Heterogeneity Aware Shared DRAM Cache for Integrated Heterogeneous Architectures

A Thesis

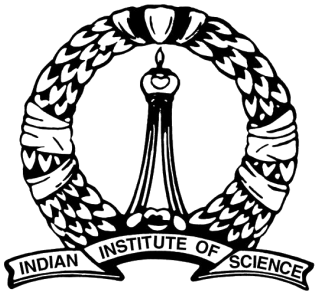
Submitted For the Degree of

Master of Science (Engineering)

in the Faculty of Engineering

by

Adarsh Patil



Department of Computer Science and Automation

Indian Institute of Science

BENGALURU – 560 012

APRIL 2021

DEDICATED TO

My Parents

and Kumudha

Acknowledgements

My experience at IISc has been nothing short of incredible and several thanks are in order.

First and foremost, I would like to express my deepest gratitude to my research advisor Prof. R. Govindarajan for being a supportive mentor and guide. He has been patient with me and has encouraged, taught and motivated me to think clearly and critically. His humbleness, knowledge and relentless work ethic will always be a constant source of inspiration to me.

I owe a deep sense of gratitude to everyone who have made my stay at IISc comfortable. A big thanks to the department chairman Professor Jayant Haritsa and Professor Uday Reddy whose backing and encouragement towards student system admin activities and their endorsement for compute infrastructure make the department well equipped to perform research. I am grateful to the staff at CSA office and the system admins Jagdish and Pushparaj who work tirelessly to keep the department and compute resources running. Thanks are also due for the funding sources - MHRD and AMD for supporting me financially to take care of my day-to-day expenses. I have greatly enjoyed courses by Professor Matthew Jacob and Professor Jayant Haritsa and their refreshing ways of course instruction.

I want to thank everyone in the HPC lab - Jayvant, Shilpa, Abhinav, Nagendra and Arth for providing a friendly and constructive atmosphere in the lab and for their useful feedback and insightful comments on my work. It was great sharing lab discussions and deliberating over problems and findings. My thanks also go out to my wonderful friends at IISc - Arvind, Vinay, Sridhar, Srinidhi, Anirudh, Pallavi, Sathya, Divya and several others who are not mentioned here. It was a pleasure sharing the many meals, coffee breaks, ice

cream outings and the lively banter and wise cracks all through my stay here. You always made sure there was never a dull moment and made my stint here truly memorable.

I would also like to thank Vikram, Archana, Aditya, Divyanshu and the members of IISc Running Club for organizing wonderful running events on campus and giving me a chance to represent the institute at the 12 hour relay run. It was wonderful being running buddies with all of you. I will always fondly remember my morning runs at this wonderful and serene campus which provided a harmonious environment throughout the year and has always reinvigorated and rejuvenated my spirits in silence.

I must express my profound gratitude to my parents for their unconditional support for all my decisions, their counsel and lending me a sympathetic ear when I needed it. It was my parents who first brought me to visit Indian Institute of Science back in my school days to visit our family friend Dr S C Pilli (PhD, Mechanical) and the campus left a lasting impression on me. Finally, I thank Kumudha who has always been there for me, be it for technical advice or moral support. Her unyielding love has been my source of strength. My gratitude also goes to Kumudha's parents who have been so supportive of us.

Thank you very much, everyone!

Publications based on this Thesis

1. A. Patil, R. Govindarajan. HAShCache: Heterogeneity Aware Shared DRAM Cache for Integrated Heterogeneous Systems. ACM *Transactions on Architecture and Code Optimization (TACO)*, Volume 14, Issue 4, December 2017. Presented at the Conference for High Performance Embedded Architectures and Compilers (HiPEAC) 2018, Manchester, United Kingdom.

Abstract

Integrated Heterogeneous System (IHS) processors pack throughput-oriented GPGPUs alongside latency-oriented CPUs on the same die sharing certain resources, e.g., shared last level cache, network-on-chip (NoC), and the main memory. They also share virtual and physical address spaces and unify the memory hierarchy. The IHS architecture allows for easier programmability, data management and efficiency. However, the significant disparity in the demands for memory and other shared resources between the GPU cores and CPU cores poses significant problems in exploiting the full potential of this architecture.

In this work, we propose adding a large capacity stacked DRAM, used as a shared last level cache, for the IHS processors. The reduced latency of access and large bandwidth provided by the DRAM cache can help improve performance respectively of CPU and GPGPU while the large capacity can help contain the working set of the IHS workloads. However, adding the DRAM cache naively leaves significant performance on the table due to the disparate demands from CPU and GPU cores for DRAM cache 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 DRAM cache. This necessitates a heterogeneity-aware management of this shared resource for improved performance. To address this, in this thesis, we propose three simple techniques to enhance the performance of CPU application while ensuring very little or no performance impact to the GPU. Specifically, we propose (i) *PrIS*, a prioritization scheme for scheduling CPU requests at the DRAM cache controller, (ii) *ByE*, a selective and temporal bypassing scheme for CPU requests at the DRAM cache and (iii) *Chaining*, an occupancy controlling mechanism for GPU lines in the DRAM cache through pseudo-associativity. The resulting cache,

HAShCache, is heterogeneity-aware and can adapt dynamically to address the inherent disparity of demands in an IHS architecture with simple light weight schemes.

We enhance the gem5-gpu simulator to model an IHS architecture with stacked DRAM as a cache, coherent GPU L2 cache and CPU caches and a shared unified physical memory. Using this setup we perform detailed experimental evaluation of the proposed HAShCache and demonstrate an average system performance (combined performance of CPU and GPU cores) improvement of 41% over a naive DRAM cache and over 100% improvement over a baseline system with no stacked DRAM cache.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The remarkable improvements 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 addition of specialized on-chip hardware circuitry for acceleration units [6]. A study in 2010 by Koomey et. al [40] found that the amount of computation that could be done per unit of energy doubled about every 18 months. However, to reach exascale and beyond requires a thousand fold decrease in energy consumed per FLOP computed. 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 of data parallel code at lower energy. GPUs have evolved into general purpose parallel processors allowing programming these devices using standard libraries and APIs such as OpenCL [12] and CUDA [2]. These APIs have allowed writing programs that execute across heterogeneous platforms. There has been considerable effort in recent years to develop GPU implementations of a wide variety of frameworks [53, 49] and parallel algorithms from High Performance Computing (HPC) and the enterprise domains [63, 45]. GPGPU architectures and compiler transformations have also evolved over several generations and are now able to efficiently execute programs with irregular branches and memory behaviour as well.

However, several hurdles remain to create a productive programming environment in

heterogeneous systems where the GPU accelerator is present as a discrete device. Traditionally these discrete GPUs have their own independent memory systems and the CPU must copy data to the GPUs memory and back. This data movement is wasteful and expensive in terms of both latency and energy. Further, as the transfer happens over a slower PCIe bus, the separate address spaces and the need to manage two sets of data complicates programming, impeding expansion of the workloads that benefit from GPGPU computing.

1.1 Integrated Heterogeneous Systems (IHS)

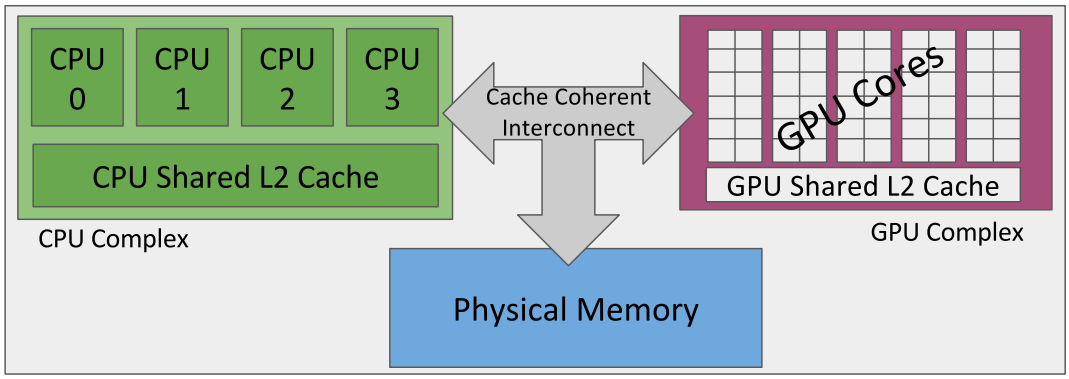


Figure 1.1: A High Level Schematic of IHS Architecture

Recently, modern processor chips (e.g., AMD APUs [4], Intel Iris [7], and NVIDIA Denver [3]) have heterogeneous processors which integrate CPU and GPU on the same die as shown in Figure 1.1. In doing so, these IHS architectures address some of the obstacles for programmability of these GPU devices. Specifically, the IHS architectures support a fully shared virtual and physical memory address space. This greatly simplifies programming as it provides the following features.

- Discrete GPUs required programmers to identify data that would be operated on by the GPU. These would then have to be explicitly copied to GPU memory. IHS architectures feature a physical memory shared with GPUs. Hence, the GPU now touches only the data required by the application, removing the programmer burden of identifying the required data structures.

- In discrete GPUs, operating on data structures with pointers such as linked lists, trees, graphs etc. required special handling by the programmer (such as deep copy) which impeded porting applications to GPU. With IHS, GPU can dereference and process pointers shared with CPU. Thus, the GPU can operate on the same application data structures as the CPU.

- In discrete GPUs, when CPUs and GPUs access the same memory region [13], the graphics driver is required to flush and invalidate the GPU caches in order for CPU and GPU to share results. However, IHS provides a fully coherent shared memory similar to multi-core CPUs. This enables authoring applications of the produce-consumer paradigm that closely couple the CPU and GPU executions.

- Traditional discrete GPU have strictly limited support for multi-process offloading. They either resort to hard partitioning of GPU cores or serialize executions of programs. IHS processors allow for Read/Write protection to be enforced by MMU units and TLBs. This allows support for preemption and context switching to maintain high throughput by co-scheduling in a multi-process environment.

- GPU can invoke operating system kernel routines for demand paging and page faults with much lesser overhead [51]. This provides the benefit to be able to run GPU programs whose dataset sizes are not constrained by the memory size.

- The HSA foundation [5] was setup to develop and define cross-vendor hardware specifications and software development tools needed to allow applications software to better use IHS architectures. HSA provides additional features such as the runtime system on IHS architectures. The HSA runtime provides support for "platform atomics" like Read Modify Write (RMW) operations as opposed to the atomics provided on a discrete PCIe based GPU which were restricted to PCIe provided atomics.

- Recent HSA runtimes [19] have proposed user-level command queuing. This feature allows user mode process to dispatch directly into those GPU queues, requiring no OS kernel switching or driver overheads. This greatly reduces GPU initialization time.

The above benefits enable several high level languages [9, 11] to also take advantage of the parallel processing synergistically with the CPU. Programmers can now write applications that seamlessly integrate CPUs with GPUs while benefiting from best attributes of each. Fine-grain parallel stream processing like face detection, compression, encryption-decryption etc., can now use the integrated GPGPU to deliver better performance.

To summarize, integrating the GPUs on chip reduces the GPU programming barrier, allows for low overhead communication between CPU and GPUs, improved energy efficiency, low overhead coherence and synchronization between CPU and GPU and thus improved performance.

1.2 3D Die-Stacked DRAM

In contrast to the processing capabilities of the modern microprocessor, DRAM memory speeds have not kept pace commensurately to serve the increasing demands of the processors. As this gap widens and processors get more capable by packing faster and larger numbers of cores, their performance will be limited by the ability to feed information into them. The speed imbalance coupled with a limited growth in pin counts has led to the memory and bandwidth wall [66, 56] for off-chip DRAM systems which often become a performance limiting factor. Newer emerging memory technologies like 3D die-stacking of DRAM provide alternatives to address some of these challenges. These devices provide large number of channels using TSVs (though-silicon-via) and have lower signalling delays. These die stacked DRAMs are touted to play an important role in allowing the scaling of multi-core and IHS processors. However, to be effective, these DRAMs require careful architecting of the plethora of design parameters. These would include but are not limited to design decisions made in the context of traditional off-chip DRAM organizations that would require to be revisited to factor the architecture of these 3D die-stacked DRAMs. These design decisions are also often interlinked with workload characteristics and processor architectures.

The advent of die-stacking technology [17] provides a way to integrate disparate silicon

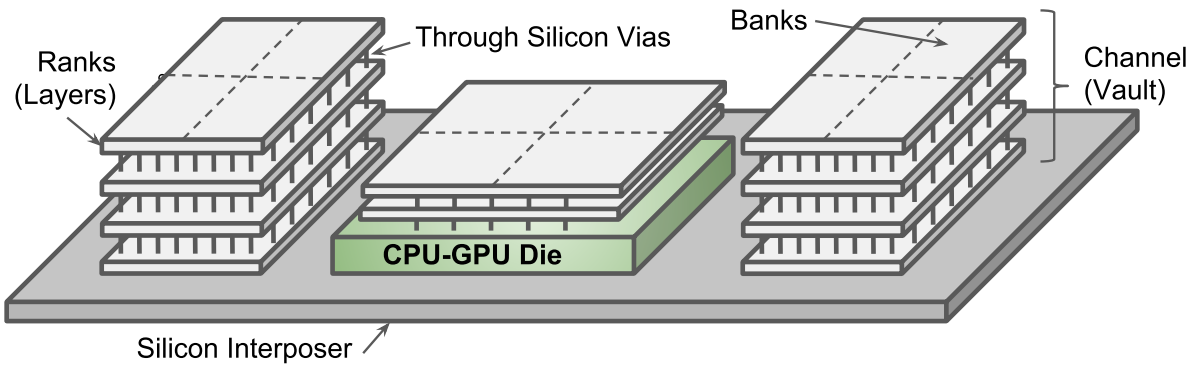


Figure 1.2: Proposed Architecture of DRAM Stacking

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 1.2. This allows the addition of a sizeable 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 [8]. The better interconnect also lowers the latency of access compared to off-chip memory [54]. Several independent studies regarding roadmaps to exascale computing [58, 65] also concur that die-stacked memory technology is a necessary cog to achieve the exascale bandwidth requirements with the targetted energy efficiency.

Principally, there have been two schools of thoughts to organize stacked DRAMs in the memory hierarchy. The first is to use it as a part-of-memory [59, 22]. These organizations deal with issues relating determination and appropriate allocation/placement of data in the stacked DRAM. Stacked DRAMs as part-of-memory allow using this capacity to increase the physical addressable memory available to the processor. Operating systems memory management and/or various software policies can also be used to allocate and page data in and out of the stacked DRAMs [48].

The second approach advocates the use of on-chip DRAM capacity as a hardware managed last level cache for improving performance of multicore CMPs [54, 28, 44, 33, 35]. This also reduces energy consumed per access for the overall system. However, stacked DRAM require careful designing since the large cache size (in the order of hundreds of

megabytes to few gigabytes) implies the total size of tags associated with it can also be quite large (order of megabytes). The large tag storage requirement has created a space/-time trade-off problem. On the one hand, we would like the latency of a tag access to be low as it contributes to both the hit latency and miss latency and hence store it in a SRAM structures (tags-in-SRAM). This SRAM storage space becomes prohibitive. Accordingly proposals have suggested storing tags in the stacked DRAM (tags-in-DRAM) alongside the data and to make it practical these schemes propose compound access [44] or multiple bursts [54] to retrieve tags efficiently. To reduce miss latency, these works propose predictors with a few kilobytes of SRAM overhead.

1.3 HAShCache: A Stacked DRAM Cache for IHS Architectures

To improve the memory subsystem of IHS and thus enhance its performance, in this thesis, we introduce a large capacity stacked DRAM, used as a hardware managed cache, as the first level shared resource in the memory hierarchies of CPU and GPU cores. There have been studies and proposals to share the on-chip last-level SRAM caches [47, 41] and non-volatile STT-RAM caches [67] in an IHS architectures. The work in HeLM [47] and TAP [41] address the issues of SRAM cache partitioning strategies between these processors in IHS architectures. OSCAR [67] focuses on optimizations for non-volatile STT-RAM based caches which have asymmetric read/write latencies and network-on-chip optimizations for IHS architectures. To the best of our knowledge this is the first work that architects stacked DRAM caches for IHS architectures.

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. Although adding the DRAM cache naively to IHS architecture improves performance, yet it leaves significant performance on the table. The reason for this is the disparate demands from CPU and GPU cores for DRAM cache and memory accesses. When a GPU kernel is launched it creates a large number

of concurrent threads which run in lock-step SIMD execution model and sends a large number of requests into the memory hierarchy causing congestion. This causes bottlenecks in request queues at the DRAM cache thus severely hampering CPU performance. This imbalance can significantly reduce the performance benefits that the CPU cores would have otherwise enjoyed with the introduction of the DRAM cache, necessitating a heterogeneity aware management of this shared resource for improved 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 DRAM cache 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 DRAM cache 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 behaviour as well as service requests for the CPU with a consistent latency without idling resources while delivering improved system throughput at lower energy.

Our organization, HAShCache, is aware of the asymmetric and contrasting requirements from the heterogeneous processors. HAShCache attempts to meet CPU 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 DRAM cache bandwidth. In the common case when CPU is running alone, HAShCache is an aggressive direct mapped DRAM cache optimized for hit times and lower miss penalty through hit/miss prediction as in [54].

However, when GPU is active we propose (i) PrIS - a DRAM cache 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 DRAM cache or data that is currently not in DRAM cache) to be temporally bypassed to utilize the under-utilized off-chip memory bandwidth (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 DRAM cache. To evaluate HAShCache, we extend the gem5-gpu [50] by adding the stacked DRAM as a memory-side hardware managed cache.

Using detailed simulations, we show that the techniques proposed in the thesis help to effectively address the disparate demands of IHS processors and improve performance of both CPU and GPU cores by 111% and 20% respectively, over a IHS processor with no DRAM cache. Further the proposed optimizations improve overall system performance, on an average by 41% over a carefully designed, but heterogeneity unaware DRAM cache.

Further, we show that our solution, HAShCache, is robust and performs well even with a wide variety of system configurations. HAShCache performs better with larger capacity DRAM cache and latest generations of aggressive off-chip DRAMs compared to a naive DRAM cache. We also show that HAShCache scales well when operating with projected stacked DRAMs of higher bandwidths, outperforming a naive DRAM cache for CPU applications and GPU applications. We also show that HAShCache performs better than the state-of-the-art DRAM scheduling scheme for IHS architectures i.e., Staged Memory Scheduler [14] and state-of-the-art DRAM cache bypass technique - Mostly Clean DRAM cache [60].

To summarize, the thesis makes the following contributions:

- We quantify the impact of a shared physical memory in IHS architectures and motivate the need for a better memory subsystem. We argue that a die stacked DRAMs, used as a hardware managed cache, can be a suitable fit for the memory system requirements of IHS architecture. Further, we show that a naive heterogeneity unaware DRAM cache can be suboptimal.

- We design an effective DRAM cache organization that is aware of this inherent heterogeneity in IHS processors. Besides tailoring the organization of the DRAM cache itself, we propose 3 heterogeneity aware mechanisms, *PrIS*, *ByE* and *Chaining* to improve the performance of the DRAM cache.

- To evaluate HAShCache, we develop simulator modules and extend gem5-gpu simulator to simulate IHS architectures with stacked DRAM as cache.
- We demonstrate using detailed simulation of co-running CPU and GPU workloads that, compared to a naive DRAM cache, the proposed HAShCache solution significantly improves the performance of CPU cores albeit with a small reduction in performance of GPU cores. Further, HAShCache outperforms a naive DRAM cache on all IHS system performance metrics.
- We show that HAShCache is robust and scales well even for projected memory system configurations. We also compare performance of our HAShCache against state-of-the-art DRAM/DRAM cache proposals, namely Staged Memory Scheduling [14] and Mostly Clean DRAM cache [60] and demonstrate that HAShCache outperforms these schemes.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present the necessary background and motivation of this work. Chapter 3 details the design and mechanisms of HAShCache. Chapter 4 elucidates our simulation infrastructure, the workloads used for evaluation, methodology and the IHS performance metrics used to evaluate the various schemes proposed. In Chapter 5, we report performance of HAShCacheand demonstrate its sensitivity and scaling properties. We also compare HAShCache mechanisms by adapting some of the state-of-the-art in the area. Chapter 6 provides a discussion of the related work in DRAM caches, IHS architectures and the IHS Benchmarks. Chapter 7 summarizes our work and notes a few potential future directions.

