of the DRAMCache, the higher caches (CPU and GPU L2) still operate at 128 byte cache line size, i.e., requests to and responses from the DRAMCache are for 128 byte cache lines. Thus, one 512 byte cache block in the DRAMCache has four 128 byte sub-blocks that can be requested by higher level caches. When a requested cache line is not found in the DRAMCache, a request for the corresponding 512 byte aligned cache block is sent to DRAM memory. Bringing in a larger block (512 byte) can improve the cache hit ratio (due to spatial locality); the miss penalty (of the DRAMCache) also increases due to the increased fetch and data transfer.

We evaluate how much of the data brought by the DRAMCache is actually used by the higher level caches in this setup. We find that on average, for 65% of the 512B blocks that are brought into the cache, at most two sub-blocks of 128 bytes are requested/used. This implies wasted capacity in the DRAMCache and wasted off-chip bandwidth. Figure 5 shows, on the primary y -axis, the performance of CPU and GPU with a 512 byte block size DRAMCache, in terms of harmonic mean of IPCs normalized to that of 128 byte block size DRAMCache. Although with the increased cache line size of 512 bytes, we observe an 11% and 8% improvement in hit rate in the DRAMCache (not shown in Figure 5) for CPU and GPU, respectively, the performance degrades by 12.2% and 10.6%, respectively, for the CPU and GPU cores compared to a 128 byte block organization. The poor performance of a large cache block size is attributed to the increased average access latency of the DRAMCache (due to larger block size fetch) and the wasted off-chip bandwidth. The secondary y -axis in Figure 5 shows the percentage increase in memory access latency for the 512 byte

²There are two commercial implementations of the stacked DRAM available on the market today. Hybrid Memory Cube (HMC) is a joint Intel-Micron standard that is not yet a JEDEC standard. HMC has a logic die (ASIC) at the base of the stacked DRAM that houses the control logic for the DRAM stacks (as implemented in the Xeon Phi Knights Landing processor [7]). On the other hand, the High Bandwidth Memory (HBM) proposed by AMD, NVIDIA, and Hynix has been ratified by JEDEC. HBM tightly couples the host compute die with an interface that is divided into independent channels. HBM is slated to be used in NVIDIA Pascal GPU and next generation AMD APU/GPU. Both these technologies promise high bandwidth up to 250GBps using TSV interconnects. The only difference in the technology would be the signaling interface. While HMC uses a SERDES interface to its stacked DRAMs logic die, HBM uses a traditional DDR signaling interface to the stacked DRAM chip.

Our evaluation in this work used a HMC-like model and timing parameters. However, HAShCache is flexible enough to be adapted using either technology. HAShCache addressing schemes, mechanisms, and associated metadata are stored in the DRAMCache controller. In the case of HMC, this would be implemented at the logic die while the same would be implemented at the host side stacked DRAM controller in the case of HBM.

Fig. 5. Performance of 512B vs. 128B block size.

DRAMCache organization over the 128 byte DRAMCache organization. We observe that memory access latency increases by an average of 75% due to the unused data fetched.

Tags-in-DRAM designs have further focused on improving access latencies by removing tag-serialization overhead using overlapped tag lookups [34] or storing TAD units [39]. These designs come close to tags-in-SRAM like access latency without concerns of spatial locality characteristics of large block sizes. Hence, HASHCache organizes data at 128 byte block size and stores data in DRAMCache as a cohesive TAD unit.

Associativity: Providing set associativity is known to improve cache hit rates by reducing conflict misses. In DRAMCaches where tag is stored in DRAM, associativity comes at a cost. Hit latencies increase due to the tag requiring to be burst out of the stacked DRAM. Hence, there is an implicit tradeoff between providing better hit-rates and reduced access latency. A higher associativity design is suitable for a GPGPU processor which can trade increased access latency for higher hit rates to make better use of the larger bandwidth of the DRAMCache. On the other hand, the CPU would suffer when using such a design due to the increased hit-latency and would instead prefer a latency-optimized direct-mapped cache [39]. Further, such a direct-mapped organization simplifies the design and eschews the need for tag-caches [25] and way locators [21] to improve hit times. HASHCache’s organization is also in line with the commercially adopted stacked MCDRAM on the Knights Landing generation of the Intel XeonPhi processor [7] where the DRAMCache is organized as a direct-mapped cache.

Miss Penalty: The access latency provided by a stacked DRAM is only slightly better (say, $0.7\times^3$) compared to off-chip DRAMs. Thus, a miss in the DRAMCache would experience a delay of $1.7\times$ as the DRAM (memory) access takes place after the miss is detected (serially). To overcome this, researchers have proposed cache line hit predictors [34, 39] which are critical to extract performance from DRAMCaches. These predictors start an early access to memory if they predict that the block will miss in the DRAMCache. Intuitively, we apply the MAP-I prediction [39] to CPU requests to start early memory access when an access is predicted to be a miss. GPU requests always proceed serially through the cache after verifying via tag match. This helps to (a) reduce the wastage of off-chip bandwidth for mis-predictions for GPU and (b) avoid large structures that will be required for making reliable predictions for GPUs which might require correlating warp, thread, and CU IDs.

Row Buffer Hits (RBHs) vs. Bank-Level Parallelism (BLP): Stacked DRAMs are organized as vaults (channels), layers (ranks), and banks within each layer as shown in Figure 1(b). Each vault

³We model our DRAM timings for stacked DRAM from DRAMSpec [36] HMC model where, for example, t_{c1} for DDR3 is 13.75ns and that for HMC is 9.9ns (~70% of DDR3). Detailed timings can be found in Section 4. Prior works assume latencies varying from $0.5\times$ [39] to $1.0\times$ [18]. Further, stacked DRAM latencies, for example in Intel MCDRAM [7], are very similar to off-chip DRAM. Given the smaller size of stacked DRAM, we feel that our assumption of $0.7\times$ is a reasonable midpoint. Further, we perform sensitivity study w.r.t. the relative access latencies for off-chip and stacked DRAM in Section 5.6.2.

has several TSVs which constitute lanes in a channel. Stacked DRAMs provide large bandwidth by organizing DRAMs as several smaller banks within layers. Given this abundant BLP, should DRAMCaches exploit this parallelism over improved RBH? In other words, should the addressing scheme of a DRAMCache be organized as RoCoRaBaCh (Row,Column,Rank,Bank,Channel)—referred to as the BLP-scheme—which distributes the cache blocks in banks of different ranks as opposed to RoRaBaCoCh—referred to as the RBH-scheme—which stores cache blocks consecutively in the row of a bank. We experimented with both addressing schemes for an IHS processor with a DRAMCache and observe that both the CPU and GPU perform on average 3% and 1% worse, respectively, when using the BLP scheme. Consequently, HASHCache is addressed using a RBH-friendly addressing scheme (RoRaBaCoCh scheme).

3.2 HASHCache Mechanism

As shown in Figure 3, despite the addition of such a carefully designed DRAMCache, the CPU still suffers significant performance losses while the GPU is relatively unaffected in an IHS architecture, compared to when they run alone. GPUs capability to context switch between warps make it more latency tolerant. This suggests that the DRAMCache should be optimized to regain the lost CPU performance without compromising the GPU performance. In this subsection, we propose three schemes for achieving this.

3.2.1 Heterogeneity-Aware DRAMCache Scheduling. DRAM devices operate at a much lower clock rate compared to the processor. Moreover, DRAM cells need to be periodically refreshed to preserve the data which further reduces the available time for accessing the device. The imbalance in request arrival rate and service times creates a queuing effect. Hence, DRAM devices have traditionally had limited size queues to hold requests until they can be serviced by the device. However, we observe that in an IHS processor the large burst of requests from the GPU quickly exhausts the available queue positions (buffer locations) at the DRAMCache controller leading to requests being rejected causing the DRAMCache to be blocked. The CPU requests, which are interleaved with the GPU requests, are few and far spaced and thus suffer large waiting time due to retries. This is compounded by the fact that GPU exploits good row buffer locality and is preferentially scheduled by the DRAMCache controller (under First-Ready First-Come-First-Serve (FR-FCFS) scheduling [9]), causing increased queue latencies for CPU requests. Increasing queue lengths beyond a certain measure increases scheduling overheads as DRAM schedulers search the queues for the most suitable request to schedule based on certain heuristics.

