

ON HETEROGENEOUS COMPUTE AND MEMORY SYSTEMS

by

Jason Lowe-Power

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 05/31/2017

The dissertation is approved by the following members of the Final Oral Committee:

Mark D. Hill, Professor, Computer Sciences

Dan Negrut, Professor, Mechanical Engineering

Jignesh M. Patel, Professor, Computer Sciences

Karthikeyan Sankaralingam, Associate Professor, Computer Sciences

David A. Wood, Professor, Computer Sciences

ACKNOWLEDGMENTS

I would like to acknowledge all of the people who helped me along the way to completing this dissertation. First, I would like to thank my advisors, Mark Hill and David Wood. Often, when students have multiple advisors they find there is high “synchronization overhead” between the advisors. However, Mark and David complement each other well. Mark is a high-level thinker, focusing on the structure of the argument and distilling ideas to their essentials; David loves diving into the details of microarchitectural mechanisms. Although ever busy, at least one of Mark or David were available to meet with me, and they always took the time to help when I needed it. Together, Mark and David taught me how to be a researcher, and they have given me a great foundation to build my career.

I thank my committee members. Jignesh Patel for his collaborations, and for the fact that each time I walked out of his office after talking to him, I felt a unique excitement about my research. Karu Sakaralingam for many spirited discussions. Our differing viewpoints always give me something to think about. And Dan Negrut for stepping out of his comfort zone to learn about computer architecture and for teaching me about ways to use hardware for truly useful computations.

I also want to thank a number of my close friends and colleagues who have helped me along the way. Marc Orr began grad school with me, and was an incredibly supportive friend and collaborator throughout our time at Wisconsin. Without Marc suggesting that

we apply for the Summer RA program after our first year, I doubt I would be writing this thesis. Lena Olson was very supportive and always up for a distracting conversation when research wasn't going the way I wanted it to. Somehow, out of all of these distracting conversations, a fruitful collaboration was born. Lena showed me how working with collaborators can have a big impact. Similarly, I also gained a lot from my collaboration with Hongil Yoon. Hongil was a great collaborator, always pushing back when I was wrong.

I would also like to thank the Multifacet group members and other computer architecture students at Wisconsin not already mentioned for their support, mentorship, and encouragement over the years. Specifically my office mates at various times, Somayeh Sardashti and Swapnil Haria provided me with a sounding board for ideas and needed distractions.

Finally, I would like to thank my family for their ever-present encouragement. To my parents, Rena and Greg Power, thank you for sparking my interest in research at a young age and your continued to encouragement and support. To my sister, Marissa Power, thank you for supporting me over the years. And, of course, I deeply thank my partner, Tiffany Lowe-Power. Without your encouragement I would not have been as successful throughout grad school (and life).

CONTENTS

Contents	iii
List of Tables	vi
List of Figures	vii
Abstract	ix
1 Introduction	1
1.1 Trends in computing	1
1.2 Contributions	2
1.3 Thesis organization	6
2 GPU Background	7
2.1 GPU architecture	7
2.2 GPGPU systems	10
2.3 GPGPU programming models	11
3 Simulation Methodology	14
3.1 gem5-gpu: A heterogeneous CPU-GPU simulator	14
3.2 Sampling methodology for native-size workloads	22

4	Tightly-integrated GPGPU Address Translation	26
4.1	<i>Introduction</i>	26
4.2	<i>Background</i>	29
4.3	<i>Simulation methodology and workloads</i>	32
4.4	<i>Designing a GPU MMU through analysis</i>	33
4.5	<i>Correctness issues</i>	43
4.6	<i>Alternative designs</i>	48
4.7	<i>Related work</i>	52
4.8	<i>Conclusions</i>	54
4.9	<i>Artifacts</i>	54
5	High Performance Analytic Databases on Tightly-integrated GPGPU Architec- tures	56
5.1	<i>Introduction</i>	56
5.2	<i>Implementation</i>	59
5.3	<i>Methodology</i>	65
5.4	<i>Results</i>	66
5.5	<i>Related work</i>	73
5.6	<i>Conclusions</i>	75
6	Implications of 3D Die-Stacked Memory for Bandwidth-Constrained Workloads	77
6.1	<i>Introduction</i>	77
6.2	<i>Case study with database scan</i>	82
6.3	<i>Potential big data machine designs</i>	84
6.4	<i>Methodology: Simple analytical model</i>	87
6.5	<i>Results</i>	89
6.6	<i>Discussion</i>	98
6.7	<i>Conclusions</i>	101

6.8	<i>Artifacts</i>	101
7	Adaptive Victim DRAM Cache: Optimizing Bandwidth for HPC Workloads	103
7.1	<i>Introduction</i>	103
7.2	<i>DRAM cache design and access amplification</i>	108
7.3	<i>Using the access amplification model</i>	112
7.4	<i>Adaptive victim cache</i>	114
7.5	<i>Evaluation</i>	127
7.6	<i>Related work</i>	139
7.7	<i>Conclusions</i>	141
7.8	<i>Artifacts</i>	141
8	Conclusions and Future Work	142
	Bibliography	147

LIST OF TABLES

4.1	Details of simulation parameters	33
4.2	Configurations under study	34
5.1	Details of hardware architectures evaluated	64
6.1	Inputs for the analytical model.	89
6.2	Cluster requirements	90
7.1	Details of the KNL-like DRAM cache operation	109
7.2	Access amplification for each design	112
7.3	Details of bandwidth optimized DRAM cache functions	125
7.4	Explanation of rows in Table 7.3.	126
7.5	NPB workload characteristics	128
7.6	Simulated system details.	129

LIST OF FIGURES

2.1	Contemporary discrete GPGPU architecture	9
2.2	Overview of a modern heterogeneous system	10
3.1	Overview of <i>gem5-gpu</i> architecture with an example configuration.	18
4.1	Example GPGPU application	31
4.2	Memory operations	35
4.3	Performance of GPU MMU	36
4.4	Page walk queue size	36
4.5	GPU L1 TLB miss rate	39
4.6	Overview of the GPU MMU designs	41
4.7	Details of the highly-threaded page table walker.	42
4.8	Sharing pattern for L2 TLB	47
4.9	Performance of alternative designs	47
4.10	Energy and area of MMU configurations	51
5.1	Overview of BitWeaving's memory layout	59
5.2	Flowchart showing the CPU-GPU interaction.	60
5.3	Performance and energy of scan microbenchmark.	66
5.4	Power for 1000 scans and energy per scan.	67

5.5	Performance and energy for TPC-H	69
5.6	Performance of aggregates	72
6.1	Memory bandwidth-capacity tradeoff	79
6.2	System diagrams	80
6.3	Performance of scan	83
6.4	Constant performance	93
6.5	Constant power	95
6.6	Constant capacity	97
6.7	Energy and power for each system.	98
7.1	Comparison of access amplification to bandwidth bloat	110
7.2	Breakdown of DRAM cache traffic	112
7.3	Example laundry count and laundry list usage	116
7.4	Operation of the laundry list and laundry counts structures	119
7.5	Details of adaptive victim cache operation	121
7.7	Access amplification for the NPB	129
7.8	Performance of NPB	133
7.9	Performance of different tag array sizes	134
7.10	Laundry list writes breakdown	135
7.11	Performance of DiRT	137
7.12	Super-frames that have a single super-block cached	137

ABSTRACT

Computer systems are at a crossroads. Instead of counting on low-level device improvements from the semiconductor industry, processors are increasingly heterogeneous, using specialized accelerators to increase performance and energy efficiency. Simultaneously, big-data is enabling new applications that consume terabytes of data in real-time, and the Internet of things is driving new data growth by enabling developers to consume data from millions of devices and users. The increasing hardware heterogeneity driven by these technology trends puts significant burdens on the application programmers. For instance, with current interfaces, developers must manually manage all aspects of accelerator computation including explicitly moving data even when the physical memory is shared between devices.

In this thesis, we increase the logical integration of physically integrated on-chip accelerators by extending conventional CPU properties to accelerators. We specifically target on-die graphics processing units (GPUs) which are found on most products from AMD, Intel, and mobile systems and are increasingly used for general-purpose computation. Logical integration between the CPU and GPU simplifies programming these heterogeneous devices. We leverage the logically integrated systems enabled by this research to improve the performance of big-data applications. We show that using integrated GPUs for analytic database workloads can increase performance and decrease energy consumption.

We also evaluate the efficacy of designing systems with heterogeneous memory in addition to heterogeneous computational units. We show that including high-bandwidth 3D-stacked DRAM can significantly improve the performance and reduce the energy consumed for analytic database workloads and other bandwidth-constrained workloads. Finally, we propose a new metric, access amplification, to help system designers reason about the best policies for using heterogeneous memory as a hardware-managed cache. Using access amplification, we design an adaptive victim cache policy which increases the performance of a DRAM cache compared to current designs.

— 1 —

INTRODUCTION

1.1 Trends in computing

There are two conflicting trends in computing today. First, due to the slowdown of Moore's Law and the end of Dennard scaling, general purpose-processors' steady improvements are waning. We can no longer expect general applications to significantly improve performance every processor generation. Using ITRS projections, Esmailzadeh et al. found that by 2024 we can only expect a $7.9\times$ average speedup compared to today's processors from traditional architectural optimizations, more than $24\times$ less than if performance had continued to follow Moore's law [36]. This result is mainly due to chip power constraints; thus, continuing to increasing compute capability requires more efficient architectural designs.

Second, new data is generated at an alarming rate: as much as 2.6 exabytes are created each day [89]. For instance, with the growth of internet-connected devices, there is an explosion of data from sensors on these devices. The ability to analyze this data is important to many different industries from automotive to health care [105]. Users want to run complex queries on this abundance of data, sometimes with real-time response latency constraints. These trends of increasing data, increasing computational complexity, and increasing performance constraints put a great strain on our computational systems,

especially under a fixed power budget.

Systems are becoming more heterogeneous to rectify these trends. Heterogeneity improves the performance and decreases the energy consumed for applications by *specializing* the hardware for a single application or a class of applications. While fully general-purpose processors' improvements are slowing, there is a large space for the specialized processor ecosystem to grow and become more efficient.

One current economically viable example of specialization is graphics processing units, or GPUs. Heterogeneous systems with GPUs began by including GPUs as an I/O device to drive the graphics display, but these devices have recently become more tightly integrated. Today, 99% of Intel's and 67% of AMD's desktop CPUs ship with an on-die GPU [109], and server chips with integrated GPUs have been announced [4].

Heterogeneity is not limited to computational specialization. Recently, systems have integrated multiple different memory technologies such as DRAM DIMMs and 3D-stacked memory [140]. This allows these systems to have memory optimized for capacity (DRAM DIMMs) and memory optimized for performance (high-bandwidth 3D-die stacked memory). With emerging memory technology [8, 59, 80], memory heterogeneity is likely to become more common.

1.2 Contributions

This thesis makes four contributions to optimizing and using heterogeneous systems.

Contribution 1: Simplifying integrated GPGPU programming

The increasing hardware heterogeneity driven by the technology trends discussed above puts significant burdens on the application programmers. With current interfaces, developers must manually manage all aspects of accelerator computation. For instance,

programmers must explicitly move data from CPU to GPU memory, even if the physical memory is shared because each device has a logically separate address space.

Coherent data access and a consistent address space simplify the programmer's interface to tightly-integrated accelerators. Parallel hardware has been ubiquitous for at least the last 15 years, and while programming parallel machines is still difficult, it is simplified by two important properties: coherent data access so when developers access a memory location it will hold the most up-to-date data and a consistent address space so programmers can use the same data structures on any processor in the system. The cache coherence hardware and the virtual memory management unit provide these properties on current CPU systems. Without coherent data access and a consistent address space, programmers manually transform and move data between the CPU and other accelerators. Our previous work provides cache coherence to integrated CPU-GPU systems [114].

In Chapter 4 we show how to provide CPU-style address translation to integrated GPUs. Conventional CPU address translation mechanisms applied directly to the GPU result in a $4\times$ performance degradation due to queuing delays at the centralized address translation hardware. However, we show how to gain the performance back by parallelizing address translation and judiciously using CPU-centric memory management unit optimizations. These techniques logically unify the physically integrated GPU and pave the way for programmers to use the integrated GPU as just another CPU core.

This chapter was originally published as “Supporting x86-64 Address Translation for 100s of GPU Lanes” in *The 20th IEEE International Symposium On High Performance Computer Architecture (HPCA)*, 2014 [116].

Contribution 2: Case study using tightly-integrate GPGUs

The logical integration of GPUs with the rest of the system enables new application domains to take advantage of the GPU. Previous work has shown GPUs can provide up to $100\times$ performance improvement, but these results are limited to relatively small problems (e.g.,

a working set of less than 8 GB). Chapter 4 and Heterogeneous System Coherence [114] enables the GPU to access the hundreds of gigabytes of system memory and significantly reduces the overheads of conventional GPU programming expanding the scope of applications that can leverage this accelerator.

In this thesis, we specifically apply these techniques to analytic databases as a case study. Analytic queries are growing in complexity, and their response time requirements are shrinking. Service providers want to analyze terabytes of data in milliseconds to provide their users with pertinent information (e.g., serving ads, suggesting purchases, and search results). To provide low latency, much of the data is stored in memory, not on disk.

Previous research had shown the significant potential of discrete GPUs; however, due to data movement costs, this potential had not been realized. In Chapter 5, we develop new algorithms to take advantage of physically and logically integrated GPUs. These new scan and aggregate algorithms are $3\times$ faster than traditional GPU algorithms and use 30% less energy than a CPU-only system.

This chapter was published as “Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries” in *The Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN’15)*, 2015 [118].

Contribution 3: Exploring 3D-stacked DRAM for bandwidth-constrained workloads

Exploiting high bandwidth to memory is one of GPUs’ biggest benefits over CPU architecture. Thus, tightly-integrating GPUs onto the same die as the CPU significantly increases the bandwidth demand of the chip making more applications memory-bandwidth constrained.

3D die stacking can help mitigate this bandwidth wall [20]. With 3D die stacking, multiple silicon dies are stacked on top of one another. 3D die-stacking enables very high-bandwidth memory systems, up to 1 TB/s in some projections [2], and decreases the

energy for communication by a factor of $3 \times$ [20].

In Chapter 6, we show integrating 3D die-stacked DRAM into analytic database appliances coupled with the algorithms from Chapter 5 can improve performance by $15 \times$. Further, we conduct a limit study showing these 3D die stacked systems provide the best power-performance tradeoff when the latency of big-data analytic workloads is paramount ($2\text{--}5 \times$ less power to meet a 10 ms response latency guarantee).

This chapter was published as “When to use 3D Die-Stacked Memory for Bandwidth-Constrained Big Data Workloads” in *The Seventh Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (BPOE 7)*, 2016 [88], and integrates a case study from the publication “Implications of Emerging 3D GPU Architecture on the Scan Primitive” in *ACM SIGMOD Record* 2015 [117].

Contribution 4: Decreasing bandwidth overheads in a heterogeneous memory system

While integrating high-bandwidth 3D die-stacked memory (HBM) can provide significant performance improvements by mitigating the off-chip bandwidth bottleneck, its capacity is limited. Therefore, systems with 3D die stacked memory will likely additionally have conventional off-chip DRAM DIMMs for their increased capacity. In a heterogeneous memory system, using HBM as a DRAM cache of off-chip RAM provides transparency to programmers. However, we show in Chapter 7 that treating HBM as a cache wastes its bandwidth because of *access amplification*. Access amplification quantifies the non-demand accesses that waste the HBM bandwidth. We show there are on average two accesses per DRAM cache request when evaluating a current production DRAM cache [140].

We propose an adaptive victim DRAM cache design that avoids access amplification by adaptively shifting between a writethrough and a write-back caching policy. Our adaptive victim cache strives to read metadata only while reading data and to write metadata only while writing data. This design performs robustly better than a current DRAM cache

design under a wide range of memory access patterns including high miss rates and high write traffic.

This chapter is currently under review for publication.

Other contributions: Simulation infrastructure

Another contribution of this thesis is new simulation infrastructure which enabled the above contributions. We developed `gem5-gpu` [115], described in Section 3.1, to simulate heterogeneous CPU-GPU platforms that did not physically exist at the time. Additionally, in Chapter 7, we focus on large high-performance computing workloads that are infeasible to simulate using conventional techniques. Therefore, we developed a sampling methodology combining previous work into a more easily usable system. We describe this methodology in detail in Section 3.2.

The work on `gem5-gpu` was published as “`gem5-gpu`: A Heterogeneous CPU-GPU Simulator” in *IEEE Computer Architecture Letters*, 2015 [115].

1.3 Thesis organization

This thesis is organized as follows. Chapter 2 discusses GPU architecture and GPU programming models used in Chapters 4 and 5. Chapter 3 describes the simulation methodology used in Chapters 4 and 7. Chapter 4 presents the design and evaluation of a memory management unit for integrated GPUs. Chapter 5 develops and evaluates new algorithms for database scan and aggregate for tightly-integrated GPUs. Chapter 6 explores system design for bandwidth-constrained workloads evaluating the tradeoffs between 3D stacked DRAM and conventional DRAM DIMMs. Chapter 7 evaluates DRAM cache designs and proposes a new DRAM cache policy to decrease bandwidth waste. Finally, Chapter 8 summarizes our work and outlines future research directions.

— 2 —

GPU BACKGROUND

In this thesis, we augment the GPU architecture in Chapter 4 and leverage the GPU microarchitecture in Chapters 5 and 6. This chapter provides details on the GPU’s microarchitecture and programming models used in this thesis.

2.1 GPU architecture

Graphic processing units (GPUs) were originally created to accelerate rendering graphics to the screen; however, in recent years they have become general purpose compute platforms [107]. These general-purpose GPUs (GPGPUs) have a number of characteristics that differ from conventional CPUs:

- GPGPUs are optimized for instruction throughput, not instruction latency.
- GPGPUs can exhibit much higher memory-level parallelism than CPUs.
- GPGPUs have high-bandwidth and low-capacity on-chip caches.
- GPGPUs devote a higher percentage of area to compute functional units than CPUs.

Modern GPUs are high-throughput devices. A GPU is made up of a set of *compute units* (CUs, SMs in NVIDIA terminology). Each CU contains many processing elements, organized into four 16-wide *SIMD units* in AMD graphics core next (GCN) architecture [7]. Each CU has a scheduler that schedules instructions onto the SIMD units. Each SIMD unit of the CU has a set of *SIMD lanes* which execute the same instruction each cycle in lock step (similar to SIMD execution units in CPUs) [78]. NVIDIA GPU architectures are similar with 16–192 SPs which execute in lock step [98]. Figure 2.1 shows an overview of GPU hardware.

GPUs provide the potential for much higher parallelism than CPUs. With well structured code, all of the SIMD lanes on the GPU can simultaneously execute instructions. Similar to some CPU architectures, GPUs leverage multithreading to increase memory-level parallelism. However, where CPU cores typically support two–eight hardware contexts, GPU CUs can support 40 or more hardware contexts. Also, by leveraging multithreading, the GPU can take advantage of high memory-level parallelism. High memory-level parallelism allows the GPU to effectively utilize high memory bandwidth and tolerate high memory latencies.

To support this high memory-level parallelism, the memory system and caches of GPUs are optimized for throughput, not latency. GPU architectures provide a much larger register file than CPUs to support the many execution contexts. GPUs typically have a 16–64 KB L1 cache per CU and a shared 1–2 MB L2 cache [7]. The general-purpose caches of GPUs are much smaller than the caches on multicore CPUs, especially when considering the capacity per-thread. Large caches are less useful for highly data-parallel applications because they often exhibit poor temporal locality.

In addition to these features, each CU of the GPU also contains two special memory system optimizations. First, between the SIMD lanes and the L1 data cache is a *coalescing unit* that takes the memory addresses generated by each of the SIMD lanes and attempts to coalesce them into the minimum number of memory references (e.g., if all 64 address

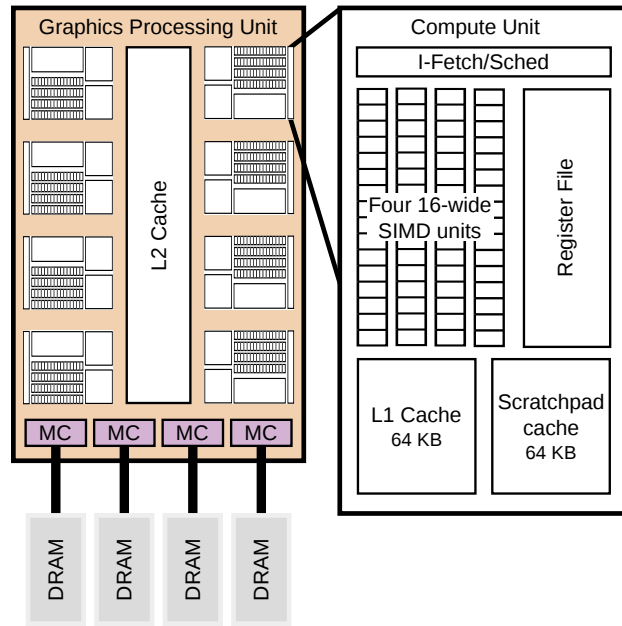


Figure 2.1: Contemporary discrete GPGPU architecture. CU microarchitecture is the same for the integrated GPU.

are consecutive and aligned the coalescer will generate four 64-byte requests instead of 64 4-byte requests). Application performance depends on good coalescing behavior, especially for memory-bound workloads.

The second GPU-specific memory optimization is the addition of a directly addressed scratchpad cache (called local memory or group memory by AMD and shared memory by NVIDIA). This small (~64 KB) cache is explicitly controlled by the programmer. Requests to the scratchpad cache do not need to access cache tags since the scratchpad is directly addressed. Memory requests to the scratchpad do not pass through the coalescer and do not access the low-bandwidth cache tags, and, therefore, the scratchpad cache is higher bandwidth than the data cache for requests that cannot be coalesced. It is common to load data into the scratchpad cache that has a poor data access pattern to avoid the high cost of uncoalesced accesses to the main memory system.

Finally, GPU architecture can be more energy efficient than CPU architecture for certain workloads (e.g. database scans). Since many SIMD lanes share a single front-end

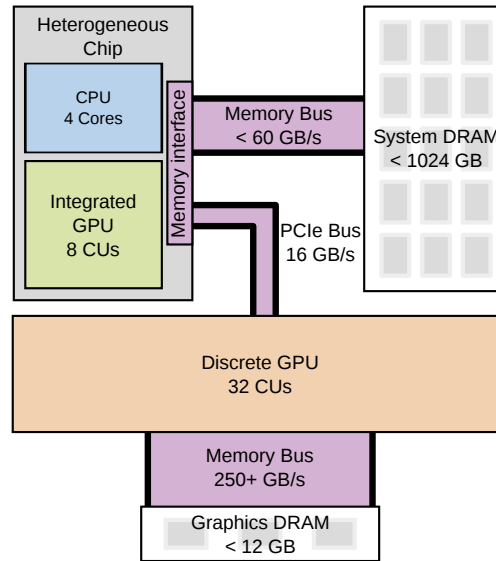


Figure 2.2: Overview of a modern heterogeneous system. Drawn comparatively to scale.

(instruction fetch, decode, etc.), this per-instruction energy overhead is amortized. On CPU architectures, the execution front-end and data movement consumes $20\text{--}40\times$ more energy than the actual instruction [73]. Additionally, the intra-CU parallelism is explicit for GPUs through the programming model, while CPUs require high energy hardware (like the re-order buffer and parallel instruction issue) to implicitly generate instruction-level parallelism, which wastes energy for data-parallel workloads.

2.2 GPGPU systems

Figure 2.2 shows a system-level view of GPGPU hardware. The discrete GPU is separated from the CPU, usually by the PCI Express (PCIe) bus, and can be optionally included in the heterogeneous system. Discrete GPUs have their own physical memory, and to use the discrete GPU all data must be copied from CPU memory to device memory via the PCIe interconnect. However, the discrete GPU is capable of the highest performance between these three architectures, with more compute resources (e.g. functional units) than the CPU and much higher memory bandwidth. Unfortunately, the discrete GPU is inappropriate

for many workloads due its limited memory capacity. To use the discrete GPU with data sets larger than about 12 GB (the largest memory available for discrete GPUs today), data must be copied at execution time over the low-bandwidth PCIe bus.

Another downside of discrete GPUs are their high power consumption. Discrete GPUs often run at higher power than CPUs for two reasons. First, they have many more compute resources simultaneously active. Second, the power drawn by discrete GPUs includes the GDDR5 memory, which is both higher bandwidth than CPU DDR3 and DDR4 memory, but also higher power due to its higher bandwidth.

The integrated GPU is co-located on the same silicon chip as the CPU cores. The integrated GPU shares hardware with the CPU, including the off-chip DRAM, memory interface, and sometimes the last-level cache. The integrated GPU often has more compute resources than the multicore CPU since GPUs have smaller caches, which frees up silicon area for more functional units.