Chapter 2

Background and Motivation

In this chapter, we first present the necessary background for the rest of the work. We then state the terminology used in the rest of the thesis. Lastly, we present the motivation for the work.

2.1 Background

In this section, we describe the necessary basics on which this thesis is based. This includes overview of IHS architectures, the basic structure and working of DRAM devices and lastly the organization of stacked DRAM caches.

2.1.1 IHS Architecture

A key feature of IHS is the unified memory model. Figure 2.1 illustrates a typical IHS architecture. It consists of multiple CPU cores with private L1 and shared L2 caches (shared among CPU cores). Alongside are multiple GPU cores or Compute Units (CUs) with private L1 and shared L2 cache (shared among GPU cores). In addition the IHS architecture features the following components.

- Shared page table: IHS allows the CPUs and GPUs to operate in a globally shared address space. Thus, address translations are to be shared across CPU and GPUs.

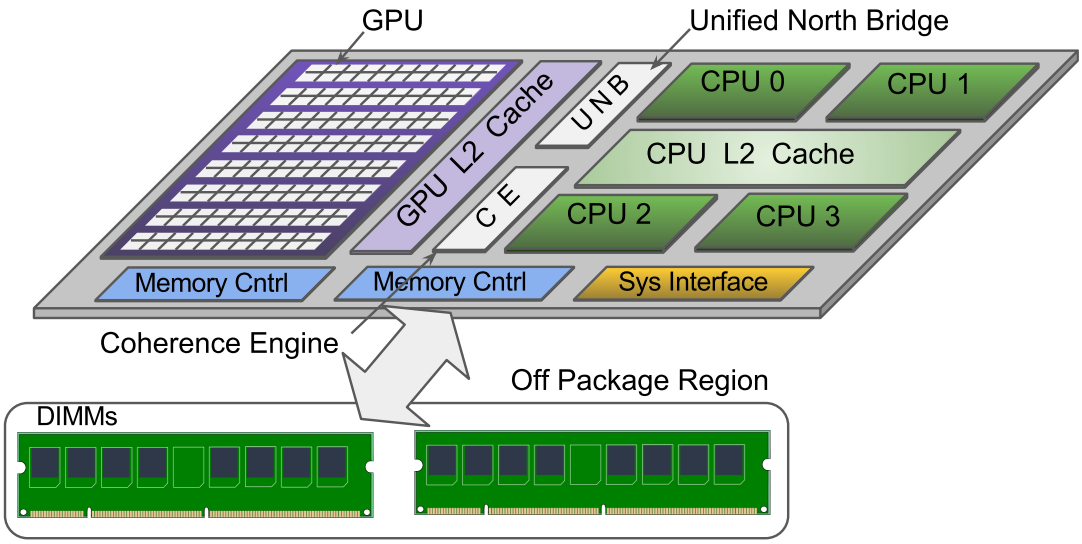


Figure 2.1: Architecture of an Integrated Heterogeneous System

Consequently, TLBs and page table walkers on CPUs and GPUs utilize a single page table in memory.

- Coherent caches: IHS architecture provides a fully coherent shared memory model between CPUs and GPUs as opposed to discrete GPUs where the driver, at the request of the programmer, must flush and invalidate the caches at required intervals in order for the CPUs and GPUs to share results. In IHS architecture, the cache coherence mechanism between the CPU caches and GPU caches allows the processors to read updated values immediately. This greatly simplifies the programming paradigm and reduces the programmer burden. It also enables fine grained offloading and efficient work sharing between the processors. Coherence is achieved using a region based Heterogeneous Coherence protocol similar to the proposal in [52].

- Shared Physical Memory System: In IHS architectures CPU and GPU processors share a single physical address space. Thus, the cores share memory controllers, channels and the physical memory devices themselves. This also avoids having to explicitly copy data to a different GPU memory as in discrete GPUs.

- Shared Network-on-Chip (NoC): A common NoC connects the CPU and GPU cores

with the caches and the memory controllers.

As IHS platforms gain widespread adoption in HPC systems, improving the performance of these platforms will become of paramount importance in the near future. [58, 65].

2.1.2 DRAM Fundamentals and Operation

Dynamic Random Access Memory (DRAM) forms the backbone of main memory in modern processors. DRAMs provide relatively large, fast and cost effective storage capacity to be used as main memory. In this section, we first describe the basics and operation of DRAM devices.

A DRAM stores information in a cell, which is a transistor-capacitor pair [34]. A DRAM device organization contains banks where each bank contains a 2D array (grid) of storage cells. A row decoder circuitry decodes the row address and activates the transistors of the corresponding row, thus reading the data out of the coupled transistors into data lines (*row activation*). A row of sense amplifiers (called row buffers) is used to sense this charge and hold it temporarily for purpose of transferring the data over the data bus. Subsequent requests to columns in this activated row are served from the row buffer. A *column access* command transfers a number of columns (consecutive bytes) to/from the row buffer. Since the read from the DRAM array is destructive i.e., the charge on the capacitor in the cells is lost, a *precharge* operation writes back contents from the sense amplifiers back to the corresponding row.

In order to improve efficiency of the DRAM systems, DRAM devices are organized hierarchically. Banks are further organized into ranks which share command buses but have separate data buses. A chip select is used to select a bank to issue a command. The banks in a rank operate in parallel and serve data independently to match the data bus width. A typical memory system would consist of multiple channels with completely independent DRAM devices having separate data and address buses. A DDRx (double data rate) DRAM transfers data on both edges of the bus clock (rising and falling). DDR increases the n-bit prefetch operation in each successive generation of DRAMs from 4n in DDR2 to 8n in DDR3. Here the "8n-prefetch" means that at each cycle a DDR3 DRAM transmits 8 bits of

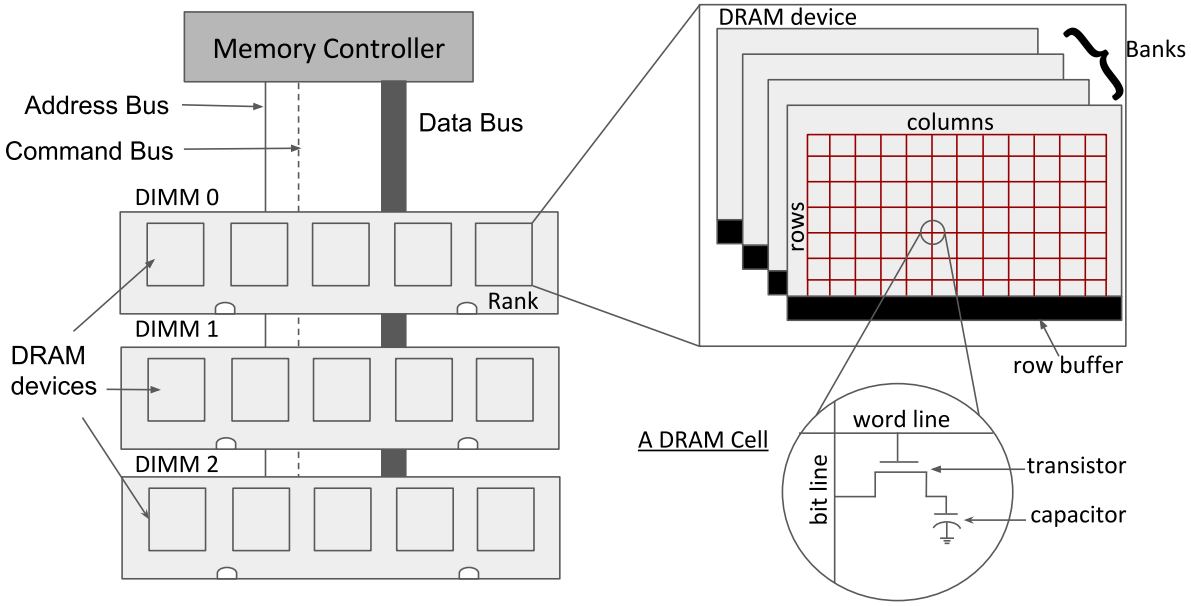


Figure 2.2: Organization of a DRAM Memory System

information from internal DRAM banks into the I/O buffers before sending them out. In other words, the minimum number of bits a DRAM column read/write performs in each operation for a DDR3 DRAM is 8 bits. This effectively increases the bandwidth without changing the core of the DRAM.

Each operation on the DRAM takes a certain fixed number of cycles. The timing cycles for the primary operations of row activation, column read, row precharge on the DRAM are denoted as t_{RCD} , t_{CL} and t_{PRE}. There are also several other timing parameters of DRAM circuits that also need to be respected during the operation of DRAMs due to the electrical constraints imposed by them. For example, t_{WR}: write recovery time for the written data to propagate into the DRAM arrays, t_{WTR}:write-to-read turnaround time, t_{CCD}: minimum column-to-column command spacing to be adhered, t_{FAW}: a rolling time-frame in which a maximum of four row activations on the same DRAM device can be engaged, t_{RRD}: a minimum row-to-row activation spacing to be maintained between two row activations and several others as specified by JEDEC [10]. The above timing constrains are complicated by the "dynamic" nature of these devices i.e., the transistors storing the charge in the DRAM are not perfect and leak charge, requiring periodic refresh. This also limits the read and

write operations that can be performed on DRAM devices. Further, two widely accepted factors that impact DRAM memory system performance are

- Row Buffer Hit (RBH): When a column access is performed to a row which is currently in the row buffer, a single column access is required to retrieve data. Thus, such an access would incur a single t_{CL} access latency without requiring any row operations. This scenario is referred to as a Row buffer hit and the corresponding parameter for a stream of accesses is known as the row buffer hit rate. Incurring row buffer hit would reduce the service time for the request.

- Bank Level Parallelism (BLP): Requests to independent banks can proceed in parallel and thus the temporal overlap can help hide the additional latencies incurred. Thus, if the requests are spread over banks the memory access latency can be amortized using this temporal overlap.

To achieve best efficiency from the DRAM requires careful scheduling of requests to extract maximum parallelism while respecting the timing constrains of the DRAM. A memory controller controls and coordinates the operation of the DRAM device while respecting timing constrains and bus conflicts. The memory access scheduler picks a request to service from read or write queues. Reordering requests allows memory controller to schedule requests to available banks while other requests wait for access. However, reordering does not compromise correctness as requests to the same address are coalesced but performed in the order that the accesses were received. For the memory access scheduler, performance is determined by the following parameters

- Waiting time: Time spent by the request waiting in the queue.

- Service time: Time taken by the request at the head of the queue to be serviced.

A popular DRAM access scheduler is the first-ready, first-come-first-service (FR-FCFS) scheduler [55]. The FR-FCFS scheduler prioritizes requests that will be a hit in the open row of DRAM banks over other requests i.e., row buffer hit requests are prioritized. If no request is a row-buffer hit, then FR-FCFS prioritizes older requests over younger ones.

Commercially, DRAMs are available as DIMMs (Dual in-line memory module) containing several DRAM devices with a standard interface. In this work, we model a JEDEC DDR3 DRAM [10] as the off-chip memory device.

Subsequently, in Chapter 3, we revisit these DRAM performance parameters to evaluate suitable DRAM addressing schemes for the new family of die stacked DRAMs. We also develop a DRAM memory access scheduler that factors in the idiosyncrasy of the requesting processor.

2.1.3 Stacked DRAM as Hardware Managed Cache

As alluded to earlier, DRAM caches require careful design to be effective. Traditional SRAM caches provide capacities of upto few tens of megabytes, have fast access times but expend higher energy and also have higher circuit costs. DRAM caches on the other hand, provide much larger capacity (several hundred megabytes to a few gigabytes) at lower power and higher density. However, they are constrained by access times. Logically, the working of DRAM caches is very similar to that of SRAM caches. Thus, cache design parameters like cache block size, set-associativity, miss penalty etc. remain pertinent parameters even in DRAM caches. However, some design decisions require to be re-examined in context of the DRAM cache characteristics.

Due to large capacity, DRAM caches have a large amount of metadata. This metadata includes tags, valid, dirty and replacement bits. This metadata size is of the order of megabytes even for reasonably sized DRAM caches. To illustrate this problem, consider a 256MB DRAM cache with 128B cache block size and with metadata of 8B per cache block. This DRAM cache organization incurs a metadata overhead of 16MB. Even with large granularity cache block size, as DRAM cache capacities increase, metadata continues to incur similar overheads. Considering a 1GB DRAM cache organized at 1KB cache block size still incurs a metadata overhead of 8MB.

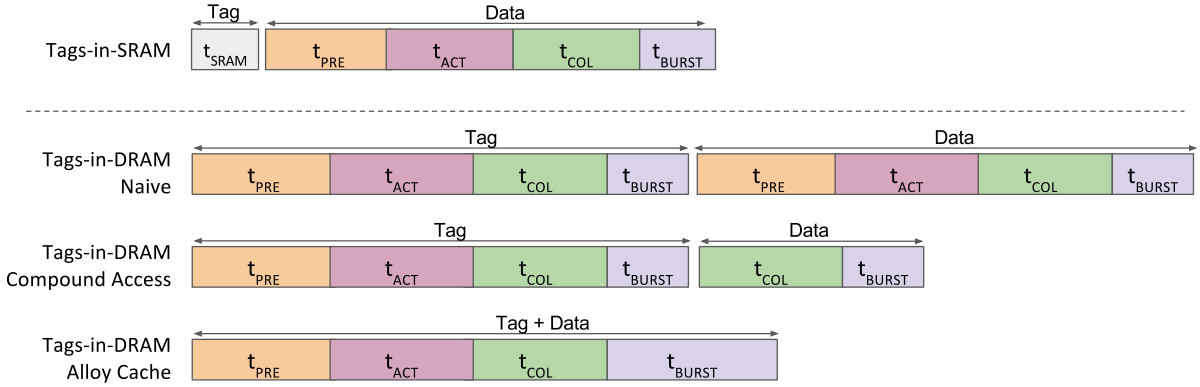


Figure 2.3: Access Latencies for Various Organizations of DRAM cache assuming a Tag Hit

Metadata Management and Access Latencies

Metadata management is critical in caches as this is on the critical path of all cache requests (hit or miss). There have been 2 design approaches to tackle the metadata issue. With the running example of a cache block hit we explain the latencies incurred by each approach and accordingly Figure 2.3 shows the timeline and sequence of events for each approach. The corresponding operation for a cache block miss would only perform the tag retrieval without the data retrieval. The first proposal called *Tags-in-SRAM* proposes to store metadata in SRAMs [35]. This proposals allocates blocks at large granularity and stores metadata in SRAM. However, large blocks waste off-chip bandwidth due to fetching of unused data. Nevertheless, metadata and tag lookup is fairy fast due to SRAM structure employed as shown in Figure 2.3(a).

The second set of proposals called *Tags-in-DRAM* [44, 54, 28, 33] store metadata within the DRAM cache. This incurs no additional SRAM costs but are subject to higher access latency since the tags have to be read from the DRAM cache. Figure 2.3(b) shows the access latencies for a naive *Tags-in-DRAM* implementation. Here there is a separate DRAM cache access for the tag and data. Each of these accesses would incur precharge (t_{pre}) and row buffer activation (t_{act}) latency, if the request row of the bank is not already in the row buffer, followed by column access (t_{col}) and data transfer (t_{burst}). Further, a *Tags-in-DRAM* organization can be optimized by placing tag and data of a set in the same row of the DRAM cache. This organization, called compound access [44], allows tag to be accessed

first from the DRAM cache and if the tag matches, the data is fetched from the matched way. Since the tag and data are stored together in the same row in the stacked DRAM, the data access will always be a row buffer hit Figure 2.3(c).

A small metadata cache [33] or predictor [44] structure is usually employed for *Tags-in-DRAM* organization to reduce DRAM cache lookups for tags. In such an organization, whenever the metadata cache contains the metadata the access would be similar to Figure 2.3(a); otherwise, it would be similar to Figure 2.3 (b) or (c) depending on whether it follows a naive or compound organization.

Alloy Cache

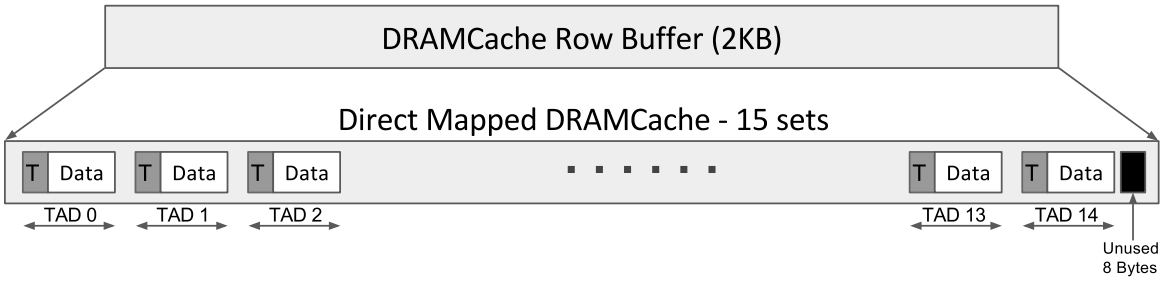


Figure 2.4: Row Buffer Layout of Alloy Cache adapted for Cache Line size of 128 Bytes

One particular proposal, we detail here is the organization of Alloy Cache proposed by Qureshi and Loh in [54]. Alloy Cache was designed and evaluated for multicore CPUs and is a latency optimized *Tags-in-DRAM* design. Alloy Cache allocates blocks at small cache block granularity (128 bytes) [1] and stores metadata for the blocks within the DRAM cache. However it is able to provide close to SRAM-like access latency to metadata by *alloying* the cache and data into blocks called TADs (Tag-and-Data). The cache is organized as a direct mapped DRAM cache and stores 15 sets in a DRAM cache row of size 2KB as shown in Figure 2.4. As a result some negligible capacity (8 bytes) of the DRAM cache row is unused.

[1] Alloy Cache was proposed with 64 byte cache line size and stored 31 sets in a DRAM cache row. We simply extend and adapt the organization for 128 byte cache line size. This was done as in our simulation setup, higher level caches (CPU and GPU L1 and L2 caches) operate at 128 byte cache line granularity. Further, we also increase the burst size to retrieve the larger 136 byte TAD (128+8 byte) instead of the 72 byte (64+8 byte) TAD.

When a cache line is probed, Alloy Cache uses a larger burst size [2] to retrieve the 136B TAD (128B data + 8B metadata) in a single DRAM cache access. Thus, a cache hit/miss decision can be made when the TAD is retrieved. In case of a hit, the data retrieved is used else the data is discarded. The Alloy Cache achieves access latency for the DRAM cache as shown in Figure 2.3(d). To further optimize the total observed memory access time, the Alloy Cache incorporates a Memory Access Predictor, called *MAP-I* predictor [54] for short, which is based on the address of the L2 miss causing instruction. The MAP-I predictor uses an array of counters hashed with the folded-XOR of the program counter to predict a DRAM cache hit/miss. They observe that such a predictor shows a good correlation between DRAM cache hit/miss information and the instruction address. MAP-I achieves a predictor accuracy of close to 95% at a cost of metadata overhead of a few bytes. Since a predictor is not perfect, they use this prediction information to start early access to off-chip memory when the predictor predicts a miss for an access to the DRAM cache. This is referred to as the Parallel-Access-Model (PAM) and this enables Alloy Cache to hide the miss latency for DRAM cache by overlapping the off-chip memory access time with the DRAM cache access time.