HASHCache reduces waiting time of CPU requests by prioritizing them at the DRAMCache controller without starving GPU requests. For this, HASHCache applies a CPU Prioritized FR-FCFS algorithm over each of the Read, Write, and Fill queues to schedule a request at each bank. The scheduler is cognizant of the request heterogeneity and searches the short queues for either a first CPU row buffer hit request or a first CPU row activation request to schedule before scheduling a GPU request in a FR-FCFS manner. For GPU requests, starvation is avoided firstly, by allowing GPU requests to be scheduled to a prepped (open) row after the CPU has completed access to that row and secondly, by allowing GPU to schedule its requests to a bank immediately, when the queue has no more CPU requests to that bank. These scheduling decisions are made subject to the device timing constraints, similar to an FR-FCFS scheduler.

Prioritization of CPU requests alone may not help, as the flood of memory requests from GPU can quickly fill the precious buffer at the DRAMCache controller, resulting in CPU requests not even entering the buffer. HASHCache overcomes this problem by guaranteeing certain minimum occupancy for CPU requests in the buffer at the DRAMCache Controller. This is accomplished

using a selective reject-retry mechanism⁴ for GPU requests when the queues reach a certain critical level. Together, these two mechanisms attempt to reduce the DRAMCache latency experienced by CPU requests. We refer to these mechanisms collectively as *PrIS* (CPU Prioritized FR-FCFS with IHS-aware Scheduling).

PrIS differentiates requests broadly as CPU or GPU requests and not within individual CPU cores or GPU CUs for scheduling.⁵ We find that in an IHS architecture the interference between CPU and GPU applications vastly overwhelms the interference between homogeneous application workloads. Hence, *PrIS* only has to make a binary selection for scheduling which greatly simplifies the scheduling algorithm. *PrIS* is a simple yet effective, single stage modified FR-FCFS algorithm that does not incur any hardware overhead in terms of multiple or large requests queues or batching stages as in [9].

PrIS picks requests to be scheduled from the input queue. The exact selection of input queue (Read, Write, or Fill Queue) is done external to this algorithm based on certain heuristics and constraints which is beyond the scope of this scheduler algorithm. Broadly, the algorithm picks one of the three types of requests: (a) seamless row buffer hit, (b) hidden bank prep, or (c) prepped row, in that strict order of priority.

A seamless row buffer hit request refers to a request that can issue a column access to an already activated row in the bank, without any further delay. A hidden bank prep request is a request that can overlap the current operation in other banks (in the same rank) and issue a request to activate or precharge a row in the requested bank. Among the hidden bank prep requests an FCFS policy is followed. A prepped row request refers to a request that needs to wait for the current column access to complete to the currently active row in a bank. Thus, choosing a prepped row leads to a bubble in the pipeline of the scheduler where the request has to wait for the row to become available for a column access command.

Additionally, the *PrIS* algorithm picks a request in the priority order of

CPU seamless row buffer hit > CPU hidden bank prep > CPU prepped row > GPU
seamless row buffer hit > GPU hidden bank prep > GPU prepped row buffer hit

We experimented with several combinations of this priority order and find that prioritizing CPU requests at all levels provides the best performance for CPU requests, while the reduction in GPU performance due to de-prioritization between the schemes was not significant.

Although *PrIS* may introduce delays in scheduling GPU requests, due to the relatively fewer numbers of memory requests coming from CPU (in comparison to the number of requests from

⁴Interactions between the memory components is modeled as a master-slave port architecture in gem5-gpu. This is common practice in most architectures that involve requests and responses. A master port sends requests while the slave port receives requests and performs appropriate action on the requests. For example, when the directory controller's memory master port sends a request to the slave port of DRAMCache Controller it enqueues the request in the appropriate queue (read or write queue). Similarly, a slave port sends out responses and the master port receives these responses and performs appropriate action. For example, when the DRAM controller responds to the a read request, the DRAMCache controller clears the corresponding Miss Status Handling Register (MSHR) and sends a response to the requester.

However, a cache can be in a blocked state when MSHRs or targets within a MSHR or WriteBuffers are unavailable. In such situations, a request is received at the DRAMCache slave port but is not able to act on it due to the cache being blocked or read/write queue being full. In such cases, the request is rejected and the reason for the blocking is remembered (i.e., the resource causing the block). When the corresponding resource becomes available, a retry message is sent to the master port from the slave port. Once the master port receives this message, the request is resent. This is called the reject-retry mechanism. The reject-retry mechanism for requests is already implemented/part of gem5-gpu [38]. This mechanism is independent of the NoC that handles routing, channel partitioning, and so forth.

⁵The request packet carries the source of the requester, i.e., GPU L2 or CPU L2 cache controller which is used to classify the request as CPU or GPU.

GPU), it is often the case that the GPU requests get serviced without much delay or starvation. In fact, our simulator framework shows that in all our experiments the GPU request is never delayed beyond 500 cycles for any of the benchmarks.⁶ This suggests that the applied heuristic in *PrIS* allows such request service reordering without causing starvation or deadlocks.

3.2.2 Heterogeneity-Aware Temporal Bypass. The large sizes of the stacked DRAMCache ensures cache lines have fairly long residency time before being evicted. Hence, DRAMCache has fairly large hit rates which leads to idling of off-chip DRAM bandwidth. Moreover, the stacked DRAM and off-chip DRAM utilize the same underlying technology and hence incur comparable latency ($0.7\times$ for DRAMCache vs. $1.0\times$ for DRAM). In an IHS architecture, the increased access latency incurred by a CPU request at the DRAMCache (see Figure 3) when the GPU is running, makes the off-chip DRAM an attractive target to direct some of the CPU requests. This leads to improved resource utilization without incurring any increased latencies for CPU requests.

To hide the latency of miss, our aggressive baseline design already incorporates a hit/miss predictor (similar to MAP-I predictor [39]) for CPUs to initiate an early access to off-chip DRAM when a miss is predicted in the DRAMCache. These requests are then enqueued in the DRAMCache queues for verification of a miss⁷ by a tag match.

When the tag is matched in the DRAMCache, in the case of a hit, data from the DRAMCache is forwarded to the requester and the DRAM memory access is squashed or its response is ignored. In the case of a miss, data from the memory is forwarded and inserted into the cache. Normally, it is expected that the access to the DRAMCache completes earlier (due to its relatively lower access latency) than the DRAM response. However, when the GPU is running, the parallel request to off-chip DRAM memory often returns earlier and waits in the Miss Status Handling Registers (MSHRs). This is due to the increased queuing delay at the DRAMCache compared to the access latency at the DRAM. HASHCache exploits this observation to bypass CPU read requests for both misses and clean lines.