Current systems show the key differences between CPUs and GPUs well. GPUs have an order of magnitude more execution units (e.g. an AMD A10-7850 has 4 CPU cores each with 5 execution units, and an AMD HD7970 discrete GPU has 32 compute units each with 64 execution units [6]), and GPUs provide much higher memory bandwidth (e.g. an AMD A10-7850 CPU has a memory bandwidth up to 34 GB/s [5], and an AMD HD7970 GPU has a memory bandwidth of 264 GB/s [6]). Additionally, the CPU has 8–16+ MB of on-chip SRAM caches, whereas the GPU only has 2–4 MB.

2.3 GPGPU programming models

New GPGPU APIs (application programming interfaces) simplify programming and increase performance of GPGPU computing. Until recently, GPGPU APIs were designed for discrete GPUs. Communication between the CPU and GPU was high-overhead, and all data was explicitly declared and copied before it was used on the GPU. However, while

GPGPUs are becoming more physically integrated with CPUs, they are also becoming more logically integrated. For instance the runtime we use in Chapter 5, heterogeneous system architecture (HSA), provides a coherent and unified view of memory [127]. Other runtimes are similar (e.g., C++ AMP [94] and CUDA unified memory [101]).

HSA support is in its infancy today and is currently only supported by platforms from AMD. However, many members of the HSA foundation (<http://www.hsafoundation.com/>)—which includes AMD, ARM, Imagination, Qualcomm, Samsung, and others—have announced support for providing GPUs with a coherent and unified view of memory. In addition to a unified view of memory, HSA also provides a low-latency user-level API for using the GPU.

Programming the discrete GPU can be high overhead, both in terms of performance and programmability. Most discrete GPU applications follow the same general outline. First, the data must be allocated and initialized within the CPU memory space. Then, memory must be allocated on the GPU. Since the GPU has its own memory, all data that is accessed on the GPU must be copied from the CPU memory to the GPU memory. After the data has been copied to the GPU, the kernel—the code to execute on the GPU—is launched. After the kernel completes, the output data must be copied back to the CPU to continue processing the data. Each of these steps is an OpenCL API call and must interact with the operating system display driver. This interaction introduces significant performance overheads.

However, programming the integrated GPU is greatly simplified. Due to HSA, after the memory is allocated and initialized in the CPU memory space, the GPU kernel is launched as if it is a CPU function. The pointers created for the CPU can be reference from the GPU. Thus, after the kernel completes there is no need to copy data back to the CPU memory space. This unified view of memory also enables the GPU to access complicated pointer-based data structures, like hashtables. Section 5.2 discusses how this support facilitates implementing group-by operations on the integrated GPU. Figure 4.1 in Chapter 4 shows

a detailed comparison between the legacy interface, NVIDIA's unified memory, and the HSA API.

— 3 —

SIMULATION METHODOLOGY

Investigating forward-looking heterogeneous systems requires simulation infrastructure. In this chapter, we discuss the simulation infrastructure developed to support the research objectives in this thesis. First, we detail `gem5-gpu`, a heterogeneous simulator built to model tightly physically and logically integrated GPUs and used in Chapter 4 to evaluate new memory management unit hardware for integrated GPUs. Then, we overview the infrastructure developed to simulate large memory bandwidth constrained high performance computing workloads used in Chapter 7 to evaluate DRAM cache policies.

3.1 `gem5-gpu`: A heterogeneous CPU-GPU simulator

To evaluate the heterogeneous CPU-GPU system in this chapter, we developed a new simulator, `gem5-gpu`. We leverage two mature simulators, `gem5` [19] and `GPGPU-Sim` [10]. `gem5` is a multicore full-system simulator with multiple CPU, ISA, and memory system models. Object oriented design, flexible configuration support, and its maturity make `gem5` a popular tool for investigating general purpose CPUs and multicore platforms. `GPGPU-Sim` is a detailed general-purpose GPU (GPGPU) simulator [10]. `GPGPU-Sim` models GPGPU compute units (CUs)—called streaming multiprocessors by NVIDIA—and

the GPU memory system.

To explore the heterogeneous system design space, we introduce the *gem5-gpu* simulator which combines the CU model from GPGPU-Sim and the CPU and memory system models from *gem5*. *gem5-gpu* builds on ideas used in related CPU-GPU simulators but makes different design choices.

gem5-gpu is the only simulator with all of the following advantages:

- Detailed cache coherence model,
- Full-system simulation,
- Checkpointing,
- Tightly integrated with the latest *gem5* simulator, and
- Increased extensibility of GPGPU programming model and entire system architecture.

By integrating GPGPU-Sim’s CU model into *gem5*, *gem5-gpu* can capture interactions between a CPU and a GPU in a heterogeneous processor. In particular, GPGPU-Sim CU memory accesses flow through *gem5*’s Ruby memory system, which enables a wide array of heterogeneous cache hierarchies and coherence protocols. *gem5-gpu* also provides a tunable DMA engine to model data transfers in configurations with separate CPU and GPU physical address spaces. Through these features *gem5-gpu* can simulate both existing and future heterogeneous processors.

gem5-gpu is open source and available at gem5-gpu.cs.wisc.edu.

The giant’s shoulders

gem5-gpu builds upon prior simulation infrastructure, *gem5* and GPGPU-Sim, briefly described below.

gem5

The gem5 simulation infrastructure (gem5.org) is a community-focused, modular, system modeling tool, developed by numerous universities and industry research labs [19]. gem5 includes multiple CPU, memory system, and ISA models. It provides two execution modes, (1) system call emulation, which can run user-level binaries using emulated system calls, and (2) full-system, which models all necessary devices to boot and run unmodified operating systems. Finally, gem5 also supports checkpointing the state of the system, which allows simulations to jump to the region of interest.

Two specific features of gem5 make it particularly well-suited for developing a heterogeneous CPU-GPU simulator. First, gem5 provides several mechanisms for modular integration of new architectural components. When new components need to communicate through the memory system, they can leverage gem5’s flexible port interface for sending and receiving messages. Additionally, the gem5 EXTRAS interface can be used to specify external code that is compiled into the gem5 binary. This interface makes it simple to add and remove complex components from gem5’s infrastructure.

Second, gem5 includes the detailed cache and memory simulator, Ruby. Ruby is a flexible infrastructure built on the domain specific language, SLICC, which is used to specify cache coherence protocols. Using Ruby, a developer can expressively define cache hierarchies and coherence protocols, including those expected in emerging heterogeneous processors.

GPGPU-Sim

GPGPU-Sim is a detailed GPGPU simulator (gpgpu-sim.org) backed by a strong publication record [10]. It models the compute architecture of NVIDIA Fermi graphics cards [100]. GPGPU-Sim executes applications compiled to PTX (NVIDIA’s intermediate instruction set) or disassembled native GPU machine code. GPGPU-Sim models the functional and timing portions of the compute pipeline including the thread scheduling logic, highly-banked

register file, special function units, and memory system. GPGPU-Sim includes models for all types of GPU memory as well as caches and DRAM.

GPGPU applications can access a multitude of memory types. *Global memory* is the main data store where most data resides, similar to the heap in CPU applications. It is accessed with virtual addresses and is cached on chip. Other GPU-specific memory types include *constant*, used to handle GPU read-only data; *scratchpad*, a software-managed, explicitly addressed and low-latency in-core cache; *local*, mostly used for spilling registers; *parameter*, used to store compute kernel parameters; *instruction*, used to store the kernel's instructions; and *texture*, a graphics-specific, explicitly addressed cache.

GPGPU-Sim consumes mostly unmodified GPGPU source code that is linked to GPGPU-Sim's custom GPGPU runtime library. The modified runtime library intercepts all GPGPU-specific function calls and emulates their effects. When a compute kernel is launched, the GPGPU-Sim runtime library initializes the simulator and executes the kernel in timing simulation. The main simulation loop continues executing until the kernel has completed before returning control from the runtime library call. GPGPU-Sim is a functional-first simulator; it first functionally executes all instructions, then feeds them into the timing simulator.

GPGPU-Sim has some limitations when modeling heterogeneous systems:

- No host CPU timing model
- No timing model for host-device copies
- Rigid cache model
- No way to model host-device interactions

Because of these limitations, researchers interested in exploring a hybrid CPU-GPU chip as a heterogeneous compute platform cannot rely on GPGPU-Sim alone.

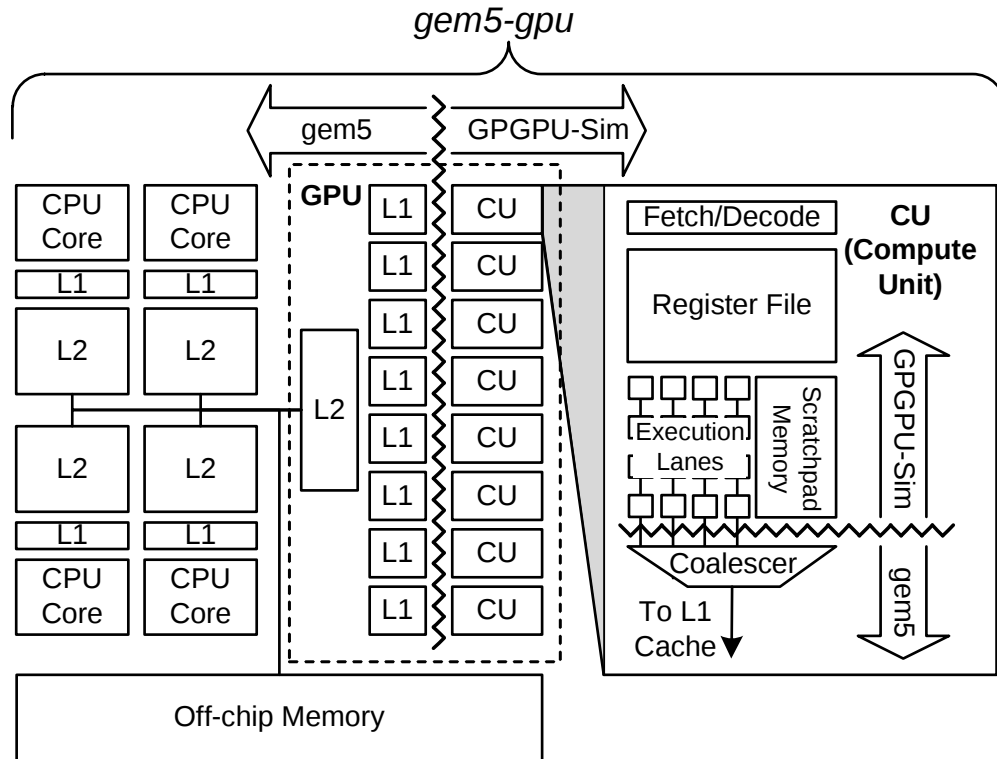


Figure 3.1: Overview of *gem5-gpu* architecture with an example configuration.

gem5-gpu Architecture

Figure 3.1 shows one example architecture *gem5-gpu* can simulate: a four core CPU and an eight CU GPU integrated on the same chip. The number of CPUs, CUs, and topology connecting them is fully configurable. Two on-chip topologies that *gem5-gpu* provides out of the box are a shared and a split memory hierarchy (i.e., integrated and discrete GPUs, respectively).

Many CUs make up the GPU, each of which has fetch/decode logic, a large register file, and many (usually 32 or 64) execution lanes. When accessing global memory, each lane sends its address to the coalescer, which merges memory accesses to the same cache block. The GPU may also contain a cache hierarchy that stores data from recent global memory accesses.

gem5 $\longleftrightarrow$ GPGPU-Sim interface

One of our goals is to have a clean interface between gem5 and GPGPU-Sim. Although there are many possible options, we chose the memory interface, as shown in Figure 3.1. *gem5-gpu* requires one small change to gem5: adding a single pseudo-instruction to gem5 to facilitate calls into the simulator for DMA engine and GPU functionality. *gem5-gpu* also provides two categories of GPGPU objects to gem5: wrappers for GPGPU-sim objects and new GPGPU objects. Some new GPGPU objects are a GPU load-store unit, a GPU TLB, and the DMA engine, which models DMA transfers between the CPU and the GPU.

Three major changes to GPGPU-Sim were necessary to integrate it into *gem5-gpu*. First, *gem5-gpu* routes general-purpose memory instructions—accesses to the global address space—from GPGPU-Sim to Ruby through gem5’s port interface. Second, since GPGPU-Sim decouples functional execution and timing state, while gem5 is execute-at-execute, we updated GPGPU-Sim to postpone functional execution of memory instructions until they reach the execute stage of the timing pipeline. Third, to combine the cycle-driven nature of GPGPU-Sim with gem5, which is event-driven, gem5’s events are used to execute each GPGPU-Sim clock cycle. This decouples the CPU from GPU, allowing it to block or continue concurrently with the GPU after a kernel launch.

Memory system modeling

gem5-gpu uses Ruby to model both the function and timing of most CU memory accesses. The load-store pipeline is modeled in gem5, including the coalescing, virtual address translation, and cache arbitration logic. By using the port interface in gem5, *gem5-gpu* has the flexibility to vary the number of execution lanes, number of CUs, cache hierarchy, etc. and incorporate other GPU models in the future.

Currently, GPGPU-Sim issues only general-purpose memory instructions to gem5, including accesses to global and constant memory. We leverage GPGPU-Sim to model memory operations to scratchpad and parameter memory. Texture and local memory are

not currently supported although they only require straight-forward simulator augmentation.

gem5-gpu supports a shared virtual address space between the CPU and GPU (i.e., the GPU using the CPU page table for virtual to physical translations). Alternatively, through a configuration option, *gem5-gpu* models separate GPU and CPU physical address spaces.

Detailed cache coherence models

gem5-gpu leverages *gem5*'s cache coherence modeling language, SLICC. When configuring *gem5-gpu*, any cache coherence protocol can be used, including the multitude that are currently distributed with *gem5*. However, these included protocols assume a homogeneous cache topology.

To augment these for heterogeneous computing, *gem5-gpu* adds a family of heterogeneous cache coherence protocols: *MOESI_hsc* (heterogeneous system coherence with MOESI states). *MOESI_hsc* uses a MOESI protocol for all of the CPU caches included in the system. For the GPU caches, we add an L2 cache controller that provides coherence between the GPU and CPU L2 caches. *MOESI_hsc* models the GPU L1 cache similar to current GPU cache architectures: write-through and only valid and invalid states. Additionally, the GPU L1 cache may contain stale data that is flushed at synchronization points and kernel boundaries.

gem5-gpu also includes a split version of *MOESI_hsc* that models architectures with separate CPU and GPU physical address spaces. When this model is used, communication between the CPU and GPU requires explicit memory copies through the DMA engine.

In addition to providing detailed cache coherence models, *gem5-gpu* can use any of the network topologies provided by *gem5* (e.g., mesh, torus, crossbar). *gem5-gpu*'s default configuration uses the cluster topology to divide the CPU and GPU into two clusters. Accesses from each cluster goes through a common interconnect to the directory and memory controllers.

Application programming interface

To avoid the complexity of implementing or interfacing existing GPGPU drivers and runtimes, *gem5-gpu* provides a slim runtime and driver emulation. Similar to GPGPU-Sim, *gem5-gpu* runs unmodified GPGPU applications by linking against the *gem5-gpu* GPGPU runtime library. When a user application calls a GPGPU runtime function, *gem5* pseudo-instructions execute to make up-calls into the simulator. This organization is flexible and extensible, making it convenient to add and test new features and integrate new GPU models.

Comparison to other simulators

In contrast to most other heterogeneous simulators, *gem5-gpu* captures interactions with execution-driven simulation rather than well-partitioned trace-driven simulation, e.g., MacSim [77]. It uses a more-detailed—therefore slower—GPU component than MV5 [91] and does not rely on the deprecated m5 simulator. It supports more flexible memory hierarchy and coherence protocols than Multi2Sim [148] or FusionSim [157] at a possible increase in simulation time.

Since the work was completed, AMD has released a GPU model as part of *gem5* [14]. This model integrates into the Ruby memory system similarly to *gem5-gpu*. The main difference is that AMD’s GPU model uses HSAIL, not PTX as the intermediate assembly language and OpenCL, not CUDA, as the main programming model.

Heterogeneous workloads

In Chapter 4, we evaluated the Rodinia workloads [27]. We modified these workloads to support the programming model enabled by implementing cache coherence and a shared virtual address space. We call our new workloads the “rodinia-nocopy” since we removed all of the copy statements from the workloads.

For most workloads, simply removing the copy from the host to device and using the host pointers was sufficient to implement the no copy version of the benchmark. However, in some cases, the memory copy transformed the data from one layout to another. In these cases, we had to further modify the workload to use the transformed data layout on the CPU as well as the GPU. We did not change the GPU kernels to have a constant comparison point with the legacy versions of the benchmarks.

Since the work in this chapter was completed, there have been many new GPU benchmarks released. Of note is Pannotia [26], whose workloads are much more irregular than Rodinia. Most of the Pannotia workloads are graph-based. Additionally, the Chai [45] workloads target “truly heterogeneous” benchmarks which use the CPU and GPU simultaneously. Both Pannotia and Chai have been ported to gem5-gpu with “nocopy” versions of the benchmarks.

3.2 Sampling methodology for native-size workloads

Simulation is an important tool for computer architecture research. It allows designers and researchers to evaluate future hypothetical systems without physically building them, which can be expensive. Additionally, it is much more flexible than physical hardware, allowing researchers to more easily explore many design parameters. However, simulation has drawbacks. One drawback is its very slow execution time.

Some important workloads, like the HPC workloads evaluated in Chapter 7, have large working set sizes and long execution times. Straightforwardly simulating these workloads is infeasible. For instance when running gem5 [19] in full-system mode with 32 cores the execution time is between $300,000\times$ and $500,000\times$ slower than native execution. To simulate one minute of native time it takes about a year of simulation time, and many workloads execute for hours!

There is a rich history of sampling methodologies for computer architecture simula-

tion [136, 155]. However, these previous techniques rely on fast-forwarding via high-level simulation, which still results in large slowdowns (e.g., $1000\times$ or more).

Therefore, we leverage recent work which uses virtualization hardware to fast-forward at near native speeds [133]. Sandberg et al. use the KVM virtualization solution built into modern Linux kernels and essentially implement another CPU model in gem5.

Our contribution beyond the state of the art discussed above is packaging these features into a set of usable scripts. An important feature of this methodology is its scalability. For instance, to simulate 100 sample points, this methodology allows the use of ten machines in parallel each producing data for ten sample points.

Description of sampling methodology

There are three phases in our sampling methodology: program analysis, random sampling, data analysis. Algorithm 1 shows an overview of our sampling algorithm.

Program analysis

First, we run the application in gem5 with the KVM CPU for its entire runtime from the beginning to the end of the region of interest. Currently, the only information we use from this phase is the total number of instructions executed during the region of interest. Section 8 discusses some ideas of how to extend this analysis to provide richer information to guide the sampling described below.

Random sampling

Currently, we use simple random sampling to choose when to run detailed simulation. Taking the instruction count from the program analysis phase, we choose n random instructions uniformly from the total number of instructions to perform detailed simulation (line 2). The simulation results beginning at a particular instruction is an *observation*. Once we have a list of instructions for which we will begin a simulation point, we start executing

Algorithm 1 Sampling algorithm

Require: n : number of observation points

Require: instructions: total number of instructions

```

  points = [] // Instructions to begin simulation
  for i := 1 to n do
2:   points  $\leftarrow$  random(0, n)
    Sort points
4:   for each point in points do
      Run application under KVM until point instructions executed
6:   Fork new process
      Warm up the system for warmup time
8:   Run detailed simulation for detailed time
      Dump statistics
10:  End new process

```

the application from the beginning under KVM. We stop the application at each of the instruction numbers in points (line 5) to perform an observation.

Then, we essentially “pause” the application running on the KVM CPU. In the meantime, we fork a new process on the host machine to simulate beginning at the observation point instruction. In this new process, we perform warmup and detailed simulation with gem5. In Chapter 7, we use a warmup for 10 ms and perform detailed simulation for 2 ms of guest time (about 10,000,000–30,000,000 instructions of detailed simulation). After the simulation, we dump all of the statistics, exit the subprocess, and continue running the KVM CPU until the next observation point.

These observation locations are non-deterministic. Due to operating system effects such as thread scheduling and I/O each time the application is executed the total number of instructions varies. Additionally, KVM does not exit at precisely the requested number of instructions. This non-determinism may cause bias in our sample statistics; we have not analyzed this methodology for bias.

Data analysis

The goal when measuring application performance is to predict the total runtime of the application, or the amount of time the applications uses to complete its *work*. We choose to measure either the floating point operations per second (FLOPs) or user-mode instructions for integer-only applications. We use the number of these instructions executed as a proxy for the the *work* the application accomplishes during a single observation. Thus, we assume a higher FLOP rate implies the application will complete in less time (e.g., have higher performance).

When comparing performance of different simulation configurations, we use the mean of the FLOPs or user-mode operations from our sample of application execution. In Chapter 7 we additionally present a 95% confidence interval for this sample mean calculated using the Student's t-distribution.

Summary

With this sampling methodology, we can evaluate natively-sized workloads. We do not require workload modification to make simulation feasible. By using unmodified workloads, we ensure we are capturing their true behavior which gives us more confidence in our conclusions. There are many ways to make this methodology even more robust and to extend it to other workloads outside of scientific computing. Section 8 describes some ideas for extending this methodology.

— 4 —

TIGHTLY-INTEGRATED GPGPU ADDRESS TRANSLATION

4.1 Introduction

Although physical integration of CPUs and GPUs is becoming widespread, the GPGPU compute platform is still widely separated from conventional CPUs in terms of its programming model. CPUs have long used virtual memory to simplify data sharing between threads, but GPUs still lag behind.

A shared virtual address space allows “pointer-is-a-pointer” semantics [126] which enable any pointer to be dereferenced on the CPU and the GPU (i.e., each data element only has a single name). This model simplifies sharing data between the CPU and GPU by removing the need for explicit copies, as well as allowing the CPU and GPU to share access to rich pointer-based data structures.

Unfortunately, there is no free lunch. Translating from virtual to physical addresses comes with overheads. Translation look-aside buffers (TLBs) consume a significant amount of power due to their high associativity [67, 68, 138], and TLB misses can significantly decrease performance [13, 18, 70, 90]. Additionally, correctly designing the memory manage-

ment unit (MMU) is tricky due to rare events such as page faults and TLB shutdown [150].

Current GPUs have limited support for virtualized addresses [34, 81, 146]. However, this support is poorly documented publicly and has not been thoroughly evaluated in the literature. Additionally, industry has implemented limited forms of shared virtual address space. NVIDIA proposed Unified Virtual Addressing (UVA) and OpenCL has similar mechanisms. However, UVA requires special allocation and pinned memory pages [101] which can have significant performance impact [95].

The HSA foundation announced heterogeneous Uniform Memory Accesses (hUMA) which will implement a shared virtual address space in future heterogeneous processors [128], but details of this support are not available in public literature.

Engineering a GPU MMU appears challenging, as GPU architectures deviate significantly from traditional multicore CPUs. Current integrated GPUs have hundreds of individual execution lanes, and this number is growing. For instance the AMD A10 APU, with 400 lanes, can in principle require up to 400 unique translations in a single cycle! In addition, the GPU is highly multithreaded which leads to many memory requests in flight at the same time.

To drive GPU MMU design, we present an analysis of the memory access behavior of current GPGPU applications. Our workloads are taken from the Rodinia benchmark suite [27] and a database sort workload. We present three key findings and a potential MMU design motivated by each finding:

1. The coalescing hardware and scratchpad memory effectively filter the TLB request rate. Therefore, the L1 TLB should be placed after the coalescing hardware to leverage the traffic reduction.
2. Concurrent TLB misses are common on GPUs with an average of 60 to a maximum of over 1000 concurrent page walks! This fact motivates a highly-threaded page table walker to deliver the required throughput.

3. GPU TLBs have a very high miss rate with an average of 29%. Thus, reducing TLB miss penalty is crucial to reducing the pressure on the page table walker, and thus, we employ a page walk cache.

Through this data-driven approach we develop a proof-of-concept GPU MMU design that is fully compatible with CPU page tables (x86-64 in this work). Figure 4.6 shows an overview of the GPU MMU evaluated in this chapter. This design uses a TLB per GPU compute unit (CU) and a shared page walk unit to avoid excessive per-CU hardware. The shared page walk unit contains a highly-threaded page table walker and a page walk cache.

The simplicity of this MMU design shows that address translation can be implemented on the GPU without exotic hardware. We find that using this GPU MMU design incurs modest performance degradation (an average of less than 2% compared to an ideal MMU with an infinite sized TLB and minimal latency page walks) while simplifying the burden on the programmer.

In addition to our proof-of-concept design, we present a set of alternative designs that we also considered, but did not choose due to poor performance or increased complexity. These designs include adding a shared L2 TLB, including a TLB prefetcher, and alternative page walk cache designs. We also analyzed the impact of large pages on the GPU TLB. We find that large pages do in fact decrease the TLB miss rate. However, in order to provide compatibility with CPU page tables, and ease the burden of the programmer, we cannot rely solely on large pages for GPU MMU performance.

The contributions of this work are:

- An analysis of the GPU MMU usage characteristics for GPU applications,
- A proof-of-concept GPU MMU design which is compatible with x86-64 page tables, and
- An evaluation of our GPU MMU design that shows a GPU MMU can be implemented without significant performance degradation.