2.2 Terminology and Keywords

We describe some of the terminology used in the rest of this thesis.

- IHS - Integrated Heterogeneous Systems - refers to the platforms which have a GPGPU integrated on chip alongside the multi-core CPU. As shown in Figure 2.1, these processors have a cache coherent interconnect between the CPU and GPU and share a unified physical memory.

- HeA - Heterogeneous Applications - refers to the workload where multiple single

[2] A larger burst size requires the DRAM cache to have a non-standard interface (that does not conform to JEDEC standards). Since stacked DRAMs are tightly integrated with the processor package, unlike off-chip DRAMs, vendors can provide such customized interfaces to reduce latency of access.

threaded CPU benchmarks and a single GPGPU application are co-run in a IHS architecture.

- HoA - Homogeneous Applications - contextually refers to either (multiple single threaded) CPU benchmarks that run on a multi-core CPU architecture (HoA CPU) or a single GPGPU application run alone on one CPU and multiple GPU CUs (HoA GPU).

- DRAM cache - vertically die stacked DRAM, used as a hardware managed cache.

- D$ - Refers to the die stacked DRAM cache as discussed above (disambiguation against data cache).

- HAShCache - Heterogeneity Aware Shared DRAM cache for Integrated Heterogeneous System - Refers to the set of all designs and mechanisms that constitute our DRAM cache.

We refer to OpenCL [12] and CUDA [2] terminology interchangeably in the rest of this document.

2.3 Motivation

The IHS architecture integrates two different classes of compute devices i.e., latency oriented CPU and throughput oriented GPUs on a single chip sharing a physical memory. In this section, we motivate the need for better memory class devices that can cater to the requirements of IHS architectures. Specifically, we show that the die stacked class of DRAMs are a good fit for IHS architecture since they provide the necessary bandwidth and latency capabilities to deliver data to the processing units. However, managing contention in these DRAMs for a heterogeneous processor architecture which have asymmetric sensitivity and demands introduces novel challenges.

2.3.1 Memory Subsystem in IHS Architecture

We first analyse the impact of CPU cores and GPU cores co-running in an IHS architecture. We use the gem5-gpu [50] simulator for this exercise. We follow experimental methodology as described in Section 4.3. The IHS has 5 CPU cores with private per CPU L1 caches (32KB I/D split) and CPU shared L2 cache (1MB) along with 8 GPU CUs with per CU private L1 cache (64KB) and GPU shared L2 cache (512KB). The shared off-chip memory is modelled after a traditional DDR3-1600 like memory, similar to the one in modern multi-core systems.

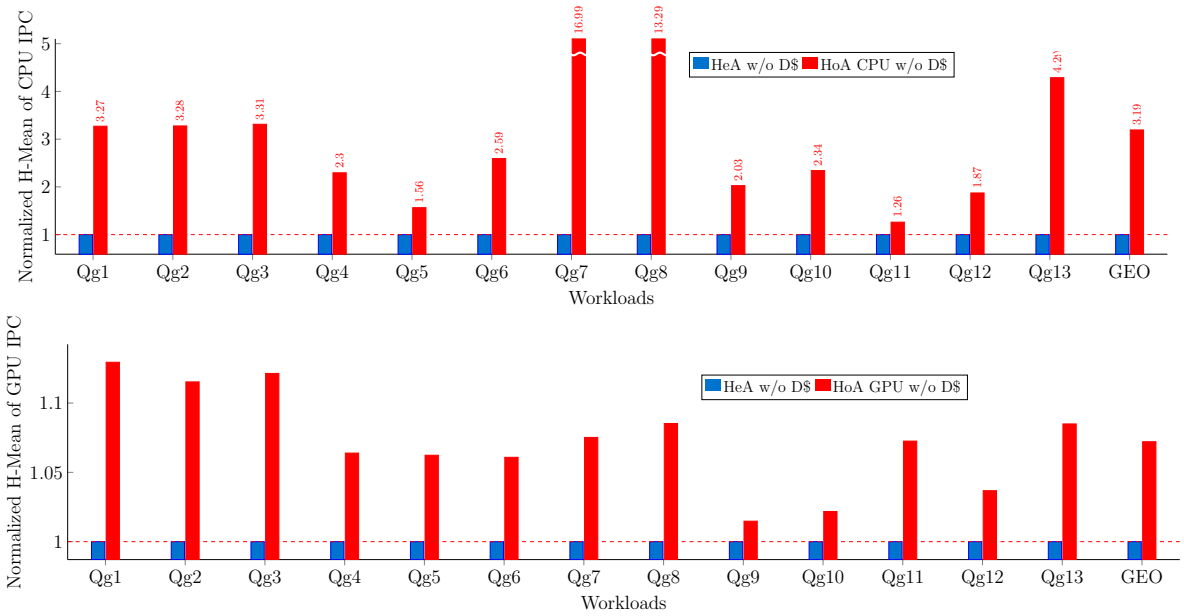


Figure 2.5: Effect of Interference in the IHS architecture due to the co-running on (a) CPU (b) GPU

Figure 2.5 shows the performance of (a)CPU and (b)GPU in terms of Harmonic Mean of IPC for HoA and HeA applications running on an IHS without a DRAM cache. We observe that the HoA CPU applications perform 3.19x better than HeA applications in an IHS architecture. On the other hand, the performance of HoA GPU applications is only marginally better by 7% compared to the on-chip GPU running HeA applications in the IHS. This shows that applications running in an IHS suffer performance losses as compared when they are run homogeneously. The loss in performance is more significant for the CPU applications

as compared to the GPU applications in an IHS.

When a GPU kernel is launched it creates large number of concurrent threads which run in lock-step SIMD execution model and sends a large number of requests into the memory hierarchy. The memory system does not provide sufficient bandwidth to cater to these large number of requests, causing congestion and 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. The GPUs hide the latency of memory access using interleaved scheduling of hundreds of concurrently running threads while CPU designs have a limited size reorder buffers (typically 128 entries) which is exhausted quickly leading to the processor stalling. This jeopardizes the overall performance of the IHS, quickly diminishing the benefits provided by integration of the on-chip GPU. This motivates the need for better memory devices that can cater to the requirements of IHS architectures.

We note here that similar observations regarding the performance reduction of co-running CPUs and GPUs in IHS as compared to their HoA performance have been made earlier in the works of [47, 67]. However, these works focus exclusively on issues of shared SRAM cache management, network-on-chip management and non-volatile STT-RAM cache management for IHS architectures.

Large computational capabilities of the IHS architecture require proportionately capable and aggressive memory system to deliver data to the computation engines as demanded. Current DRAM memory systems like DDR3, DDR4 etc. are designed for multi-core CPUs and provide small number of channels (typically 2 to 4), yielding a total memory bandwidth of a few tens of GB/s. On the other hand, on-chip SRAM caches can only provide capacities of a few tens of MBs which will not be able to contain the working set requirements of IHS architectures. Even with non-volatile STT-RAM caches the device characteristics introduce challenges in the operation and the die sizes restrict the capacities that they can provide. Hence, we infer that the memory hierarchy needs be revisited to support IHS architectures effectively.

2.3.2 Addition of Die-Stacked DRAM cache to IHS

To avoid the performance degradation due to co-running in IHS we propose addition of a large capacity die stacked DRAM, used as a hardware managed cache. The immense bandwidth provided by these DRAM class devices can improve the performance of throughput-oriented GPU while the reduced latency of access can assist in improving the performance of latency-oriented CPUs. 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 DRAM cache help improve performance of IHS processors.

To design an effective DRAM cache for IHS requires balancing conflicting set of design parameters. A DRAM cache design that offers lower hit-time and hence a lower miss penalty for the previous level caches (L2 cache) would thus benefit the CPUs. While on the other hand, the GPU cores would benefit from higher bandwidth and higher hit-rates in DRAM cache even at the cost of higher hit-time. This necessitates the need for a careful design of the DRAM cache for IHS processors, that can handle the GPGPU bursty behaviour as well as service requests for the CPU with a consistent latency without idling resources while delivering improved system throughput at lower energy.

In this work, we assume a cache organization similar to Alloy cache [54] with a block size of 128 bytes and study the problems and challenges in using the DRAM cache for IHS architectures. We justify these design decisions in the Chapter 3. 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 DRAM cache (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 [3]. Additional details relating to the experimental methodology and workloads are described in Chapter 4

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

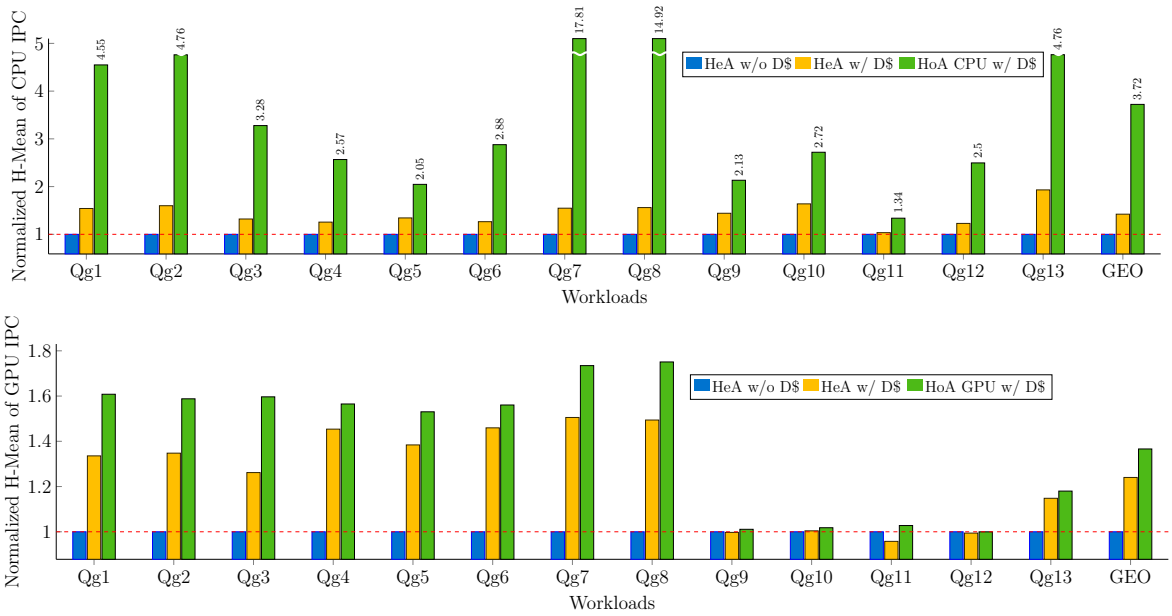


Figure 2.6: Performance Comparison of (a) CPU (b) GPU when running alone, co-run, with and w/o DRAM cache

First, we evaluate the benefits of having a large DRAM cache for the CPU in an IHS architecture. Figure 2.6(a) presents the performance improvement for CPU cores due to the addition of a stacked DRAM cache (in terms of harmonic mean of IPCs, normalized to HeA without a DRAM cache) when co-run with a GPU and when run alone. As can be seen from the figure, the addition of a stacked DRAM cache improves performance of CPU applications in an IHS architecture, on an average by just 42%. However, when the CPU runs alone with a stacked DRAM cache it achieves a 3.72x better performance than that achieved in IHS without a DRAM cache. Although this HoA-CPU performance cannot be achieved with co-running applications in IHS (interference effects cannot be removed), there still exists a significant opportunity for improvement. This gap in performance can be attributed to the unmanaged heterogeneity and interference in the DRAM cache organization.

We further investigate the cause of this performance gap. Figures 2.7 presents the DRAM cache access latency experienced by a CPU request (in terms of CPU cycles) on the primary (left) y-axis and the CPU hit rates on the secondary (right) 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 DRAM cache by 113% while hit rates of the CPU

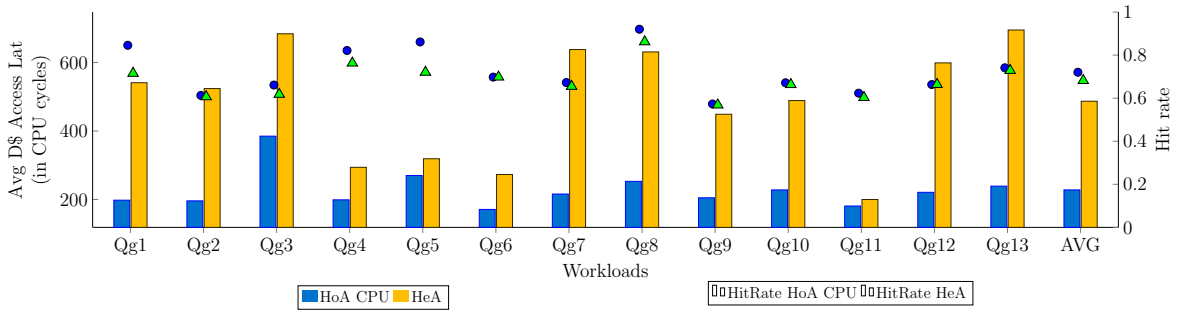


Figure 2.7: DRAM cache CPU Access Latency and Hit Rate

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 DRAM cache 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 applications on the GPU cores. Figure 2.6(b) presents GPU performance obtained by addition of a stacked DRAM cache. We observe that with the introduction of DRAM cache in an IHS, the GPU performance improves on an average by 24%. This performance is within 10% of its HoA GPU performance with DRAM cache when the GPU application is co-run with CPU applications (HeA) and without them (HoA). Thus, co-running with CPU applications has only a minor impact on the GPU performance.

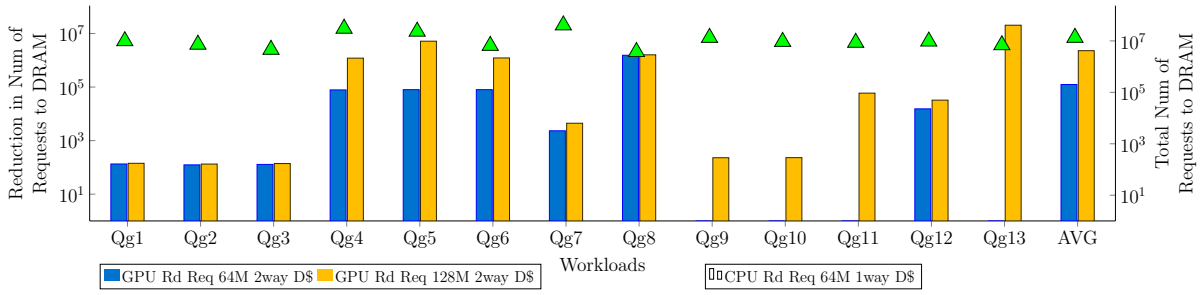


Figure 2.8: GPU Miss reduction with 2-way Associativity

While the above discussion indicate that the design decisions of heterogeneity aware DRAM cache are influenced by the latency-sensitive CPU applications, we emphasize the need to effectively utilize the large capacity and the higher bandwidth of DRAM cache by

GPU applications. Towards this goal, we experimented with two DRAM cache configurations of higher associativity i.e., a 64MB 2-way associative and a 128MB 2-way associative. We observe that the hit rate for GPU applications improves, on an average, by 3.6% and 5% respectively. While the improvement in hit-rate is not significant, as the number of GPU requests is very large (2 or 3 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 DRAM cache bandwidth by the GPU application. Figure 2.8 illustrates this by plotting the reduction in number of GPU misses in log scale on the primary y-axis (left), observed by introducing 2-way associativity over our direct mapped DRAM cache. The GPU misses reduce by an average of 10^5 for 2-way associative cache with half the number of sets (64MB 2way associative DRAM cache) and 10^6 for a 2-way associative cache with the same number of sets (128MB 2way associative DRAM cache). Few benchmarks like Qg9, Qg10, Qg11 and Qg13 show very small to no reduction in GPU DRAM requests for the 64MB 2-way associative DRAM cache. In contrast to this, the total number of CPU read requests to the DRAM, as shown in Figure 2.8 on the secondary y-axis (right), are in the order of 10^7 on an average. This considerable reduction in GPU misses and increased GPU hit rate translates to improved bandwidth utilization of the DRAM cache. Hence, this makes a case for improving the associativity for GPU requests by introducing some pseudo-associativity in the DRAM cache, without impacting the hit-time offered by a direct-mapped tag-and-data (TAD) cache.

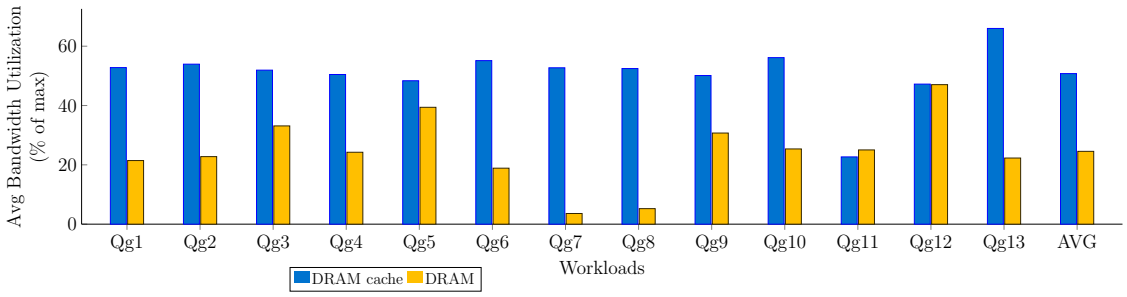


Figure 2.9: Average Bandwidth Utilization for DRAM and DRAM cache as a Percentage of Maximum Bandwidth

We also study the bandwidth utilization characteristics of the DRAM cache and DRAM

buses in an IHS architectures with a stacked DRAM cache. We simulate an IHS architecture as before with a 64MB stacked DRAM cache having peak bandwidth 40GB/s and an off-chip DRAM having a peak bandwidth of 25.6GB/s. Figure 2.9 shows the average bandwidth utilization of DRAM caches and DRAMs, as a percentage of maximum (peak) bandwidths provided by these devices, for IHS architectures running CPU and GPU applications concurrently. We observe that while on an average 51% of DRAM cache bandwidth has been utilized, only 24.5% of the off-chip DRAM bandwidth has been utilized. Stacked DRAMs provide high capacities (compared to SRAM caches) and substantial bandwidths (compared to off-chip DRAMs). Due to the large capacities, the DRAM cache is able to cache and serve most of the data with very high hit rates. This leaves the off-chip bandwidths under-utilized. Hence, to be able to improve IHS performance and achieve improved resource utilization we need to utilize all available memory bandwidth sources in the system i.e., DRAM cache bus and the off-chip DRAM bus effectively.

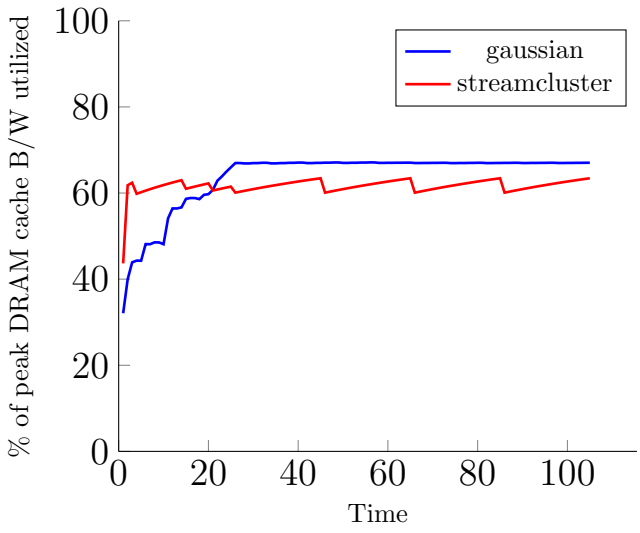


Figure 2.10: Percent of Peak DRAM cache Bandwidth Utilized when GPU application is running

Further, to understand the bandwidth utilization characteristics of the GPU kernel Figure 2.10 plots the percentage bandwidth utilization of DRAM cache sampled periodically during the running of only the GPU kernel. We observe that as soon as the gaussian kernel starts execution about 70% of peak bandwidth of DRAM cache is utilized while about

64% of peak bandwidth is utilized for streamcluster. Thus when the GPU kernel executes the abundant parallelism and concurrent execution on large number of GPU cores creates large number of requests to the DRAM cache, consequently almost saturating its bandwidth. This requires that the underutilized off-chip DRAM bus be put to use immediately on start of execution of the kernel on the GPU.