For this, HASHCache uses a Bypass Enabler (*ByE*). *ByE* uses a counting Bloom filter [12, 17] that tracks the dirty lines in the DRAMCache and provides a space-efficient way to determine if a given request can be bypassed. The property of a Bloom filter to answer “definitely not in set” allows us to bypass requests correctly, i.e., without verifying tags in the set of the DRAMCache. On a write request when a cache line becomes dirty in the DRAMCache, the address is hashed into *ByE* and the corresponding counters are incremented. When a dirty line is evicted from the DRAMCache, *ByE* attempts to remove the entry from the Bloom filter by decrementing the corresponding locations.⁸ *ByE* bypasses CPU requests only when the GPU cores are executing the kernel. For this, all CPU read requests lookup into *ByE* as shown in Figure 6. If the Bloom filter search returns a negative result, the address is guaranteed to be not dirty in the DRAMCache. Thus, the request can safely be bypassed to utilize the off-chip DRAM bandwidth. Further, when the bypassed CPU requests return from the off-chip DRAM access, these requests are directly forwarded to the requester and are not inserted into the cache. Firstly, this allows *ByE* to ensure that future write requests for the line do not hit in the DRAMCache as an increase in dirty lines would lead to reduced bypass

⁶It is also possible to modify this scheme to reset priority/switch back to a baseline FR-FCFS scheme after servicing some number of CPU requests to avoid possible GPU starvation.

⁷This is required to ensure that misprediction does not result in using data from the DRAM for lines modified (dirty) in DRAMCache.

⁸Counting Bloom filters use saturating counters. If counter saturates, decrementing it can lead to false negatives (dirty lines predicted as clean lines and wrongly bypassed). Hence, saturated counters are never decremented. Although this may increase the false positives (clean lines being predicted as dirty), which only reduces the benefits obtained by *ByE*, it does not affect functional correctness. In our implementation, we observe that on average just 2% of 2-bit counters in the Bloom filter saturate out of the 512K counters.

Fig. 6. Working of HASHCache with ByE.

efficiency. Secondly, this allows *ByE* to reduce some of the bloat caused by a Miss Fill [15] into the DRAMCache.

All write requests and GPU read requests proceed serially after looking into the cache. *ByE* does not bypass any write requests as it would otherwise require a back-invalidation of the cache line, if present in the DRAMCache, which would need a full DRAMCache access.

We find that a small 2-bit counting Bloom filter implemented with two H_3 hash functions [40] and 512K entries per controller is sufficient to produce reasonable bypass efficiency with a tolerable mis-prediction rate. The total overhead for *ByE* is 256KB for a 64MB DRAMCache, which is less than 0.4% of the cache size.⁹

3.2.3 Heterogeneity-Aware Spatial Occupancy Control. The schemes proposed in the previous two subsections, *PrIS* and *ByE*, attempt to improve the latency of CPU requests. The mechanism described in this section details HASHCache’s approach to improve the utilization of the DRAMCache for GPU requests, in order to exploit the higher bandwidth provided by it. We make the following observations and inferences: (i) For the GPU to be able to better utilize the DRAMCache bandwidth, the hit rates for GPU workloads should be large enough that the GPU does not have to frequently use the relatively constricted off-chip DRAM buses. (ii) As noted in Section 2, GPUs can trade access latencies for higher hit rates. Further, providing associativity for GPU requests improves the hit rate. (iii) The working sets of CPU applications tend to be limited to few tens of MBs due to the limited amount of MLP that can be exploited by the CPUs. Thus, providing a larger than certain share of cache leads to no further improvements in hit rates and IPC for CPU. Nevertheless, the CPU can still gain from some share of the DRAMCache due to reduced latency of access. (iv) Given that GPU can exploit much higher MLP using several thousands of threads, the relatively small GPU L2 cache provides limited filtering of traffic and has significantly high miss rates while, on the other hand, CPUs have sufficiently sized L2 cache sizes to be able to retain blocks longer before re-requesting a block.

The above observations lead to the following conflicting requirements. It is important to ensure that the CPU requests have a certain share (minimum occupancy) in the DRAMCache to ensure the benefits of lower latency, while to effectively use the larger share of DRAMCache for GPU requests, it may be required to increase associativity of the DRAMCache. However, such an associativity should not unduly increase the hit latency for CPU requests.

To accomplish the above goals, HASHCache uses the *Chaining* scheme which introduces (pseudo) associativity mainly for GPU requests, while ensuring a certain minimum occupancy

⁹Although the metadata size increases linearly with capacity of DRAMCache, the Bloom filter overhead is still 0.4% of DRAMCache size. Another way to manage this is to distribute the address space into a multiple partitioned *ByE* structure. However, we do not explore such techniques in this work.

Fig. 7. HASHCache. (a) Row organization (2KB). (b) Access path in the HASHCache.

for CPU lines. The *Chaining* scheme uses a linear probing-like technique inspired by the collision resolution mechanism of a hash map. To ensure minimum occupancy for CPU requests, *Chaining* maintains a low-threshold value (l_{cpu}) and when the occupancy of CPU lines¹⁰ reaches this threshold, *Chaining* ensures GPU data does not replace data brought in by CPU.

In the other situations, HASHCache modifies the replacement policy in the DRAMCache depending on the requesting core type. For a GPU request that is evicting another GPU line, HASHCache looks for a line belonging to a CPU to replace within the same row in the next three consecutive locations, i.e., if B is the original cache block, then the blocks considered for insertions are $(B + 1)\%N_s$, $(B + 2)\%N_s$, and $(B + 3)\%N_s$, where N_s is the number of blocks in a DRAMCache row (page). Hence, the inserted block always lies in the same row as the original cache block. Note that for every set, there could be at most one chained set, providing a pseudo-associativity of at most 1. We refer to this inserted location as the *chained block* and the actual cache block the request mapped to as the *original block*. The location of the *chained block* is then represented as a 2-bit offset and is stored along with the metadata for the original set (see Figure 7(a)). When a cache block is evicted, if it was a *chained block*, to unchain it (from the *original block*), the offset stored in the reverse chain bits field in the metadata for the *chained block* is used. The chain dirty bit field (Figure 7(a)) in the metadata indicates whether the chained location, if any, holds modified data. This is used to optimize the access path for CPU and reduce the adverse effect of a double set lookup for latency-sensitive CPU requests as shown in Figure 7(b). *Chaining* relies on the hit/miss predictor to have started an early access to memory. This avoids the second set lookup for CPU if the parallel memory (PAM) access has returned and the chained block is clean. Additionally, each tag also stores 1 bit information about the owner of the block (CPU or GPU). This bit is used to make quick replacement decisions locally. The additional metadata required for *Chaining* is only 6 bits which can easily be accommodated in the existing 8 byte metadata. Lastly, the unused 8 bytes (at the end of each row(page)) is used to store ownership information of each block in the row (15 bits). This information is used to make the *Chaining* replacement decision.

As explained earlier, when the l_{cpu} threshold is reached, GPU lines are not allowed to evict CPU blocks and such GPU requests contending to evict a CPU line are forced to chain to another block belonging to a GPU and evicting that instead, thus maintaining the l_{cpu} occupancy for CPU. In the very rare case that a GPU block is not found within the three consecutive locations, the request is not inserted into the cache. Thus, even with the *Chaining* scheme HASHCache follows the simple

¹⁰As mentioned earlier, we classify a data as CPU data or GPU data based on whose request last accessed the data in the DRAMCache. An alternate equally feasible design point is to classify the data as CPU or GPU data based on which request brought the data into DRAMCache, although the CPU and GPU may subsequently access it. In our setup, only the CPU core executing the GPGPU application possibly shares data with the GPU and hence we do not expect to see a significant difference in the results.

Table 1. GPU Fill Request Insertion Policy

Threshold Reached?	Original Set	Not Chained	Chained Set	
			CPU	GPU
No	CPU	Replace original	Replace chained	Replace original
	GPU	Chain to nearest CPU set	Replace chained	Replace original
Yes	CPU	Chain to nearest GPU Set	Do not insert	Replace chained
	GPU	Replace original	Replace original	Replace original

Threshold Reached: $O_{gpu} \leq (1 - l_{cpu})$ where O_{gpu} is the current GPU occupancy in DRAMCache row.

static replacement policy, i.e., evict the conflicting cache line according to Table 1 for GPU lines and direct mapped for CPU lines. HASHCache does not require one to maintain LRU/MRU recency information for replacement.