4.2 Background

General background on GPU architecture is covered in Chapter 2. In this section, we focus on the GPU programming model details that relate to shared virtual memory and background on CPU MMU designs.

GPU programming model

Current GPGPU programming models consider the GPU as a separate entity with its own memory, virtual address space, scheduler, etc. Programming for the GPU currently requires careful management of data between CPU and GPU memory spaces.

Figure 4.1 shows an example CUDA application. Figure 4.1a shows a simple kernel that copies from one vector (in) to another (out). Figure 4.1b shows the code required to use the `vectorCopy` kernel using the current separate address space paradigm. In addition to allocating the required memory on the host CPU and initializing the data, memory also is explicitly allocated on, and copied to, the GPU before running the kernel. After the kernel completes, the CPU copies the data back so the application can use the result of the GPU computation.

There are many drawbacks to this programming model. Although array-based data structures are straightforward to move from the CPU to the GPU memory space, pointer-based data structures, like linked-lists and trees, present complications. Also, separate virtual address spaces cause data to be replicated. Even on shared physical memory devices, like AMD Fusion, explicit memory allocation and data replication is still widespread due to separate virtual address spaces. Additionally, due to replication, only a subset of the total memory in a system is accessible to GPU programs. Finally, explicit separate allocation and data movement makes GPU applications difficult to program and understand as each logical variable has multiple names (`d_in`, `h_in` and `d_out`, `h_out` in the example).

Beginning with the Fermi architecture, NVIDIA introduced “unified virtual addressing”

```
__device__ void vectorCopy(int *in, int *out) {
    out[threadId.idx] = in[threadId.idx];
}
```

(a) Simple vector copy kernel

```
void main() {
    int *d_in, *d_out;
    int *h_in, *h_out;
    // allocate input array on host
    h_in = new int[1024];
    h_in = ... // Initial host array
    // allocate output array on host
    h_out = new int[1024];
    // allocate input array on device
    d_in = cudaMalloc(sizeof(int)*1024);
    // allocate output array on device
    d_out = cudaMalloc(sizeof(int)*1024);
    // copy input array from host to device
    cudaMemcpy(d_in, h_in, sizeof(int)*1024, HtD);
    vectorCopy<<<1,1024>>>(d_in, d_out);
    // copy the output array from device to host
    cudaMemcpy(h_out, d_out, sizeof(int)*1024, DtH);
    // continue host computation with result
    ... h_out
    //Free memory
    cudaFree(d_in); cudaFree(d_out);
    delete[] h_in; delete[] h_out;
}
```

(b) Separate memory space implementation

```
int main() {
    int *h_in, h_out;
    // allocate input/output array on host
    h_in = cudaHostMalloc(sizeof(int)*1024);
    h_in = ... // Initial host array
    h_out = cudaHostMalloc(sizeof(int)*1024);
    vectorCopy <<<1,1024>>> (h_in, h_out);
    // continue host computation with result
    ... h_out
    //Free memory
    cudaHostFree(h_in); cudaFree(h_out);
}
```

(c) “Unified virtual address” implementation

```

int main() {
    int *h_in, h_out;
    // allocate input/output array on host
    h_in = new int[1024];
    h_in = ... // Initial host array
    h_out = new int[1024];
    vectorCopy <<<1,1024>>> (h_in, h_out);
    // continue host computation with result
    ... h_out
    delete[] h_in; delete[] h_out;
}

```

(d) Shared virtual address space implementation

Figure 4.1: Example GPGPU application

(UVA) [101] (OpenCL has a similar feature as well). Figure 4.1c shows the implementation of vectorCopy with UVA. The vector-copy kernel is unchanged from the separate address space kernel. In the UVA example, instead of allocating two copies of the input and output vectors, only a single allocation is necessary. However, this allocation requires a special API which creates difficulties in using pointer-based data structures. Separate allocation makes composability of GPU kernels in library code difficult as well, because the allocation is a CUDA runtime library call, not a normal C or C++ allocation (e.g. new/malloc/mmap). Memory allocated via cudaMallocHost can be implemented in two different ways. Either the memory is pinned in the main memory of the host, which can lead to poor performance [100], or the data is implicitly copied to a separate virtual address space which has the previously discussed drawbacks.

Figure 4.1d shows the implementation with a shared virtual address space (the programming model used in this chapter). In this implementation, the application programmer is free to use standard memory allocation functions. Also, there is no extra memory allocated, reducing the memory pressure. Finally, by leveraging the CPU operating system for memory allocation and management, the programming model allows the GPU to take page faults and access memory mapped files. HSA hUMA takes a similar approach [128].

CPU MMU design

The memory management unit (MMU) on CPUs translates virtual addresses to physical address as well as checks page protection. In this chapter we focus on the x86-64 ISA; however, our results generalize to any multi-level hardware-walked page table structure. The CPU MMU for the x86-64 ISA consists of three major components: 1) logic to check protection and segmentation on each access, 2) a translation look-aside buffer to cache virtual to physical translations and protection information to decrease translation latency, and 3) logic to walk the page table in the case of a TLB miss. In this work, we use the term MMU to refer to the unit that contains the TLB and other supporting structures. Many modern CPU MMU designs contain other structures to increase TLB hit rates and decrease TLB miss latency.

The x86-64 page table is a 4-level tree structure. By default, the TLB holds page table entries, which reside in the leaves of the tree. Therefore, on a TLB miss, up to four memory accesses are required. The page table walker (PTW) traverses the tree from the root which is found in the CR3 register. The PTW issues memory requests to the page walk cache that caches data from the page table. Requests that hit in the page walk cache decrease the TLB miss penalty. Memory requests that miss in the page walk cache are issued to the memory system similar to CPU memory requests and can be cached in the data caches.

The x86-64 ISA has extensions for 2 MB and 1 GB pages in addition to the default 4 KB page size. However, few applications currently take advantage of this huge page support. Additionally, it is important for an MMU to support 4 KB pages for general compatibility with all applications.

4.3 Simulation methodology and workloads

We used a cycle-level heterogeneous simulator, gem5-gpu [115] (Section 3.1), to simulate the heterogeneous system. Results are presented for 4 KB pages. Large page support is

CPU	1 core, 2 GHz, 64 KB L1, 2 MB L2
GPU	16 CUs, 1.4 GHz, 32 lanes
L1 cache (per-CU)	64 KB, 4-way set associative, 15 ns latency
Scratchpad memory	16 KB, 15 ns latency
GPU L2 cache	1 MB, 16-way set associative, 130 ns latency
DRAM	2 GB, DDR3 timing, 8 channel, 667 MHz

Table 4.1: Details of simulation parameters

discussed in Section 4.6. Table 4.1 shows the configuration parameters used in obtaining our results. Section 3.1 contains the details of gem5-gpu.

We use a subset of the Rodinia benchmark suite [27] for our workloads. We do not use some Rodinia benchmarks as the input sizes are too large to simulate. The Rodinia benchmarks are GPU-only workloads, and we use these workloads as a proxy for the GPU portion of future heterogeneous workloads (details on changes made to the Rodinia workloads in Section 3.1). We add one workload, sort, to this set. Sort is a database sorting kernel that sorts a set of records with 10 byte keys and 90 byte payloads. All workloads are modified to remove the memory copies, and all allocations are with general allocators (new/malloc/mmap). Although we are running in a simulation environment and using reduced input sized, many of our working sets are much larger than the TLB reach; thus, we expect our general findings to hold as working set size increases.

As a baseline, we use an ideal, impossible to implement MMU. We model an ideal MMU with infinite sized per-CU TLBs and minimal latency (1 cycle cache hits) for page walks. This is the minimum translation overhead in our simulation infrastructure.

4.4 Designing a GPU MMU through analysis

We meet the challenges of designing a GPU MMU by using data to evolve through three architectures to our proof-of-concept recommendation (Design 3). Design 3 enables full compatibility with x86-64 page tables with less than 2% performance overhead, on average. Table 4.2 details the designs in this section as well as the additional designs from Section 4.6.

	Per-CU L1 TLB entries	Highly-threaded page table walker	Page walk cache size	Shared L2 TLB entries
Ideal MMU	Infinite	Infinite	Infinite	None
Section 4.4				
Design 0	N/A: Per-lane MMUs		None	None
Design 1	128	Per-CU walkers	None	None
Design 2	128	Yes (32-way)	None	None
Design 3	64	Yes (32-way)	8 KB	None
Section 4.6				
Shared L2	64	Yes (32-way)	None	1024
Shared L2 & PWC	32	Yes (32-way)	8 KB	512
Ideal PWC	64	Yes (32-way)	Infinite	None
Latency	1 cycle	20 cycles	8 cycles	20 cycles

Table 4.2: Configurations under study. Structures are sized so each configuration uses 16 KB of storage.

We start with a CPU-like MMU (Design 0) and then modify it as the GPU data demands. Design 0 follows CPU core design with a private MMU at each lane (or “core” in NVIDIA terminology). Design 0 has the same problems as a CPU MMU—high power, on the critical path, etc.—but they are multiplied by the 100s of GPU lanes. For these reasons, we do not quantitatively evaluate Design 0.

Motivating Design 1: Post-coalescer MMU

Here we show that moving the GPU MMU from before to after the coalescer (Design 0 → Design 1) reduces address translation traffic by 85%.

GPU memory referencing behavior differs from that of CPUs. For various benchmarks, Figure 4.2 presents operations per thousand cycles for scratchpad memory lane instructions (left bar, top, blue), pre-coalescer global memory lane instructions (left bar, bottom, green), and post-coalescer global memory accesses (right bar, brown). The “average” bars represent the statistic if each workload was run sequentially, one after the other.

Figure 4.2 shows that, for every thousand cycles, the benchmarks average:

- 602 total memory lane instructions,

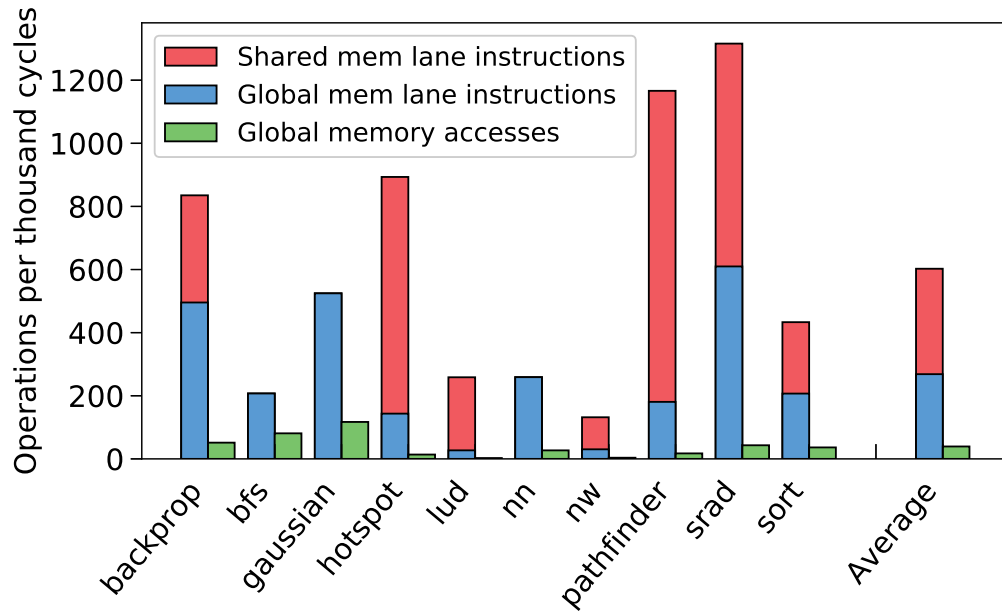


Figure 4.2: All memory operations, global memory operations, and global memory accesses per thousand instructions.

- 268 of which are global memory lane instructions (with the other 334 to scratchpad memory), and
- Coalescing reduces global memory lane instructions to only 39 global memory accesses.

In total, the rate of memory operations is reduced from 602 to 39 per thousand cycles for an 85% reduction.

Although the coalescing hardware is effective, the benchmarks do show significant memory divergence. Perfect coalescing on 32 lanes per CU would reduce 268 global memory lane instructions (per thousand cycles) by $32\times$ to 9, which is much less than the 39 observed.

To benefit from this bandwidth filtering, *Design 1* includes a private per-CU L1 TLB after scratchpad memory access and after the coalescing hardware. Thus, the MMU is only accessed on global memory accesses. Figure 4.6 shows Design 1 in light gray and Table 4.2 details the configuration parameters.

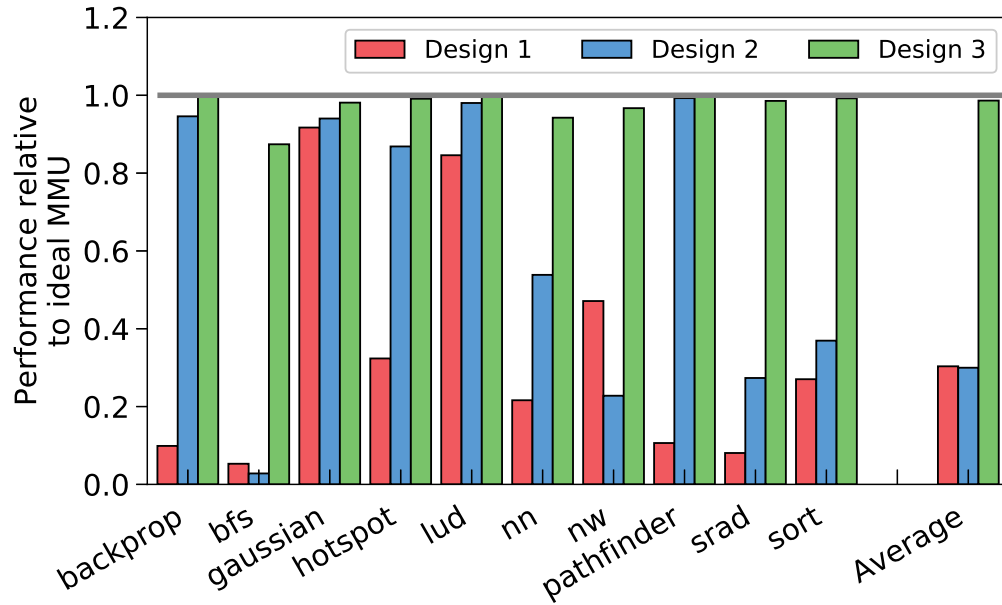


Figure 4.3: Performance of each design relative to an ideal MMU. See Table 4.2 for details of configurations.

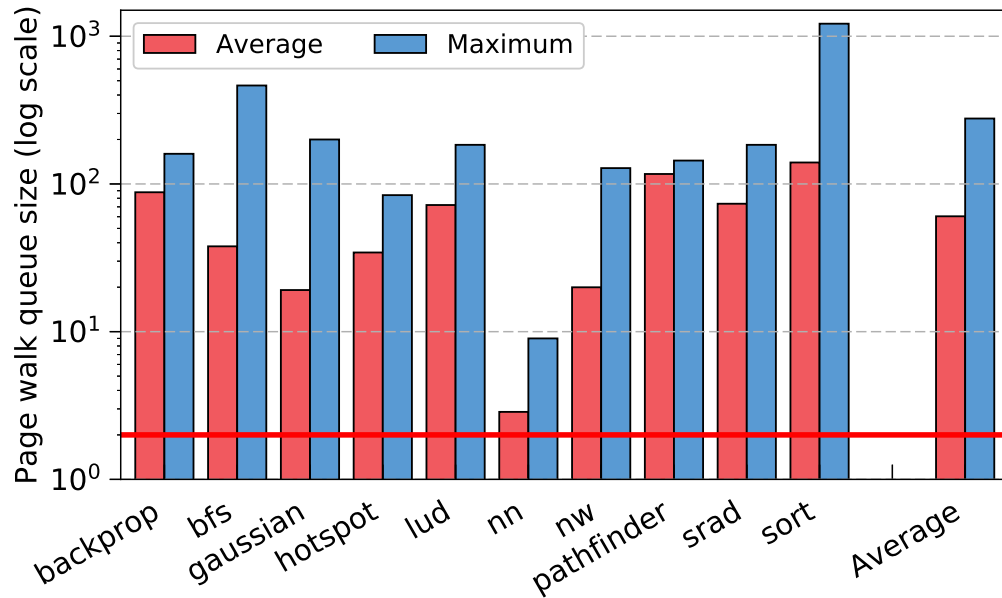


Figure 4.4: Average and max size of the page walk queue for each per-CU MMU in Design 1. Log scale, bold line at 2.

Motivating Design 2: Highly-threaded page table walker

Here we show that Design 1 fails to perform well (average performance is 30% of an ideal MMU), isolate the problem to bursts of TLBs misses (60 concurrent), and advocate for a highly-threaded PTW (Design 2).

Now that we have mitigated the bandwidth issues, we might expect Design 1 to perform well; it does not. For each benchmark and the average, Figure 4.3 shows the performance of Design 1 (leftmost, red) compared to an ideal MMU with an impossibly low latency and infinite sized TLBs. Performance is good when it is close to the ideal MMU's 1.0. (Designs 2 (blue) and 3 (green) will be discussed later.)

Figure 4.3 results show that Design 1 (blue) performs:

- Poorly on average (30% of ideal's),
- Sometimes very poorly (about 10% of ideal for backprop, bfs, and pathfinder), and
- Occasionally adequately (gaussian and lud).

These performance variations occur for various reasons. For example, bfs is memory bound—having few instructions per memory operation—making it particularly sensitive to global memory latency. On the other hand, gaussian and lud perform well, in part because the working set sizes are relatively small.

Investigating Design 1's poor performance lead to an obvious culprit: bursts of TLB misses. For each benchmark and the average, Figure 4.4 shows the average (left, red) and maximum across CUs (right, blue) occupancy of the page walk queue when each page walk begins. Note that the y-axis is logarithmic.

Figure 4.4 shows that when each page walk is issued:

- An average of 60 page table walks are active at that CU, and
- The worst workload averages 140 concurrent page table walks.

Moreover, additional data show that, for these workloads, over 90% of page walks are issued within 500 cycles of the previous page walk, which is significantly less than the average page walk latency in this design. Also, almost all workloads use many more than 100 concurrent page walks at the maximum. Therefore, these workloads will experience high queuing delays with a conventional blocking page table walker. This also shows that

the GPU MMU requires changes from the CPU-like single-threaded pagetable walker of Design 1.

This high page walk traffic is primarily because GPU applications can be very bandwidth intensive. GPU hardware is built to run instructions in lock step, and because of this characteristic, many GPU threads simultaneously execute a memory instruction. This, coupled with the fact each CU supports many simultaneous warp instructions, means GPU TLB miss traffic will be high.

Therefore, *Design 2* includes a shared multi-threaded page table walker with 32 threads. Figure 4.6 shows how Design 2 builds on Design 1 in dark gray and Table 4.2 details the configuration parameters. The page walk unit is shared between all CUs on the GPU to eliminate duplicate hardware at each CU and reduce the hardware overhead. On a TLB miss in the per-CU L1 TLBs, the shared page walk unit is accessed and executes a page walk.

Recent work by Yoon et al. has further studied the page table walk bandwidth bottleneck in heterogeneous systems [156]. In this thesis, we do not constrain the bandwidth at the shared page walk unit. This has a small performance impact on the Rodinia workloads analyzed in this thesis. However, Yoon et al. show this is a poor assumption when running graph-based and irregular workloads [156].

Motivating Design 3: Add a page walk cache

Here we show that Design 2 performs much better than Design 1 for most workloads but still falls short of an ideal MMU (30% of ideal on average). For this reason, we introduce Design 3 that adds a shared page walk cache to perform within 2% of ideal.

The second bars in Figure 4.3 (blue) show the performance of each benchmark for Design 2. Results from this figure show that Design 2:

- Often performs much better than Design 1, but

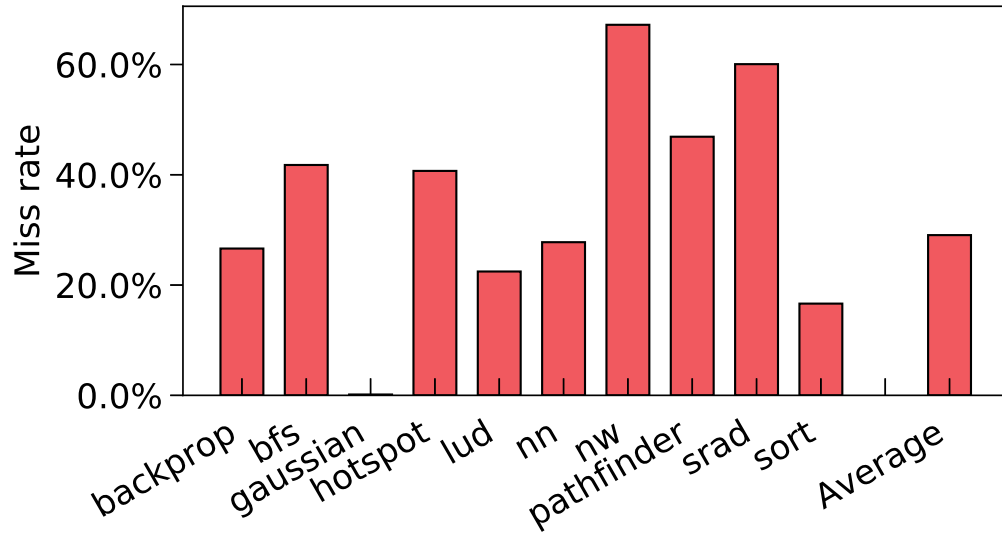


Figure 4.5: Miss rate for a 128 entry per-CU L1 TLB averaged across all CUs.

- That these benefits are inconsistent and short of ideal

We find the workloads that perform best have been tuned to tolerate long latency memory operations and the addition of the TLB miss latency is hidden by thread parallelism.

On the other hand, some workloads, e.g., bfs and nw, actually perform worse with Design 2 than Design 1. We isolated the problem to the many requests from one CU queuing in front of another CU's requests rather than being handled more round robin as in the single-threaded page-table walker of Design 1. While this specific effect might be fixed by changing PTW queuing from first-come-first-serve, we seek a more broadly effective solution.

To better understand why Design 2 falls short of ideal, we examined the TLB miss rates. Figure 4.5 shows the miss rates (per-CU, 128 entry) for each benchmark and the average.

Figure 4.5 results show that per-CU TLB miss rates:

- Average 29% across benchmarks and
- Can be as high as 67% (nw).

Needless to say, these rates are much higher than one expects for CPU TLBs. Gaussian is a low outlier, because of high computational density (compute operations per byte of

data) and small working set (so all TLB misses are compulsory misses).

This high miss rate is not surprising. With many simultaneous warps and many threads per warp, a GPU CU can issue memory requests to a very large number of pages. In addition, many GPU applications exhibit a memory access pattern with poor temporal locality reducing the effectiveness of caching translations. If an access pattern has no temporal locality (e.g. streaming), even with perfect coalescing, each CU could potentially access 128 bytes per cycle. This translates to only 32 cycles to access an entire 4 KB page, in the worst case.

As discussed previously, the global memory access rate on GPUs is quite low (39 accesses per thousand cycles on average) and consequently the TLB request rate is small. Therefore, it's possible that even though the GPU TLB exhibits high miss rates the miss traffic (misses per cycle) could be relatively low. However, this is not the case. There is an average of 1.4 TLB misses per thousand cycles and a maximum of 13 misses per thousand cycles.

We investigated several alternatives to improve on Design 2, discussed further in Section 4.6, and settled on one. Our preferred *Design 3* includes a page walk cache with the shared page walk unit to decrease the TLB miss latency. Figure 4.6 shows how Design 3 builds on Design 2 in black and Table 4.2 details the configuration parameters.

Returning to Figure 4.3, the third bars (brown) show performance of Design 3 relative to an ideal MMU (1.0):

- Design 3 increases performance for all benchmarks over Design 2 and
- Design 3 is within 2% of the ideal MMU on average.

This increase in overall performance occurs because the page walk cache significantly reduces the average page walk time reducing the number of cycles the CU is stalled. Adding a page walk cache reduces the average latency for page table walks by over 95% and correspondingly increases the performance. The performance improvement of Design

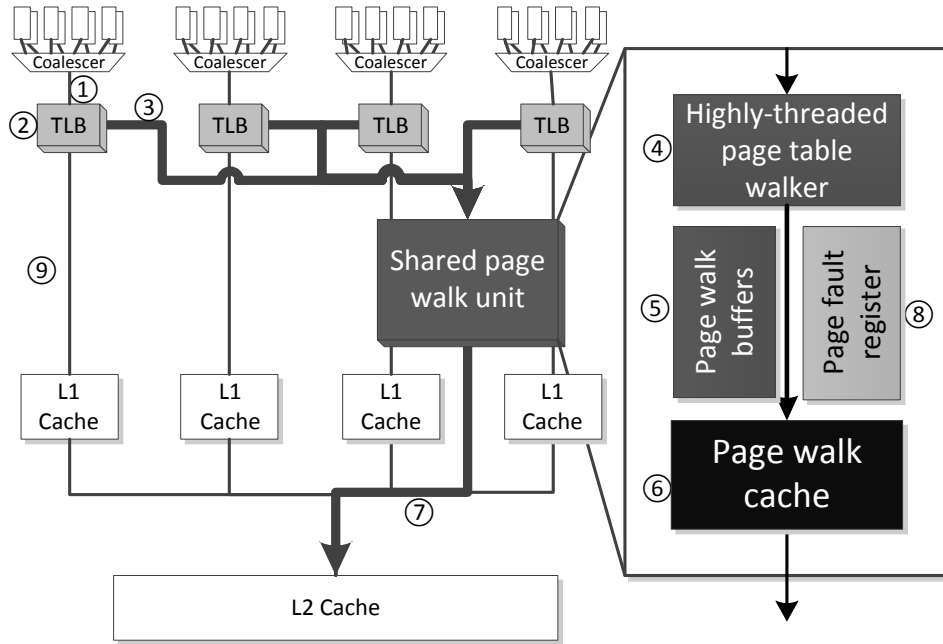


Figure 4.6: Overview of the GPU MMU designs. *Design 1* is shown in light gray. *Design 2* is shown in light and dark gray. *Design 3* is shown in light and dark gray and black.