Based on the above motivation study we conclude that (i) there is significant performance impact of co-running of CPUs and GPUs in IHS architectures with the standard DDRx like memory interfaces (ii) there are significant performance benefits that could be obtained with the introduction of the stacked DRAM cache for the latency-sensitive CPU applications. However, in a naive implementation of the DRAM cache, these benefits can be lost due to interference from the co-running GPU application. Hence it is important to carefully architect the DRAM cache 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 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 larger cache partition for the CPU first. The second bar in Figure 2.11(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 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 2.11(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%. 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

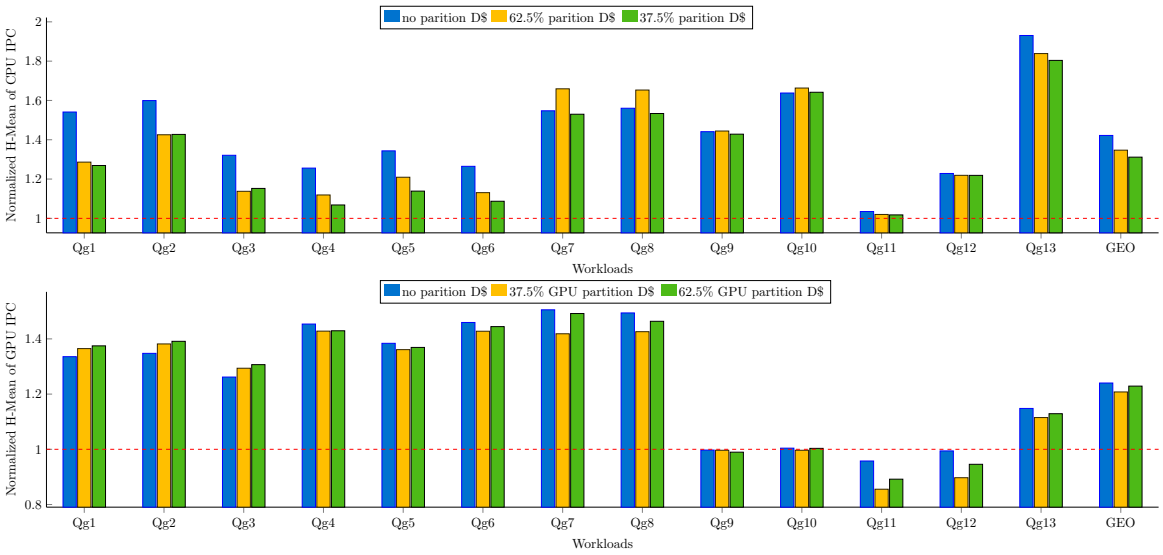


Figure 2.11: Performance comparison of (a) CPU (b) GPU when with un-partitioned D$ and partitioned D$

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 [29, 26, 23] 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.

2.4 Summary

In this chapter, we presented the basics of IHS architecture, DRAM memory system and the DRAM cache organizations which form the background for this work. We then motivated

the need for better memory systems for supporting IHS architecture. We showed that the stacked DRAM cache device characteristics hold the potential to improve performance of IHS processors but in a naive implementation these benefits are negated. Further, we also studied the sources of inefficiencies in a naive implementation of DRAM cache for IHS architectures.

Chapter 3

HAShCache Design and Mechanisms

In this chapter, we describe the details of HAShCache design and the workings of the different HAShCache mechanisms.

3.1 HAShCache Design

In this section, we describe our organization of the DRAM cache for IHS architectures. The DRAM cache is the first level shared cache between the memory hierarchies of CPUs and GPUs. The CPU cores share a SRAM L2 cache of 1MB among themselves while GPU cores share a SRAM L2 cache of 512KB among them homogeneously. HAShCache adapts an aggressive direct mapped cache design with tags stored in DRAM as TAD units i.e., Alloy Cache as described in Section 2.1.3, with a cache line size of 128 byte. We first state and rationalize these design decisions.

3.1.1 Metadata Overhead

The metadata requirement for DRAM caches, even for caches of size 64MB, is large and in the order of few MBs as discussed in Section 2.1.3. For example, the metadata of 4MB is required to be stored for a 64MB cache size assuming 128 byte block size and 8 byte metadata per cache block. These metadata sizes are already in the range of the capacities of last level SRAM caches on modern multicore processors. With DRAM cache sizes projected to

be in the order of few Gigabytes in the next few years, this metadata requirement problem poses a significant challenge. The large storage requirement along with the associated cost if it has to be stored in SRAM, has driven DRAM cache designs to either use larger block sizes (of size 2KB or 4KB [35, 37]) to reduce metadata overhead or co-locate metadata alongside data in the DRAM cache [44, 54, 33]. In the former case, misses waste precious off-chip bandwidth in the absence of spatial locality while the latter design faces slow tag matches.

To understand the spatial locality characteristics of large blocks in a IHS configuration, we experimented with 512 byte block organization for the DRAM cache. The 512 byte block size has been shown to provide a good trade-off between increased hit rates and reduced metadata. Earlier studies on DRAM cache designs for multi-core CPUs [28] also favour 512 byte block size, and hence we limit the block size exploration study upto 512B bytes. For DRAM caches that use large block sizes without footprint prediction the larger block sizes are unlikely to give any additional benefits, other than reduction in tag-match delay (due to SRAM tag caches). However since we are considering direct-mapped Alloy Cache, the tag lookup overhead is unlikely to give any additional benefit. 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 bye) 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 an average for 65% of the 512B blocks that are brought into the cache, at most 2 sub blocks of 128 byte are requested/used. This implies wasted capacity in the DRAMCache and wasted off-chip bandwidth. Figure 3.1 on the primary y-axis (left), shows the performance of CPU and GPU using 512 byte block

DRAM cache normalized to the performance of 128 byte block DRAM cache. We observe that the 512 byte block consistently under performs the 128 byte block DRAM cache by 12.2% and 10.6% for the CPU and GPU respectively in Figure 3.1. The secondary y-axis (right), shows the increase in the average memory access latency in percentage for a 512 byte block DRAM cache as compared to a 128 byte DRAM cache. We observe that despite a 11% and 8% improvement in hit rate in the DRAM cache for CPU and GPU respectively the average memory access latency increases by an average of 75% for the 512 byte block cache. The increased hit rate comes at the cost of wasted bandwidth and increased off-chip DRAM latency as some of the sub blocks fetched are not used. This study indicates a smaller block size for DRAM cache (∼128 bytes) would perform well for IHS workloads.

As discussed in Section 2.1.3, Tags-in-DRAM designs have further focused on improving access latencies by removing tag-serialization overhead using overlapped tag lookups [44] or storing TAD units [54]. These designs come close to tags-in-SRAM like access latency without concerns of spatial locality characteristics of large block sizes. Further, using a non-standard burst length tags can even be retrieved along with data for no additional latency overhead from the stacked DRAM cache.

Hence, HAShCache organizes data at 128 byte block size and stores data in DRAM cache as a cohesive TAD unit.

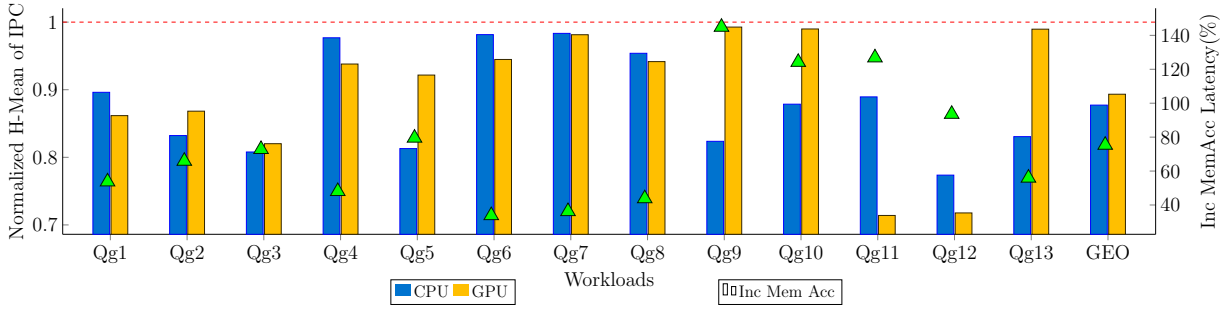


Figure 3.1: Performance of 512B vs 128B block size

3.1.2 Associativity

Providing set associativity is known to improve cache hit rates by reducing conflict misses. In DRAM caches 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 trade-off between providing better hit-rates and reduced access latency. As shown in Section 2.3 (Figure 2.8), 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 DRAM cache. 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 [54].

The large performance decline for CPUs due to co-running requires HAShCache to be organized as a direct mapped cache to achieve better hit time for improved CPU performance. Further, such a direct mapped organization simplifies the design and eschews the need for tag-caches [33] and way locators [28] to improve hit times. HAShCache's organization is also inline with the commercially adopted stacked MCDRAM on the Knights Landing generation of the Intel XeonPhi processor [8] where the DRAM cache is organized as a direct mapped cache.

3.1.3 Miss Penalty

Past research works in stacked DRAMs have assumed an access latency which ranges from 0.5x to 1.0x of off-chip DRAM, while recent data sheets advocate a value closer to 1.0x, especially for large capacity stacked DRAM of close to 1GB. In our work, with a smaller stacked DRAM size of 64MB and 128MB, we assume an access latency of 0.7x compared to the off-chip DRAM latency. Even with this latency, a miss in the DRAM cache would experience a delay of 1.7x as the DRAM (memory) access takes place after the miss is detected (serially). To overcome this, researchers have proposed cache line hit predictors [44, 54] which are critical to extract performance from DRAM caches. These predictors start an early access to memory if they predict that the block will miss in the DRAM cache.

Starting an early access to off-chip main memory can remove the DRAM cache lookup serialization and effectively overlap some of memory access latency with the DRAM cache tag lookup.

Intuitively, we apply the MAP-I prediction (Instruction Based Memory Access Predictor) [54] to CPU requests to start early memory access when an access is predicted to be a miss. The MAP-I mechanism works by using the address of the instruction (Program Counter) making the access to the DRAM cache to make a prediction about a hit or miss. Such a correlating predictor has been shown to have reasonable accuracy of about 95% for multi-core CPUs [54]. For GPUs, the MAP-I predictor is not used and the 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 (b) avoid large structures that will be required for making reliable predictions for GPUs which might require correlating warp, thread, and CU IDs.

3.1.4 Row Buffer Hits (RBH) vs Bank Level Parallelism (BLP)

Stacked DRAMs, similar to commodity off-chip DRAMs, are organized as vaults (channels), layers (ranks) and banks within each layer as shown in Figure 1.2. This organization is similar to that of off-chip DRAMs as explained in Chapter 2. Each vault has several TSVs which constitute lanes in a channel. Each layer shares a command, address and data bus and each bank is operated and accessed independently. As discussed in Section 2.1.2, DRAMs exploit this parallelism, called Bank Level Parallelism (BLP), to improve effective memory bandwidth. Given this abundant BLP in stacked DRAMs, we ask the question should DRAM caches addressing scheme favour exploiting this parallelism over improved RBH? In other words, we ask the question should the addressing scheme of a DRAM cache be organized as RoCoRaBaCh (Row,Column,Rank,Bank,Channel) — referred to as the BLP-scheme — which distributes a cache block 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 the addressing schemes for an IHS processor with a

DRAM cache. Figure 3.2 shows the performance of a DRAM cache that uses a BLP addressing scheme normalized to the performance of a RBH addressed DRAM cache. We observe that both the CPU and GPU applications experience on an average 3% and 1% lower H-Mean IPC respectively when using the BLP scheme. Hence we choose the RBH friendly addressing scheme (RoRaBaCoCh scheme) to address HAShCache.

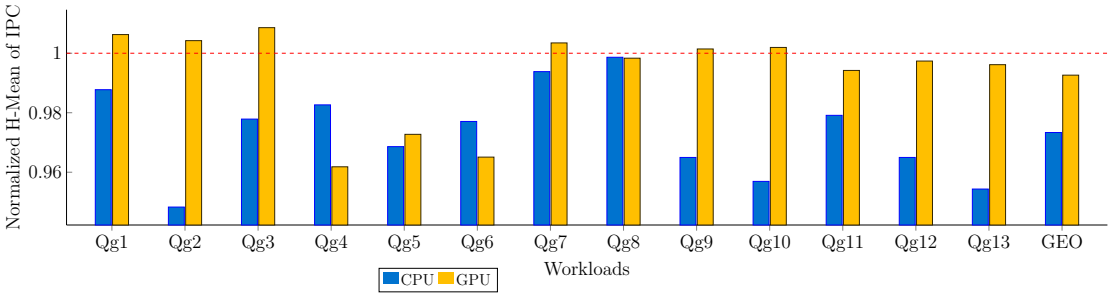


Figure 3.2: Performance of BLP-scheme vs RBH-scheme

To summarize, our HAShCache organizes DRAM cache as follows

- 128 byte block size with tags stored in DRAM rows (tags-in-DRAM)
- Direct Mapped DRAM cache with a longer burst length to retrieve tag along with data
- Early Memory access predictor for CPU requests, while GPU proceed serially via tag match
- Uses an addressing scheme that attempts to maximize row buffer hits

3.2 HAShCache Mechanisms

As noted in Figure 2.6, despite the addition of such a carefully designed DRAM cache, the CPU applications when run along with GPU applications in an IHS architecture do not achieve the full benefits of the DRAM cache compared to when they are run alone. The GPU applications on the other hand are relatively unaffected by the co-running of CPU applications. This can be attributed to the GPUs capability to context switch among warps

to make it latency tolerant. This suggests that the DRAM cache should be optimized to regain the lost CPU performance without compromising the GPU performance. In this section we propose three schemes for achieving this. For each mechanism we state the objective and then articulate its working.

3.2.1 Heterogeneity-Aware DRAM cache Scheduling

Objective: Reduce the large access latencies for CPU requests at DRAM cache

Premise

In an IHS architecture, the CPU requests experience bursts of requests from the co-running GPU application. The imbalance in request arrival rates between CPU cores and GPU cores causes a significant increase of 113% on an average, in memory access latency for CPU requests as observed earlier in Figure 2.7. This increase in memory access latency causes the CPU performance to decrease as compared to when it ran alone without the co-running GPU workload.

As detailed in Chapter 2.1.2, 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 limited size buffer (meant for holding the requests till they are serviced) at the DRAM cache controller. This leads to requests being rejected causing the DRAM cache 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 further compounded by the fact that GPU exploits good row buffer locality and is preferentially scheduled by the DRAM cache controller (under FR-FCFCS scheduling [14]), 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.

To reduce this increased access latencies for CPU requests at DRAM cache, we propose a heterogeneity-aware DRAM cache access scheduling mechanism called *PrIS* (CPU

Prioritized FR-FCFS with IHS aware Scheduling).

Mechanism Overview

HAShCache reduces waiting time of CPU requests by prioritizing them at the DRAM cache 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 DRAM cache controller, resulting in CPU requests not even entering the buffer (queue). HAShCache overcomes this problem by guaranteeing certain minimum occupancy for CPU requests in the buffer at the DRAM cache Controller. This is accomplished using a selective reject-retry mechanism for GPU requests when the queues reach certain critical level. Together these two mechanisms attempt to reduce the DRAM cache latency experienced by CPU requests. We refer to these mechanisms collectively as *PrIS* (CPU *Pr*ioritized FR-FCFS with *I*HS aware *S*cheduling).

PrIS differentiates requests broadly as CPU or GPU requests and not within individual CPU cores or GPU CUs for scheduling. 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.

Hardware Overhead

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 [14]. We also compare the performance of *PrIS* against the mechanism in [14] subsequently in Chapter 6.

PrIS Algorithm Details

The complete *PrIS* scheduling algorithm is stated in Algorithm 1. *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 3 types of requests; (a) seamless row buffer hit, (b) hidden bank prep or (c) prepped row, in that strict order of priority. These are explained below.

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 best performance for CPU requests while the reduction

in GPU performance due to de-prioritization between the schemes was not significant.

Thus, *PrIS* is able to reduce access latencies by reducing waiting delay and queue latencies for CPU requests at DRAM cache.

3.2.2 Heterogeneity-Aware Temporal Bypass

Objective: Utilize the under-utilized off-chip DRAM Bandwidth

Premise

The large sizes of the stacked DRAM cache ensures cache lines have fairly long residency time before being evicted. Hence, DRAM cache has fairly large hit rates which leads to idling of off-chip DRAM bandwidth as shown in Section 2.3. Moreover the stacked DRAM and off-chip DRAM utilize the same underlying technology and hence incur comparable latency (0.7x for DRAM cache vs 1.0x for DRAM). Thus, directing some requests to off-chip allows improved resource utilization and allows us to exploit the aggregate bandwidth of both DRAM cache and DRAM buses without incurring significant latency overheads.

Further, we take cue from another observation made from requests which are mispredicted as misses in the DRAM cache. As mentioned in Section 2.1.3, to hide the latency of miss, our aggressive baseline design already incorporates a hit/miss predictor (similar to MAP-I predictor [54]) for CPU requests to initiate an early access to off-chip DRAM when a miss is predicted in the DRAM cache. These requests are then enqueued in the DRAM cache queues for verification of a miss [1] by a tag match.

Once the tag is compared (matched) against the requesting address in the DRAM cache either of the following 2 cases ensue:

(a) In the case of a hit, data from the DRAM cache is forwarded to the requester and the DRAM memory access is squashed or its response is ignored.

(b) In the case of a miss, data from the memory is forwarded and inserted into the cache.

[1] This is required to ensure that misprediction does not result in using stale data from the DRAM for lines modified (dirty) in DRAM cache

Algorithm 1: *PrIS* DRAM cache Scheduling Policy

```
Input: DRAM cache Queue
Output: selected_req to be scheduled
all bool variables are initialize to false
while not at end of Queue do
    read rank, bank and row of current_req
    if rank.isAvailable then
        if row.isOpen then
            if bank.colAllowedAt <= minColAt then
                /* seamless row buffer hit                          */
                if req.isCPU then
                    selected_req = current_req
                    break
                else if !found_seamless_gpu then
                    selected_req = current_req
                    found_seamless_gpu = true
                end
            else if !found_hidden_bank && !found_prepped_cpu &&
             !found_seamless_gpu then
                /* req to prepped row                               */
                if req.isCPU then
                    selected_req = current_req
                    found_prepped_cpu = true
                    found_prepped = true
                else if !found_seamless_gpu then
                    selected_req = current_req
                    found_prepped = true
                end
            end
        else if !found_earliest_cpu && !found_seamless_gpu then
            /* earliest bank that can be issued hidden cmd          */
            /* executed only once per scheduling decision           */
            found_hidden_bank, earliest_bank = find_earliest_bank()
            if earliest_bank == bank &&
            (found_hidden_bank || !found_prepped) then
                if req.isCPU then
                    selected_req = current_req
                    found_earliest_cpu = true
                    found_earliest = true
                else if !found_seamless_gpu then
                    selected_req = current_req
                    found_earliest = true
                end
            end
        end
    end
end
```

Normally, it is expected that the access to the DRAM cache 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 MSHRs. This is due to the increased queuing delay at the DRAM cache compared to the access latency at the DRAM. HAShCache exploits this observation to bypass CPU read requests for both misses and clean lines thereby circumventing this waiting delay. In a IHS architecture, with a co-running GPU application a DRAM cache hit for a CPU request often experiences a latency which is higher than 1.0x of off-chip DRAM memory as shown in Figure 2.7. This makes the off-chip DRAM an attractive target to direct some of the CPU requests, even those that are likely to be DRAM cache hits. Such a scheme can lead to improved resource utilization of off-chip bandwidth without incurring any increased latencies for CPU requests.