We now summarize the insertion policy used by the *Chaining* mechanism in HASHCache. For all CPU fill (insertion) requests, the data is always inserted in the original block, and the victim block is evicted—removing chaining, if any, using the reverse chain bit. For a GPU fill request, if the low threshold mark for CPU occupancy is not reached, then the *Chaining* scheme replaces a CPU location, either from the original location or from the chained location, as indicated in row 1 of Table 1. For a GPU fill request, if the original location belongs to GPU and does not have a chained location, then block is inserted in one of the nearest CPU blocks ($(B + 1)\%N_s$ or $(B + 2)\%N_s$ or $(B + 3)\%N_s$). If such a nearest CPU block is not found, the request is inserted into the original block itself. If the original location is chained, then the scheme replaces the chained location, if that belongs to CPU or the original location itself, as indicated in row 2 of Table 1. When the CPU occupancy has hit low threshold, then a GPU fill request replaces the original location if it belongs to GPU (row 4 in Table 1). If the original location belongs to CPU and does not have a chained block, then the GPU request is chained to the next nearest GPU location. If the original location is chained, but the chained location belongs to GPU, then the fill request replaces that. Otherwise, the fill request is not inserted in the DRAMCache (see row 3 in Table 1). Thus, the *Chaining* scheme ensures, as far as possible, the CPU requests can find the data in the original location, while the GPU requests attempt to exploit pseudo-associativity for increased cache occupancy. In all cases (for both GPU and CPU requests), the access is satisfied with at most two tag matches, either in the original location or in the chained location (identified by the chain bits).

In essence, HASHCache uses this *Chaining* mechanism to force occupancy control in the DRAMCache. *Chaining* is able to (i) ensure a minimum occupancy for the CPU lines while effectively allowing the GPU to occupy the rest of the DRAMCache by providing pseudo-associativity; (ii) remain as a direct-mapped cache for the majority of the CPU requests; and (iii) avoid forcing eviction of hot GPU lines while also avoiding storing of dead lines in the cache. Lastly, this scheme is dynamic and allows one to adaptively set CPU occupancy threshold l_{cpu} based on the workloads requirements. This occupancy control mechanism does not incur any additional storage and uses the unused space in the DRAMCache rows. Once the GPU finishes kernel execution, HASHCache returns to a direct-mapped cache as the CPU lines inserted into the DRAMCache occupy *chained blocks* thereby unlinking chains.

4 EVALUATION METHODOLOGY

In this section, we describe the experimental methodology used in our evaluation.

HASHCache Configuration: The DRAMCache we evaluate here is a memory-side, first-level shared cache between the two split cache hierarchies of CPUs and GPUs. Access to memory is

Table 2. Configuration of the Simulated System

CPU Core	Five 4-wide OoO x86 cores @2.5GHz
CPU Caches	32KB eight-way split I/D private L1 Cache, 1MB eight-way shared split L2 cache, 128B lines
GPU Core	Eight Fermi SMs@700MHz, 2×2 GTO warp sched Up to 32 threadblocks, 64 warps of 32 threads, 64K registers, 96KB scratch memory
GPU Caches	64KB four-way private L1 cache, 512KB, 16-way assoc L2 coherent cache
Stacked DRAM	Two vaults, HMC_2500_x64, 2KB Page, $t_{CL}-t_{RCD}-t_{RP}-t_{RAS}=9.9ns-10.2ns-7.7ns-21.6ns$ Eight layers/vault, four banks/layer, 64 byte burst size, peak bandwidth 40GB/s Refresh related: $t_{REFI}=3.9us$, $t_{RFC}=59us$
Off-chip DRAM	Two channels, DDR3_1600_x64, 1KB page, $t_{CL}-t_{RCD}-t_{RP}-t_{RAS}=13.75ns-13.75ns-13.75ns-35ns$ One rank/channel, eight banks/rank, 64 byte burst size, peak bandwidth 25GB/s Refresh related: $t_{REFI}=7.8us$, $t_{RFC}=260us$

interleaved across multiple memory controllers. To avoid a biased interleaving due to uniformly striped address patterns, a basic XOR-based address hashing mechanism is employed. Each memory controller manages a 4GB DDR3 memory device and a 32MB stacked DRAM vault. The stacked DRAM vault caches data from the corresponding 4GB memory device that the controller is responsible for. This setup ensures that there are no cross bus requests between controllers. Since our simulation setup has the limitations of having a one-to-one channel mapping between a stacked DRAM vault and a off-chip DRAM channel, our model does not provide sufficient channel-level parallelism as is essential in a stacked DRAM device. To circumvent this limitation, we increase the number of layers (ranks) to provide a higher amount of parallelism in our stacked DRAM. However, the bank capacity is retained as would be in a standard stacked DRAM. The DRAMCache is non-inclusive [24] of the L2 caches above it and uses a write no-allocate policy.

The NoC is modeled as a hierarchical cross-bar topology. One cross bar connects the CPU and caches to each other and another cross bar connects the GPU to the caches. A third cross bar connects the LLCs of the CPU and GPU to the DRAMCache memory controller. The DRAMCache controller does the request scheduling and dispatches commands to the stacked DRAM devices. The DRAMCache controller and off-chip DRAM controller are connected by a point-to-point link. The DRAMCache controller forwards the request to the off-chip DRAM controller if necessary.

Simulator: We evaluate the performance of the DRAMCache over an IHS processor using a modified version of the cycle accurate simulator gem5-gpu [38] which is configured to simulate cache coherent unified CPUs and GPUs. Our IHS consists of five CPU cores (four-wide OoO cores, operating at 2.5GHz) with a 32KB private L1 cache (split I/D), and a 1MB shared L2 cache (shared across all CPU cores). The IHS also consists of eight Fermi-like compute units, operating at 700MHz with a private 64KB L1 and 512KB shared L2 cache (shared across all CUs). The details of the IHS configuration are given in Table 2. The private L1 of CUs are non-inclusive of the shared GPU L2 cache and can hold stale data. However, the GPU L2 cache is coherent with all levels of the CPU hierarchy. The CPU caches and the GPU L2 cache are kept coherent in the simulator. Table 2 provides the simulator setup details. Our simulator also respects all significant timing and functional parameters for the stacked DRAMCache (including refresh, data bus, request queues, scheduling algorithms, command signaling, and clock frequencies) using the DRAMCtrl [22] model. We have also faithfully modeled a Fill Queue [14] for fill requests to insert data into the cache on the return path from main memory. We also model MSHR and WriteBuffers with their associated latencies to realize the precise working of caches.

Table 3. Workloads

Name	Multi-program SPEC2006	Rodinia
Qg1	cactusADM;gcc;bzip2;sphinx3	needle
Qg2	astar;mcf;gcc;bzip2	needle
Qg3	gcc;libquantum;leslie3d;bwaves	needle
Qg4	astar;soplex;cactusADM;libquantum	<i>k</i> -means
Qg5	milc;mcf;libquantum;bzip2	<i>k</i> -means
Qg6	bzip2;gobmk;hmmer;sphinx3	<i>k</i> -means
Qg7	soplex;milc;cactusADM;libquantum	Gaussian
Qg8	milc;libquantum;gobmk;leslie3d	Gaussian
Qg9	astar;milc;gcc;leslie3d	hotspot
Qg10	gcc;gobmk;leslie3d;sphinx3	hotspot
Qg11	astar;cactusADM;libquantum;sphinx3	srad
Qg12	astar;mcf;gobmk;sphinx3	streamcluster
Qg13	astar;cactusADM;libquantum;sphinx3	lud

Further, our baseline setup with a naive DRAMCache is equivalent to the on-chip shared caches in [47] without the NoC-related improvements, which is orthogonal to the ideas presented in this work.