3 is in part due to reducing the occupancy of the page walk buffers since the page walk latency decreased. This fact is most pronounced for bfs and nw, which suffered from queuing delays.

Summary of proof-of-concept design (Design 3)

This section summarizes our proof-of-concept Design 3 and presents additional details to explain its operation. Figure 4.6 details Design 3. The numbers correspond to the order in which each structure is accessed. Design 3 is made up of three main components, the per-CU post-coalescer L1 TLBs, the highly-threaded page table walker, and a shared page walk cache, discussed below.

Per-CU post-coalescer L1 TLBs—Each CU has a private TLB that is accessed after coalescing and scratchpad memory to leverage the traffic reduction. On TLB hits, the memory request is translated then forwarded to the L1 cache. On TLB misses, the warp instruction is stalled until the shared page walk unit completes the page table lookup and

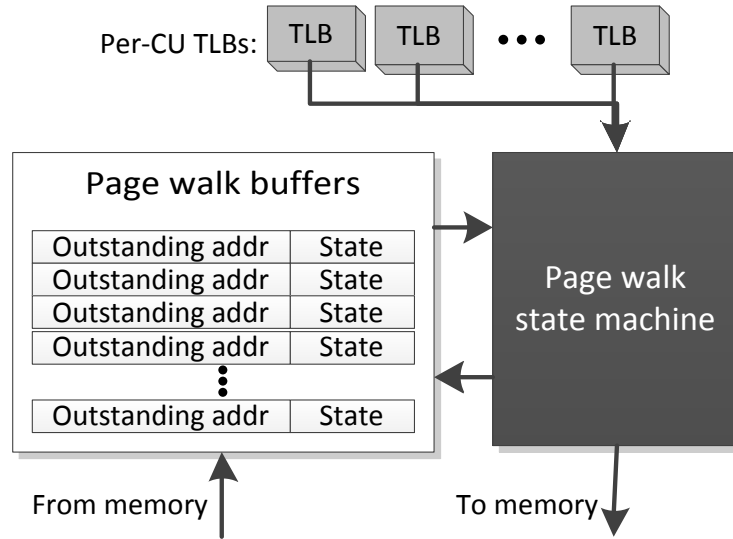


Figure 4.7: Details of the highly-threaded page table walker.

returns the virtual to physical address translation. Stalling at this point in the pipeline is common for memory divergent workloads, and the associated hardware is already present in the CUs.

Highly-threaded page table walker—The PTW design consists of a hardware state machine to walk the x86-64 page table and a set of page walk buffers that hold the current state of each outstanding page walk, shown in Figure 4.7. On a TLB miss, the page walk state machine allocates a page walk buffer entry and initializes the outstanding address to the value in the CR3 register, which holds the address of the root of the page table. Next, the page walk state machine issues the memory request corresponding to the buffer entry. When memory responds, the page walk buffers are queried for the match, and the state for that request is sent to page walk state machine, which then issues the next memory request or returns final translation.

The PTW also has supporting registers and logic to handle faults that occur when walking the page table. Concurrent page faults are serialized and handled one at a time by the operating system. Page faults are discussed in detail in Section 4.5.

This design can also extend to other ISAs that have hardware page table walkers. For

example, the ARM MMU also defines a hardware page table walker and this can be used in place of the x86-64 page walk state machine included in our current design.

Page walk cache—Many modern CPUs contain a small cache within the memory management unit to accelerate page table walks. Since the latency of the L2 cache of a GPU is very long (nearly 300 cycles) a cache close to the MMU decreases the page walk latency significantly. We use a page walk cache design similar to AMD [15]. It caches non-leaf levels of the page table, decreasing the number of L2 cache and DRAM accesses required for a page table walk. Other page walk cache designs are discussed in Section 4.6. However, we do not see a significant performance improvement over Design 3 with these alternatives.

4.5 Correctness issues

In this section, we discuss the issues and implementation of page faults and TLB shutdown for the GPU architecture. We expect these events to be rare. For instance, often workloads are sized to fit in main memory virtually eliminating major page faults. Nevertheless, we correctly implement page faults and TLB shutdown in Linux 2.6.22 on top of gem5 full-system simulation, to our knowledge a first in public literature.

Page fault handling

Although rare, the GPU memory management unit architecture must be able to handle page faults to have correct execution.

There are two different categories of page faults, major and minor. Below we give details on how each is handled.

Minor page faults—A minor page fault occurs when the operating system has already allocated virtual memory for an address, but it has not yet allocated a physical frame and written the page table. Minor page faults do not require accessing the disk or other backing storage device. Minor page faults often occur when sharing virtual memory between

processes, copy-on-write memory, and on the initial accesses after memory allocation. The last is common in our workloads as there are many large calls to malloc in which the memory is not touched by the CPU process. As an example, in Figure 3c, `h_out` is allocated by the CPU process, but is not accessed until the `copyVector` kernel and causes a minor page fault. Minor page faults are low latency, about 5000 cycles on average in our workloads. The operating system only needs to allocate a physical frame and modify the page table with the new physical page number. Since the page fault is low latency, we stall the faulting warp instruction in the same way as a TLB miss.

Major page faults—For major page faults, the operating system must perform a long-latency action, such as a disk access. Stalling an application for milliseconds—while likely correct—wastes valuable GPU execution resources. To handle this case, the GPU application, or a subset thereof, could be preempted similar to a context switch on a CPU. However, this technique has drawbacks since the GPU’s context is very large (e.g., 2 MB of register file on NVIDIA Fermi [101]). Another possibility is to leverage checkpointing, and restart the offending applications after the page fault has been handled. There are proposals like iGPU [92] to reduce the overhead of these events. However, none of our workloads have any major page faults so we do not focus on this case.

GPU page fault handler implementation

Although there are many ways to implement handling page faults, in this GPU MMU architecture we chose to slightly modify the CPU hardware and make no modifications to the operating system. Our page fault handling logic leverages the operating system running on the CPU core similar to CPU MMU page fault logic. We use this design for two reasons. First, this design does not require any changes to the GPU execution hardware as it does not need to run a full-fledged operating system. Second, this design does not require switching contexts on the GPU as it can handle minor page faults by stalling the faulting instruction.

The only CPU change necessary is modification of the microcode that implements the IRET (return from interrupt) instruction. When a page fault is detected by the page table walker logic, the address which generated the fault is written into the page fault register in the GPU page walk unit. Then, the page fault proceeds similar to a page fault generated by the CPU MMU. The faulting address is written into the CPU core's CR2 register, which hold the faulting address for CPU page faults, and a page fault interrupt is raised. Then, the page fault handler in the operating system runs on the CPU core. The operating system is responsible for writing the correct translation into the page table, or generating a signal (e.g., SEGVFAULT) if the memory request is faulting. Once the page fault handler is complete, the operating system executes an IRET instruction on the CPU core to return control to the user-level code. To signal the GPU that the page fault is complete, on GPU page faults, we add a check of the GPU page fault register in the IRET microcode implementation. If the address in that register matches the CR2 address then the page fault may be complete (it is possible the operating system could have finished some other interrupt instead). To check if the page fault is complete, the page walk unit on the GPU performs a second page table walk for the faulting address. If the translation is found, then the page fault handler was successful and the page fault register on both the page walk unit and the CPU are cleared. If the second page walk was not successful then the GPU MMU continues to wait for the page fault handler to complete.

Page fault discussion

We implemented the above technique to handle page faults in the gem5 simulator running Linux (version 2.6.22.9) in full-system mode. Using this implementation, the Linux kernel correctly handles minor page faults for all of our GPU applications.

For this implementation, the page fault handling logic assumes that the GPU process is still running on the CPU core. This will be true if, for instance, the application can run on the CPU and GPU at the same time, or the CPU runtime puts the CPU core into a low

power state and does not change contexts.

However, the CPU runtime may yield the CPU core while the GPU is running. In this situation, page faults are still handled correctly, but they are longer latency as there is a context switch to the process which spawned the GPU work before the operating system begins handling the page fault. This is similar to what happens when other process-specific hardware interrupts are encountered. Another option is to include a separate general purpose core to handle operating system kernel execution for the GPU. This core can handle any page faults generated by the running GPU kernel in the same way as described above.

TLB flushes and shutdown

The GPU MMU design handles TLB flushes similarly to the CPU MMU. When the CR3 register is written on the CPU core that launched the GPU application, the GPU MMU is notified via inter-processor communication and all of the GPU TLBs are flushed. This is a rare event, so performance is not a first order concern. Since there is a one-to-one relation between the CPU core executing the CPU process and the GPU kernel, on TLB a shutdown to the CPU core, the GPU TLBs are also flushed. The GPU cannot initiate a shutdown, only participate. When a CPU initiates a shutdown, it sends a message to the GPU MMU which responds after it has been handled by flushing the TLBs. If the GPU runtime allows the CPU processes to be de-scheduled during GPU kernel execution, TLB flushes and shutdown become more complicated. However, this can be handled in a similar way as page faults. If TLB shutdown to GPU CUs becomes common, there are many proposals to reduce the overheads for TLB shutdown [129, 145].

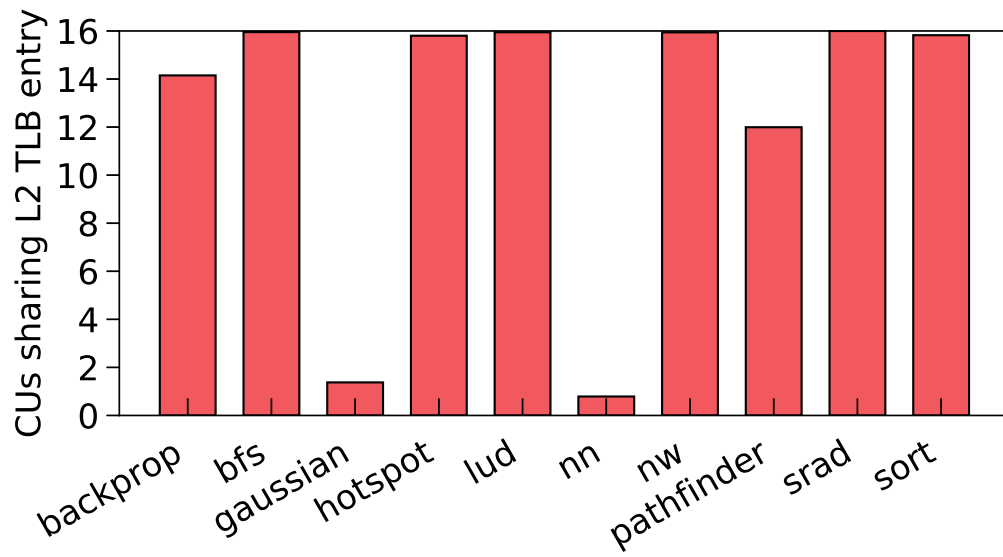


Figure 4.8: Sharing pattern for the 512 entry L2 TLB. 16 sharers implies all CUs sharing each entry.

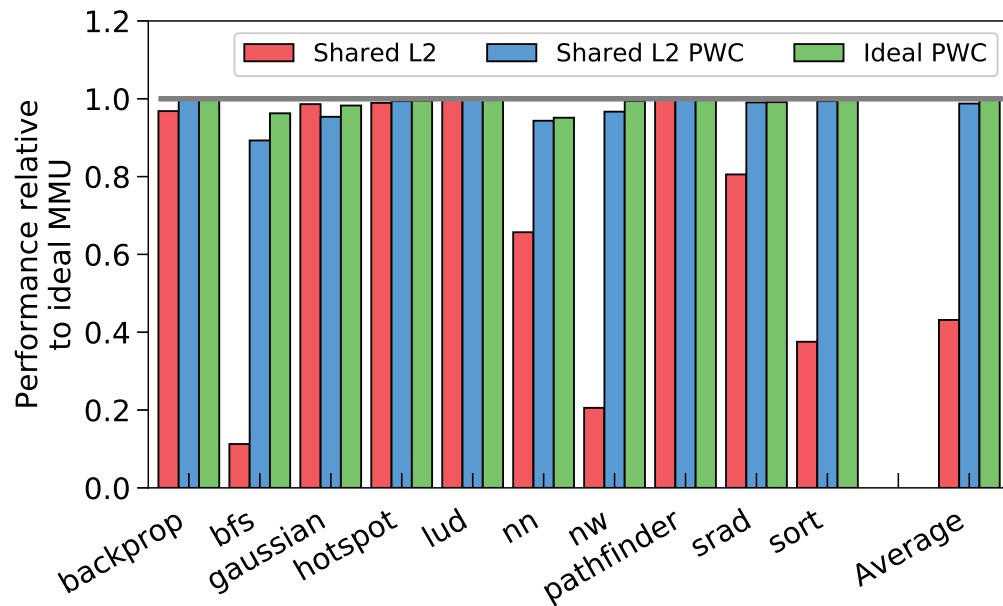


Figure 4.9: Performance of a shared L1 TLB and an ideal PWC relative to the ideal MMU. See Table 4.2 for details of configurations.

4.6 Alternative designs

Here, we discuss some alternative designs we considered as we developed our proof-of-concept design. These design either do not perform as well or are more complex than Design 3. We first evaluate the addition of a shared L2 TLB and then alternative page walk cache designs. Next, we discuss adding a TLB prefetcher and the impact of large pages. Finally, we consider the impact of alternative MMU designs on area and energy.

Shared L2 TLB

A shared L2 TLB can capture multiple kinds of data sharing between execution units [18]. This cache can exploit sharing of translations between separate CUs, effectively prefetching the translation for all but the first CU to access the page. The shared L2 TLB can also exploit striding between CUs where the stride is within the page size (e.g., CU 1 accesses address 0x100, CU 2 0x200, etc.).

The shared L2 TLB is most effective when each entry is referenced by many CUs. Figure 4.8 shows the number of CUs that access each L2 TLB entry before eviction. 16 sharers show all CUs share each entry, and one sharer shows no overlap in the working sets. Figure 4.8 shows most applications share each L2 TLB entries with many CUs and some share entries with all CUs. In these cases, the shared L2 TLB can improve performance by sharing the capacity of the L2 TLB between CUs.

Due to this potential, we investigated two MMU designs with shared L2 TLBs. The first design (Shared L2) has private L1 TLBs with a shared L2 TLB and no page walk cache. In this design the area that is devoted to the page walk cache in Design 3 is instead used for the L2 TLB. The second design we evaluate (Shared L2 & PWC) contains private L1 TLBs, a shared L2 TLB, and a page walk cache. In this design each L1 TLB size is reduced to accommodate an L2 TLB. All of the designs evaluated use a total of 16 KB of storage for their implementation.

Figure 4.9 shows the performance of these two designs relative to an ideal MMU in the first two bars. Parameters for each configuration are in Table 4.2.

Shared L2—per-CU private L1 TLBs and a shared L2 TLB: With a shared L2 TLB, many applications perform as well as the ideal, like Design 3 (Figure 4.9 leftmost bars in blue). However, bfs, nw, and sort perform at least 2x worse. For these applications, decreasing the page walk latency is very important as the L1 TLBs experience a high miss rate. nn sees a slowdown when using the Shared L2 design because there is no sharing of TLB entries between CUs. Thus, area dedicated to a page walk cache is more useful for this workload. On average, there is more than a 2x slowdown when using a shared L2 TLB instead of a page walk cache.

Shared L2 & PWC—per-CU private L1 TLBs, a shared L2 TLB, and a page walk cache: The second bars (green) in Figure 4.9 show the performance with both a shared L2 TLB and a page walk cache compared to an ideal MMU. In this configuration, even though the L1 TLB size is reduced, performance does not significantly decrease; the average performance is within 0.1%. Using both a shared L2 TLB and a page walk cache achieves the benefits of both: it takes advantage of sharing between CUs and reduces the average page walk latency. We chose to not include an L2 TLB in our proof-of-concept design as it adds complexity without affecting performance.

Alternative page walk cache designs

In this work, we use a page walk cache similar to the structure implemented by AMD [15]. In this design, physical addresses are used to index the cache. Other designs for a page walk cache that index the cache based on virtual addresses, including Intel-style translation caches, have been shown to increase performance for CPUs [11]. We chose to use an AMD-style page walk cache primarily for ease of implementation in our simulator infrastructure.

To evaluate the possible effects of other page walk cache designs, we evaluated our workloads with an ideal (infinitely sized) page walk cache with a reduced latency to model

a single access, the best case for the translation cache. The rightmost (brown) bars in Figure 4.9 show the performance with a 64 entry L1 TLB and the ideal page walk cache compared to an ideal MMU. The ideal page walk cache increases performance by an average of 1% over our proof-of-concept Design 3. For bfs the ideal page walk cache increases performance by a more significant 10% over Design 3 as this workload is sensitive to the page walk latency. From this data, a different page walk cache design may be able to increase performance, but not significantly.

TLB prefetching

We evaluated Design 3 with the addition of a one-ahead TLB prefetcher [70]. The TLB prefetcher issues a page walk for the next page on each L1 TLB miss and on each prefetch buffer hit. The TLB prefetcher does not affect the performance of our workloads. Prefetching, on average, has a less than 1 impact on performance. One hypothesis as to the ineffectiveness of TLB prefetching is the bursty-ness of demand misses. Other, more complicated, prefetching schemes may show more performance improvement, but are out of the scope of this thesis.

Large pages

Large pages reduce the miss rate for TLBs on CPUs. On GPUs, due to the high spatial locality of accesses, large pages should also work well. When evaluated, for all of our workloads except gaussian, 2 MB pages reduce the TLB miss rate by more than 99%, resulting in more than 100 times fewer TLB misses. As previously mentioned, since the working set for Gaussian fits in the TLB, large pages do not provide as much benefit, although the miss rate is reduced by over 80%.

Large pages work well for the workloads under study in this chapter, but may not perform as well for future, larger memory footprint, workloads. Additionally, to maintain compatibility with today’s CPU applications, the MMU requires 4 KB pages. Alternatively,

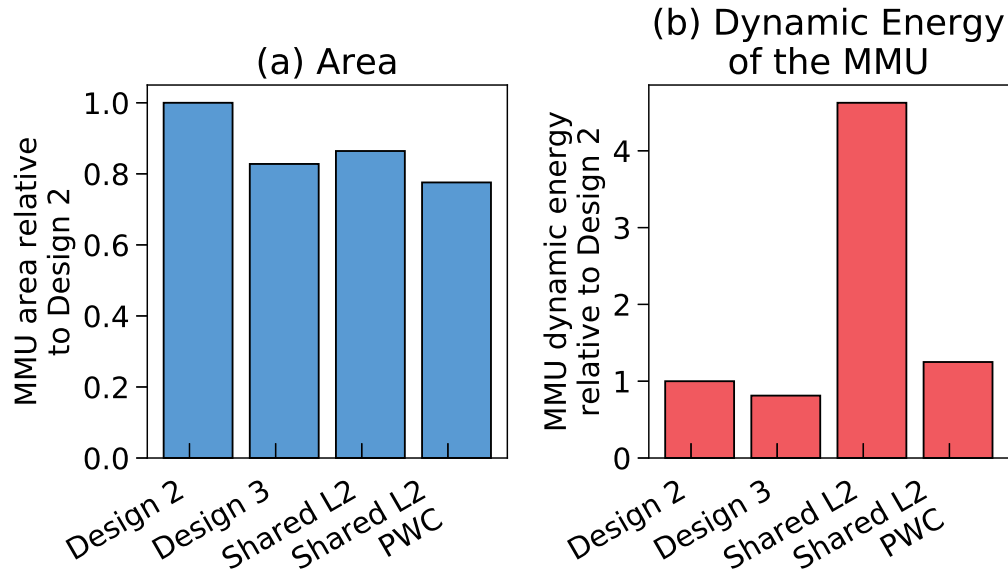


Figure 4.10: Energy and area of MMU configurations relative to Design 2

requiring applications to use large pages would place a burden on the application developer to use a special allocator API for memory that is accessed by the GPU.

Energy and area

Figure 4.10 shows the relative area and energy of the MMU for a subset of designs. We used a combination of Cacti [96] and McPAT [83] to determine relative area and dynamic access energy of each structure.

Figure 4.10a shows that all configurations are less area than the L1 TLB-only Design 2. This is because Design 2 has many large and highly associative TLBs, one per CU. The other configurations that share structures can amortize the overheads and provide higher performance. The shared L2 design is much more energy hungry than the other configurations. This is because the L1 TLBs do not filter a large percentage of requests (the average miss rate is 27%) and accesses to a large associative structure are high energy.

Design 3 with only a page walk cache shows a modest energy reduction (20%) and increases performance significantly over the Design 2. This energy reduction comes from having smaller highly associative structures (L1 TLBs) and a larger lower associativity

structure (the page walk cache). The design with both a page walk cache and shared L2 TLB (Shared L2 & PWC) has the smallest area, but has a modest energy increase (25%) over Design 2. Here, there is a tradeoff between energy and area reduction.

4.7 Related work

In this work we build on research accelerating CPU MMUs. Page walk caches and translation caches improve MMU performance by accelerating the page table walk [15, 58]. Barr et al. explored other MMU cache structures as well and found that a unified translation cache can outperform both the page walk cache and the translation cache [11].

Sharing resources between CPU cores has been studied as a way to decrease CPU TLB miss rate. Bhattacharjee and Martonosi examine a shared last level TLB [17] and a shared page walk cache [16]. These accelerate multi-threaded applications on the CPU by sharing translations between cores. These works target multithreaded applications on the CPU and apply similarly to the GPU since, in the common case, all CUs of the GPU are running the same application.

There are also several patents for industrial solutions for GPU virtual address translation [34, 146]. However, these patents have no evaluation and little implementation details.

There are multiple attempts to reduce the complexity of the GPGPU programming model through software [41, 130]. While these frameworks simplify the code for straightforward applications, like the UVA implementation of the vectorcopy example presented, it is still difficult to represent complex data structures.

GPUfs presents a POSIX-like API for the GPU that allows the GPU to access files mapped by the operating system. Similar to GPUfs, this chapter simplifies programming the GPU by providing developers with a well-known interface (shared virtual address space). Additionally, as a consequence of correctly handling page faults on the GPU, this

chapter also provides a method for the GPU to access files. GPUfs accomplishes these goals on current hardware through a complicated software library whereas this chapter implements solutions for future hardware similar to how conventional CPUs solve these problems.

Concurrent with our work, Pichai et al. also investigate address translation for GPG-PU's [111]. Both their and our work show that modest hardware changes can enable low-overhead GPU address translation, but with different designs and emphasis. For example, Pichai et al. use a 4-ported TLB and PTW scheduling, while we use a single-ported TLB and highly-threaded PTW. Pichai et al. additionally explore address translation effects on GPU warp scheduling, while we explore MMU correctness issues, like page faults, in a CPU-GPU system using gem5-gpu full-system simulation.

Since the work on this chapter was complete, there have been some related work. Vesely et al. evaluated the performance of GPU address translation on real hardware available from AMD [149]. Similar to the conclusions in this chapter, they found that GPU TLB misses are very high latency ($25\times$ CPU TLB miss latency) and workloads with poor memory coalescing particularly suffer from address translation overhead.

Yoon et al. further investigated address translation overheads by specifically targeting the shared page walk unit bandwidth [156]. In this chapter, we did not consider the bandwidth at the shared page walk unit (e.g., we assumed an infinite bandwidth). Yoon et al. show that limiting this bandwidth significantly degrades performance on emerging irregular GPU workloads. Building off of the ideas in this chapter of filtering bandwidth, Yoon et al. propose using a practical virtual cache hierarchy for the GPU which further filters bandwidth from the shared page walk unit.

4.8 Conclusions

As heterogeneous systems become more prevalent, it is becoming more important to consider how to program these systems. Applications developers assume a shared virtual address space between all of the CPU cores, and in this chapter, we extend this assumption to one other device in the system, the GPGPU. As GPGPUs can issue 100s of per-lane instructions per cycle, supporting address translation appears formidable. Our analysis, however, shows that a non-exotic GPU MMU design performs well with commonly-used 4 KB pages: per-CU post-coalescer TLBs, a shared 32-way highly-threaded page table walker, and a shared page walk cache.