To achieve this, we propose a bypass mechanism for CPU requests that are hits to clean lines in the DRAM cache. For correctness reasons, hits to dirty (modified) lines in the DRAM cache should not be bypassed. We call this heterogeneity-aware temporal scheme as *ByE*.

Mechanism Overview

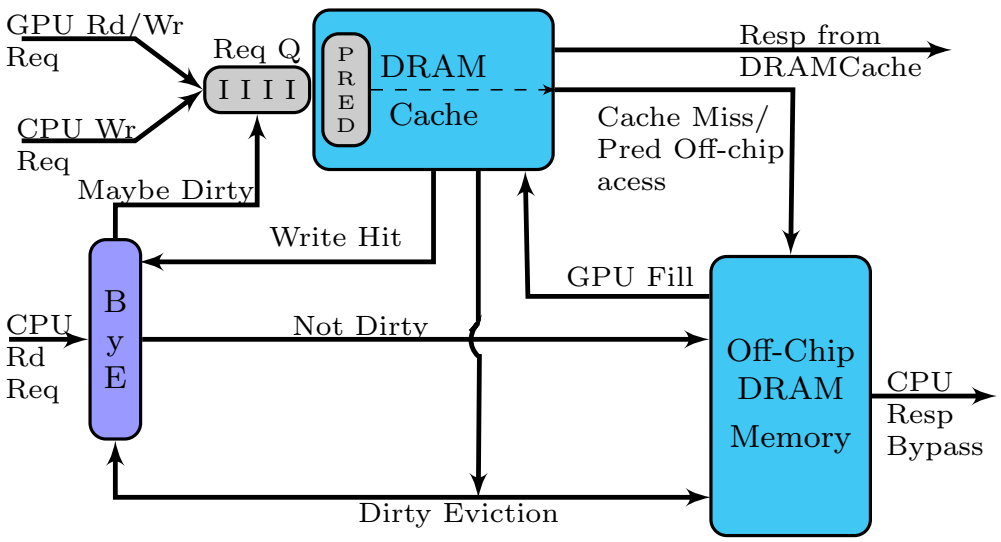


Figure 3.3: Working of HAShCache with *ByE*

To allow the CPU requests to bypass the DRAM cache, HAShCache needs to be able to track and lookup dirty lines in the DRAM cache. However, maintaining a list of all dirty lines in the DRAM cache would require prohibitively large storage structures. Thus, HAShCache uses an approximate tracker called Bypass Enabler (*ByE*). *ByE* uses a counting Bloom filter [18, 24] that tracks the dirty lines in the DRAM cache 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 DRAM cache. On a write request when a cache line becomes dirty in the DRAM cache, the address is hashed into *ByE* and the corresponding counters are incremented. When a dirty line is evicted from the DRAM cache, *ByE* attempts to remove the entry from the Bloom filter by decrementing the corresponding locations [2].

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 3.3. If the Bloom filter search returns a negative result, then the address is guaranteed to be not dirty in the DRAM cache. Thus, the request can safely be bypassed to utilize the off-chip DRAM bandwidth.

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 DRAM cache, which would need a full DRAM cache access.

Insertion Policy

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,

[2]Counting 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 in our scheme, saturated counters are never decremented. While 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 an average just 2% of the 2-bit counters in the Bloom-filter saturate out of the 512K counters. Further, saturation of Bloom filter is a cause of concern for large DRAM caches and long running systems. Clearing the filter would require time consuming operation of scrubbing dirty blocks from cache. However, it is possible to design a per rank or a per bank bloom filter. When the bloom filter for a rank or bank saturates (or Bloom filter produces large number of false positive) the corresponding cache lines in the rank or bank can be scrubbed. The rest of the DRAM can continue regular operation. Only the requests corresponding to the rank or bank being scrubbed will face higher latency.

this allows *ByE* to ensure that future write requests for the line do not hit in the DRAM cache as increase in dirty lines would lead to reduced bypass efficiency. Secondly, this allows *ByE* to reduce some of the bloat caused by a Miss Fill [23] into the DRAM cache. Thus, the no-insert policy and the *ByE* act as complementary self balancing mechanisms - trading off CPU hit rates in DRAM cache for better access latency at the off-chip DRAM.

ByE Hardware Overhead

We find that a small 2-bit counting Bloom filter implemented with two H_3 hash functions [57] and 512K entries per controller is sufficient to produce reasonable bypass efficiency with a tolerable misprediction rate. The total overhead for *ByE* is 256KB for a 64MB DRAM cache which is less than 0.4% of the cache size.

Thus, *ByE* is designed to improve performance by achieving a better bandwidth balance between the DRAM cache and the off-chip main memory.

3.2.3 Heterogeneity-Aware Spatial Occupancy Control

Objective: Allow GPU to better use DRAM cache Bandwidth

Premise

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 DRAM cache for GPU requests, in order to exploit the higher bandwidth provided by it.

First, we make the following observations and inferences:

(i) 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 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 DRAM cache due to reduced latency of access.

(ii) Further as CPU applications are latency sensitive, the DRAM cache hit latency for CPU requests should not be unduly stretched.

(iii) For the GPU to be able to better utilize the DRAM cache 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.

(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.

(v) As noted in Section 2.3, GPUs can trade access latencies for higher hit rates. Thus, providing associativity for GPU request to improve the hit rate would be beneficial.

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

To accomplish the above goals, HAShCache uses a *Chaining* scheme which introduces (pseudo) associativity mainly for GPU requests, while ensuring a certain minimum occupancy for CPU lines. The chaining scheme described below can be easily implemented as an algorithmic finite state machine.

Mechanism Overview

The proposed *Chaining* scheme uses a linear probing [39] like technique, inspired by the collision resolution mechanism of a hash map, as shown in Figure 3.4. To ensure minimum occupancy for CPU requests, *Chaining* maintains a low-threshold value (l_{cpu}) and when

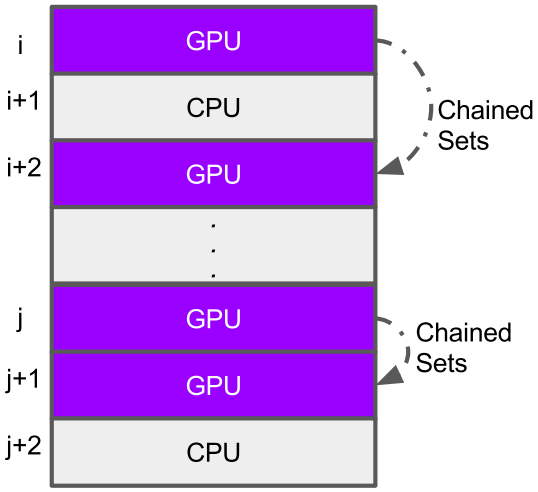


Figure 3.4: Conceptual Working of *Chaining* in HAShCache

the occupancy of CPU lines [3] 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 DRAM cache 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 number of blocks in a DRAM cache row (page). Hence, the inserted block always lies in the same DRAM cache row as the original cache block. Note that for every set, there could be at most 1 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 3.5(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

[3] As mentioned earlier we classify a data as CPU data or GPU data based on whose request last accessed the data in the DRAM cache. An alternate equally feasible design point is to classify the data as CPU or GPU data based on which request brought the data into DRAM cache 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 significant difference in the results.

field (Figure 3.5(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 3.5(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.

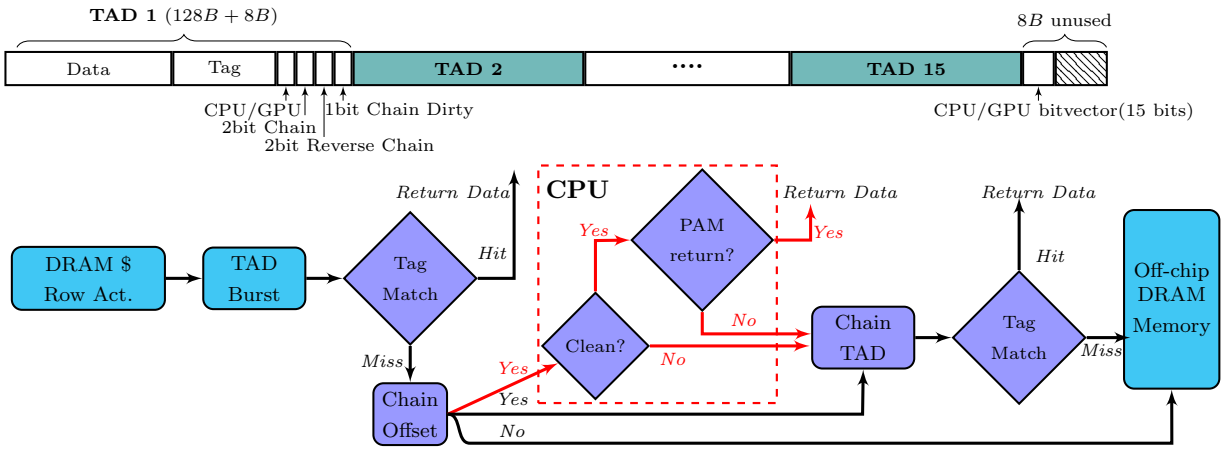


Figure 3.5: HAShCache Row Organization and Access Path of a Request for *Chaining*

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 byte (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 3 consecutive locations the request is not inserted into the cache.

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

Table 3.1: *Chaining* Mechanisms GPU Fill Request Insertion Policy

Insertion Policy

We now summarize the insertion policy used by *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. If the set that the CPU occupied here is a *chained block*, chaining is removed using the reverse chain bit i.e., the chain bits in the *original block* are reset [4].

The memory controller maintains a running GPU occupancy counter for each row in the DRAM cache. This information is accommodated in the unused bits at the end of the row in the DRAM cache and can be cached in SRAM structures for fast lookups. Using this current occupancy counter the controller decides the fill policy for the GPU fill request. HAShCache checks whether the threshold reached is determined by comparing the current GPU occupancy with the set l_{cpu} threshold value as : $O_{gpu} \leq (1 - l_{cpu})$ where O_{gpu} is the current GPU occupancy in the DRAM cache row. 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 3.1. For a GPU fill request, if the original location belongs to GPU and does not have

[4]Additional bandwidth required for resetting chain bits in the *original block* in this scenario has been accounted for in the simulation. We also account for the additional bandwidth required to occasionally access CPU/GPU bit vector at the end of the DRAM row. We find that the average bandwidth overhead for chaining across our benchmarks is just 4.7%.

a chained location, then block is inserted in one of the nearest CPU block [$(B+1)\%N_s$ or $(B+2)\%N_s$ or $(B+3)\%N_s$]. If such a nearest CPU block is not found, the request is inserted into 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 3.1. 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 3.1). 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 DRAM cache (see row 3 in Table 3.1). 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 2 tag matches, either in 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 DRAM cache. *Chaining* is able to

(i) ensure a minimum occupancy for the CPU lines while effectively allowing the GPU to occupy the rest of the DRAM cache by providing pseudo-associativity.

(ii) remain as a direct mapped cache for majority of the CPU requests.

(iii) avoid forcing eviction of hot GPU lines while also avoiding storing of dead lines in the cache.

Chaining Hardware Overhead

This occupancy control mechanism does not incur any additional storage and uses the unused space in the DRAM cache rows. Once the GPU finishes kernel execution HAShCache returns to a direct mapped cache as the CPU lines inserted into the DRAM cache occupy *chained blocks* thereby unlinking chains.

3.3 Summary

In this chapter, we presented the HAShCache organization and mechanisms. We first showed the conscientious design decisions made and substantiated them with experiments of DRAM cache for IHS architecture. We then detail the principles and working of the three HAShCache mechanisms - *PrIS*, *ByE* and *Chaining*. For each of the mechanisms, we describe the issues they try to address, describe their working in detail and the hardware overheads incurred. HAShCache mechanisms are heterogeneity-aware, lightweight (incurring minimal or no hardware overhead) and can dynamically adapt to the inherent disparity of demands in an IHS architecture.

Chapter 4

Simulation Infrastructure

In this chapter, we articulate the simulator setup and enhancements done to simulate the IHS architecture with the shared stacked DRAM cache. We then outline the experimental methodology, benchmarks comprising of composite heterogeneous workloads used in our evaluation and the parameters used to report system performance in our evaluation.

4.1 The gem5-gpu Simulator

The gem5-gpu [50] is a heterogeneous cycle accurate CPU-GPU simulator used to simulate a hybrid system with both CPU and GPU types of cores. It integrates gem5 [16] for the multi-core CPU simulation and gpgpu-sim [15] for simulating the compute units of a general purpose GPU with a flexible and configurable memory hierarchy using gem5's Ruby memory system [46]. It can model a system with multi-core CPUs and a discrete GPU with a "split" memory hierarchy or an integrated heterogeneous system with tightly coupled CPU and GPU cores with a "shared" cache coherent interconnect. We refer to the gem5-gpu from here on running an IHS architecture with a shared virtual and physical address spaces as described in Chapter 1. The gem5-gpu can run unmodified operating system and application binaries. The simulator is also performance validated against hardware and is completely open source [50]. For our evaluation we run gem5-gpu in System Call emulation mode with the base operating system performing the CUDA driver emulation.

4.1.1 Addition of Shared 3D Die-Stacked DRAM cache

The Ruby cache and memory hierarchy in gem5-gpu is authored using the domain specific language SLICC (Specification Language for Implementing Cache Coherence) [46]. SLICC allows the definition of cache levels and behaviour of the coherence protocol. The directory controller is the memory side coherence engine and participates in the coherence protocol. These directory controllers have message queues to send and receive memory requests from the DRAM Controllers. The DRAM controllers are responsible for the operation of the DRAM devices. These DRAM controllers queue requests and orchestrate memory access scheduling to the DRAM according to the type of memory device (e.g., DDR, LPDDR, GDDR, HMC etc). To study the interference of CPU and GPU traffic we model the DRAM cache as the first shared level of cache between the heterogeneous CPUs and GPUs while the L2 SRAM caches are shared between homogeneous CPU cores and GPU cores. We modified the IHS architecture to include a shared split L2 cache shared across multi-core CPUs instead of the default per CPU private L2 cache. For this we modified the existing VI_Hammer coherence protocol [50] to simulate the shared L2 Cache.

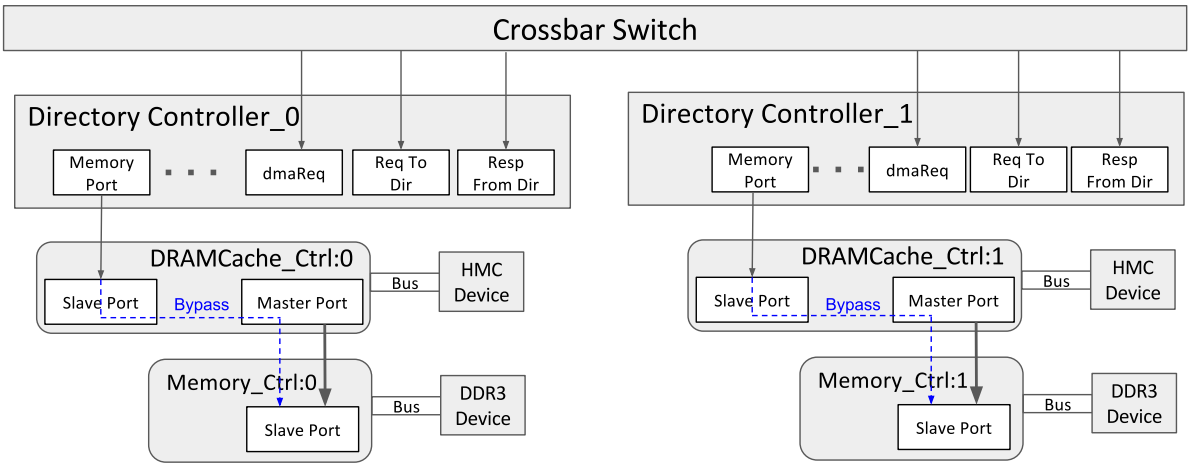


Figure 4.1: Memory Hierarchy as Modelled in Simulator

The DRAM cache we evaluate here is a memory-side [1, 26] shared cache between the 2 split cache hierarchies of CPUs and GPUs. Memory-side caches are ones that are outside the coherence domain as shown in Figure 4.1. They are logically just in front

of the memory and serve to increase the memory's effective bandwidth and reduce the average latency of memory accesses. It does not introduce any new states to the directory in the coherence protocol i.e., there is no state called dirty in DRAM cache, or there is no need to snoop it separately and a single memory state is enough to serve data from it. Further, access to memory is interleaved across multiple memory controllers. To avoid a biased interleaving due to uniformly strided address patterns, in our simulation we employ a basic XOR-based address hashing mechanism. Each 4GB DDR3 memory device is *paired* with a 32MB chunk of stacked DRAM vault. The stacked DRAM vault caches data from the corresponding *paired* 4GB memory device that it is coupled with. This setup ensures that there are no cross bus requests between controllers. Since our simulation setup has the limitations of having a 1-to-1 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 higher amount of parallelism in our stacked DRAM. However, the bank capacity is retained as would be in a standard stacked DRAM. The DRAM cache is non-inclusive [32] of the L2 caches above it and uses a write no-allocate policy.

4.1.2 DRAM cache Controller Design

The DRAM cache Controller consists of several constituent components as shown in Figure 4.2. The highlighted components (gray fill) in the figure correspond to the components that are unique/exclusive to a DRAM cache controller and are not present in a conventional DRAM controller. These include fill queue, MSHR and WriteBuffer.

DRAM cache controllers similar to conventional off-chip DRAM controllers consists of request queues. Each request in the queue corresponds to the address of a single burst of data from the stacked DRAM device. In our design, cache lines are stored as TAD units (tag-and-data) [54] and each TAD is divided into 2 bursts, where each burst corresponds to the width of the data bus of the stacked DRAM. Conventional DRAM controllers enqueue requests into a read or a write queue depending on the type of request. However, more subtly unlike conventional DRAM devices, each write request in the DRAM cache entails a

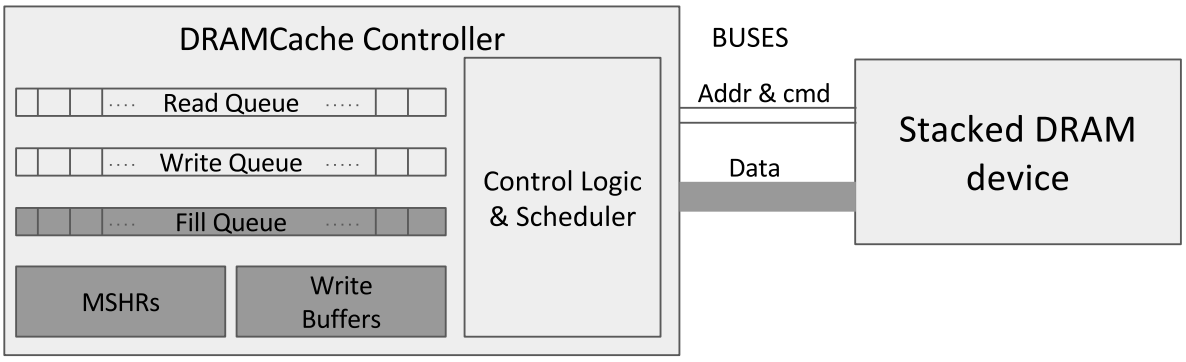


Figure 4.2: DRAM cache Controller Components

tag read and match followed by a write to the data of the TAD unit only if the tag matches (i.e., request is a hit). Thus, both read and write requests in a DRAM cache perform a DRAM read operation. To avoid complicating the model and implementation, we choose to hold all write requests, irrespective of hit or miss, in the same write queue. This allows us to apply uniform priority to all write requests as these are not in the critical path. Our stacked DRAM cache also models a third type of queue called a Fill Queue [21] which enqueues requests to insert data into the cache when the requests return from main memory. A fill request performs a DRAM write operation only.