Workloads: Applications having high and medium memory intensive behaviors from SPEC CPU2006 suite [23] were chosen to form a multi-programmed workload. We use the Rodinia benchmark suite [13] to represent GPU applications that offload kernel computation to GPU CUs. These Rodinia applications are modified to elide the *memcpy* api calls so as to run with unified virtual and physical address spaces. Combinations of quad-core multi-programmed workloads were coupled with a full Rodinia benchmark to create a representative mix of applications that would run in an IHS system. These composite workloads embody different levels of total memory intensity produced by the CPU and GPU cores. Our simulation of the simultaneous activity on both CPU and GPU cores using a co-running multi-programmed workload along with Rodinia-nocopy benchmarks allows us to demonstrate the interleaving of memory traffic at the stacked DRAM and off-chip DRAM. We also measure the footprint of these workloads in terms of the number of unique 128B cache blocks accessed at the DRAMCache. The memory footprints range from 70MB to 650MB for quad-core CPU workloads and from 5.5MB to 135MB for the GPU application. The smaller footprints obligate us to use a smaller stacked DRAMCache capacity of 64MB for all workloads and configurations to make pertinent observations.

Simulation Methodology: We fast-forward the initialization phase of each workload up until just before the launch of the first kernel of the GPU program. We ensure that each core is fast-forwarded by at least 2 billion instructions and in total each quad-core workload runs 20 billion instructions on average in the fast-forward phase. This is accomplished by adding a pause phase to the Rodinia benchmarks for the duration until the initialization quota of the SPEC programs is complete. We then warm the cache until the fastest core completes 250 million instructions. During the warm-up phase the GPU program is not executed. Timing simulations were then run for at least 250 million instructions for each CPU core, resulting in a total of more than 1 billion instructions across all the CPU cores. As is the norm, when a core finishes its quota of 250 million instructions, it continues to execute until all the cores have completed.

The Rodinia application uses an extra fifth CPU core and offloads to the integrated GPU. These applications are modified to execute in a *conditioned loop* such that there is no corruption of data

structures in the program. The *conditioned loop* is run infinitely and represents the Region of Interest (ROI) of the Rodinia benchmark. This region includes sections of CPU activity and GPU offloads in the execution, as is typical of a HSA program which will exhibit an interleave of offloaded regions and serial CPU sections. However, only the performance statistics for the first execution of the *conditioned loop* in the program are considered. This is done as the first loop represents the true run of the GPU kernel. The number of *conditioned loops* executed depends on the length of the simulation and the IPC achieved by the GPU. Also, subsequent loops can achieve better hit rates in cache due to the data fetched by the earlier loops, thus not corroborating with the true performance achieved. In cases where the ROI is longer than the complete run of the CPU workloads, the statistics for the last completed kernel are used.

Coherence and Memory Consistency: The DRAMCache evaluated here is a memory-side cache [1, 18, 44]. These caches are outside the coherence domain. These caches do not require one to add additional states to the coherence protocol and do not need to be snooped separately. They are logically just in front of the memory and serve to reduce the average latency of memory accesses and increase the memory's effective bandwidth.

The consistency model followed for each type of cores (CPU cores and GPU cores) is different and is dictated by the corresponding processor architecture, i.e., strong memory consistency for the CPU and weak/release consistency for GPU. Cache line evictions and explicit fence operations flush data from the L2 SRAM cache to memory and could be cached in the DRAMCache. Subsequent requests to memory for that cache line first look up the DRAMCache (or bypass in the case of *ByE*) before sending the request to the off-chip DRAM. Further discussion about cache coherence and memory consistency models for IHS architecture is beyond the scope of this work.

Performance Metric: We report the performance of each processor using the intuitive metric of harmonic mean of IPC for the CPU cores and GPU CUs, which is defined as

$$H-MEAN_{CPU} = \frac{n_{cpu}}{\sum_{i=1}^{n_{cpu}} \frac{1}{IPC_i^{CPU}}}, \quad H-MEAN_{GPU} = \frac{n_{gpu}}{\sum_{i=1}^{n_{gpu}} \frac{1}{IPC_i^{GPU}}},$$

where IPC_i^{CPU} and IPC_i^{GPU} are the instructions per cycle achieved by the i th CPU core and the i th GPU CU, respectively. In this work, we report CPU and GPU $H-MEAN$ (and the improvement in them) independently. This is done to identify the impact of CPU and GPU applications on each other, as well as to understand the impact of the proposed modifications on each of the application types. We report the combined system performance using the combined $H-MEAN$ of all CPU and GPU cores, as in

$$H-MEAN_{IHS} = \frac{n_{cpu} + n_{gpu}}{\sum_{i=1}^{n_{cpu}} \frac{1}{IPC_i^{CPU}} + \sum_{i=1}^{n_{gpu}} \frac{1}{IPC_i^{GPU}}}.$$

Lastly, we also report combined system performance using the weighted speedup metric [43], which is defined as

$$WeightedSpeedup = \sum_{i=1}^{n_{cpu}} \frac{IPC_i^{CPU_{IHS}}}{IPC_i^{SP}} + \sum_{i=1}^{n_{gpu}} \frac{IPC_i^{GPU_{IHS}}}{IPC_i^{GPU}},$$

where $IPC_i^{CPU_{IHS}}$ and $IPC_i^{GPU_{IHS}}$ denote the IPC achieved by the i th CPU core and the i th GPU CU when running in an IHS setup, respectively; whereas, IPC_i^{SP} and IPC_i^{GPU} denote the same when the i th CPU core and the i th GPU CU are running standalone, respectively.

5 RESULTS

In this section, we evaluate the results of the proposed HASHCache mechanism and the optimizations discussed in Section 3. Figure 8(a) and (b) presents the performance improvement in terms

Fig. 8. Speedups obtained by adding a stacked DRAMCache for (a) CPU and (b) GPU.

of harmonic mean of IPC, for CPU and GPU, respectively, normalized to the baseline without a DRAMCache. We report the performance improvement due to the introduction of DRAMCache (naive), and the different HASHCache mechanisms (*PrIS*, *ByE*, and *Chaining* and some combinations of them) in this section.

5.1 *PrIS* DRAMCache Scheduling

Prioritizing CPU requests with our *PrIS* scheduler at the DRAMCache controller leads to considerable performance benefits for the CPU. By using *PrIS*, the average access latency of the DRAMCache for CPU requests reduces by an average of 15.3% and up to 48.9% over a naive DRAMCache (presentation of detailed data is omitted due to space constraints). Hence, *PrIS* is able to improve the performance of the CPU by an average of 35% over a naive DRAMCache. However, on the flip side, giving aggressive priority to all CPU requests reduces the performance of GPU by 10% despite the GPU being able to tolerate larger memory access latencies. For some of the benchmarks like Qg3, Qg10, Qg11, and Qg12 the high priority given to CPU requests by *PrIS* impacts the GPU, causing the GPU performance to reduce marginally below the baseline IHS architecture without a DRAMCache. Note, however, that, in these workloads, the introduction of DRAM cache (naive) itself improves performance only marginally. Our mechanisms (*ByE* and *Chaining*) further aim to reduce this performance drop for the GPU by ensuring (a) there are fewer CPU requests in the DRAMCache queues and (b) GPU requests, despite being deprioritized at DRAMCache, have a better hit rate in the DRAMCache and avoid accessing the off-chip DRAM.

5.2 *ByE* for Temporal Bypass

ByE attempts to achieve improved performance by directing some requests to be served from the off-chip DRAM, thus achieving improved resource utilization and bandwidth balance in the process. *ByE* alone achieves 12% improvement in CPU performance and a 3% improvement in GPU performance.