We focused on the x86-64 ISA in this work. However, our findings generalize to any ISA with a hardware walked and tree-based page table structure. The proof-of-concept GPU MMU design analyzed in this chapter shows that decreasing the complexity of programming the GPU without incurring significant overheads is possible, opening the door to novel heterogeneous workloads.

We build on this work in Chapter 5 by leveraging these tightly-integrated GPGPUs. The logical integration provided by the hardware described in this chapter enables performance improvements for analytic database workloads, as we will show in Chapter 5.

4.9 Artifacts

In the process of writing this chapter, the following artifacts were generated. All of the artifacts are available at <http://research.cs.wisc.edu/multifacet/gpummu-hpca14/>.

- Changes to gem5-gpu to implement the optimized MMU described in this chapter. Most of this work is integrated into the upstream gem5-gpu (<https://gem5-gpu.cs.wisc.edu/>).
- Benchmarks were modified to support a shared virtual address and minor bug fixes

were added. Both binary and sources are available.

- Other supporting files used for full-system simulation.
- Data generated from running the simulator with specific configurations.
- Scripts to process the data and generate the graphs.

— 5 —

HIGH PERFORMANCE ANALYTIC DATABASES ON TIGHTLY-INTEGRATED GPGPU ARCHITECTURES

In Chapter 4, we showed how to design a memory management unit for the GPU which allows application developers to use pointer-is-a-pointer semantics with physically integrated GPGPUs. A shared virtual address space and cache coherence [114] between the CPU and GPU enables new workloads to take advantage of the GPU. In this chapter, we explore one of these workloads: analytic database systems.

5.1 Introduction

Businesses from all sectors perform complex analyses on the data they collect from their customers, supply chain, etc. The kinds of analytic database queries we focus on in this chapter include choosing which ad to serve on a webpage, customer recommendation engines, and traditional data warehouse queries. The performance of analytic queries is increasingly important with many of these use cases requiring real-time response latency (milliseconds to seconds). Due to these low-latency response requirements, many analytic databases store all of their data in memory and rarely, if ever, access the disk.

In this chapter, we limit our focus to scan-aggregate queries for two reasons. First,

scans and aggregates are data-parallel operations that are strong candidates for offloading to the GPU, and in this work we make our argument by focusing our attention on scan-aggregate queries in in-memory settings. Second, scans are an important primitive and the workhorse in high-performance in-memory database systems like SAP HANA [38, 153], Oracle Exalytics [43], IBM DB2 BLU [122] and Facebook’s Scuba [3].

A series of scan algorithms have been developed in the database community to exploit this parallelism using hardware artifacts such as the parallelism within regular ALU words (e.g., [84, 123]), and SIMD (short vector units) to accelerate scans (e.g., [29, 65, 84, 123, 152, 153, 159]). The GPU hardware is designed for highly data-parallel workloads like scan-aggregate queries and we show that it can provide higher performance and higher energy efficiency than CPUs. As more processors include integrated on-die GPUs, it will become important for database systems to take advantage of these devices.

Because of the GPU’s potential for increased performance, there has been work accelerating database operations such as sort, scan, aggregate, and join with *discrete GPUs* (e.g., [46, 47, 69, 134]). These works show discrete GPUs can improve performance for some database operations by over $20\times$.

However, mainstream database systems still do not commonly use GPGPUs, because there are many overheads associated with discrete GPUs that negate many of their potential benefits. Due to the limited memory capacity of discrete GPUs, when accessing large data sets, the data must be copied across the relatively low bandwidth (16 GB/s) PCIe interface. This data copy time can be up to 98% of the total time to complete a scan using the discrete GPU. Additionally, applications must frequently access the operating system-level device driver to coordinate between the CPU and the GPU, which incurs significant overhead. Many previous works have discounted the overhead of copying the data from the CPU memory, which results in optimistic performance predictions. *We find that by including these overheads, the discrete GPU can be $3\times$ slower than a multicore CPU.* This result shows that although discrete GPUs may seem a good fit for performing scans, due to their limited

memory capacity they are not practical today. However, we argue that this situation is likely to change due to recent GPU hardware trends.

Today, many systems are integrating the GPU onto the same silicon die as the CPU, and these *integrated GPUs* with new programming models reduce the overheads of using GPGPUs in database systems. These integrated GPUs share both physical and virtual memory with the CPU. With new GPGPU APIs like heterogeneous system architecture (HSA) [127], integrated GPUs can transparently access all of the CPUs' memory, greatly simplifying application programming (see Figure 5.2b and Section 2.3). HSA reduces the programming overhead of using GPUs because the CPU and GPU share a single copy of the application data, making it possible to make run-time decisions to use the CPU or the GPU for all or part of the query. As integrated GPUs become more common, it will be important for database systems to take advantage of the computational capability of the silicon devoted to GPUs. Furthermore, GPGPUs have become far easier to program, making it more economical to write and maintain specialized GPU routines.

We implement an in-memory GPU database system by leveraging the fast-scan technique BitWeaving [84] and the database denormalization technique WideTable [85], and a new method to compute aggregates on GPUs efficiently. Using this implementation, we show the benefits of our approach for an important, but admittedly limited, class of scan-aggregate queries. We find that the integrated GPU can provide a speedup of as much as $2.3\times$ on scans and up to $3.8\times$ on aggregates. We also evaluate our algorithms on 16 TPC-H queries (using the WideTable technique) and find that by combining the aggregate and scan optimizations, the integrated GPU can increase performance by an average of 30% (up to $3.2\times$) and decrease energy by 45% on average over a four-core CPU. Thus, we conclude that it is now practical for database systems to actually deploy methods for GPUs in production settings for in-memory scan-aggregate queries. With the proliferation of integrated GPUs, ignoring this computational engine may leave significant performance on the table.

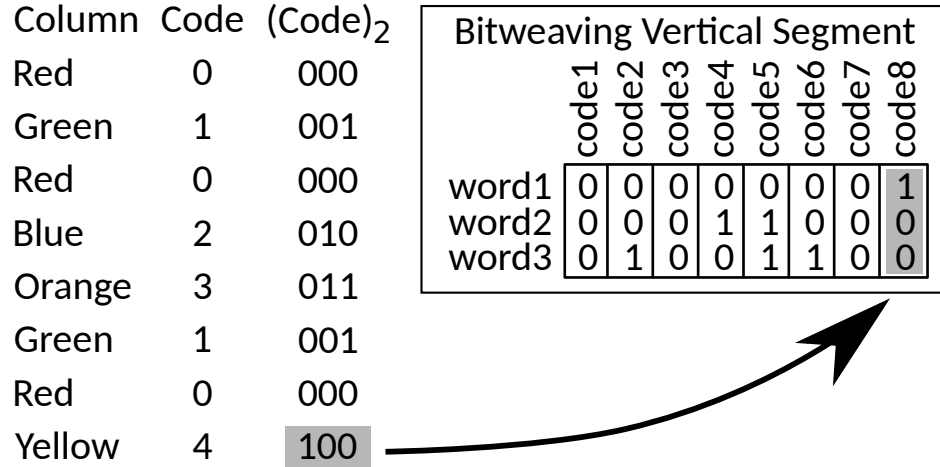


Figure 5.1: Overview of BitWeaving’s memory layout assuming an 8-bit word. The actual implementation uses 128-bit word for the SIMD CPU and 16,384-bit word for the GPU.

5.2 Implementation

To study scan-aggregate query processing on GPUs, we leverage previous work accelerating analytical scan query processing on the CPU: i.e., we use the BitWeaving scan algorithm [84]. BitWeaving uses a coded columnar layout, packing multiple codes per word. The output of the scan is a bitvector where each 1 bit corresponds to a matching row.

In this work, we use the BitWeaving/V scan algorithm. This algorithm encodes columns using order-preserving dictionaries and then stores the encoded values for a column grouped by the bit position. At a high-level, BitWeaving/V can be thought of as a column store at the bit-level with algorithms that allow evaluating traditional SQL scan predicates in the native bit-level columnar format using simple bitwise operations, such as XOR, AND, and binary addition. (See Li and Patel [84] for details). Figure 5.1 shows an example of a BitWeaving/V representation for a sample column.

The BitWeaving/V method needs small modifications to execute efficiently on GPUs. The largest change to adapt BitWeaving to the GPU is to increase the underlying word size to the logical SIMD width of the GPU. Thus, instead of packing coded values into one 64-bit word as in CPU algorithms, or 128- or 256-bit words in SIMD scan implementations,

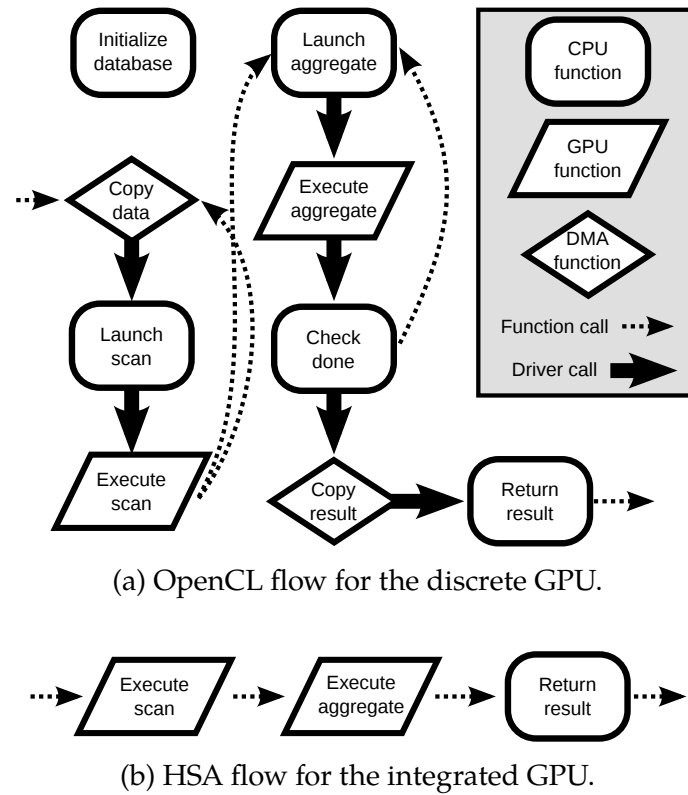


Figure 5.2: Flowchart showing the CPU-GPU interaction.

the GPU scan implementation uses a logical 16,384-bit word (256 consecutive 64-bit words).

For the discrete GPU, Figure 5.2a shows an overview of how the CPU and the GPU interact using OpenCL—the legacy GPU API. After the database is initialized, when a query arrives, the columns which the query references must be copied to the device, which is a DMA call. Next, the scan is launched to the GPU, which requires a high-latency user-mode to kernel-mode switch and high-latency PCIe communication (shown with bold arrow in Figure 5.2a). Then, the scan is actually executed on the GPU, which may take significantly less time than the previous operations. Once the (possibly many) scan operations have completed, the aggregate operation is launched in a similar fashion incurring the high OS kernel overheads. Finally, the result must be copied back to the CPU memory again paying the OS kernel and PCIe latencies.

Performing this scan on the discrete GPU requires many calls to the operating system's

kernel display driver, shown as bold arrows in Figure 5.2a. These driver calls incur significant overhead for short running GPU jobs such as column scans. Additionally, since the GPU cannot hold the entire database in its memory, each column must be copied to the GPU before the scan kernel can begin. As most queries require scans over multiple columns, this scan loop is repeated many times.

However, most of these overheads are eliminated on integrated GPUs. Figure 5.2b shows an overview of how the CPU and the GPU interact in our system using HSA. With HSA, integrated GPUs do not interface through the OS kernel driver. Instead, they use user-mode queues which provide low-latency to offload work to the GPU [127]. Therefore, the overhead to use the integrated GPU is simply a few writes to coherent memory with no driver involvement. Additionally, since the integrated GPU shares memory with the CPU, the copies used in the OpenCL version are eliminated. As Figure 5.2b shows, executing a query on the integrated GPU with HSA only requires a few function calls similar to how the CPU executes the query.

Aggregate computation

Scan queries generally tend to have aggregation operations in analytic environments, and the aggregate component can dominate the overall query execution time in some cases. In the simplified, scan-based queries that we evaluate in this chapter, a significant percentage of the time is spent in aggregates, an average of 80% across the queries that we use in this chapter. In fact, we find that some queries spend over 99% of their execution time in the aggregate phase.

Since aggregate performance can dominate the overall query performance in some scan-aggregate queries, we investigate offloading the aggregate computation to the GPU (in addition to offloading the scan computation). Tightly-integrated programming models like HSA 2.3 significantly decrease the complexity of offloading the aggregate computation to the GPU. As discussed below, parts of the aggregate computation are more efficiently

performed on the CPU and other parts are more efficiently performed on the GPU. It is easier to take advantage of this fine-grained offloading with current integrated GPUs and the HSA programming model.

In many modern systems, the output of a selection operation produces a result bitvector. In this bitvector, a bit value of 1 indicates the the tuple at the corresponding index position was selected by that operation. To actually fetch the data for the selected tuple, the index value must be converted to an absolute offset that points to the actual memory location of the selected tuple. Instead of evaluating every valid bit sequentially, we batch many valid column offsets and call the computational function only once, as shown in Algorithm 2. We split each column into blocks of 2^{20} or about one million codes. This approach optimizes the aggregate primitive in two ways. This algorithm gives much better memory locality when searching for valid bits in the bitvector, and by batching, we decrease the overhead of function calls on the CPU and launching jobs to the GPU.

Algorithm 2 Aggregate algorithm

Require: bitvector that encodes matching tuples
 split into 2^{20} -bit blocks
Require: reduce functor to compute the aggregate
 offsets = [] // Indices of matching tuples
 2: **for** each bv_block in bitvector **do**
 for each word in bv_block **do**
 4: **for** each true bit in word **do**
 offsets.append(word_index * 64 + bit_index)
 6: // call functor for group-by/agg.
 reduce(offsets) // Offloaded to integrated GPU
 8: clear(offsets)

This algorithm takes a bitvector with the selected tuples and a functor (see Algorithm 3 and 4) that is the actual aggregate computation. The bitvector is split into blocks which are each operated on in serial. For each word (8-bytes) of the bitvector block, we search for valid bits (lines 2–4). For each bit that is 1, which corresponds to a selected tuple, we insert the absolute offset into the column into an offsets array. Then, we execute the user-provided functor which computes the aggregate given the computed offsets for that column block.

We find that it is more efficient to use the CPU to compute the offsets than the GPU. When creating the offsets array, many threads may simultaneously try to add new offsets, and it is inefficient for the GPU to coordinate between these threads [32]. Therefore, when performing aggregates on the GPU, we first compute the offsets on the CPU and only perform the reduction operation on the GPU.

Many scan-aggregate queries include group-by operations in their aggregate phase. The number of groups in the group-by statements vary greatly, from a single group (i.e., a simple reduction) to millions of different groups. For group-by operations with multiple columns the number of groups can be even larger.

We investigate two different algorithms to compute group-by aggregate: direct storage (Algorithm 3) and hashed storage (Algorithm 4). The direct storage algorithm allocates space for every group and directly updates the corresponding values. The hashed storage algorithm uses a hash of the group ID instead and does not require a one-to-one mapping. The hash-based algorithm is required when there are many groups or the groups are sparse as the direct storage method would use too much memory.

Algorithm 3 Direct group-by (reduce with direct storage)

Require: offsets a list of tuple indices

```

1: // array for storage of aggregate value for each group
2: static aggregates[num_groups]
3: for each offset in offsets do
4:   // Materialize the column data and calculate group_id
5:   // Perform aggregate based on op
6:   aggregates[group_id] (op) data

```

In the direct group-by algorithm, we store the aggregate values of each group directly in an array (aggregates on line 2). These values are stored statically to persist across multiple instances of this function. For each offset in the provided offsets array, we look up the value contained in the tuple at that offset, which may result in accessing multiple columns. Then, we perform the aggregate operation (op on line 6) to update the stored value of the aggregate. To perform averages and other similar operations, we can also count the

number of matches in a similar fashion.

Algorithm 4 Hashed group-by (reduce with hashtable)

Require: offsets a list of tuple indices

```

1: static hashtable(predicted_size)
2: for each offset in offsets do
3:   // Materialize the column data and calculate group_id
4:   entry = hashtable.get(group_id) // get group entry
5:   // Perform aggregate based on op
6:   entry.value (op) data

```

In the hashed group-by algorithm, the values for each group’s aggregate are stored in a hashtable instead of in a simple array. We can use profiling to predict the size for the hashtable (line 1). The hashed algorithm performs the same steps as the above direct algorithm, except it acts on values in the hash entries instead of the direct storage array. The hash table is implemented with linear probing and uses the MurmurHash function.

To use these algorithms in a parallel environment, we use synchronization to guard the reduction variables. The GPU does not have mature support for inter-thread communication. Therefore, we use lock-free algorithms for the hashtable in Algorithm 4 and a local reduction tree for the direct storage algorithm (Algorithm 3).

	CPU	Integrated GPU	Discrete GPU
Part name	A10-7850K	R7	HD 7970
Cores / CUs	4	8	32
Clock speed	3700 MHz	720 MHz	1125 MHz
Mem. bandwidth	21 GB/s	21 GB/s	264 GB/s
Total memory	16 GB (shared)		3 GB
Rated TDP	95 W (combined)		225 W

Table 5.1: Details of hardware architectures evaluated. (Note: TDP for the CPU and integrated GPU does not include the memory power. TDP for the discrete GPU includes the GDDR5 memory.)

5.3 Methodology

There is a large space of potential hardware to run a database system. For a constant comparison point, we use AMD CPU and GPU platforms in our evaluation. We use a four-core CPU and two different GPUs, an integrated GPU that is on the same die as the CPU, and a discrete GPU connected to the CPU via the PCIe bus. Table 5.1 contains the details of each architecture.

For power measurements, we use the full-system power measured with a WattsUp meter. The WattsUp meter is connected between the system under test and the power supply (wall jack). Since all of our runs last for minutes, we use the energy reported by the WattsUp meter as our primary metric for energy usage.

For the results presented for multiple CPU cores, we used OpenMP to parallelize the scan and the aggregate functions. Each function performs a highly data-parallel computation on the entire column of data. Thus, the fork-join parallelism model of OpenMP is appropriate.

We use the HSA programming interface described in Section 2.3 for the integrated GPU case. The GPU kernels (scan, simple aggregate, direct- and hashed-group-by) are written in OpenCL. For the integrated GPU, all of the data is allocated and initialized by the CPU in the main physical memory and directly referenced by the integrated GPU. The discrete GPU uses similar OpenCL kernels, but the data is explicitly copied from the CPU memory to the GPU, which is not necessary when using the integrated GPU and HSA.

We use TPC-H as a proxy of analytic database workloads [147]. TPC-H is a decision support benchmark that examines large amounts of data with complex queries meant to model business questions. TPC-H consists of 22 queries and the total size of the database is configurable. The TPC-H benchmark is an industry standard test of database performance with many organizations submitting TPC-H performance results on real systems.

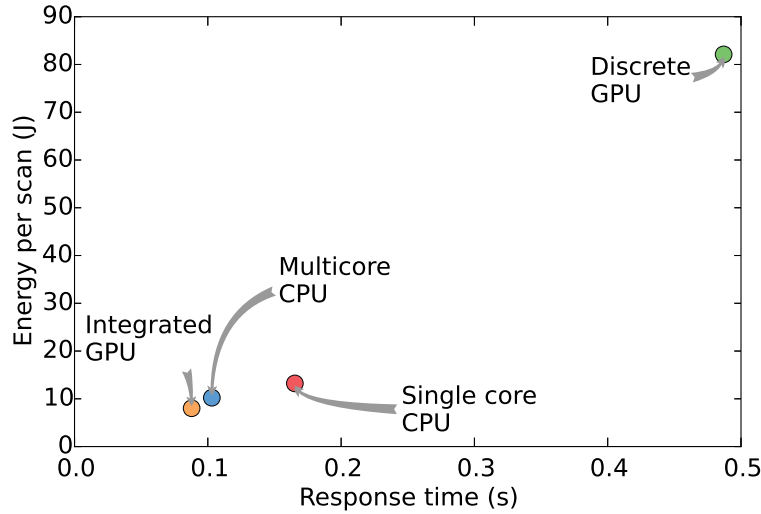


Figure 5.3: Performance and energy of scan microbenchmark.

5.4 Results

In this section, we first discuss the performance and energy characteristics of scans on discrete and integrated GPUs. Next, we evaluate our scan and aggregate algorithms on a set of scan-aggregate TPC-H queries. Finally, we show the tradeoffs between the two GPU aggregate algorithms discussed and compare their performance to the CPU.

Scans (discrete vs. integrated GPUs)

We first investigate the performance of the scan operator on the two GPU architectures. The scan microbenchmark that we run performs a simple selection operation (e.g., `SELECT count(*) from TABLE, omitting the actual count operation`) on a single column. The microbenchmark applies a predicate which has a 10% selectivity. Thus, this benchmark measures the raw cost associated with scanning a column excluding the costs to process the result bitvector that is produced.

Figure 5.3 shows the performance and energy of the scan operation on a discrete GPU and an integrated GPU. This figure shows the number of seconds to complete a scan of 1 billion 10-bit codes averaged over 1000 scans. The total size of the column is about 1 GB.

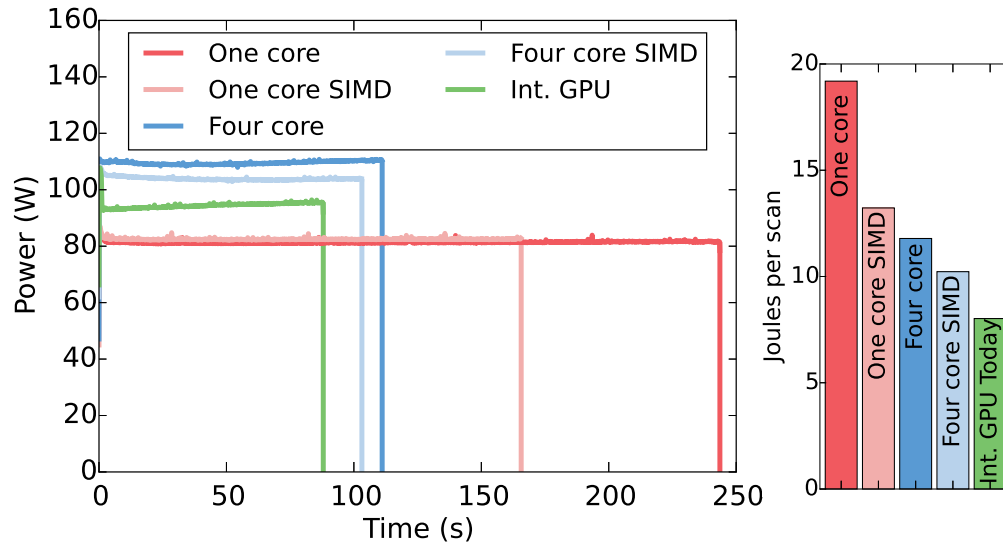


Figure 5.4: Power for 1000 scans and energy per scan.

Performance is shown on the x-axis (response time), and energy is shown on the y-axis.

Figure 5.3 shows that performing a scan on the integrated GPU is both $3\times$ faster and $3\times$ lower energy than performing a scan on the discrete GPU. Although the discrete GPU has many more compute units and much higher memory bandwidth, the time to copy the data from the CPU memory to the GPU memory dominates the performance (about 98% of the execution time). This time can be decreased by overlapping the memory copy with computation; however, the performance on the discrete GPU is fundamentally limited by the low bandwidth and high latency of the PCIe bus.

The integrated GPU has a slightly faster response time than the multicore CPU (17% speedup), but its energy consumption is significantly lower (27% less energy consumed). Since the integrated GPU and CPU share the same memory interface, the scan performance should be similar because scan is a bandwidth-bound computation. However, the power consumption of the integrated GPU is lower than the multicore CPU. The full-system power when using the integrated GPU is about 90 W compared to 110 W for the multicore CPU. Thus, in today's systems, if energy consumption is important, the integrated GPU provides a significant benefit over the multicore CPU for simple scans.

Figure 5.4 shows the power and energy consumed by the CPU and GPU when performing 1000 scans. The data was obtained by measuring the full system power; thus, it includes all of the system components (e.g., disk, motherboard, DRAM, etc.). The right side of the figure shows the total energy consumed (power integrated over time).

Figure 5.4 shows that even though the four core configuration and the integrated GPU take more power than the one core CPU, they execute much faster, resulting in lower overall energy. Additionally, the integrated GPU is more efficient than the four core CPU in power and performance.

In the future, it's likely that the GPU will become even more energy efficient compared to the CPU. Each GPU compute unit (CU) (similar to a CPU core) has lower power per performance than a CPU core [73]. Also, each GPU CU is smaller area per performance than CPU cores. Thus, there can be many more GPU CUs than CPU cores, exemplified by our test platform.

In the rest of this chapter, we do not evaluate the discrete GPU for two reasons. First, the integrated GPU outperforms the discrete GPU for the scan operation when the copy and initialization overheads are included. Second, it is difficult to implement the aggregate on the discrete GPU due to the complex data structures used in our algorithms (e.g., a hashtable). It may be feasible to design optimal scheduling strategies and new data structures for the discrete GPU. However, we find that the integrated GPU provides performance improvement and energy savings without paying the high programming overhead of the discrete GPU.