We also model MSHR and WriteBuffers with their associated latencies to realize the precise working of caches. An MSHR is allocated on a cache line miss in the DRAM cache and subsequent requests to that cache line are coalesced and added as targets in the MSHR. A single fetch request is sent to memory corresponding to an MSHR. Once the memory request returns, a response with the data is sent for each target within the MSHR before it is deallocated. Subsequently, a fill request is also created to insert the TAD back into the DRAM cache. A WriteBuffer is allocated when (a) a dirty cache line is evicted from the DRAM cache, thus requiring a write back to memory (b) a write request misses in the DRAM cache. Since the DRAM cache is designed with a write no-allocate policy, write misses in DRAM cache are directly forwarded to the memory. The WriteBuffer is deallocated when the write to memory returns.

When scheduling a request to the DRAM cache, the control logic first selects a queue. The DRAM cache control logic prioritizes reads over writes as reads are on the critical

path. This is done by allowing the write queue to fill up to a certain high threshold (*write_high_threshold*) before switching the bus for writes. Switching the bus from read to write or vice versa incurs a delay and hence a minimum number of writes (*min_writes_per_switch*) are performed before switching the bus back to servicing read requests. If the read queue is empty, we avoid starving the writes and prevent the device from being underutilized/idle by allowing the bus to switch to writes if the write queue has a certain minimum number of requests (*write_low_threshold*). The DRAM cache control logic considers the fill queue as part of the write queue thresholds and switches to servicing a fill queue request in the write mode, only when a fill queue high threshold is reached (*fill_high_threshold*). Before enqueuing read and write requests into the respective queue, the fill queue is checked for a match [1]. If the read request is found in the fill queue i.e, an outstanding fill request for the same address exists, it is directly serviced from there without accessing the stacked DRAM. If a write request is found in the fill queue, the request is coalesced in the fill queue. Once the queue is picked, the DRAM cache scheduler then picks a request to be serviced from the queue according to the scheduling algorithm. The address of the request is decoded according to the address mapping policy and appropriate address and commands are sent over the bus while respecting the device timings and refresh constraints.

To simplify data management and ensure correctness we choose not to implement a backing store for the DRAM cache and instead share the backing store instance of the off-chip DRAM itself.

Reject-Retry Mechanism

Interactions between the components, for example DRAM cache Controller and off-chip DRAM controller, is implemented as a master-slave port architecture. A master port sends requests while the slave port receives requests and invokes appropriate functions to act on the requests. For example, when the directory controller's memory master port sends

[1] We find that for our benchmarks the average length of the fill queue ranges from 4 to 9 entries. Hence such a fill queue lookup mechanism, before enqueuing reads and writes into the respective queue, is a plausible design.

a request to the slave port of DRAM cache Controller it enqueues the request in the appropriate 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 a read request, the DRAM cache controller clears the corresponding 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. At such a time, when a request is received at the DRAM cache slave port but is not able to act on it due to the cache being blocked or read/write queue being full or due to mechanism restrictions as in *PrIS*, 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. This mechanism is independent of the network-on-chip that handles routing, channel partitioning etc.

4.1.3 System Configuration

The gem5-gpu is configured to run 5 CPU cores (4-wide out-of-order 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 8 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 4.1. The private L1 of CUs are non-inclusive of the shared GPU L2 cache and can hold stale data. However, GPU L2 cache is coherent with all levels of the CPU hierarchy. The CPU caches and GPU L2 cache are kept coherent in the simulator. Table 4.1 provides the simulator setup details. Our simulator also respects all significant timing and functional parameters for the stacked DRAM cache (including refresh, data bus, request queues, scheduling algorithms, command signalling and clock frequencies) using the DRAMCtrl [30] model.

CPU Core	five 4-wide out-of-order x86 cores @2.5 GHz
CPU Caches	32KB 8-way split I/D private L1 Cache, 1MB 8-way shared split L2 Cache, 128B lines
GPU Core	eight Fermi SMs@700MHz, 2x2 GTO warp sched upto 32 threadblocks, 64 warps of 32 threads, 64K registers, 96KB scratch memory
GPU Caches	64KB 4-way private L1 cache, 512KB, 16-way assoc L2 Coherent Cache
Stacked DRAM	2 vaults, HMC_2500_x64, 2KB Page t_{CL}-t_{RCD}-t_{RP}-t_{RAS}=9.9ns-10.2ns-7.7ns-21.6ns 8 layers/vault, 4 banks/layer 64 byte burst size, Peak bandwidth 40GB/s Refresh related: t_{REFI}=3.9us t_{RFC}=59us
Off-chip DRAM	2 channels, DDR3_1600_x64, 1KB page t_{CL}-t_{RCD}-t_{RP}-t_{RAS}=13.75ns-13.75ns-13.75ns-35ns 1 rank/channel, 8 banks/rank 64 byte burst size, Peak bandwidth 25GB/s Refresh related: t_{REFI}=7.8us t_{RFC}=260us

Table 4.1: Configuration of the Simulated System

4.2 Workloads

Our goal is to study the performance impact of the large last level shared DRAM cache in IHS architectures. Hence we run multiple applications in parallel on the multi-cores and the GPGPU. Applications having high and medium memory intensive behaviours from SPEC CPU2006 suite [31] were chosen to form a multi-programmed workload that run completely on the CPU. Table 4.2 classifies the CPU benchmarks into low, medium and high memory intensity based on the observed MPKI values as seen at the L2 cache when running alone. We create a multi-programmed workload with 4 SPEC benchmarks each. We use the Rodinia benchmark suite [20] to represent GPU applications that offload kernel computation to a GPU CUs. These Rodinia applications are modified to elide the *memcpy* API calls so as to run with unified virtual and physical address spaces (Rodinia-nocopy).

For the Rodinia-nocopy benchmarks we used *needle*, *k-means*, *gaussian*, *hotspot*, *srad*,

Low (MPKI $\leq$ 10)	astar, cactusADM, gcc, gobmk, hmmer, sphinx3
Medium (10 <MPKI $\leq$ 20)	bwaves, bzip2, leslie3d, libquantum, milc
High (MPKI >20)	mcf, soplex

Table 4.2: MPKI of CPU Stand alone Benchmarks

Low (MPKI $\leq$ 10)	hotspot, lud
Medium (10 <MPKI $\leq$ 20)	kmeans, needle, srad
High (MPKI >20)	streamcluster, gaussian

Table 4.3: MPKI of GPU Stand alone Benchmarks

streamcluster and *lud*. The rest of the programs either could not be compiled for IHS architecture or could not run (results in forward progress deadlock errors in the modified simulator). The Rodinia benchmarks use the fifth CPU core to run the initialization section before offloading the kernel to GPU. Thus, our IHS simulates five CPU cores and eight GPU CUs. Table 4.3 categorizes the Rodinia-nocopy benchmarks into Low, Medium and High based on the observed MPKI values (Instructions here refer to thread instructions) at the L2 cache.

Combinations of these SPEC quad-core multi-programmed workloads were coupled with a full Rodinia-nocopy benchmark to create a representative mix of applications 4.4 that would run in an IHS system. These composite workloads embody different levels of total memory intensity produced by the CPU and GPU cores.

We also measure the footprint of these workloads in terms of number of unique 128B cache blocks accessed at the DRAM cache. 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 obligates us to use a smaller stacked DRAM cache capacity to make pertinent observations.

4.3 Simulation Methodology

We fast-forward the initialization phase of each workloads up until just before the launch of the first kernel of the GPU program. We ensure that each executes at least 2 Billion instructions in fast forward and a total of 20 billion instructions of the IHS workloads is

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	k-means
Qg5	milc;mcf;libquantum;bzip2	k-means
Qg6	bzip2;gobmk;hmmer;sphinx3	k-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

Table 4.4: Composite Heterogeneous Workloads

executed on an average in the CPU cores. 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. The benchmarks were then check-pointed after the above fast forward phase. Subsequent detailed simulations with different configurations were run from this checkpoint. We then warm the cache until the fastest core completes 250 million instructions. During the warmup 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 and the kernel is executed on he GPU core using the gpgpu-sim component. As is the norm, when a core finishes its quota of 250 million instructions, it continues to execute until all the cores have completed.

As stated earlier, the Rodinia application uses the 5^{th} CPU core and offloads the kernel 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 region and serial CPU section. 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.

4.4 Performance Metrics

Our study concerns the performance of the IHS architecture, in which multiple single threaded CPU programs and GPU program are co-run and contend for the shared DRAM cache. In such a co-run setup, the progress made by the applications (workloads) is different compared to when running homogeneously (i.e., only ran CPU applications on CPU cores or GPU kernels on GPU cores). Further, the performance impact of the shared DRAM cache is not same on each of the CPU benchmarks as well as the GPU benchmark. We require to define metrics that combine IPC meaningfully without biasing the system performance to either CPU or GPU cores. Hence, we report a multitude of performance metrics to holistically determine the performance of the IHS processor with the DRAM cache and our optimizations.

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

$$H\text{-}MEAN_{CPU} = \frac{n_{cpu}}{\sum_{i=1}^{n_{cpu}} \frac{1}{IPC_i^{CPU}}} \; and \; H\text{-}MEAN_{GPU} = \frac{n_{gpu}}{\sum_{i=1}^{n_{gpu}} \frac{1}{IPC_i^{GPU}}}$$

where IPC_i^{CPU} and IPC_i^{GPU} are the instructions per cycle achieved by the i^{th} CPU core and i^{th} GPU CU respectively and n_{cpu} and n_{gpu} are number respectively the number of CPU cores and GPU cores. In this work, we report CPU and GPU $H\text{-}MEAN$ (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 type.

To report combined system performance we use 2 metrics; the H-Mean of all IPCs and Weighted Speedup for all IPCs. This is important as we would need to see a fair system performance metric without biasing the metric to either CPU or GPU performance. Thus, we report combined system performance using the combined H-MEAN of IPC all CPU and GPU cores which is defined as,

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

Combined system performance using the weighted speedup metric [61] 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 a IHS setup respectively. IPC_i^{SP} denotes the IPC of the i^{th} CPU application running in a single core in a stand alone mode. IPC_i^{GPU} denote the IPC of the i^{th} GPU CU when the GPU application is running in a stand-alone mode across all GPU CUs.

4.5 Summary

In this chapter, we presented the simulator extensions to the gem5-gpu simulator to model a DRAM cache for an IHS architecture. We described the workloads created to evaluate the proposed design and the simulation methodology. Finally, we define the comprehensive performance metrics used to measure the IHS system performance for the various configurations.

Chapter 5

Performance Evaluation of HAShCache

In this chapter, we evaluate the performance of the proposed HAShCache mechanisms discussed in Chapter 3 and show that HAShCache is able to improve the performance of IHS. Further, we show that HAShCache solution is robust and scales well by performing sensitivity study for the various components in Section 5.2. Lastly, we compare our schemes quantitatively with a couple of state-of-the-art techniques in Section 5.3

5.1 HAShCache Mechanisms

In this section, we present the performance of HAShCache mechanisms (*PrIS*, *ByE* and *Chaining*) and some combinations of them. We examine the impact of these mechanisms on CPUs, GPUs and the performance of the IHS system as a whole. Throughout, we use the experimental methodology as described in Section 4.3.

In the following subsections we discuss the results in detail for each of the HAShCache schemes in detail. Figure 5.1 (a) and (b) presents the performance improvement in terms of Harmonic Mean of IPC, for CPU and GPU respectively, normalized to the baseline without a DRAM cache. We report the performance improvement due to the introduction of DRAM cache (Naive), and then the different HASHCache mechanisms and their combinations. In all cases the CPU applications are co-run with the GPU applications. For comparison reasons, we also show the performance of HoA CPU — CPU applications (multi-programmed

workloads) when run without the GPU application — and HoA GPU — GPU application run without CPU applications.

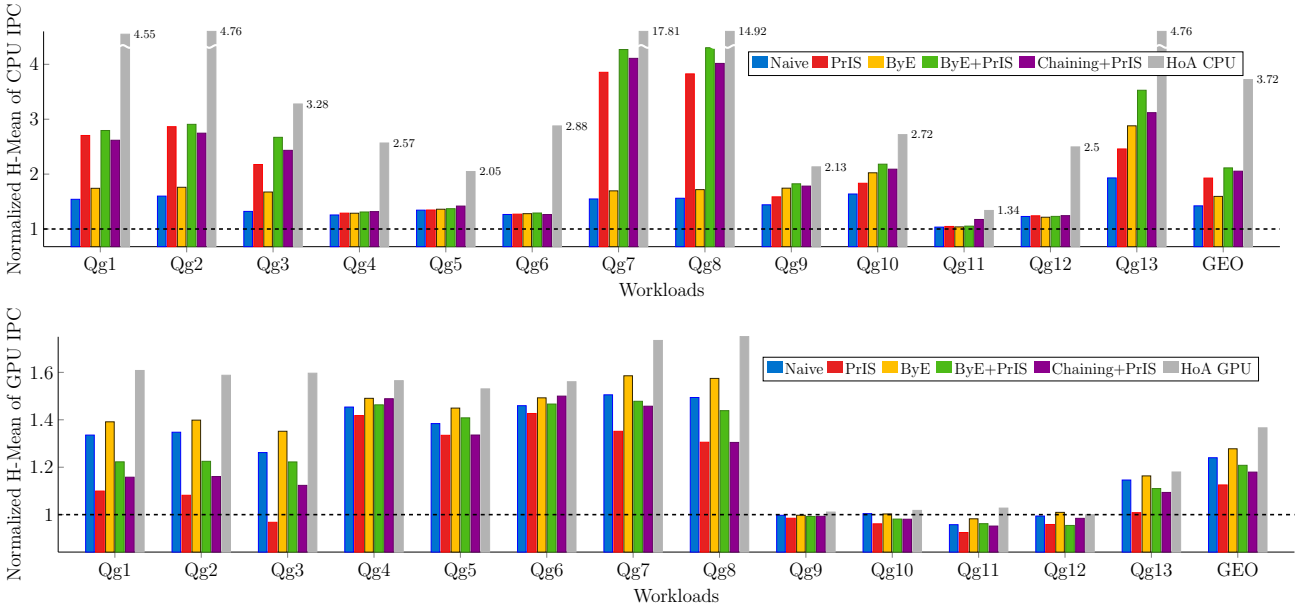


Figure 5.1: Speedups obtained by adding a stacked DRAM cache for (a) CPU (b) GPU

5.1.1 PrIS DRAM cache Scheduling

Prioritizing CPU requests with our *PrIS* scheduler at the DRAM cache controller leads to considerable performance benefits for the CPU applications. By using *PrIS*, the average access latency of the DRAM cache for CPU requests reduces by an average of 15.3% and upto 48.9% over a naive DRAM cache (see Figure 5.2). Hence, *PrIS* is able to improve the performance of the CPU by an average of 35% over a naive DRAM cache. 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 DRAM cache. 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 DRAM cache queues (b) GPU requests, despite being de-prioritized at DRAM cache, have a better hit rate in the DRAM cache and avoid accessing the off-chip DRAM.

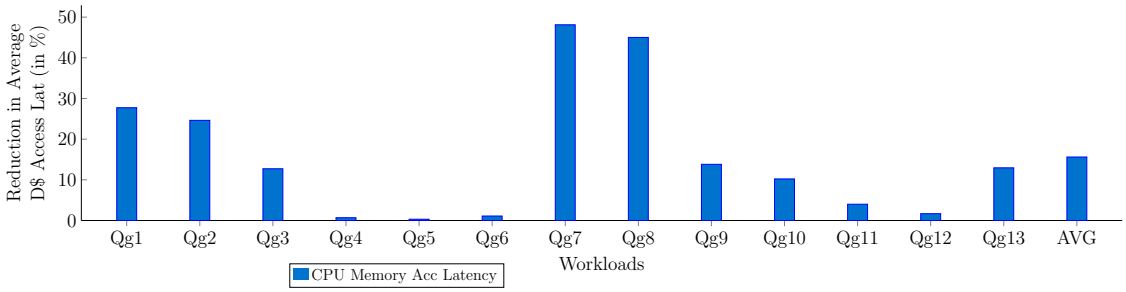


Figure 5.2: Reduction in DRAM cache Access Latency for CPU requests with *PrIS* over a naive DRAM cache

5.1.2 ByE for Temporal Bypass

ByE attempts to improve performance by directing some CPU requests to be served from the off-chip DRAM, thus achieving improved resource utilization and bandwidth balance in the process. This also ensures that the overall aggregate system bandwidth can be exploited efficiently. *ByE* alone achieves 12% improvement in CPU performance and a 3% improvement in GPU performance over a naive DRAM cache.

The CPU performance improvements are primarily due to bypassed requests facing reduced queuing delays at DRAM. Figure 5.3 shows the percent reduction in total memory access latency on primary y-axis (left) for CPU read requests achieved by *ByE* over an already aggressive naive DRAM cache which employs a hit/miss predictor for CPU requests. The total memory access latency for CPU read requests reduces by an average of 28%. The already high hit rates for GPU in the DRAM cache coupled with the no-fill policy for bypassed CPU requests ensures fewer GPU requests at the off-chip DRAM which leads to less congestion.

Figure 5.3 also shows the percentage of incoming read requests bypassed by *ByE*, on the secondary y axis. *ByE* is able to bypass on an average about 37% of incoming read requests. On an 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 DRAM cache. We discuss more on a sensitive study of Bloom filter later in this section. Further, the reduced set contention and less number of CPU requests in DRAM cache queues reduces congestion for GPU, which in turn leads to small performance benefits for the GPU as well improving the Harmonic Mean of GPU IPC by 3% over naive DRAM cache.

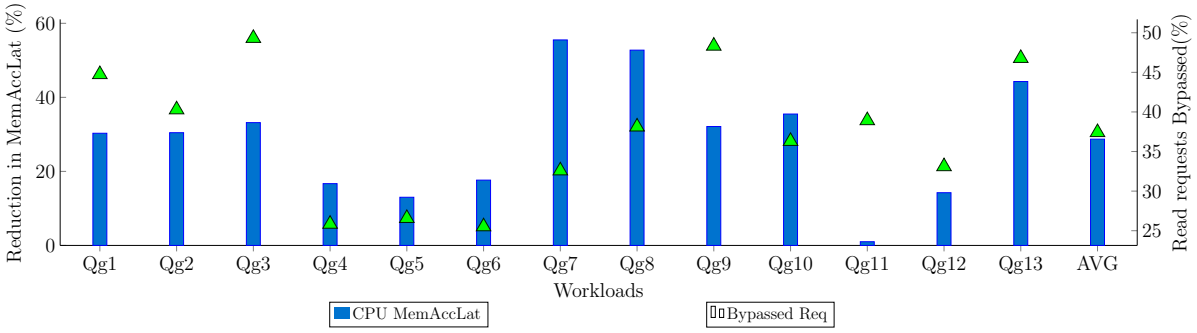


Figure 5.3: Total MemAccLat reduction with *ByE* and % of bypassed read requests for CPU requests

Combining *PrIS* with *ByE* allows for the non-bypassed CPU requests at the DRAM cache 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 DRAM cache while degrading GPU performance by just 3%.