The CPU performance improvements are primarily due to bypassed requests facing reduced queuing delays at DRAM. Figure 9 shows the percent reduction in total memory access latency for CPU read requests achieved by *ByE* over an already aggressive naive DRAMCache which employs a hit/miss predictor for CPU requests (primary *y*-axis). The total memory access latency for CPU

Fig. 9. Total MemAccLat reduction with *ByE* and percentage of bypassed read requests for CPU requests.

read requests reduces by an average of 28%. The already high hit rates for GPU in the DRAMCache coupled with the no-fill policy for bypassed CPU requests ensures fewer GPU requests at the off-chip DRAM, which leads to lesser congestion. Figure 9 also shows the percentage of incoming read requests bypassed by *ByE*, on the secondary *y*-axis. *ByE* is able to bypass on average about 37% of incoming read requests. On average, 23% read requests are to dirty lines in the cache which cannot be bypassed. The remaining 40% are false positives in our Bloom filter implementation which could have bypassed the DRAMCache. We discuss more on a sensitive study of Bloom filter later in this section. Further, the reduced set contention and lesser number of CPU requests in DRAMCache queues reduces congestion for the GPU, which in turn leads to small performance benefits for the GPU as well improving the harmonic mean of the GPU IPC by 3% over naive DRAMCache.

Combining *PrIS* with *ByE* allows for the non-bypassed CPU requests at the DRAMCache to be served with a higher priority and hence reducing the queuing delays. *ByE+PrIS* performs better than just *PrIS* by 10% for CPU and 7% for GPU. Overall, *ByE+PrIS* achieves 48.5% improvement in CPU performance over a naive DRAMCache while degrading GPU performance by just 3%.

The somewhat high false-positive ratio in our bypass mechanism is due to the large number of dirty blocks in the DRAMCache and the relatively small size of the Bloom filter. We also experimented with three hash-functions while also optimally increasing the array capacity to 312KB (20% larger) to reduce aliasing and increase the efficacy of bypass. However, the CPU performance improves by only 2.1% while the GPU remains largely unaffected.

5.3 Chaining for Spatial Occupancy Control

As discussed in Section 3.2.3, the *Chaining* mechanism improves hit rates for GPU while guaranteeing some occupancy for the CPU in the DRAMCache. We empirically determine a suitable low occupancy threshold of the CPU (l_{cpu}) to be 20% for all our workloads. Chaining alone performs no better than a naive cache as the queuing latencies overwhelm any improvements in hit rates. However, when chaining is coupled with *PrIS*, the increased hit rates reduce the performance drop caused by *PrIS* for the GPU from 10% to 5.2% (i.e., 4.8% performance improvement over *PrIS*). For the CPU, guaranteed occupancy in the DRAMCache and the secondary effect of lower congestion at off-chip DRAM allows CPU requests to be serviced with lower delays. This improves the performance of CPU by 7%, over only *PrIS*.

Overall, *Chaining+PrIS* improves CPU performance by 44.7% while degrading GPU performance by merely 7% over a naive DRAMCache.

5.4 System Performance with HASHCache

We now holistically examine the performance improvements due to the introduction of our HASHCache mechanisms, in CPU and GPU cores together. From Figure 8 we observe that, adding a naive DRAMCache can achieve an average of 42% and 24% improvement in CPU and GPU cores, respectively; whereas, HASHCache achieves significant speedups of (205%, 17.5%)

Fig. 10. (a) Harmonic mean of IHS with HASHCache. (b) Weighted speedup of IHS with HASHCache.

and (211%,20.4%) for (CPU,GPU) using heterogeneity-aware mechanisms of *ByE* and *Chaining*, respectively. This comes within 16% and 13% of the ideal no interference performance for the GPU (final gray bar in Figure 8(b)) and within 81% and 76% of the ideal no interference performance for the CPU (final gray bar in Figure 8(b)) for each of the schemes. Further, for memory intensive combination of CPU and GPU workloads like Qg7 and Qg8 which see significant degradation in the performance of both processors due to interference, adding a DRAMCache can improve performance up to 430% for CPU and 48% for GPU over a baseline system with no stacked DRAMCache. In comparison, the naive DRAMCache implementation only brings 55% and 56% improvement in the CPU and GPU performance, respectively.¹¹

Figure 10(a) plots the performance improvement as a harmonic mean of IPCs of IHS IPC (combined CPU cores and GPU CUs), normalized to our baseline IHS architecture without a DRAMCache. Overall, with simple heterogeneity-aware management of the stacked DRAMCache, IHS systems can achieve on average 200% (up to 400%) performance improvement while a naive DRAMCache is able to achieve just 41.8% improvement.

As a comprehensive system metric, Figure 10(b) plots the weighted speedup normalized to the baseline of IHS without DRAMCache. Adding a DRAMCache to IHS processors naively achieves an improvement of merely 3%. In fact, for some workloads like Qg3 and Qg11 adding a DRAMCache without careful considerations can lead to negative performance impact. However, our heterogeneity-aware mechanisms are able to achieve on an average of 16% and 15% improvement in performance over a IHS architecture without a DRAMCache. This improvement corresponds to a 12.9% and 11.5% improvement for each of the schemes over a carefully designed but heterogeneity-unaware DRAMCache for the IHS processors.

We also observe that in the case of a naive DRAMCache, on average 58.22% and 23.93% of the peak DRAMCache and peak off-chip DRAM bus bandwidth is utilized. With *PrIS+ByE*, the off-chip DRAM bus utilization improves by 23.8% with negligible reduction (less than 1%) of DRAMCache bus utilization. Further, with *PrIS+Chaining*, the DRAMCache bus utilization improves by 35.12% while off-chip DRAM bus utilization reduces by 2.64%.

5.5 Comparison with Related Work

The objectives of *ByE* and *PrIS* are similar to the Mostly Clean DRAMCache (MCC) [42] and Staged Memory Scheduling (SMS) [9], respectively. Hence, we compare our work with MCC (adapted for IHS architecture) and SMS in Figure 11 which presents the performance of (a) CPU and (b) GPU normalized to an IHS with no DRAMCache. Overall, *ByE* performs 7.8% better than MCC in terms of CPU IPC. This is because when a page (row) is marked as write-through in MCC, all requests to that page are bypassed oblivious of the source of the request. The large number of GPU requests

¹¹We note here that combining *PrIS*, *ByE*, and *Chaining* schemes is likely to improve performance. However, as the objectives of *Chaining* and *ByE* are somewhat conflicting, this requires careful consideration of replacement (fill) policies in the DRAMCache. We defer this to future work.

Fig. 11. Comparison of HASHCache against MCC [42] and SMS [9]. (a) CPU and (b) GPU.

Fig. 12. Sensitivity study with 128MB DRAMCache. (a) CPU and (b) GPU.

to the cache lines in the write-through (clean) pages quickly exhausts the available MSHR entries, which leads to the DRAMCache being blocked for subsequent requests. For workloads Qg9, Qg10, and Qg12 the GPU workloads are relatively less intensive and MCC tends to perform slightly better than *ByE*. The GPU performance is comparable for both approaches.

The striped bars in Figure 11 show the performance of the *PrIS* and *SMS* for (a) CPU and (b) GPU. For *SMS* implementation, we proportionally scale down the total hardware requirements to the same size as that of *PrIS* (128 entries). We observe that *PrIS* performs on-par with *SMS* for almost all CPU workloads. For workloads like Qg2 and Qg12, *SMS* performs marginally better than *PrIS* as it is able to better manage CPU inter-application interference using Shortest-Job-First (SJF) scheduling. However, for the GPU, *SMS* performs 4% better than *PrIS* due to its batching algorithm that is able to admit row buffer hit requests into the queues, which leads to improved row buffer hit rate (9% better). Our combined mechanism of *PrIS+ByE* proposed in this work performs better than each of the prior compared schemes for both CPU and GPU.