TPC-H

For the workload, we used sixteen TPC-H queries on a pre-joined dataset as was done in Li et al. and Sun et al. [85, 143]; i.e., we pre-joined the TPC-H dataset and ran queries on the materialized dataset (WideTable) using scans and aggregate operations. The queries not included have string-match operators, which have not been implemented in our system.

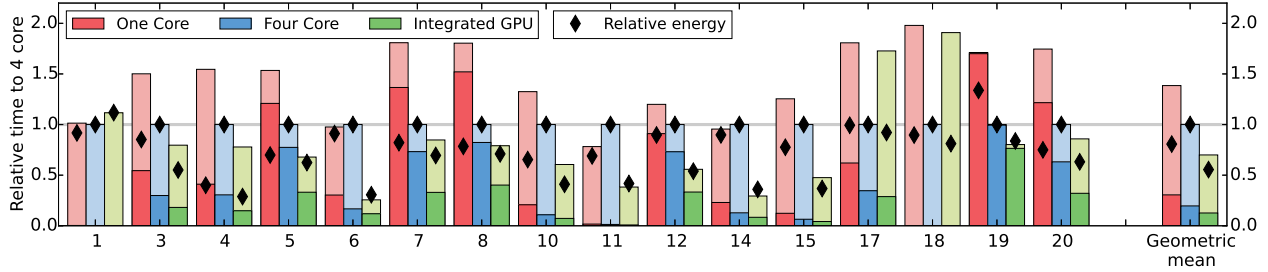


Figure 5.5: Performance and energy for 16 TPC-H queries normalized to the four-core CPU. Lower bold portion of each bar is the scan time and the upper shaded portion of each bar is the rest of the time spent in aggregate, group-by, and sort operations. Energy is shown on the right axis with diamonds relative to the four-core CPU.

The queries evaluated show a range of different behaviors: the percent of time in scan operations ranges from about 0% to more than 99%; the number of scans performed ranges from zero scans to 18 scans; and the columns vary in bit width from 3- to 32-bits. We run each query with a scale-factor 10 data set.

Execution time of the TPC-H queries relative to the multicore CPU is shown in Figure 5.5. The execution time of each query is broken into two sections: the time to perform the scan (the darker bottom part in each graph), and the time to execute the rest of the query. The non-scan parts of the queries include aggregate, group-by, and sort operations. Only the aggregate and group-by operations were ported to the GPU; all sort operations execute on the CPU.

The two queries which see the most performance improvement using the integrated GPU are query 6 and query 14. For these two queries, the GPU is able to significantly increase the performance of the aggregate computation. The reason for this performance improvement is that these queries perform a simple reduction (one group) and touch a large amount of data. These two queries have a relatively high selectivity (1–2%) and access large-width columns. For these two queries, the increased parallelism of the GPU can take advantage of the memory bandwidth in the system better than the multi-core CPU platform.

For some queries, the integrated GPU significantly outperforms the multicore CPU in

the scan operation (e.g., query 5 and query 12). The reason for this higher performance is that these queries spend a large percentage of their execution time in single table multi-column (STMC) predicates. STMC predicates scan through two columns comparing each row (e.g., in query 12 `l_commitdate < l_receiptdate`). For STMC scans there is an even smaller compute to memory access ratio than for single-column predicate scans. Thus, the integrated GPU can use the available memory bandwidth more effectively than the multicore CPU.

Overall, the integrated GPU increases performance compared to the multicore CPU of both the scan and aggregate portion of the TPC-H queries evaluated as shown by the geometric mean in Figure 5.5. The integrated GPU shows a 35% average reduction in response time for scans, a 28% reduction in response time on aggregates, and an overall 30% average performance improvement.

The diamonds in Figure 5.5 show the the whole-system energy of the each TCH-H query evaluated relative to the multicore CPU (right axis, measured with a WattsUp meter). This data includes both the scan and the aggregate portions of each query. Surprisingly, the single-core CPU uses less energy than the multicore CPU, on average. The reason for this behavior is that the aggregate portion of the queries is not energy-efficient to parallelize. Performing parallel aggregates results in a performance increase, but not enough to offset the extra power required to use all four CPU cores.

Figure 5.5 also show that the integrated GPU is more energy-efficient when performing whole queries. The integrated GPU uses 45% less energy than the multicore CPU and 30% less energy than the single-core CPU.

We found that some aggregate queries perform poorly on the integrated GPU (Q1, Q5, Q7, and Q8). Thus, we use a mixed-mode algorithm for the aggregate computation (Section 5.4 presents a more detailed analysis). Additionally, we found that some aggregate queries (Q1, Q6, Q11, and Q14) perform poorly when parallelized. The aggregate operations in these queries do not parallelize efficiently due to overheads from locks to keep the shared

data structures coherent. For the queries that aggregate operations perform poorly on the multicore CPU or the integrated GPU, we use a single CPU core to execute the aggregate operation.

Aggregates

In this section, we compare the aggregate algorithms discussed in Section 5.2. We use a microbenchmark which applies a selection predicate on a table with two (10-bit) integer columns and then performs an aggregate operation returning the sum of the values in the first column grouped by the second column's value. We vary the codes size of the two columns and the selectivity of the query.

Figure 5.6 shows the performance of the two group-by algorithms on the integrated GPU assuming a 0.1%, a 1.0%, and a 10% selectivity in the scan portion of the query. We found that selectivities above about 10% were dominated by the time to materialize the codes, not the group-by algorithm. Each graph shows the performance of the two algorithms discussed in Section 5.2, direct (Algorithm 3) and hashed (Algorithm 4). Since Algorithm 3 is only appropriate when there are a small number of groups and Algorithm 4 is only appropriate when there are a large number of groups, each algorithm was only evaluated on a subset of the evaluated groups (x-axis).

Figure 5.6 shows that for the simple reduction and a large numbers of groups, the GPU outperforms the CPU when using the hashed group-by algorithm. However, there is a range of groups for which neither the hashed nor the direct group-by algorithm performs well on the GPU. For this range, the CPU significantly outperforms the GPU (up to about 1024 groups). Therefore, we advocate using a mixed-mode algorithm. Luckily, the HSA programming model makes it possible to perform a simple run-time decision.

The reason the GPU performs poorly with a small number of groups is two-fold. First, when using the direct algorithm, the GPU must allocate memory to store the local results for every single GPU thread, which can be more than 10,000. This memory overhead means

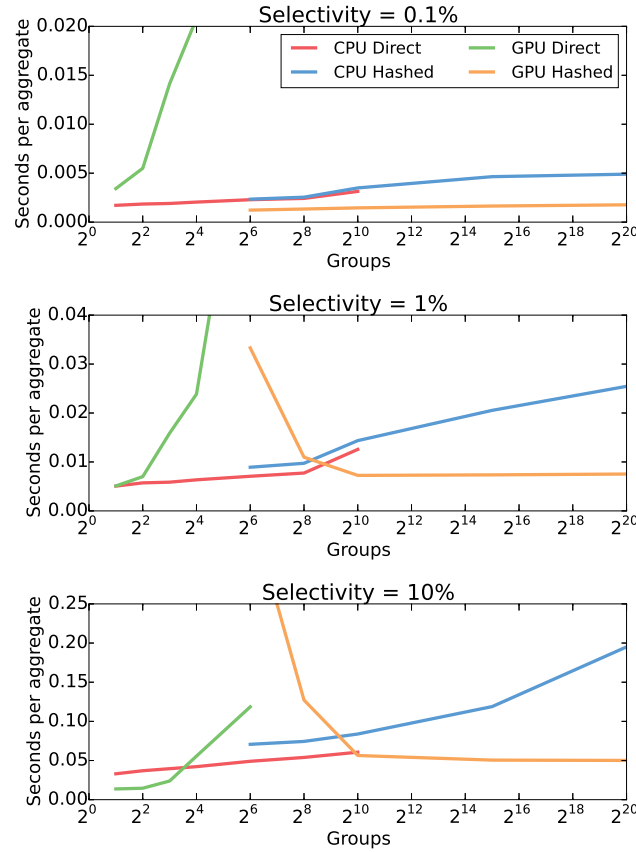


Figure 5.6: Performance of aggregates comparing number of groups with three selectivities.

the direct algorithm can only be used when there are 64 groups or fewer. Additionally, this per-thread memory is not optimized on the GPU. Second, when using the hashed storage algorithm, for small numbers of groups (less than 1024), the hashed storage algorithm performs poorly on the GPU because there is contention for the data in the hash table. The contention causes the lock-free algorithm we use to perform poorly.

We also find that selectivity affects the speedup of the GPU over the CPU for the aggregate primitive for the simple reduction (group-by with one group) and hashed group-by algorithm. For a simple reduction (one group), as the selectivity increases and the total number of records that are materialized grows, the GPU is able to achieve more speedup over the CPU. When the selectivity is higher, there is more memory parallelism available and the GPU performs better. Additionally, we use the CPU to compute the offsets and the

GPU only computes the reduction value. Thus, with very low selectivity rates, the CPU time dominates the aggregate and performing the entire aggregate on the CPU has the highest performance.

5.5 Related work

GPU acceleration. There is a rich body of work accelerating database applications on GPUs, but this prior work focuses on discrete GPU systems. Govindaraju et al. showed that the GPU provides a significant performance improvement for some database computations [47]. He et al. [51] show that GPU joins can provide a $2\times$ to $7\times$ performance improvement over their CPU counterparts. This work was extended to show promise for other database operations, like sort [46, 134], tree search [75], and join [52, 76, 86].

Fang et al. [37] use compression to reduce the total data transferred to discrete GPUs, mitigating the data transfer bottleneck associated with the PCIe bus, because of the low bandwidth between system and GPU memory. Their compression technique can increase the effective bandwidth of the PCIe bus up to 56 GB/s reducing the data-transfer bottleneck. This work eliminated some of the problems with discrete GPUs, but integrated GPUs achieve the same goal of hiding memory copy times without requiring compression.

Kaldewey et al. [69] use a new feature in NVIDIA GPUs, unified virtual addressing, which allows the GPU to access CPU DRAM memory. By accessing the CPU memory, the GPU can reference an order of magnitude more memory than what current GPU hardware provides. The authors show an almost $5\times$ speedup when using the GPU compared to the CPU algorithms.

In contrast to prior work, we are the first to evaluate physically (on the same chip) and logically (with a shared address space) integrated GPUs for database scans. This architecture provides many of the same performance benefits as seen by Kaldewey et al. due to cache coherence and a high bandwidth on die interconnect.

SIMD optimizations. There is a rich body of work on leveraging SIMD instructions to speed up database operations. The first work in this field was from Zhou and Ross [159]. They implemented several key database operations, including scans, aggregation, index operations, and joins using the SIMD instructions, and showed that there are significant benefits in using the SIMD instructions for database query processing.

The SIMD-scan [152, 153] method is the state-of-the-art scan method that uses SIMD instructions. This technique keeps data in compressed form (in main memory) and uses the SIMD instructions speed up the scan. The technique loads compressed column values into four banks in a SIMD word in parallel, and then processes the compressed data simultaneously using SIMD instructions. SIMD instructions are also critical for implementing efficient sorting algorithms [134], and the sort-based join algorithms [76]. According to the analysis in the work [134], merge sort is generally faster than radix sort for large SIMD widths on future architectures. Similarly, it has been shown that sort-merge join has a potential to outperform hash join on upcoming architecture with wider SIMD registers [76]. SIMD instructions are also used to improve the performance of searching index trees [75], computing aggregate functions [112], and looking up bloom filters [113].

Full-table scans. There has been an increasing interest in efficient full table scan methods for main memory database systems. Many scan methods have been proposed to exploit the parallelism available at the word-level in modern processors, e.g., the BitWeaving methods [84], the scan primitive used in IBM Blink system [65, 123], and the SIMD-based scan implementation used in SAP HANA [152, 153]. In this chapter, we focus on the BitWeaving method, but our analysis of the scan performance on modern and future GPUs is also generally applicable to the other scan methods.

The full table scan primitive has also been well optimized for modern architecture in other aspects. Gold et al. [44] studied the scan performance on a network processor. Lin et al. [120] proposed algorithms to share the CPU cache when performing concurrent scans on modern multi-core processors. Switakowski et al. [144] proposed a buffer manager

design to apply cooperative scans on Vectorwise. Advanced thread scheduling method was also proposed to efficiently share caches on multicore processors [28]. Holloway et al. [53] present a scan generator to create highly optimized code for scan operations over compressed data. These methods are complementary to our proposed methods.

Specialized processors for database systems. The need for special hardware to speed up database operations arose in the early 1980's. However, in the initial effort, it had been shown that designing and manufacturing such specialized hardware was not economical, as such specialized hardware simply could not keep up with the rate at which commodity computing (general purpose processors) were getting cheaper and more powerful [22].

There is a resurgent interest in considering specialization in the processors, driven by the increasing demand for high-performance analytics engines and the customization capability of future processors [23], e.g., programmable accelerator, and customizable logic. At the extreme of this trend, there is a growing set of research in building highly specialized hardware for database operations (e.g., Oracle SPARC M7 [110] and Q100 [154]).

We echo the need for specialized hardware but think that there is a balance between highly specialized hardware which has the problems associated with traditional database machine and very general hardware (like current CPUs that have hit the power wall). We speculate that general-purpose data-parallel hardware (like a GPU) provides an economically viable middle ground between this highly-specialized hardware and very general hardware.

5.6 Conclusions

Previous works have shown the huge potential of using GPUs for database operations. However, many of these works have neglected to include the large overheads associated with discrete GPUs when operating on large in-memory databases. We show current *physically and logically integrated* GPUs enabled by our previous research (Chapter 4 and

HSC [114]) mitigate the problems associated with discrete GPUs showing a modest speedup and energy reduction over multicore CPUs for scan-aggregate queries.

To achieve the full potential of GPUs, we must re-architect the system to provide higher bandwidth to the integrated CPU-GPU chip. In Chapter 6, we explore the performance potential and the tradeoffs of a CPU-GPU system with high-bandwidth die-stacked memory.

— 6 —

IMPLICATIONS OF 3D DIE-STACKED MEMORY FOR BANDWIDTH-CONSTRAINED WORKLOADS

In Chapter 5, we showed that there are performance benefits to using integrated GPUs compared to traditional multicore CPU architectures. However, if we ignore the overheads of moving data, there could be even larger benefits (almost $16\times$) when using the discrete GPU due to its higher memory bandwidth. In this chapter, we explore the performance, power, and cost tradeoffs when designing a system for bandwidth-constrained workloads like the workload in Chapter 5 that combines the best of the discrete GPU (high memory bandwidth) and integrated accelerators (low overhead offloading).

6.1 Introduction

New data is generated at an alarming rate: as much as 2.6 exabytes are created each day [89]. For instance, with the growth of internet-connected devices, there is an explosion of data from sensors on these devices. The ability to analyze this data is important to many different industries from automotive to health care [105]. Users want to run complex queries on this abundance of data, sometimes with real-time constraints.

To meet the latency requirements of real-time queries, many big data systems keep all of

their data in main memory. In-memory queries provide lower response times than queries that access disk or flash storage, and are increasingly popular [158]. The in-memory data may be the entire data corpus, or the hot parts of the data set.

In this chapter, we evaluate bandwidth constrained workloads with large data sets. Many big data workloads are memory intensive, performing a small number of computational operations per byte of data loaded from memory. This low computational intensity means these workloads are increasingly constrained by memory bandwidth, not the capabilities of the processor. This problem is exacerbated by the increasing gap between processor speed and memory bandwidth [24]. One example workload that fits these characteristics is the database workload detailed in Chapter 5.

As a solution to this memory bandwidth gap, industry is currently investing heavily in 3D die-stacked memory technologies [5, 20, 25, 56, 79, 102, 104, 108, 124, 131, 132]. 3D die-stacked memory enables DRAM chips stacked directly on top of compute chip with through-silicon vias (TSVs). TSVs enable high-bandwidth low-power memory accesses. Currently only high-performance graphics processors use die-stacked memory. However, many major companies such as IBM, Intel, Samsung, Micron, 3M and others are making significant investments in 3D die-stacking.

We evaluate three different server architectures, two that are available from system providers today, and a potential future architecture based on 3D die-stacking. First, the *traditional server* represents a large-memory system assembled from commodity components. We use a Dell PowerEdge R930 as our example traditional server [35], configured with the maximum memory bandwidth per processor. Second, the *big-memory server* represents a specialized system designed to support a very large amount of main memory. This system is a database machine designed for in-memory database systems. We use an Oracle M7 as our example big-memory server [106]. Finally, the *die-stacked server* represents a novel big data machine design built around many memory-compute stacks [21, 74]. Section 6.3 and Figure 6.2 contains details of these three systems. Throughout this work, we assume these

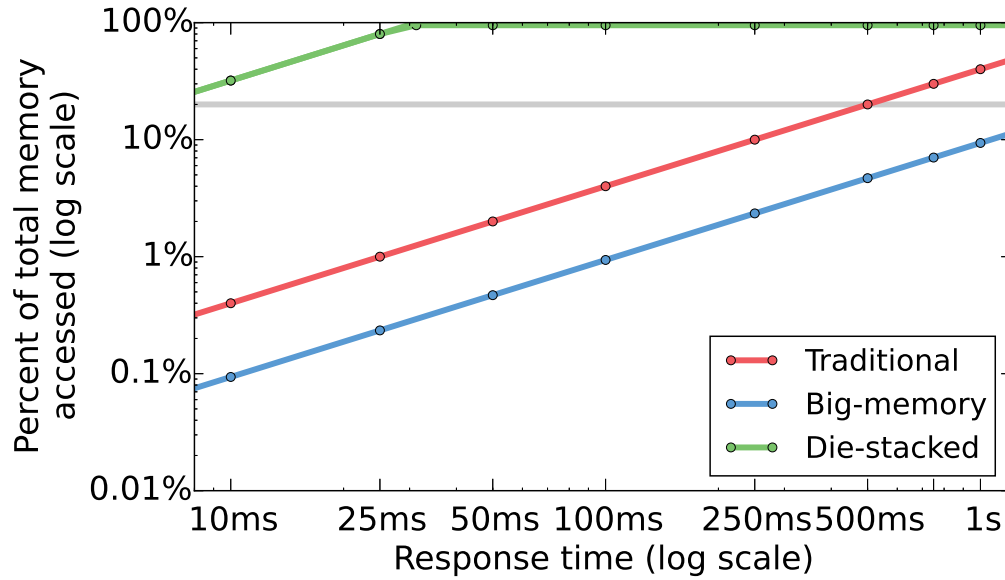


Figure 6.1: Comparison of the memory bandwidth-capacity tradeoff in current commercial systems.

system architectures are deployed in a cluster of computing systems.

To demonstrate the memory bandwidth gap, Figure 6.1 shows the amount of time it takes to read a fraction of the total memory capacity on a log-log scale. The amount of time, on the x-axis, represents the performance service-level agreement (SLA) between the service provider and the user. For instance, if the provider promises “real-time” access to the data (e.g., the data is queried to provide information to render a web-page) the required latency may be as little as 10–20ms. Since we focus on bandwidth-bound workloads, we use the amount of data accessed (read or written) by the query to measure its complexity (y-axis). In Figure 6.1, we show this complexity as the percent of the entire main memory capacity that is accessed on each query. As a point of reference, Figure 6.1 shows a line at 20% of memory capacity, which is the amount of an analytic database that is referenced in a single query in a popular analytic benchmark [147]. Przybylski makes a similar argument for scaling bandwidth with capacity with the “fill frequency” metric [119].

Figure 6.1 shows that the die-stacked design is better than the traditional and big-memory designs for low-latency bandwidth-limited workloads. The reason for this benefit is the die-stacked system has a much higher bandwidth-capacity ratio (memory bandwidth

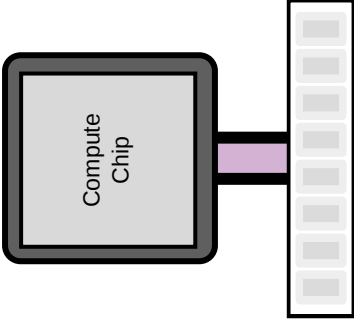
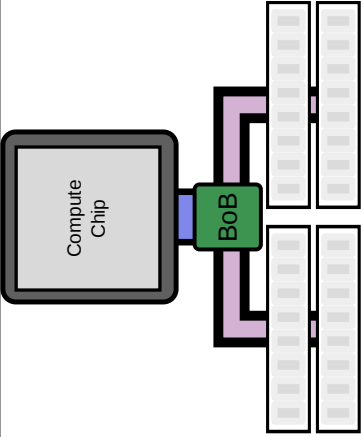
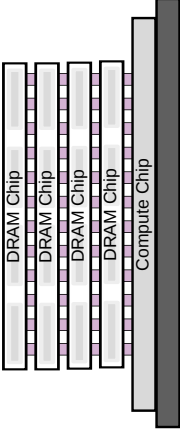
	Traditional	Big-memory	Die-stacked
	 (a) A traditional server memory configuration. In this configuration, DRAM DIMMs are directly connected to the compute chip. This is a commodity system available today that is configured for the maximum bandwidth-capacity ratio.	 (b) Big-memory server memory configuration. In this configuration, DRAM DIMMs are connected to a buffer-on-board (BoB), and the BoB is connected to the compute chip. This system is available today and is configured for the maximum potential DRAM capacity.	 (c) A 3D die-stacked server configuration. In this configuration, DRAM chips are integrated with the compute chip with through-silicon vias. This system is not yet available. It has a very high bandwidth-capacity ratio, and is tailored to bandwidth-bound memory workloads.
Off-socket bandwidth	102 GB/s	192 GB/s	256 GB/s
Memory per socket	256 GB	2 TB	8 GB
Sockets per blade	4	1	9
Blades needed for 16 TB capacity	16	8	228
Aggregate bandwidth for cluster	6.4 TB/s	1.5 TB/s	512 TB/s
Commercial example	PowerEdge R930 [35]	Oracle M7 [106]	N/A

Figure 6.2: Diagrams of the three system designs we consider in this chapter and potential configurations for each system.

per byte of memory capacity): $80\text{--}341\times$. This performance benefit exists even though the off-socket bandwidth is only $1.3\text{--}2.5\times$ higher than the other systems because the capacity per socket is much smaller for the die-stacked system ($32\text{--}256\times$). If a workload is bandwidth-constrained a higher bandwidth-capacity ratio will increase its performance. For instance, to access 20% of the total memory capacity, the big-memory server takes over 2 seconds, the traditional server takes 500 ms, and the die-stacked server takes less than 10 ms, $50\times$ faster than the current systems.

Although one could conclude from Figure 6.1 that the die-stacked system is always better than the current systems, we find that the die-stacked system *may be better* under some constraints. We use a simple back-of-the-envelope-style model to characterize *when* the die-stacked architecture beats the current architectures for in-memory big data workloads. We investigate designing cluster systems under three different system design constraints: performance provisioning (the cluster is required to meet a certain SLA), power provisioning (the cluster has a certain power budget), and data capacity provisioning (the cluster has a set DRAM capacity).

We find:

- When provisioning for performance, die-stacking is preferred only under aggressive SLAs (e.g., 10 ms) and the traditional and big-memory systems require memory over provisioning (up to $213\times$ more memory than the workload requires).
- When provisioning for power, die-stacking is preferred with large power budgets and the big-memory system provides the most memory capacity.
- When provisioning for capacity, die-stacking reduces response time by up to $256\times$ and uses less energy, but die-stacking uses more power (up to $50\times$).

6.2 Case study with database scan

As a first step to explore the performance potential of 3D die-stacking, we evaluate the scan algorithm from Chapter 5 on a hypothetical system with 3D die-stacked memory.

Methodology

For a constant comparison point, we use AMD CPU (A10-7850K) and GPU (HD7970) platforms in our evaluation. We use a four core CPU at 3.7 GHz and two different GPUs, an integrated GPU that is on the same die as the CPU, and a discrete GPU connected to the CPU via the PCIe bus. The GPUs have 8 CUs at 720 MHz and 32 CUs at 1125 MHz, respectively with the same microarchitecture (GCN [7]). The theoretical memory bandwidth for the CPU-GPU chip is 21 GB/s, and the discrete GPU's memory bandwidth is 264 GB/s. We use a single-socket system in our evaluation, but integrated CPU-GPU chips should scale to multiple sockets similar to CPU-only chips.

We use the discrete GPU as a model to predict the performance of the future die-stacked system. According to the projection in [2, 33, 108], 264 GB/s is a reasonable assumption for the memory bandwidth in a first-generation die-stacked system, and the 32 CU chip in the discrete GPU will fit into a package like Figure 6.2c.

3D architecture performance

To measure the efficacy of the GPU for the scan operation, we measured the performance, power, and energy consumed for the CPU and GPU hardware. Figure 6.3 shows the time per scan over a 1 billion entry column (about 1 GB of data).