Another result to be noted here is that, unlike *PrIS* or *PrIS+ByE*, *ByE* is able to improve not just CPU application performance but also achieves improved performance for GPU applications over a naive DRAM cache. *ByE* achieves this by being able to reduce the interference of CPU and GPU requests at DRAM cache, thus allowing the GPU to better utilize the DRAM cache bandwidth and capacity while the redirected CPU requests are served out of off-chip DRAM memory.

The somewhat high false positive ratio in our bypass mechanism is due to the large number of dirty blocks in the DRAM cache and the relatively small size of the Bloom filter. We also experimented with 3 hash-functions while also optimally increasing the array capacity to 312KB (20% larger) to reduce aliasing and increase the efficacy of bypass. The

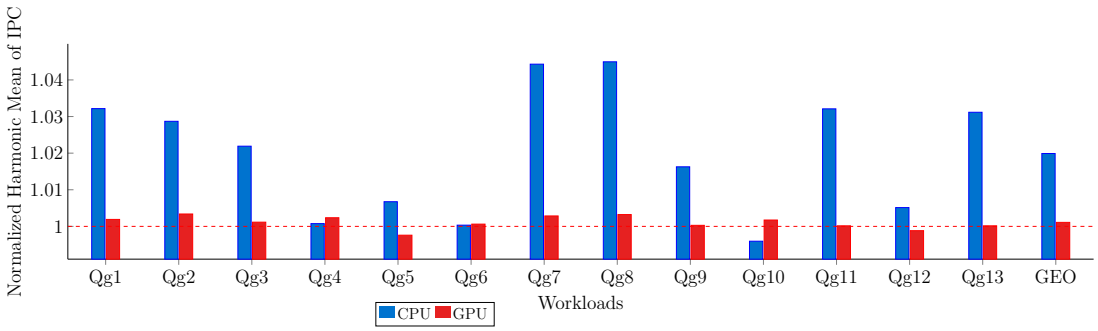


Figure 5.4: Performance with a larger Bloom Filter

performance of CPU and GPU using the larger Bloom filter (in terms of Harmonic Mean of IPC) normalized to the original bloom filter size is shown in Figure 5.4. We observe that the CPU performance improves only by 2.1% on an average while the GPU remains largely unaffected. Hence, we retain the original Bloom filter (256KB) with 2 hash functions for our *ByE* implementation.

5.1.3 Chaining for Spatial Occupancy Control

As discussed in Section 3.2.3, *Chaining* mechanism improves hit rates for GPU while guaranteeing some occupancy for the CPU in the DRAM cache. 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 reduces 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 DRAM cache and the secondary effect of lower congestion at off-chip DRAM allows CPU requests to be serviced with lower delays. This improves performance of CPU by 7%, over only *PrIS*.

Overall *Chaining+PrIS* improves CPU performance by 44.7% while degrading GPU performance by merely 4.8% over a naive DRAM cache.

5.1.4 Summary of HAShCache Mechanisms Performance

We now holistically examine the performance improvements due to the introduction of our HAShCache mechanisms, in CPU and GPU cores together. From Figure 5.1 we observe that, adding a naive DRAM cache can achieve an average of 42% and 24% improvement in CPU and GPU cores respectively. Whereas, HAShCache achieves significant speedups of (105%,17.5%) and (111%,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 5.1) and within 81% and 76% of the ideal no interference performance for the CPU (final gray bar in Figure 5.1(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 performance of both processors due to interference, adding a DRAM cache can improve performance upto 330% for CPU and 48% for GPU over a baseline system with no stacked DRAM cache. In comparison, the naive DRAM cache implementation only brings 55% and 56% improvement in the CPU and GPU performance respectively.

5.1.5 System Performance with HAShCache

We now examine aggregate IHS performance using system metrics - H-Mean of IHS IPCs and Weighted Speedup of IHS IPCs, as discussed in Section 4.4. First, Figure 5.5(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 DRAM cache. Overall with simple heterogeneity aware management of the stacked DRAM cache, IHS systems can achieve, on an average, 103% (upto 270%) performance improvement while a naive DRAM cache is able to achieve just 41.8% improvement.

Second, Figure 5.5(b) plots the weighted speedup normalized to the baseline of IHS without DRAM cache. Adding a DRAM cache to IHS processors naively achieves an improvement of merely 3%. In fact for some workloads like Qg3 and Qg11 adding a DRAM cache without careful considerations can lead to negative performance impact. However,

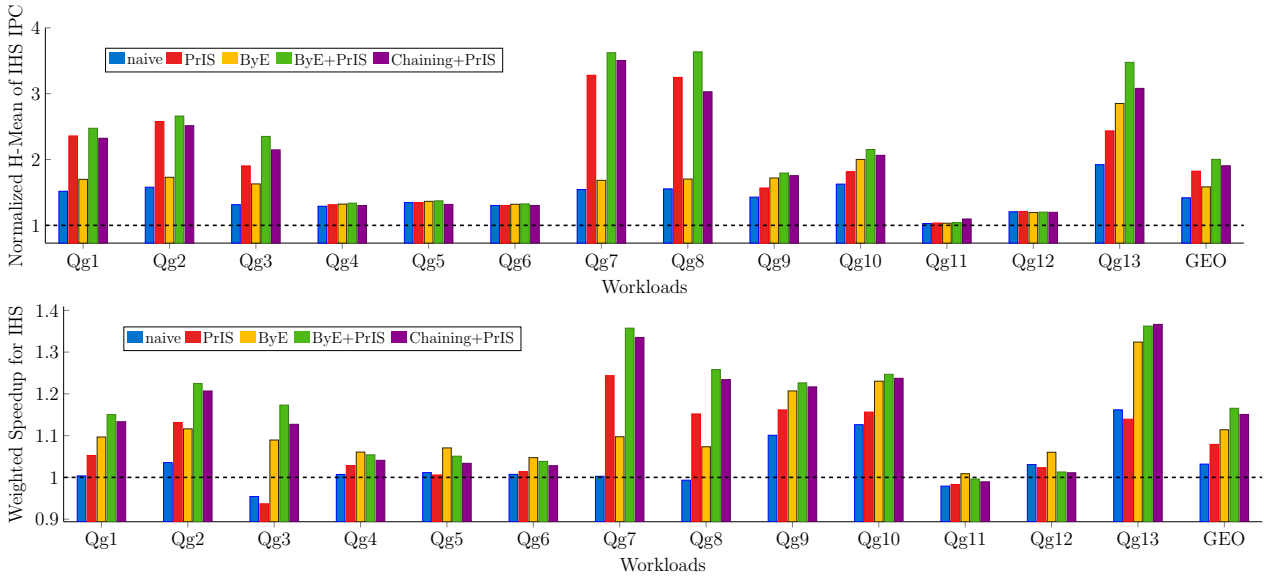


Figure 5.5: (a)Harmonic Mean (b)Weighted Speedup of IHS with HAShCache

our heterogeneity-aware mechanisms are able to achieve on an average of 16% and 15% improvement in performance over a IHS architecture without a DRAM cache. This improvement corresponds to a 12.9% and 11.5% improvement for each of the schemes over a carefully designed but heterogeneity unaware DRAM cache for the IHS processors.

5.2 Sensitivity Study

Next we study the sensitivity of HAShCache mechanisms with regard to a few memory system parameters.

5.2.1 Larger Capacity DRAM cache

We conduct a sensitivity study for HAShCache mechanisms with a larger 128MB stacked DRAM cache. Figure 5.6 presents the performance of a naive DRAM cache 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 DRAM cache. We observe that the larger naive DRAM cache is able to achieve 54.5% and 29.8% improvement for CPU and GPU processors. HAShCache mechanisms continue to provide an average improvement of 46% and 39% for the CPU,

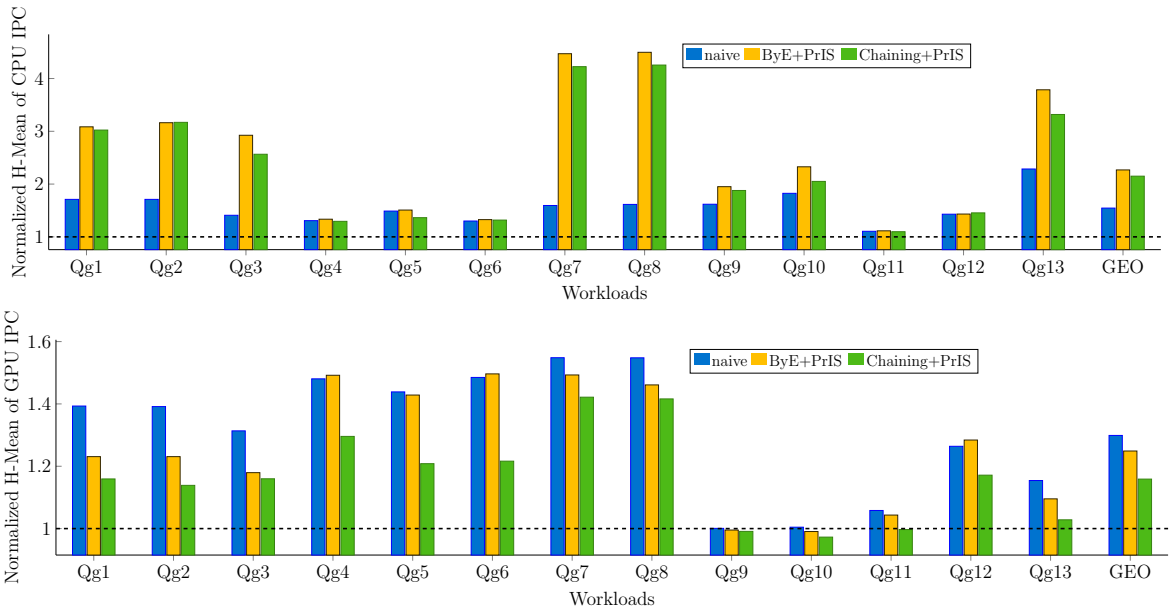


Figure 5.6: Sensitivity Study with 128MB DRAM cache (a)CPU (b)GPU

on an average, above a heterogeneity-unaware DRAM cache and 126% and 115% over the baseline IHS. In the larger DRAM cache, 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 larger GPU footprint.

5.2.2 Off-chip DRAM with same latency as DRAM cache

Although we expect the smaller sizes of stacked DRAMs to have lower latency than off-chip DRAMs, commercial stacked DRAM like those in Intel's Knights Landing [8] have access latencies similar to off-chip DRAMs. Hence, we scale up the off-chip DRAM to 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 5.7 presents the performance (H-Mean of IPC) of (a)CPU (b)GPU with a naive DRAM cache and HAShCache normalized to a IHS system with no DRAM cache. 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 DRAM cache while decreasing GPU performance by 4.4%. Thus, HAShCache mechanisms continue to benefit even when stacked DRAM caches have comparable latency to off-chip DRAM memory.

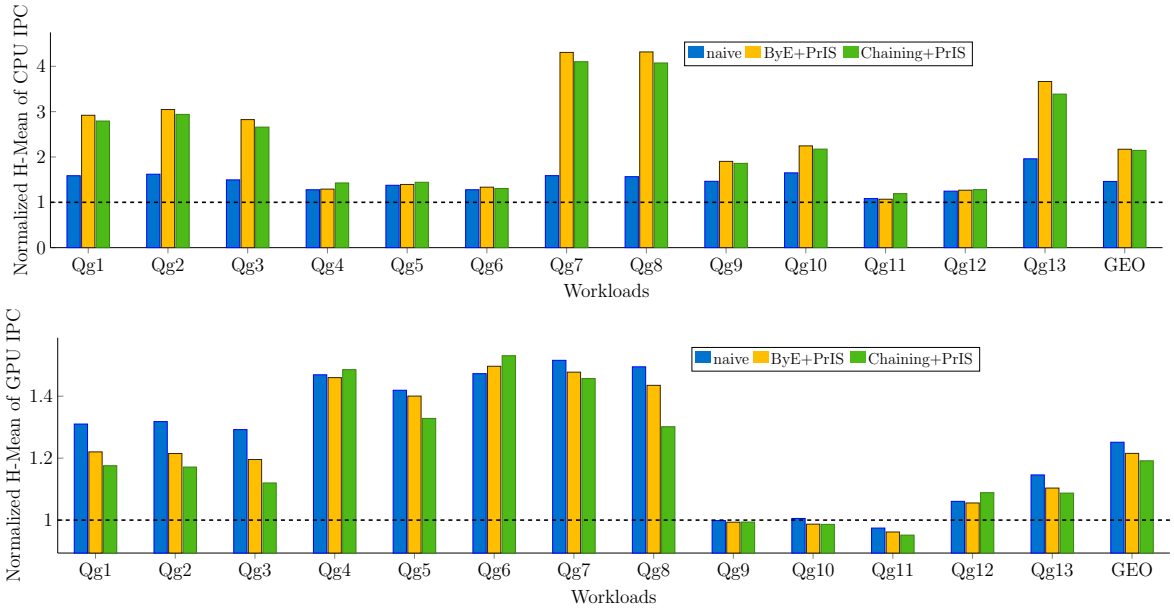


Figure 5.7: Sensitivity Study with DDR3-2133 off-chip DRAM (a)CPU (b)GPU

5.2.3 Higher Bandwidth DRAM cache

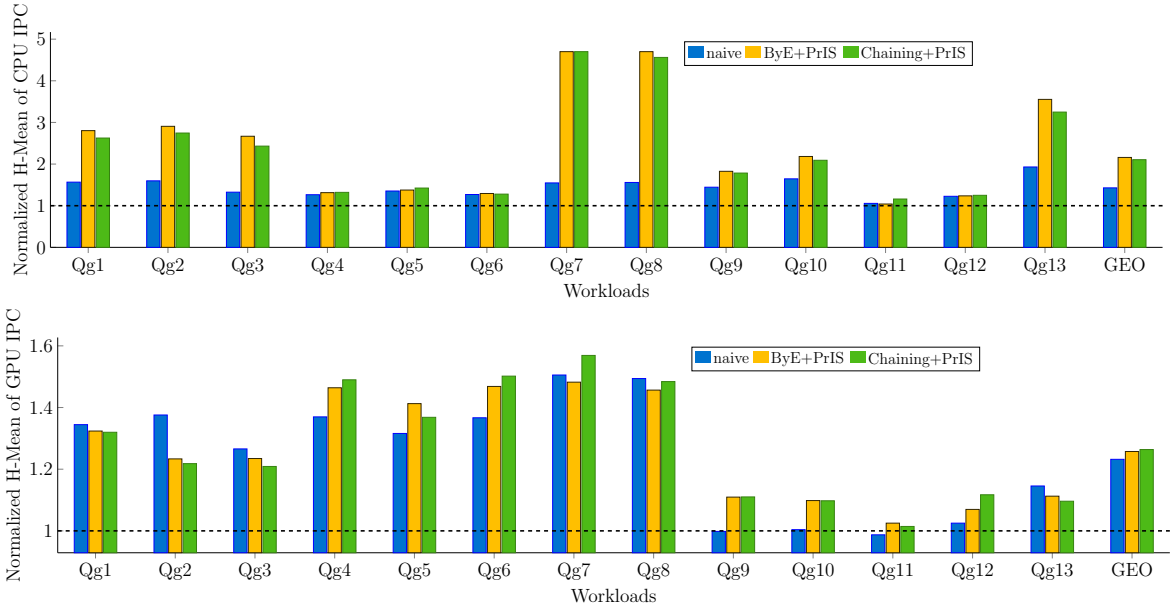


Figure 5.8: Sensitivity Study with 2x Bandwidth for DRAM cache (a)CPU (b)GPU

We carry out experiments for HAShCache with a higher bandwidth for stacked DRAM cache (80GB/s). We achieve this by doubling the burst length of the stacked DRAM cache

devices. Again, Figure 5.8 presents the performance of a naive DRAM cache and HAShCache for (a)CPU and (b)GPU. We observe that our mechanisms scale with increased DRAM cache bandwidth and *ByE+PrIS* performs 51% better and *Chaining+PrIS* performs 47.3% better than a naive DRAM cache for CPU. GPU performs also improves by 1.9% and 2.4% over a naive DRAM cache. Notably for GPU benchmarks like *streamcluster* and *gaussian*, *Chaining+PrIS* performs better than *ByE+PrIS*.

5.2.4 Larger capacity CPU last level SRAM cache

We perform a sensitivity study with a 2MB L2 CPU SRAM cache (double the capacity of baseline configuration). We have not considered increasing GPU L2 size, as such increase (to 1MB or 2MB) is unlikely to make any significant difference due to the streaming nature of these programs.

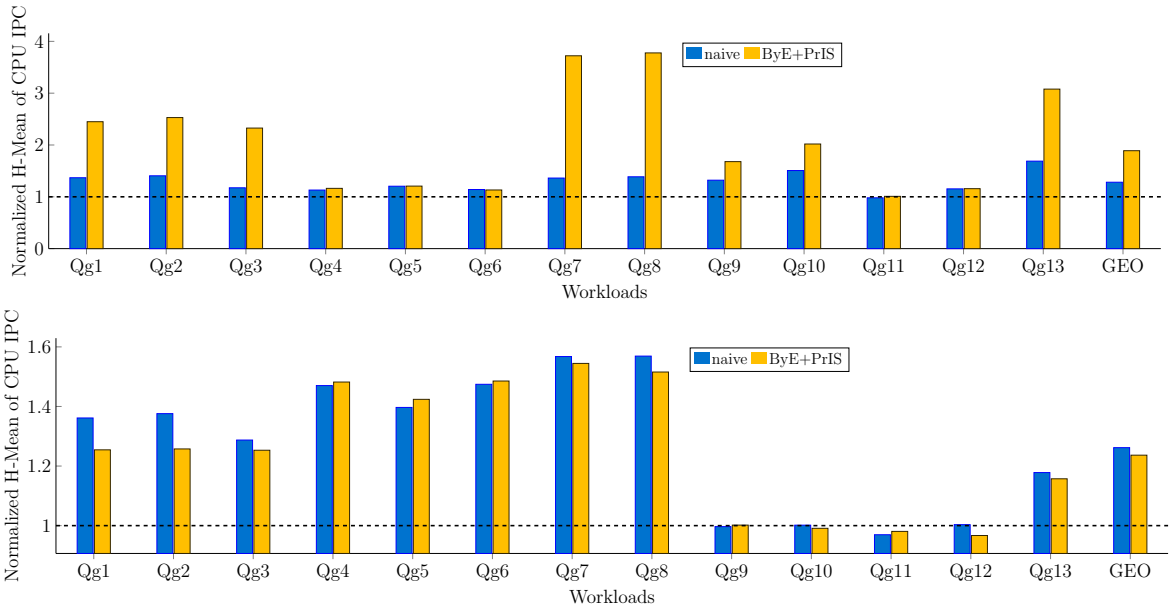


Figure 5.9: Sensitivity Study with 2x CPU SRAM LLC cache (a)CPU (b)GPU

Figure 5.9 presents the performance of (a)CPU and (b)GPU in an IHS with a 2MB L2 CPU SRAM LLC for a naive DRAM cache and our *ByE+PrIS* mechanism, normalized to a IHS system with no DRAM cache. We observe that our mechanism continues to benefit the CPU, showing an average speedup of 47% over a naive DRAM cache. Although for the

CPU, the performance benefit of adding a naive DRAM cache is lower due to the larger CPU L2 cache (28% for IHS with 2MB CPU L2 as compared to 42% for the 1MB CPU L2), yet our mechanism improves performance for CPU workloads over a naive DRAM cache. With the same configuration, the GPU also benefits due to the reduced number of CPU requests at the DRAM cache. *ByE+PrIS* mechanism performs, on an average, only 2% lower than a naive DRAM cache (as compared to the 2.6% loss in the configuration with 1MB CPU L2).

5.3 Comparison with Related Work

The objectives of *ByE* and *PrIS* are, respectively, similar to the Mostly Clean DRAM cache (MCC) [60] and Staged Memory Scheduling (SMS) [14] respectively.