5.6 Sensitivity Study

5.6.1 Larger Capacity DRAMCache. We conduct a sensitivity study for HASHCache mechanisms with a larger 128MB stacked DRAMCache. Figure 12 presents the performance of a naive DRAMCache and our mechanisms *ByE+PrIS* and *Chaining+PrIS* for the (a) CPU and (b) GPU in terms of H-Mean of IPC normalized to an IHS with no DRAMCache. We observe that the larger naive DRAMCache is able to achieve 54.5% and 29.8% improvement for CPU and GPU processors, respectively. HASHCache mechanisms continue to provide an average improvement of 46% and 39% for the CPU, on average, above a heterogeneity-unaware DRAMCache, and 226% and 215% over the baseline IHS.¹²

5.6.2 Off-chip DRAM with Same Latency as DRAMCache. Although we expect the smaller sizes of stacked DRAMs to have lower latency than off-chip DRAMs, commercial stacked DRAMs like those in Intel's Knights Landing [7] have access latencies similar to off-chip DRAMs. Hence, we scale up the off-chip DRAM to a DDR3-2133-like device with latencies (13.09ns-13.09ns-13.09ns-33ns) for t_{CL} - t_{RCD} - t_{RP} - t_{RAS} . As before, Figure 13 presents the performance (H-Mean of IPC) of (a) CPU and (b) GPU with a naive DRAMCache and HASHCache normalized to an IHS system

¹²In the larger DRAMCache, Chaining+PrIS results in 15.8% improvement over baseline (10% decrease over naive); we expect the scheme to catch up the lost performance (over naive) in benchmarks with a larger GPU footprint.

Fig. 13. Sensitivity study with DDR3-2133 off-chip DRAM. (a) CPU and (b) GPU.

Fig. 14. Sensitivity study with 2x bandwidth for DRAMCache. (a) CPU and (b) GPU.

with no DRAMCache. For *PrIS+ByE*, CPU performance improves by 48% while reducing merely 3% of GPU performance. For *PrIS+Chaining*, CPU performs 47% better than a naive DRAMCache while decreasing GPU performance by 4.4%.

5.6.3 Higher Bandwidth DRAMCache. We carry out experiments for HASHCache with a stacked DRAMCache of 80GB/s. We achieve this by doubling the burst length of the stacked DRAMCache devices. Again, Figure 14 presents the performance of a naive DRAMCache and HASHCache for (a) CPU and (b) GPU. We observe that our mechanisms scale with increased DRAMCache bandwidth and *ByE+PrIS* performs 51% better and *Chaining+PrIS* performs 47.3% better than a naive DRAMCache for CPU. GPU performance also improves by 1.9% and 2.4% over a naive DRAMCache. Notably, for GPU benchmarks like streamcluster and Gaussian *Chaining+PrIS* performs better than *ByE+PrIS*.

6 RELATED WORK

State-of-the-art stacked DRAM device designs have primarily focused on improving the performance of multi-core architectures. In [16, 41], the stacked DRAMs are organized as part of memory in view of the large capacities provided by these devices. The designs propose hardware management schemes for swapping hot pages into and out of the stacked DRAM devices. On the other hand, several designs propose to use the stacked DRAM as transparent hardware-managed cache. Sectorized caches such as [26, 28] allocate large blocks and avoid wastage of bandwidth by intelligently fetching useful blocks only. In Section 3.1, we have broadly alluded to some of the works of DRAMCache designs and organizations [21, 25, 34, 39] that organize DRAMCache at smaller system-sized blocks (64B or 128B). These designs intelligently manage metadata and tag lookup serialization by using approximate hardware structures. HASHCache extends and adapts the simplicity and effectiveness of the Alloy Cache organization for IHS architecture after carefully considering the implications of each design decision on performance. While these works have focused on improving multi-core CPU performance, our work shows that there exists a large potential for performance improvements by using stacked DRAMCache in an IHS architecture.

Orthogonal to these efforts, there have also been proposals such as [37] to expose the stacked DRAM to the applications and/or system software by providing special allocation calls or using intelligent page management algorithms in system software to place hot data in the high bandwidth memory. Besides these designs incurring the obvious overheads of software modification

to improve performance, they also require a good understanding of program behavior and IHS architecture knowledge.

Recently, researchers have pointed out the need to utilize all the available bandwidth from both on-chip and off-chip DRAM devices due to comparable access latencies [15, 18, 20, 30, 42] to extract best performance and improve resource utilization. However, HASHCache uses the ingrained disparity in the request rates and their implication on the performance of each core to optimize bandwidth balance in a heterogeneity-aware manner.

Complementary to our work, there have been efforts to improve the performance of IHS systems in [29] by throttling the GPU cores using intelligent warp scheduling and avoiding congestion in the NoC [32]. These techniques are orthogonal to ours, which explores managing the interference in the memory subsystem of IHS architectures. To manage shared on-chip SRAM caches, Lee and Kim [31] propose heterogeneity-aware schemes that are built on top of UCP and RRIP schemes for managing shared resources. While our chaining mechanism is in line with this to ensure minimum occupancy for CPU requests, it also goes by introducing pseudo-associativity and improving hit-rates (specifically for GPU requests). Mekkat et al. [35] further propose shared SRAM cache management for IHS workloads that uses runtime metrics, like cache sensitivity of each workload, to allocate cache capacities. Despite larger capacities, DRAMCaches have higher latencies and hence will not be able to adapt quickly to SRAM occupancy management schemes proposed in these works.

Zhan et al. [47] propose improving the performance of IHS architectures by replacing on-chip SRAM caches with slightly larger STT-SRAM caches that are non-volatile but have asymmetric read/write energy and latencies. They focus on NoC-related optimizations through NoC reordering/batching schemes and differential CPU/GPU, read/write prioritization. The NoC optimizations are orthogonal and can be supplemented to the ideas proposed in this work. The performance improvement due to introduction of STT-RAM in IHS architectures is equivalent to that observed in our naive DRAMCache.

The authors in [27] propose a QoS-aware memory scheduler to avoid the GPU from missing a frame rendering deadline. However, in our IHS architecture the GPU is used to accelerate general purpose code and hence *PrIS* does not consider such deadlines for the memory controller.

In [48], the authors propose to use the GPU side stacked DRAM as another level of cache for the GPU memory hierarchy before going to the host memory. They use additional structures to maintain memory coherence information both on the CPU and discrete GPU side. However, complementary to our approach, this stacked DRAMCache on the discrete GPU is primarily used for caching data from the GPU cores while HASHCache is shared by both CPU and the integrated GPU throughout the execution. Several of our design decisions include tradeoffs which are heterogeneity-aware and provide improved access latency for CPU cores as well as high bandwidth for GPU cores.

Workload suites such as Chai [19] and Hetero-Mark [45] were designed to collaboratively engage both CPU and GPU cores simultaneously and take full advantage of the IHS architecture. The Chai benchmark suite was developed independently and concurrent to this work.

7 FUTURE WORK AND CONCLUSION

A detailed study of the energy efficiency of IHS systems with a stacked DRAM as cache is deferred to future work as this requires additional enhancements to the simulator owing to the heterogeneous nature of the cores. Intuitively, as a result of the significant speedups obtained using HASHCache the static energy dissipation of the system will be lower. The design and optimization of NoC and interconnect topology for IHS architecture is outside the purview of this work and

is also deferred as a future work. We would also like to explore the design of the virtual memory subsystem shared by CPU and GPU cores in an IHS architecture in the future.

In this work, we presented a case for performance improvement of an IHS processor by the addition of a stacked DRAMCache. We quantify the effects of interference due to co-running on each processor and show that the heterogeneity adversely affects CPU performance compared to the GPU. We carefully design an effective DRAMCache organization for IHS processors and improve IHS performance using three simple and effective heterogeneity-aware techniques: a DRAMCache scheduler, a spatial bypass, and a temporal occupancy control. HASHCache achieves significant improvement of 200% in overall system performance on average over a baseline system with no DRAMCache and 41% over a heterogeneity-unaware DRAMCache. This work thus shows that there are significant benefits to using a stacked DRAMCache for IHS processors, far exceeding the usefulness of such devices in homogeneous GPUs and multi-core CPU systems.