The multicore CPU is not an efficient platform for performing the scan primitive. Figure 6.3 shows that the speedup of four cores over one core is only 60%. Scan is an embarrassingly parallel operation, so we expect almost perfect scaling from the scan algorithm. The reason the CPU does not scale linearly is that the CPU memory system is not designed

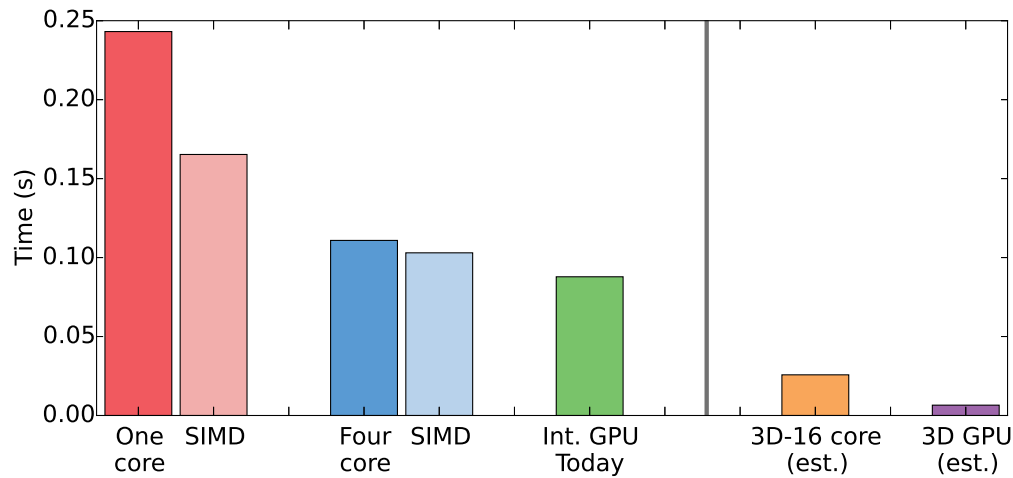


Figure 6.3: Performance of scan on 1 billion 10-bit codes. Averaged over 1000 scans.

to support high bandwidth applications. CPU caches are built for low-latency, not high bandwidth.

The integrated GPU sees a small improvement over the multicore with SIMD, a 17% speedup. This improvement principally is because the integrated GPU's more efficient cache hierarchy. GPUs are designed for these types of high-bandwidth applications and have a separate memory controller which can exploit the memory-level parallelism of the application better than the CPU.

There are two characteristics of 3D architecture that significantly affects the performance of the scan primitive. First, by stacking a GPU die with the CPU die, each processor type is more highly optimized and can take more total area, thus increasing the overall compute capability of the system. Second, since the chip-memory interconnect uses TSVs, the memory bandwidth is orders of magnitude higher than current systems. These attributes together create a higher performance and more energy efficient system than today's architectures.

The right side of Figure 6.3 shows the estimated performance of the CPU and the GPU in a 3D die-stacked system. To estimate the CPU's speedup, we assume that adding $4\times$ more cores will result in a $4\times$ speedup. This is a very aggressive estimate; the CPU is unlikely to get perfect speedup when adding more cores. There is only a 60% speedup

from one to four cores; however, the added bandwidth will increase the CPU performance too. We err on the side of overestimating the CPU performance.

To estimate the 3D GPU's performance, we run the scan operation on a current discrete GPU with 32 CUs ($4\times$ more execution resources than the integrated GPU) with a bandwidth similar to future die-stacked systems [33]. We discard all of the overheads associated with using the discrete GPU, since on a die-stacked system, the overheads will be minimal, as we found with the integrated GPU.

Using this projection, we find that a die-stacked GPU system can provide $15.7\times$ higher performance than today's non-stacked multicore CPU system. The 3D GPU is also $3.9\times$ faster than the aggressive 3D CPU estimate.

In addition to the performance gains, 3D die-stacked DRAM enables lower energy per access by using smaller and shorter wires than off-chip DRAM. Coupled with the GPU's efficiency, it is likely that the energy benefit of this system is much higher than the performance benefit.

We predict that to take advantage of the increasing memory bandwidth from 3D die-stacking, database designers must embrace high-bandwidth, highly data-parallel architectures, and we have shown that the GPU is a good candidate. Looking forward, CPU architecture will continue to become more efficient, including for data-parallel workloads. However, GPUs will also increase in efficiency at a similar rate. We believe that as these trends come to fruition, it will become increasingly important for database designers to leverage high-bandwidth data-parallel architectures to keep pace with the highest possible performance.

6.3 Potential big data machine designs

Figure 6.2 shows diagrams of the three different server architectures we investigate in this chapter: a traditional server, a big-memory server, and a die-stacked server. The traditional

server and big-memory server are systems that can be purchased today, and the die-stacked system is a future-looking machine architecture proposal. When evaluating these server designs, we assume many servers are networked together into a larger cluster.

The traditional server, shown in Figure 6.2a, is based on a commodity Intel Xeon platform [57]. The main memory is accessed directly from the memory controller on the compute chip and there are four 25.6 GB/s memory controllers on the chip for a total of 102 GB/s of off-chip memory bandwidth. Each memory controller can have up to 6 DRAM DIMMs connected. However, since our workload is bandwidth bound, we evaluate a system configured for the minimum capacity while still achieving the maximum bandwidth. Thus, we only populate each channel with two DDR4 DIMMs, since two DIMMs are required to achieve the peak off-chip bandwidth. Each DDR4 DIMM can have 32 GB of capacity, for a total of 256 GB of main memory capacity per socket. A similar system to what we evaluate is available from OEMs including Dell (e.g., the PowerEdge R930 [35]).

The big-memory system (Figure 6.2b) is a proxy for database appliance solutions from companies such as Oracle and IBM. Rather than directly access main memory like the traditional server, these systems use buffer-on-board chips to increase the maximum memory capacity. These buffer-on-board chips also increase the peak off-chip bandwidth by using different signaling protocols than DRAM DIMMs. Each buffer-on-board communicates both with the compute chip via a proprietary interconnect and up to eight DRAM DIMMs via the DDR4 standard. We use the Oracle M7 [106] as an example big-memory system. The M7 has eight buffer-on-board controllers per socket, for a total memory capacity of 512 GB and 192 GB/s per socket.

Finally, we also evaluate a novel architecture which combines compute chips with tightly-integrated memory via 3D die-stacking as shown in Figure 6.2c. This design is similar to the nanostore architecture [124]. Die-stacking technology increases the potential off-chip bandwidth by significantly increasing the number of wires between the chip and

the memory. There are currently no systems which use this technology in the database domain. However, this technology is gaining a foothold in the high-performance graphics community [5, 102]. We assume a system that uses the high bandwidth memory (HBM) 2.0 standard [61, 104]. With HBM 2.0, each stack can be up to eight DRAM chips high and each chip has 8 gigabits of capacity, for a total capacity of 8 GB per stack.

For all of the platforms, we assume the same DRAM chip technology: 8 gigabit chips. This gives a constant comparison in terms of capacity and power. Additionally, assuming 3D die-stacked DRAM reaches commodity production levels, the cost-per-byte of 3D DRAM will be about the same as the cost of DRAM DIMMs.

For these systems, we assume the limiting factor is the bandwidth, not the compute capability. There are many possible designs for a highly data-parallel system. Some examples include re-architecting CPUs by increasing the vector SIMD width and changing their cache management policies, Intel’s Xeon Phi processor which has 72 simple in-order cores with wide SIMD lanes [50], and GPU architecture which is an example of a highly data-parallel architecture that has already shown economic viability. Although there are many possible embodiments, we believe that any highly data-parallel architecture will share many characteristics with GPGPUs. These characteristics include:

- Large number of simultaneous threads to generate the memory-level parallelism needed to hide memory latency,
- Wide vector execution units that can issue vector memory accesses, and
- Many execution units to operate on data at memory-speed.

Additionally, it is likely that programming these devices will be similar to programming current GPGPUs. For instance, OpenCL is a flexible language which can execute on GPUs, CPUs, and other accelerators like the Xeon Phi. We focus on GPGPUs as an example data-parallel architecture that has already shown economic viability. We do not make any assumptions on the underlying architecture.

6.4 Methodology: Simple analytical model

Since it would be cost prohibitive to actually build and test these systems and simulation is many orders of magnitude slower than native execution, we use a simple analytical model to predict the performance of analytic database queries on each system. Additionally, an analytic model allows us to explore much larger design space than simulation. This model uses data from two sources. First, we use data from manufacturers as in Table 6.2, and second, we assume that the query is constrained either by the memory bandwidth or the rate at which instructions can be issued by the processor.

To calculate the compute performance and power, we performed experiments on a modern hardware platform. We used the scan algorithms from Chapter 5 and ran the tests on an AMD GPU [7]. We chose to perform our tests using a GPU as it is currently the highest performing and most energy efficient hardware platform for scans [117]. We found that each GPU core uses about 3 watts when performing the scan computation, and each core can process about 6 GB/s of data. Finally, we assume that each GPU chip has a maximum of 32 cores.

Equations 6.1–6.10 show the details of our simple model. The inputs to our model, are shown in Table 6.1. All of the inputs come from datasheets or manufacturer information [35, 57, 61, 104, 106]. A *memory module* refers to the minimum amount of memory that can be added or removed. In the traditional system this refers to a DRAM DIMM. In the big-memory system a buffer-on-board and its associated memory is considered a module. Finally, in the die-stacked system, each stack of DRAM is considered a module. The *memory channel* is how many memory modules can be connected to a single compute chip.

$$\text{mem modules} = \frac{\text{db size}}{\text{module capacity}} \quad (6.1)$$

$$\text{compute chips} = \left\lceil \frac{\text{mem modules}}{\text{mem channels}} \times \frac{1}{\text{channel modules}} \right\rceil \quad (6.2)$$

$$\text{chip bandwidth} = \text{mem channels} \times \text{channel bandwidth} \quad (6.3)$$

$$\text{chip perf} = \min\{\text{core perf} \times \text{chip cores}, \text{chip bandwidth}\} \quad (6.4)$$

$$\text{chip cores} = \left\lceil \frac{\text{chip perf}}{\text{core perf}} \right\rceil \quad (6.5)$$

$$\text{mem power} = \text{mem modules} \times \text{module power} \quad (6.6)$$

$$\text{compute power} = \text{chip cores} \times \text{core power} \times \text{compute chips} \quad (6.7)$$

$$\text{blades} = \left\lceil \frac{\text{compute chips}}{\text{blade chips}} \right\rceil \quad (6.8)$$

$$\text{response time} = \frac{\text{percent accessed} \times \text{db size}}{\text{chip perf} \times \text{compute chips}} \quad (6.9)$$

$$\text{power} = \text{mem power} + \text{compute power} + \text{blades} \times \text{blade power} \quad (6.10)$$

There are three inputs to our model that are workload dependent. The *core perf* which represents the rate the core can process data, the database size (*db size*) which represents the required memory capacity, and the *percent accessed* which we use as a proxy for query complexity. The percent accessed is the amount of data that a single query or operation over the data touches. The performance estimation is based on the performance of the scan operation, but other database operations in analytic databases can be performed with similar algorithms as the scan operator [39].

To investigate the tradeoffs between the three systems, we chose a workload size of 16 TB to represent a large analytic database and a complexity of 20% of that data, or 3.2 TB as there are some analytic queries that access 20% of the entire database corpus [147]. Although it is unlikely that a single operation will touch 3.2 TB of data, each query is made up of many operations, that together could touch many of the database columns. Additionally, in the future, the data size will be increasing along with the query complexity.

In Section 6.5 we evaluate the systems with our model under three different conditions, constant response time, constant power, and constant memory capacity. Equations 6.1–6.10 show our model for a constant memory capacity. We modify this slightly when holding other characteristics constant. For constant response time, we assume an increased number of sockets to support the required bandwidth. For constant power, we first assume each

	Traditional server	Big-memory server	Future die-stacked
module capacity	32 GB	512 GB	8 GB
channel bandwidth	25.6 GB/s	48 GB/s	256 GB/s
memory channels	4	4	1
channel modules	2 ¹	1	1
module power	8 W	100 W	10 W
blade chips	4	1	9

core perf	core power	chip cores	db size	percent accessed
6 GB/s	3 W	32	16 TB	20%

Table 6.1: Inputs for the analytical model.

blade is fully populated, then compute the total blades that can be deployed given the power budget.

The model we use is released online as an IPython notebook at https://research.cs.wisc.edu/multifacet/bpoe16_3d_bandwidth_model/.

6.5 Results

In this section, we investigate the tradeoffs between performance and cost in designing a real-time big data cluster using the model described in Section 6.4. We consider the three designs discussed in Section 6.3. Our performance metric is the response time (i.e., the latency to complete the operation). We chose latency instead of the throughput for two reasons. First, in big data workloads, the latency to complete operations is increasingly important with many applications requiring “real-time” or “interactive” results. Second, a lower latency design implies the design has higher throughput. With no parallelism between queries, throughput is equivalent to the inverse of latency. However, higher throughput does not always imply lower latency due to parallelism.

Cost is more difficult to quantify than performance. Rather than try to distill cost into a

¹The traditional server can have up to six DRAM DIMMs per channel, but must have at least two to take advantage of the full DDR bandwidth. Therefore, we use two as the modules per channel to maximize the memory bandwidth-capacity ratio.

	Traditional	Big-memory	Die-stacked
Number of blades	800	1700	228
Number of chips	3200	1700	1700
Cluster bandwidth	320 TB/s	320 TB/s	384 TB/s

Table 6.2: Requirements of a cluster of each system architecture given a 10 ms SLA.

single metric, we look at two important factors in the cost of building a large scale cluster. We explore the power consumed for the cluster. Often, power provisioning is one limiting factor in datacenter design [12]. Additionally, energy is a significant component of the total cost of ownership of a large-scale cluster. Assuming the cluster is active most of the time, power is a proxy for this energy cost.

The second cost factor we study is the total memory capacity of the cluster. Memory is a significant cost when building big data clusters. For instance, memory is over 70% of the cost of a Dell PowerEdge R930, if it is configured with the maximum memory [35].

We present the results of our model under three different constraints by holding different cost-performance factors constant.

- *Performance provisioning*—Constrained response time (SLA): This represents designing a system to meet a performance requirement.
- *Power provisioning*—Constrained power: This represents designing a system to fit a power envelope in a datacenter.
- *Data capacity provisioning*—Constrained DRAM capacity: This represents designing a system around the data size of a workload.

Performance provisioning

When designing a cluster to support an interactive big data workload, response time (or SLA, e.g., 99.9% of queries must complete in 10 ms) is one possible constraining factor. The SLA time includes many operations (e.g., multiple database queries, webpage rendering,

and ad serving); we focus on only one part of the entire response time: one query to an analytic database. We look at three different SLA response times, 10 ms, 100 ms, and one second. We present this spectrum of SLA response times since “real-time” may mean different response times to different providers.

Similar to over provisioning disks to increase disk bandwidth and I/O operations per second, for the traditional and big-memory systems, we must over provision the total amount of main memory to increase the memory bandwidth. To meet a certain SLA, the aggregate bandwidth of the cluster must be greater than the SLA response time divided by the amount of data accessed. For instance, assuming a 10 ms SLA and a 3.2 TB working set, each system must support 320 TB/s of aggregate bandwidth. For a cluster based on the traditional system to support 320 TB/s, it needs 800 blades and 3200 compute chips. Table 6.2 shows the requirements for all three systems given a 10 ms SLA.

Figure 6.4 shows the power (Equation 6.10) and memory capacity (based on the required bandwidth to meet the SLA) for the clusters given a 10 ms SLA (top row), 100 ms SLA (second row), and one second SLA (bottom row). This figure shows that if high performance (e.g., a 10 ms SLA) is required, the die-stacked system can provide significant benefits. With a 10 ms SLA the die-stacked architecture uses almost $5\times$ less power, and does not need to be over provisioned at all.

The traditional and big-memory systems have significantly over provisioned memory capacity to be able to complete the workload in 10 ms. This over provisioning means the traditional and big-memory platforms will be very expensive for this workload. These platforms are over provisioned by a factor of $50\times$ and $213\times$, respectively. These systems have low bandwidth compared to their memory capacity, and thus, require high memory capacity when performance is important. However, the die-stacked system is the opposite: it has low capacity and high bandwidth, which is the most cost-effective system architecture when high performance is required.

Figure 6.4 also shows the power and capacity for each system when the SLA is 100 ms

and one second (second and third rows). When the performance requirement is relaxed, the traditional and big-memory servers do not need to be as over provisioned, saving both power and memory cost. In both of these cases, the die-stacked system uses about the same or more power than the current systems. The die-stacked system uses more power because it requires more compute chips, more blades, and more memory modules to have the same capacity as the traditional and big-memory systems. This increased number of chips puts the die-stacked system at a disadvantage when the SLA is less strict.

At an SLA of about 60 ms the power for the traditional and die-stacked systems is the same. At this point, the only benefit to the die-stacked system is that it is less over provisioned, and could reduce the cost of extra DRAM, assuming die-stacked DRAM is the same cost as DDR4 DRAM. As the complexity of the query increases (i.e., more data is accessed on each query) the crossover point moves to higher SLAs. For instance, if the query touched 50% of the data the crossover point is about 170 ms. Also, as the density of the memory increases, the crossover point increases. For instance, if the main memory was $8\times$ denser (e.g., using 64 Gb DRAM chips), the crossover point is about 800 ms. Crossover points at higher SLAs imply the die-stacked system is cost-effective in more situations.

Power provisioning

Provisioning a datacenter for power distribution and removing the resulting heat is a large portion of the fixed cost of a datacenter [12]. To reduce the capitol investment overheads, it is important for the datacenter to use the amount of power provisioned for it. Any provisioned power that is not used is wasted fixed cost. Therefore, we investigate the tradeoffs given a fixed power budget. We choose to hold the power of each system constant and one megawatt, 100 kW, and 50 kW. As a point of reference, one megawatt is the power of a small datacenter [12]. We assume each system is used in a cluster that exactly meets the power budget (e.g., there are over 1300 traditional blades given 1 MW).

Figure 6.5 shows the performance (Equation 6.9) and memory capacity (based on the

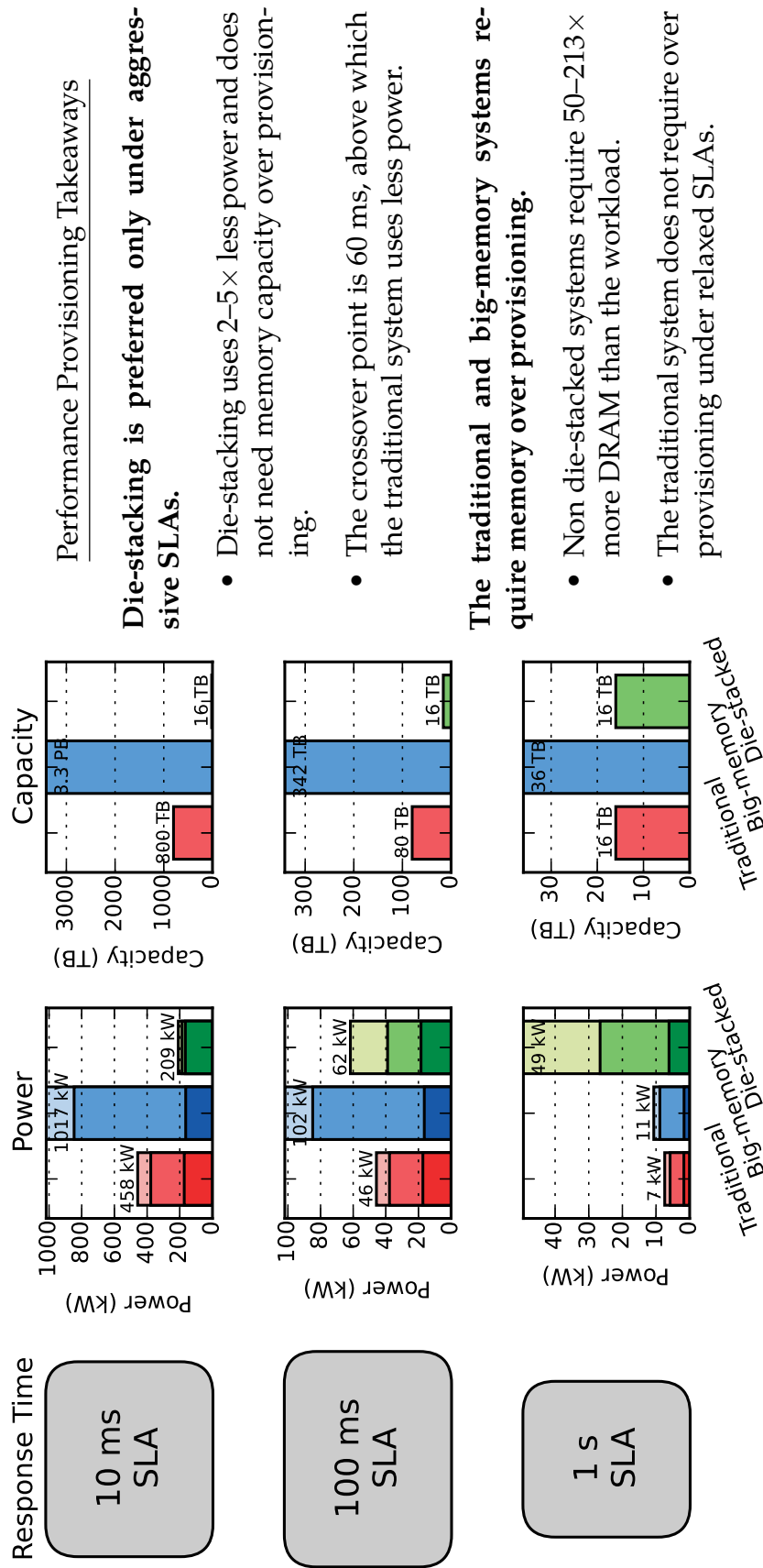


Figure 6.4: Power and memory capacity of a cluster based on traditional, big-memory, and die-stacked servers with constant response times of either 10 ms, 100 ms, or one second (1000 ms). The power for each is broken down into overhead power (light, top), memory power (middle), and compute power (bottom, dark). Memory capacity must be over provisioned (16 TB required database size) to achieve the required bandwidth to meet the SLA for the traditional and big-memory systems.

maximum number of blades for the given power budget) of each system architecture when provisioned to use 1 MW of power (top row), 100 kW (middle row), and 50 kW (bottom row). At 1 MW each configuration can meet aggressive real-time SLA constraints of 10 ms. However, the die-stacked architecture has a $5\times$ higher performance than the traditional and big-memory systems. This decrease in response time translates into increased throughput in a real-world system.

The total memory capacity of each system given 1 MW of power is shown on the right side of Figure 6.5. Similar to Figure 6.4, the traditional and big-memory systems can support much higher capacity than the die-stacked system. However, this increased capacity comes with an increased cost.

When the power limit is strict (e.g., 50 kW), the response time for the die-stacked system drops below the response time of the traditional and big-memory systems. The reason for this behavior is that the die-stacked memory has a higher power per capacity (watts/bytes) due to its increased memory interface width and higher bandwidth. Thus, when the power is strictly constrained, the die-stacked system is limited in the amount of compute power it can use. In fact, for the 50 kW configuration, the die-stacked system only has enough power to use one core per compute chip.

Data capacity provisioning

Figure 6.6 shows the estimated performance (Equation 6.9) and power (Equation 6.10) using our model for three workload sizes: 160 TB (top row), 32 TB (middle row), and 16 TB (bottom row). So far, all of the results has assumed a 16 TB workload size. This data assumes that the workload only accesses 3.2 TB per query for each workload size. Thus, for the larger workload sizes, the queries touch a smaller percentage of the total data. If the complexity changed at the same rate as the capacity, all of the response times would be constant between different sizes.