SMS [14] is a DRAM memory access scheduler for IHS architectures. SMS decouples the various tasks for memory access scheduling into three stages. The first stage groups requests based on row-buffer locality from the same requester core. The second stage focuses on policies for inter-application request scheduling i.e., picking one of the request queues from first stage. The third stage consists of FIFO queues that issue DRAM commands while respecting DRAM timing constraints. We use a total request queue size of 64 entries for the first stage in SMS. For each of the four CPU cores we use a 10 entry first stage request queue and a single 24 entry first stage request queue for all the GPU cores which is similar to the specifications in the original SMS work [14]. We note that the total request queue size of 64 entries is the same for *PrIS* as well. Additionally, for the second and third stage of SMS we use a simple 10 entry FIFO queue each.

MCC [60] is a DRAM cache scheme proposed for multi-core CPUs, that attempts to improve performance by dispatching some of the requests to off-chip DRAM. For this they use a hybrid write policy in the DRAM cache that supports write-through and write-back policies for different rows (pages) of the DRAM cache. By permitting only a limited number of pages to be in write-back mode, MCC is able to bound the amount of dirty data in the DRAM cache. This enables the *Self-Balancing Dispatch (SBD)* mechanism to redirect some of these requests to the off-chip DRAM to effectively use the under-utilized off-chip DRAM

bandwidth. For MCC, we implement the *Dirty Region Tracker (DiRT)* structure as proposed in the original work [60], which tracks the number of writes to different pages using a counting Bloom filter and a Dirty List that holds the list of pages that are operated in write-back policy mode. The DiRT structure lookup determines if the request can be sent off-chip. If the lookup into DiRT returns that the access was to a clean row (page), the request is sent to the off-chip DRAM, else it is enqueued in the DRAM cache queue.

A more comprehensive description of the working and mechanisms of SMS and MCC can be found in Section 6. Here we quantitatively compare HAShCache with MCC (adapted for IHS architecture) and SMS. Figure 5.10 presents performance of CPU and GPU normalized to a IHS with no DRAM cache.

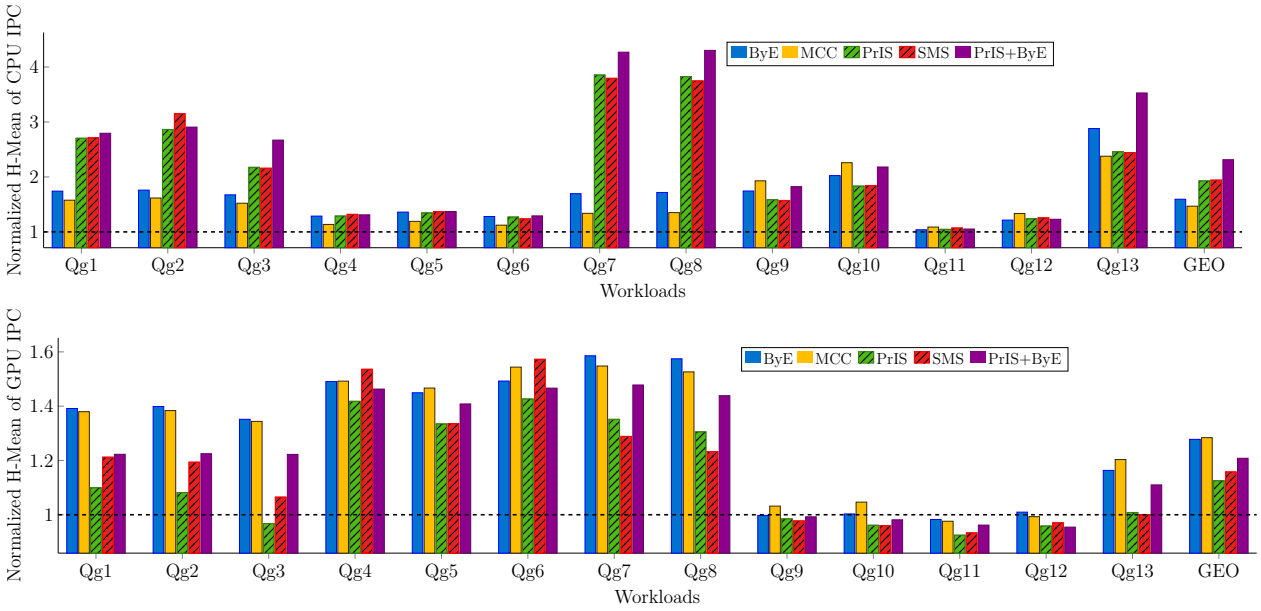


Figure 5.10: Comparison of HAShCache against MCC [60] and SMS [14] (a)CPU (b)GPU

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 source of the request. The large number of GPU requests to the cache lines in the write-through (clean) pages quickly exhausts the available MSHR entries which in turn leads to the DRAM cache 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 5.10 show the performance of the *PrIS* and SMS for CPU and 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. This SJF scheduling reduces latencies for CPU applications as they are serviced quicker, hence reducing the waiting time. 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).

Finally, we note that our combined mechanism of *PrIS+ByE* proposed in this work performs better than each of prior compared schemes for both CPU and GPU.

5.4 Summary

In this chapter, we evaluated the performance of HAShCache its constituent mechanisms and some of their combinations, first on individual CPU and GPU cores and subsequently on IHS architecture as a whole. Next we perform sensitivity studies for capacity, bandwidth and latencies of the memory subsystem, thus showing that HAShCache is robust and performs well for wide variety of memory system parameters. Finally we also present quantitative comparison of HAShCache with two of the state-of-the-art mechanisms — Staged Memory Scheduling [14] and Mostly Clean DRAM cache [60].

Chapter 6

Related Work

In this chapter, we compare out work with related work. In Section 6.1, we present various works on stacked DRAMs. Section 6.2 discusses related work on IHS architectures and organization. Finally in Section 6.3, we present related work in benchmark development for IHS architectures that use both CPU and GPU synergistically.

6.1 3D Stacked DRAM devices

Stacked DRAM devices have primarily been organized to improve the performance of multi-core architectures. Principally there have been two schools of thought for using the stacked DRAM devices

- as an addressable part-of-memory stacked DRAM
- as a hardware managed stacked DRAM Cache

6.1.1 Part-of-Memory

In [59, 22], due to the large capacities provided by these devices, stacked DRAMs are organized as part-of-memory . The designs propose hardware management schemes for swapping hot pages into and out of the stacked DRAM devices. These designs can potentially suffer from large swapping overheads due to the large and disparate working sets of

IHS workloads increasing the number of hot pages in the system. Orthogonal to these, in [48] the authors propose 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. These designs besides incurring the obvious overheads of software modification to improve performance, also require a good understanding of program behaviour and IHS architecture knowledge. They also incur significant overheads due to TLB shootdowns [48].

6.1.2 Hardware Managed Cache

There are several works that propose various organizations for using stacked DRAMs as hardware managed cache [44, 54, 28, 33, 60, 23, 43]. However, all of these works were done with homogeneous multi-core CPUs or GPUs. Our work builds on this body of literature and looks at cache management issues for heterogeneous architectures. We look at some of the significant classes of work below.

- **Cache Granularity**: In the works of [44, 54, 33, 28], the authors propose caching at smaller granularities (less than 512B) and store the resulting large metadata in the rows (pages) of the stacked DRAM. These works also propose various types of predictors like *miss-map* [44], *MAP-I* [54], caching a subset of tags [33], way predictor [28] etc. to determine if a line is present in the DRAM cache. HAShCache adapts the Alloy Cache organization and the *MAP-I* predictor [54] to IHS architectures. In the works of [35], the authors propose allocation at larger page granularity (e.g., 2KB, 4KB) but only fetching useful footprint of the page into the stacked DRAM cache. In Section 3.1, we have justified the design decision of caching granularity in HAShCache for IHS architecture.

- **Access Partitioning:** The works of [60, 26, 29, 23] look at access partitioning between DRAM cache and off-chip DRAM, as the latency of access to these devices is mostly similar. Partitioning the accesses between both these DRAM class devices can improve the effective bandwidth of the system. HAShCache uses the ingrained

disparity in the requests rates and their implication on performance of each core to optimize bandwidth balance in a heterogeneity aware manner. We also compared the performance of HAShCache with Mostly Clean DRAM cache [60] in Section 5.3.

- **TLB assisted DRAM caches:** These DRAM cache organizations piggyback on the TLB translation mechanisms to infer the presence of blocks in the cache. Tagtables [25] achieves this by flipping the page table organization, which requires modification to the TLBs and page walker which are hardware managed in x86 architectures. Tagless DRAM cache [43] provides a more elegant solution where the translation is done from virtual address to a die-stacked cache address, moving the translation to physical address to off-critical path. The GPUs in IHS architectures have several threads running concurrently which already stresses the address translation structures like TLB and page walkers. Overloading these structures to further add DRAM cache meta-data would require careful design decisions and we defer this to future work.

6.2 IHS architectures

Complementary to our work, these body of works discussed below propose optimizations to improve the performance of IHS architectures.

Warp-scheduling and Network-on-Chip: There have been efforts to improve performance of IHS systems in [38] by throttling the GPU cores using intelligent warp scheduling. The work in [42] deals with designing an effective network-on-chip interconnect for IHS architectures.

On-chip shared cache management: To manage shared on-chip SRAM caches, Lee et al. [41] propose heterogeneity aware schemes that are built on top of UCP and RRIP schemes for managing shared resources. While our chaining mechanism is somewhat similar to this to ensure minimum occupancy for CPU requests, it also goes beyond by introducing pseudo-associativity and improving hit-rates (specifically for GPU requests). Mekkat et al. [47] further propose shared SRAM cache management which uses runtime metrics, like

cache sensitivity of each workload, to allocate cache capacities. Despite larger capacities, DRAM caches have higher latencies and hence will not be able to adapt quickly to SRAM occupancy management schemes proposed in these works.

Zhan et al. [67] propose improving performance of IHS architectures by replacing on-chip SRAM caches with a slightly larger STT-SRAM cache that is non-volatile but has 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 DRAM cache.

Memory Access Scheduling: Ausavarungnirun et al. [14] propose Staged Memory Scheduling (SMS) for main memory (DRAM) in IHS processors and is similar in spirit to our *PrIS*, which is applied at the DRAM cache. *PrIS* uses a relatively simpler approach for prioritizing CPU requests while ensuring that GPU requests are not significantly starved. We also compare the performance of HAShCache against SMS in Section 5.3. The authors in [36] 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 which is purely bandwidth-oriented. Hence *PrIS* does not consider such deadlines for the memory controller. A. Stevens [62] proposes QoS and regulating mechanisms for the requests into the shared interconnect to the DDR memory controller between a bandwidth-oriented GPU/VPU sharing and a latency-sensitive CPU. Again the GPU considered is a latency-critical processor for display/rendering. Our *PrIS* mechanism is a heterogeneity-aware memory scheduler which is independent of the interconnect.

Heterogeneous Coherence: As noted earlier, IHS architectures provide a coherent memory between CPU and GPU. The coherence protocol accordingly has to be altered to avoid GPU CUs overwhelming the caches with coherence requests due to their large multi-threaded execution model. In [52], Power et. al. tailor the coherence protocol for IHS architectures. It uses a region-based coherence and reduces large resource requirements.

6.3 Benchmarks

To exploit the full potential of IHS architecture an application (or benchmark) needs to be able to use both the types of cores suitably in its working. Recently Juan et al. developed the Chai benchmarks suite [27] that consists of workloads that collaboratively use both and CPU and GPU for execution. The Chai benchmarks follow data partitioned, fine and course grain task partitioned execution paradigm. These benchmarks could provide better insights into working of IHS programs. We could not use these benchmarks as they were released only in April 2017 and would require significant work to port them for our simulator framework. Hetero-Mark [64] is another benchmark that provides collaborative pattern between CPU and GPU. However, Hetero-Mark has not been authored to be compatible to run with the gem5-gpu simulator that we use for our experimentation. However, for the purposes of our study the shared DRAM cache is receiving requests simultaneously from CPU and GPU which already simulates a busy IHS architecture. Hence, broadly we expect the inferences of HAShCache to be similar with these benchmarks as well.

6.4 Summary

In this chapter, we described the related work in the area of memory systems, IHS architecture and collaborative workloads for these architectures. To the best of our knowledge, no existing approach are aimed at die stacked DRAM organization for IHS architecture. We discuss how HAShCache mechanisms differ from state-of-the-art techniques. We show that even though the area of DRAMs and DRAM cache are explored by several works, incorporating the effects of heterogeneity in IHS architectures is an important consideration while designing DRAM caches.

Chapter 7

Conclusions

In this chapter, we summarize the HAShCache solution and discuss potential avenues for future work.

7.1 Summary

Stacked DRAM caches require careful design to be able to improve performance of workloads. There are several conflicting factors that require careful consideration and trade-offs. This is compounded by the fact that in IHS architectures the processors themselves are heterogeneous and have disparate requirements from the DRAM cache i.e., latency for CPUs vs throughput for GPUs.

In this work, first we demonstrated that current memory hierarchy requires to be revisited for IHS architectures to be able to perform upto capacity. We presented a case for performance improvement of an IHS processor by addition of a stacked DRAM cache. We quantify the effects of interference due to co-running on each processor and show that the heterogeneity adversely affects CPU performance as compared to the GPU. Consequently, we carefully design an effective DRAM cache organization for IHS processors by evaluating the design space of DRAM caches. Further, we propose three simple and effective heterogeneity-aware techniques — a heterogeneity-aware DRAM cache scheduler (*PrIS*), a heterogeneity-aware temporal bypass (*ByE*) and a heterogeneity-aware spatial occupancy

control (*Chaining*) scheme to improve the performance IHS architectures.

We develop the simulation modules and infrastructure to simulate an IHS architecture with a DRAM cache. We use a suitable methodology for detailed simulation of HAShCache- and adapt metrics to measure performance of the proposed system. Using this experimental setup we show that HAShCache achieves significant improvement of 100% in overall system performance on an average over a baseline system with no DRAM cache and 41% over a heterogeneity unaware DRAM cache. We also show that HAShCache mechanisms scale very well with stacked DRAMs of larger capacity, higher bandwidth and even with access latencies of off-chip DRAMs. This work, thus shows that there are significant benefits of using a stacked DRAM cache for IHS processors, far exceeding the usefulness of such devices in homogeneous GPUs and multi-core CPU systems.

7.2 Future Work

An interesting direction for future work would be to explore differential granularity caching for IHS architectures in the stacked DRAM cache for CPU and GPU cores since GPU benchmarks have good spatial locality. Larger blocks with critical sub block first and early restart [34] consequently might prove beneficial as well. Prefetching GPU lines without overly polluting the cache would be another approach to serve GPU lines out of the DRAM cache thus better utilizing the large stacked DRAM bandwidth. However, such techniques need to be mindful that both the memory and cache are slower DRAM devices.

Exploring ways of combining *PrIS*, *Chaining* and *ByE* as a unified technique 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 DRAM cache.

It would also be interesting to explore the performance of HAShCache mechanisms in Collaborative Heterogeneous Applications benchmarks (Chai) [27] where the CPU and GPU are used concurrently by the same application. DRAM cache management schemes using program hints about the collaboration patterns such as data partitioning, fine or

coarse grain task partitioning would also be a interesting future work.

Since the IHS architectures use a shared virtual and physical address space, using TLBs to make the DRAM cache tagless might be a practical solution. However this requires careful consideration of the TLB reach and page walker state machine in IHS architectures.

References

[1] The compute architecture of intel processor graphics gen9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf.

[2] Cuda. cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] Fastest processors, smartphones, and tablets - nvidia tegra. http://www.nvidia.com/object/tegra.html.

[4] The future of the apu - braided parallelism. http://developer.amd.com/wordpress/media/2013/06/2901_final.pdf.

[5] Hsa foundation standards. http://www.hsafoundation.com/standards/.

[6] Integrated cryptographic and compression accelerators on intel architecture platforms. https://www.intel.ru/content/dam/www/public/us/en/documents/solution-briefs/integrated-cryptographic-compression-accelerators-brief.pdf.

[7] Intel graphics opencl. https://software.intel.com/en-us/node/540387.

[8] Intel xeonphi processor datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-processor-x200-product-family-datasheet.pdf.

[9] Java sumatra. http://openjdk.java.net/projects/sumatra/.

[10] Jedec: Ddr3 sdram standard. http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[11] Julia gpu. https://github.com/JuliaGPU/HSA.jl.

[12] Opencl. www.khronos.org/opencl.

[13] Unified memory in cuda. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[14] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 416–427, Washington, DC, USA, 2012. IEEE Computer Society.

[15] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[17] 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. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 469–479, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[19] I. Bratt. Hsa queueing. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 1–43, Aug 2013.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Arpit Joshi Cheng-Chieh Huang, Vijay Nagarajan. Dca: a dram-cache-aware dram controller. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November. 2016.

[22] C. C. Chou, A. Jaleel, and M. K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, Dec 2014.

[23] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 198–210, New York, NY, USA, 2015. ACM.

[24] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[25] S. Franey and M. Lipasti. Tag tables. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 514–525, Feb 2015.

[26] Jayesh Gaur, Mainak Chaudhuri, Pradeep Ramachandran, and Sreenivas Subramoney. Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2017.

[27] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[28] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 38–50, Dec 2014.

[29] Nagendra Gulur, R. Govindarajan, and Mahesh Mehendale. Microrefresh: Minimizing refresh overhead in dram caches. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 350–361, New York, NY, USA, 2016. ACM.

[30] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi. Simulating dram controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210, March 2014.

[31] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[32] C. C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan. C3d: Mitigating the numa bottleneck via coherent dram caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[33] Cheng-Chieh Huang and Vijay Nagarajan. 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, pages 51–60, New York, NY, USA, 2014. ACM.

[34] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[35] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee. Efficient footprint caching for tagless dram caches. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 237–248, March 2016.

[36] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 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, pages 850–855, New York, NY, USA, 2012. ACM.

[37] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 25–37, Washington, DC, USA, 2014. IEEE Computer Society.

[38] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. Managing gpu concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 114–126, Washington, DC, USA, 2014. IEEE Computer Society.

[39] Don Knuth. Notes on "open" addressing, 1963.

[40] J. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, March 2011.

[41] J. Lee and H. Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[42] Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. Design space exploration of on-chip ring interconnection for a cpu-gpu heterogeneous architecture. *J. Parallel Distrib. Comput.*, 73(12):1525–1538, December 2013.

[43] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A fully associative, tagless dram cache. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 211–222, June 2015.

[44] Gabriel H. Loh and Mark D. Hill. 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, pages 454–464, New York, NY, USA, 2011. ACM.

[45] L. Luo, M. Wong, and W. m. Hwu. An effective gpu implementation of breadth-first search. In *Design Automation Conference*, pages 52–55, June 2010.

[46] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[47] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. 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, pages 225–234, Piscataway, NJ, USA, 2013. IEEE Press.

[48] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 188–200, Washington, DC, USA, 2015. IEEE Computer Society.

[49] Prasanna Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[50] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, Jan 2015.

[51] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578, Feb 2014.

[52] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[53] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[54] Moinuddin K. Qureshi and Gabe H. Loh. 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, pages 235–246, Washington, DC, USA, 2012. IEEE Computer Society.

[55] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[56] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, New York, NY, USA, 2009. ACM.

[57] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[58] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, July 2015.

[59] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, Dec 2014.

[60] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–257, Dec 2012.

[61] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

[62] Ashley Steves. Qos for high-performance and power-efficient hd multimedia. In *ARM Whitepaper*. ARM, 2010.

[63] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[64] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

[65] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, and Onur Kayiran. Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[66] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[67] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie. Oscar: Orchestrating stt-ram cache traffic for heterogeneous cpu-gpu architectures. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.