REFERENCES

- [1] Intel Corporation. 2015. The Compute Architecture of Intel Processor Graphics Gen9. Retrieved from <https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf>.
- [2] NVIDIA Corporation. 2017. CUDA C Programming Guide. Retrieved from <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- [3] NVIDIA Corporation. 2017. Fastest Processors, Smartphones, and Tablets - Nvidia Tegra. Retrieved from <http://www.nvidia.com/object/tegra.html>.
- [4] Benedict R Gaster and Lee Howes. 2011. The Future of the APU - Braided Parallelism, AMD Fusion Developer Summit 2011. Retrieved from http://developer.amd.com/wordpress/media/2013/06/2901_final.pdf.
- [5] HSA Foundation. 2016. HSA Foundation Specification Version 1.1. Retrieved from <http://www.hsafoundation.com/standards/>.
- [6] Intel Corporation. 2017. OpenCL Platform with Intel Graphics. Retrieved from <https://software.intel.com/en-us/node/540387>.
- [7] Intel Corporation. 2017. Intel XeonPhi Processor Datasheet. Retrieved from <http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-processor-x200-product-family-datasheet.pdf>.
- [8] Khronos Group. 2012. OpenCL 2.2. Retrieved from <https://www.khronos.org/opencl/>.
- [9] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, 416–427.
- [10] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2014. A logic-base interconnect for supporting near memory computation in the hybrid memory cube. In *Workshop on Near-Data Processing (WoNDP, MICRO-47)*. 1–6.
- [11] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die stacking (3D) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, 469–479. DOI : <http://dx.doi.org/10.1109/MICRO.2006.18>
- [12] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. DOI : <http://dx.doi.org/10.1145/362686.362692>
- [13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, Washington, DC, 44–54. DOI : <http://dx.doi.org/10.1109/IISWC.2009.5306797>
- [14] Arpit Joshi Cheng-Chieh Huang, and Vijay Nagarajan. 2016. DCA: A DRAM-cache-aware DRAM controller. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
- [15] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: Techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, 198–210. DOI : <http://dx.doi.org/10.1145/2749469.2750387>
- [16] C. C. Chou, A. Jaleel, and M. K. Qureshi. 2014. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12. DOI : <http://dx.doi.org/10.1109/MICRO.2014.63>

- [17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293. DOI : <http://dx.doi.org/10.1109/90.851975>
- [18] J. Gaur, M. Chaudhuri, P. Ramachandran, and S. Subramoney. 2017. Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 13–24. DOI : <http://dx.doi.org/10.1109/HPCA.2017.46>
- [19] Juan Gómez-Luna, Izzat El Hajjy, Li-Wen Changy, Victor García-Floreszx, Simon García de Gonzaloy, Thomas B. Jablin, Antonio J. Pena, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. IEEE, 43–54.
- [20] Nagendra Gulur, R. Govindarajan, and Mahesh Mehendale. 2016. MicroRefresh: Minimizing refresh overhead in DRAM caches. In *Proceedings of the 2nd International Symposium on Memory Systems (MEMSYS'16)*. ACM, New York, USA, 350–361.
- [21] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. 2014. Bi-modal DRAM cache: Improving hit rate, hit latency and bandwidth. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 38–50. DOI : <http://dx.doi.org/10.1109/MICRO.2014.36>
- [22] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 201–210.
- [23] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* (Sept. 2006), 1–17. DOI : <http://dx.doi.org/10.1145/1186736.1186737>
- [24] C. C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan. 2016. C3D: Mitigating the NUMA bottleneck via coherent DRAM caches. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12. DOI : <http://dx.doi.org/10.1109/MICRO.2016.7783739>
- [25] Cheng-Chieh Huang and Vijay Nagarajan. 2014. ATCache: Reducing DRAM cache latency via a small SRAM tag cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, 51–60. DOI : <http://dx.doi.org/10.1145/2628071.2628089>
- [26] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee. 2016. Efficient footprint caching for tagless DRAM caches. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 237–248. DOI : <http://dx.doi.org/10.1109/HPCA.2016.7446068>
- [27] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, 850–855. DOI : <http://dx.doi.org/10.1145/2228360.2228513>
- [28] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison cache: A scalable and effective die-stacked DRAM cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, 25–37. DOI : <http://dx.doi.org/10.1109/MICRO.2014.51>
- [29] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungrinun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2014. Managing GPU concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, 114–126. DOI : <http://dx.doi.org/10.1109/MICRO.2014.62>
- [30] Dongwoo Lee, Sangheon Lee, Soojung Ryu, and Kiyoun Choi. 2017. Dirty-block tracking in a direct-mapped DRAM cache with self-balancing dispatch. *ACM Trans. Archit. Code Optim.* 14, 2, Article 11 (May 2017), 25 pages. DOI : <http://dx.doi.org/10.1145/3068460>
- [31] J. Lee and H. Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*. 1–12. DOI : <http://dx.doi.org/10.1109/HPCA.2012.6168947>
- [32] Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. 2013. Design space exploration of on-chip ring interconnection for a CPU-GPU heterogeneous architecture. *J. Parallel Distrib. Comput.* 73, 12 (Dec. 2013), 1525–1538. DOI : <http://dx.doi.org/10.1016/j.jpdc.2013.07.014>
- [33] G. H. Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the 2008 International Symposium on Computer Architecture*. 453–464. DOI : <http://dx.doi.org/10.1109/ISCA.2008.15>
- [34] Gabriel H. Loh and Mark D. Hill. 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, 454–464. DOI : <http://dx.doi.org/10.1145/2155620.2155673>
- [35] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. 2013. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE Press, Piscataway, NJ, 225–234.

- [36] O. Naji, C. Weis, M. Jung, N. Wehn, and A. Hansson. 2015. A high-level DRAM timing, power and area exploration tool. In *Proceedings of the 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 149–156. DOI : <http://dx.doi.org/10.1109/SAMOS.2015.7363670>
- [37] Mark Oskin and Gabriel H. Loh. 2015. A software-managed approach to die-stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Washington, DC, 188–200. DOI : <http://dx.doi.org/10.1109/PACT.2015.30>
- [38] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. 2015. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Comput. Archit. Lett.* 14, 1 (Jan. 2015), 34–36. DOI : <http://dx.doi.org/10.1109/LCA.2014.2299539>
- [39] Moinuddin K. Qureshi and Gabe H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, 235–246. DOI : <http://dx.doi.org/10.1109/MICRO.2012.30>
- [40] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. 2007. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, 123–133. DOI : <http://dx.doi.org/10.1109/MICRO.2007.24>
- [41] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. 2014. Transparent hardware management of stacked DRAM as part of memory. In *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24. DOI : <http://dx.doi.org/10.1109/MICRO.2014.56>
- [42] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. 2012. A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–257. DOI : <http://dx.doi.org/10.1109/MICRO.2012.31>
- [43] Allan Snaveley and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, 234–244. DOI : <http://dx.doi.org/10.1145/378993.379244>
- [44] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.
- [45] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16)*. 1–10. DOI : <http://dx.doi.org/10.1109/IISWC.2016.7581262>
- [46] Thiruvengadam Vijayaraghavan, Yasuko Eckert, and Gabriel H. Loh. 2017. Design and analysis of an APU for exascale computing. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*.
- [47] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie. 2016. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. DOI : <http://dx.doi.org/10.1109/MICRO.2016.7783731>
- [48] Amir Kavayan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. 2016. UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs. *ACM Trans. Archit. Code Optim.* 13, 4, Article 35 (Dec. 2016), 25 pages. DOI : <http://dx.doi.org/10.1145/2996190>

Received June 2017; revised November 2017; accepted November 2017