This figure shows in all cases that the die-stacked system can perform a query $256\times$ faster

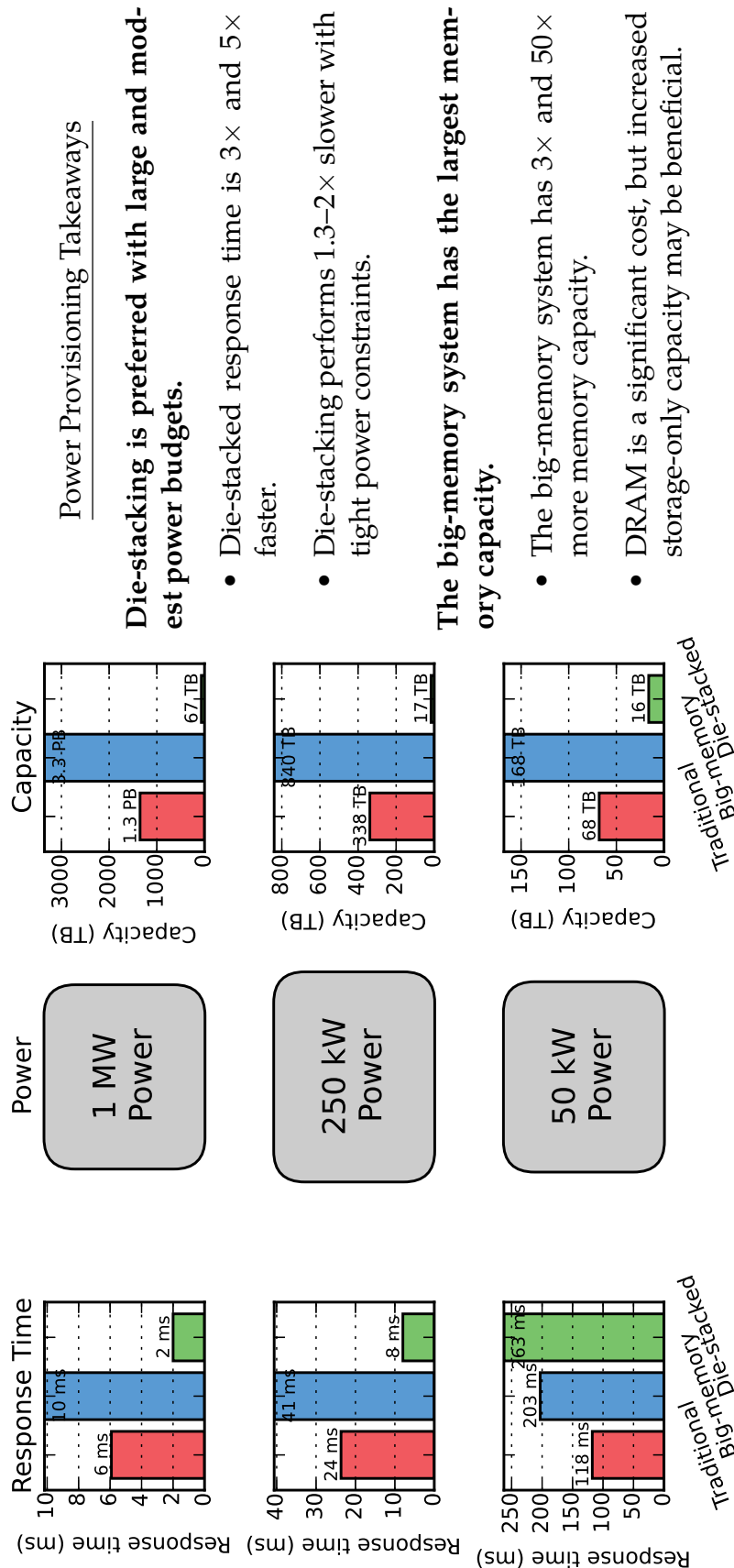


Figure 6.5: Response time and memory capacity of a cluster based on traditional, big-memory, and die-stacked servers with constant power envelope of either 1000 kW (1 MW), 250 kW, or 50 kW. Memory capacity is over provisioned if the power budget allows and capacity and performance are constrained by the allowed power. Each cluster is configured with multiple blades to exactly meet the power budget.

than a big-memory machine and $60\times$ faster than a traditional server. The die-stacked system improves performance for two reasons. First, its aggregate bandwidth is much higher. For instance, a die-stacked system with 16 TB has an aggregate bandwidth across all stacks of 512 TB/s. The traditional server and big-memory servers have an aggregate bandwidth of 6.4 TB/s and 1.5 TB/s, respectively. Second, to handle this increased bandwidth, the die-stacked system has many more compute chips, which in turn increases power (center column). In fact, the die-stacked system uses $26\text{--}50\times$ more power than the traditional and big-memory systems. This increased power usage significantly increases the cost of deploying a die-stacked cluster.

Interestingly, even though the die-stacked system has $340\times$ more bandwidth than the big-memory system, it is only $256\times$ faster. This discrepancy is because there is not enough compute resources to keep up with the available bandwidth in the die-stacked system. As compute chips become more capable or the density of main memory increases, the die-stacked system will increase the gap between other systems.

Figure 6.7a shows the energy consumption of each system configured with a 16 TB database. Even though the die-stacked system's power consumption is large, the die-stacked system is more energy efficient than either the traditional or big-memory system, using about $5\times$ less energy. The die-stacked system does not need to use high-power chip-to-board interconnects or extra buffer-on-board chips. The increased energy efficiency also comes from the fact that the die-stacked system "races to halt". All of the extra energy from shared resources like the power supply, caches, disks, etc. is minimized since the die-stacked system completes the operation more quickly. The relative energy efficiency of each system does not depend on the performance, power, or capacity provisioning.

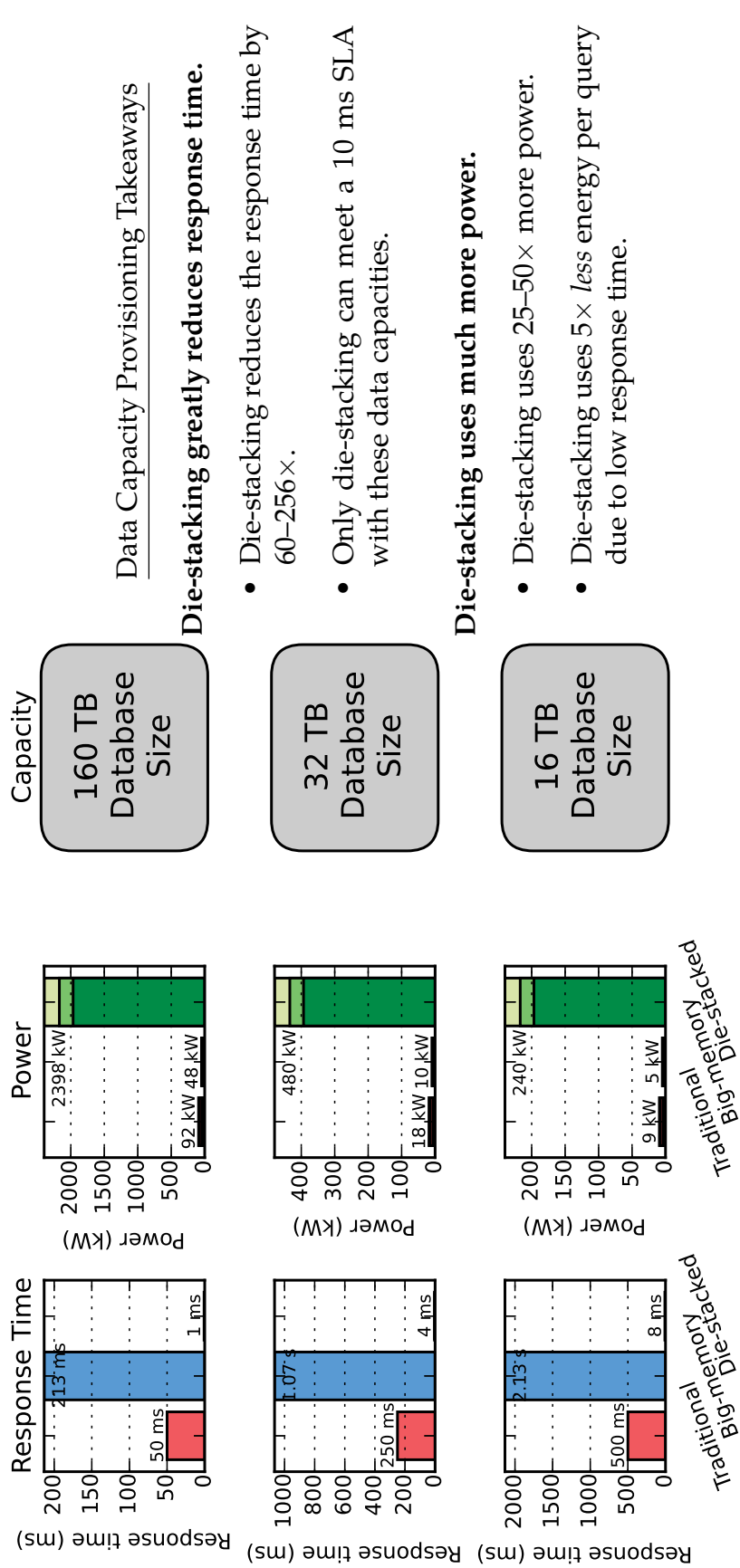
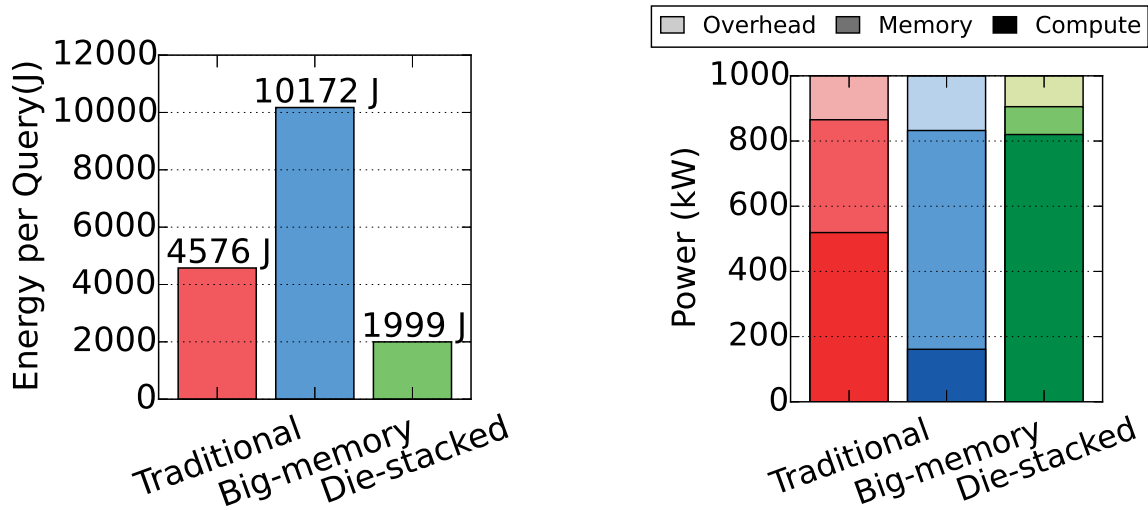


Figure 6.6: Response time and power of a cluster based on traditional, big-memory, and die-stacked servers with constant memory capacity of either 160 TB (10× data size), 32 TB (2× data size), or 16 TB (1× data size).



(a) Energy for each system assuming each cluster is configured for a 16 TB database size (Figure 6.6 bottom row).

(b) Breakdown of the components of power between compute power, memory power, and overhead power for a cluster with one megawatt of power.

Figure 6.7: Energy and power for each system.

6.6 Discussion

In this section, we discuss the implications of the model results on designing a big data machine. We first discuss the important areas for computer architects to focus to further increase the benefits of the future die-stacked architecture over the current server designs. We then discuss some of the deficiencies of our analytical model.

Improving system efficiency

Figure 6.7b shows the breakdown of power between the compute, memory, and overhead for each system as a percentage of the whole system's power assuming a 1 MW power provisioned cluster. The compute power is computed from Equation 6.7, the memory power is computed from Equation 6.6, and the overhead power assumes each blade (Equation 6.8) uses an extra 100 W for peripheral components. This figure shows that the traditional and big-memory servers' power is dominated by the memory chips. However, the die-stacked server power is dominated by the compute chips. None of the systems have a large portion

of the power from the blade overhead component.

Figure 6.7b implies that architects should focus on reducing the compute energy in future die-stacked systems. If the compute portion of power of the die-stacked system was reduced by $10\times$:

- The die-stacked system would use less power when meeting high performance SLAs, but it does not fundamentally change the SLA tradeoffs.
- The capacity of the die-stacked system increases significantly when you have a fixed power budget of more than 100 kW.
- The die-stacked system has reduced power in the case of fixed data capacity, but it does not fundamentally change the response time tradeoffs.
- The die-stacked system would have increased energy efficiency.

Alternatively, architects could concentrate on increasing the main memory density. One promising technology that increases density over DRAM is non-volatile memories [141]. Increasing the density of main memory does not directly affect the performance in any system architecture. However, increased density does reduce the required power (especially for the die-stacked system) as fewer chips and fewer blades are required to meet the capacity requirement. Increasing the density by $8\times$ affects our results in the following ways:

- The die-stacked architecture becomes cost-effective at higher SLAs, but it is still less cost-effective than the traditional system with a one second SLA.
- With fixed power provisioning, increasing the density does not change the performance, but does increase the capacity for all systems.
- For a fixed data capacity, increasing the density decreases the power required for all systems and makes the performance worse for the traditional and big-memory systems due to the lower bandwidth-capacity ratio.

Model deficiencies

In this work, we use a high-level model of the performance and power of a big data cluster. There are many important factors that we ignore. First, we only focus on one potential big data workload. However, we expect that our results generalize to any bandwidth-bound workload. Second, the inputs we use for our model significantly affect the results. We use data from manufacturers and datasheets. However, it is likely these numbers vary between providers and will change over time. For these reasons, we have released our model as an interactive online application for readers to choose the numbers which are best for their systems. See https://research.cs.wisc.edu/multifacet/bpoe16_3d_bandwidth_model/.

Also, we do not know whether the compute energy is due to data movement or actual computation. Since we use real hardware to measure the energy of a computation, we cannot break down the compute energy into its components. More deeply understanding the power and energy implications in die-stacked systems is interesting future work.

In this work, we assume that each processor only accesses its local memory and ignored the time required for communication between processors in the cluster. Even with regular data-parallel workloads like those we evaluated, there will be some amount of communication between processors. Largely, these overheads will be the same for each of the systems we evaluated. However, it is possible that since the die-stacked system has less data per processor that the communication overheads will be larger than for systems with more data per processor. Our model does not consider the communication between processors.

Finally, our model only applies to workloads that are limited by memory bandwidth. We defined workload complexity in terms of the data required for the computation. However, another common measure of complexity is “arithmetic intensity”: the ratio of compute operations (e.g., FLOPS) to memory operations. Our model does not directly take this characteristic into consideration. Some big data workloads do few compute operations per byte of data, which causes these workloads to be memory-bandwidth bound. Additionally,

as more big data applications take advantage of vector extensions like SIMD their arithmetic efficiency (bytes per cycle) will be higher putting more pressure on the memory bandwidth.

6.7 Conclusions

We investigated how the system architecture affects the power and memory capacity cost-performance tradeoff for three possible designs, a traditional server, a big-memory server, and a novel die-stacked server for one big data workload. We find the traditional server and the big-memory server are not architected for low-latency big data workloads. These current systems have a poor ratio of memory bandwidth to memory capacity. To increase the performance of big data workloads, these systems need a larger bandwidth-capacity ratio, increasing the memory bandwidth without increasing the memory capacity.

On the other hand, the die-stacked system has a bandwidth-capacity ratio that is too large to be cost-effective in many situations. The die-stacked system provides greatly increased bandwidth, but to the detriment of memory capacity. In fact, we require over 2000 stacks to meet the same capacity as eight big-memory systems. Because the die-stacked system requires so many stacks to reach a high capacity, the power requirements may be a cost-limiting factor.

We find the die-stacked architecture of today is cost-effective with complex queries that touch a significant fraction of the total data and there is a tight (10 ms) SLA. Looking forward, architects should focus on reducing the compute power in these die-stacked systems or increasing the memory density. These changes will allow the die-stacked architecture to be cost-effective in more situations.

6.8 Artifacts

The model used in this chapter is available at https://research.cs.wisc.edu/multifacet/bpoe16_3d_bandwidth_model/. This link provides details on the model and the code to

generate all of the graphs in Section 6.5.

— 7 —

ADAPTIVE VICTIM DRAM CACHE: OPTIMIZING BANDWIDTH FOR HPC WORKLOADS

7.1 Introduction

Similar to other workloads discussed in this dissertation, high performance computing (HPC) applications often have high memory bandwidth demands, especially when executed on highly-parallel hardware like GPUs or many-core processors. As discussed in Chapter 6, to meet this bandwidth demand, system designers are including high-bandwidth memory (HBM) on package through 3D die-stacking (e.g., Intel’s Knights Landing [140] and NVIDIA’s Tesla P100 [103]). *The main benefit of this memory comes from increased bandwidth, not reduced access latency, since it uses the same underlying DRAM technology.*

Chapter 6 showed that using solely HBM as main-memory has significant power costs. A straightforward way to combine a limited capacity HBM with DRAM DIMMs is as another level of cache between the on-chip SRAM last-level cache (LLC) and main memory [30, 55, 62, 87, 93, 121] since caches are transparent to programmers.

There are two primary differences between SRAM and DRAM caches that influence

DRAM cache design. First, SRAM caches typically separate the tag and state into a different structure from the data [151], but DRAM caches use the same array structure to hold the tags, state, and data. Second, each level of SRAM cache typically has lower latency, but DRAM caches have comparable latency to main memory.

To support high bandwidth, DRAM caches must have much more parallelism than SRAM caches, and interference between the metadata accesses (tag and cache block state) and the data accesses can have a significant performance impact. Many works have shown that interference between DRAM requests hurts performance for main memory [60, 97, 125, 142] and this is exacerbated by auxiliary accesses (e.g., tag, dirty bit reads) in DRAM caches.

We introduce a new metric to quantify this interference: *access amplification*. We define access amplification to be the total memory transactions divided by the number of demand memory requests. Access amplification is similar in spirit to write amplification in NAND-flash disks [54]. The impact of DRAM cache metadata accesses was initially explored by Chou et al. [31] which showed there is bandwidth waste in some DRAM cache designs. Access amplification differs from Chou et al.’s “bandwidth bloat” in two ways. First, access amplification counts memory transactions instead of raw bytes since using ECC to store cache tags requires reading and writing the entire cache block for all accesses. Second, access amplification considers all read and write requests from the LLC “useful”, not just read hits. We show that access amplification is a better predictor of performance than bandwidth bloat, especially for workloads with high DRAM cache miss rates.

We use access amplification, the NAS parallel benchmarks, and detailed gem5 simulations [19] to analyze alternative DRAM cache designs. The NAS parallel benchmarks (NPB) are widely used and are considered to be representative of an important class of HPC workloads [1, 9]. For these workloads, DRAM cache miss rates vary from 0% to nearly 100% and write to read ratios vary from 0% writes to 100% writes, sometimes within different phases of a single application. Due to their varying phase behavior and large working

set sizes, NPB workloads put different demands on a DRAM cache than general-purpose workloads (e.g., SPEC).

Our analysis starts with the DRAM cache of a commercially available many-core system. Then, we discuss how victim caches can reduce access amplification and potentially improve performance. However, for some workloads, reducing access amplification to the DRAM cache results in excessive traffic to the off-package main memory DRAM. Thus, we introduce an adaptive victim cache design that dynamically balances these competing factors.

KNL-like mostly-inclusive cache

Intel's Knights Landing (KNL) [140] is a commercially-available multicore processor that targets HPC workloads. According to published literature, KNL implements a DRAM cache with a mostly-inclusive policy (i.e., DRAM cache evictions do not invalidate the LLC or higher levels of the cache hierarchy). We call our model of this DRAM cache *KNL-like*, since we had to assume certain details not included in the public literature.

Our model implements two demand cache accesses: cache reads that result from LLC-misses and cache writes from LLC-writebacks. HBM accesses occur as follows.

Cache read hits cause one DRAM cache access: ① read the tag & data. The KNL design leverages Qureshi and Loh's Alloy cache which stores the tag and data together in a direct-mapped cache eliminating excess accesses [121].

Cache read misses cause three DRAM cache accesses: ① read the tag & data to check the tag and retrieve dirty data on a miss, ② write the cache tag to mark the frame as busy (KNL uses a "busy" state instead of MSHRs since 1000s of misses may be outstanding [139]), and ③ write the new data once it returns from main memory.

Cache writes cause two DRAM cache accesses: ① read the tag & data to check if the cache frame holds dirty data (from the victim block) that must be written back to memory and ② write the new data.

This analysis shows that both cache read misses and cache writes result in excess

accesses increasing the access amplification above 1.0. Simulation results show that, on average, access amplification for the NPB workloads is 2.03, or two DRAM cache accesses for each demand request (Section 7.5).

Dirty victim cache

Changing the cache policy from mostly-inclusive to a *dirty victim cache* reduces access amplification by eliminating the two excess accesses on cache read misses.

Cache read hits are the same as KNL-like.

Cache read misses cause a single DRAM cache access: ① read the tag & data to check the tag. There is no need to reserve a location in the cache since the fill is delayed until LLC writeback.

Cache writes are handled the same as KNL-like. However, the LLC must also writeback clean blocks to the DRAM cache to implement the victim cache policy.

For the NPB workloads, we find that this dirty victim cache design reduces access amplification to 1.75, leading to performance improvements over the KNL cache design by 6%, on average (Section 7.5). We leverage a clean-evict bit in the LLC [71] to write clean data at most once.

Clean victim cache

A victim cache policy eliminates excess accesses on cache read misses, but cache writes must still check whether a cache frame holds dirty data that must be written back to main memory. These excess accesses can be eliminated by implementing a clean victim cache policy, where the contents of the DRAM cache are always consistent with the main memory.

Cache read hits are the same as KNL-like cache.

Cache read misses are the same as the dirty victim cache.

Cache writes cause a single DRAM cache access: ① write the tag & data. In a clean victim cache, it is always safe to overwrite the cache frame with new tag and data. Like the

dirty victim cache, the LLC must writeback clean data to a clean victim cache.

For the NPB workloads, we find that the clean victim design reduces access amplification to 1.24, outperforming the dirty victim cache for workloads with relatively low write traffic (Section 7.5). However, because a clean victim cache cannot store dirty data, all LLC-writebacks of modified data must update main memory. For write-heavy workloads, the limited bandwidth of off-package DRAM becomes a bottleneck, degrading performance toward a write-through cache despite the very low access amplification. For these workloads, a dirty victim cache performs better.

Adaptive victim cache

To address these differing workload properties, we propose an adaptive victim cache design that seeks to balance the performance characteristics of both the dirty victim cache and the clean victim cache designs. By default, this policy behaves like a clean victim cache, but falls back to a dirty victim cache policy for workload phases with high write intensity.

Our adaptive victim cache design uses on-chip SRAM to track metadata only about *dirty* cache frames, achieving most of the performance of full SRAM tags with less than 3% of the area. We use the following mechanisms.

1. **Laundry counts** track the number of dirty blocks within a cache region. Cache writes to clean regions (i.e., laundry count = 0) do not need to read the tag & data.
2. **Laundry list** track tags of regions with dirty blocks, allowing writes with laundry list tag matches to proceed without first reading the tag & data. This optimizes for a common case in HPC workloads that data is written repeatedly.
3. **Proactive writeback** to keep the cache mostly clean by cleaning the cache segments in the laundry counts when sufficient main memory bandwidth is available.

The laundry counts, laundry list, and proactive writeback allow our adaptive victim cache to behave the same as a clean victim cache when the write traffic is low, and like a dirty

victim cache under high write traffic workloads dynamically adapting to the workload or the workload phase.

Our proposed adaptive victim cache reduces access amplification from 2.03 in the KNL-like design to 1.35 (Section 7.5). This is within 7% of an unrealistic SRAM-tag design (1.26). Our design is decentralized, and each HBM channel operates fully independently, requiring 65 KB per HBM channel (total of 1040 KB) of on-chip SRAM (full SRAM-tags require 36 MB). Our adaptive victim design performs at least as well as either the dirty or clean victim designs for each workload. On average, our design increases performance by 10% over the KNL and is within 2% of SRAM tags.

7.2 DRAM cache design and access amplification

In our system, we assume there are many CPU clusters, each containing a CPU core, split L1 I/D caches, and a private L2 cache. There is a unified shared SRAM L3 LLC on-chip, split into multiple banks to support high-bandwidth.

Backing this LLC, is a memory-side DRAM cache. Since the DRAM cache is memory-side, it does not participate in the on-chip coherence traffic. All coherence requests are handled by the SRAM LLC. The storage for the DRAM cache is *multiple channels* of high-bandwidth memory (HBM). Addresses are striped across these channels using the lowest-order bits for the highest bandwidth [72]. We consider this HBM as on-package: 3D-stacked with TSVs [20, 61], 2.5D-stacked with a silicon interposer [5, 20], or attached via some other high-bandwidth interconnect [108].

There are two types of requests to the DRAM cache: *LLC-read* and *LLC-writeback*. LLC-read requests are issued to the DRAM cache when an LLC miss occurs, and fetches data from the DRAM cache. In the baseline mostly-inclusive KNL design, LLC-writeback requests are issued when the LLC evicts a modified block.

	HBM actions	Cache frame state		HBM actions	Main-memory actions	HBM actions
		Tag match	Frame dirty			
LLC-Read	Read tag & data	Hit	-	-	-	-
	Read tag & data	Miss	Clean	Write busy*	Fetch data	Fill*
	Read tag & data	Miss	Dirty	Write busy*	Writeback data, Fetch data	Fill*
LLC-Writeback	Read tag & data*	Hit	-	Fill	-	-
	Read tag & data*	Miss	Clean	Fill	-	-
	Read tag & data*	Miss	Dirty	Fill	Writeback data	-

Table 7.1: Details of the KNL-like DRAM cache operation. Non-demand requests are marked with an asterisk. The colors correspond to Figure 7.2.

Baseline KNL-like DRAM cache

Table 7.1 shows the detailed function of our baseline DRAM cache design based on public information available for the Knights Landing (KNL) DRAM cache [140]. We assume a latency optimized design like Alloy cache [121] that is direct-mapped and stores the tags with data in the DRAM rows. We store the tags with the ECC data in the DRAM. The JEDEC HBM specification has 16 bits per 128-bit bus for ECC, which allows enough space for 20 bits of tag and metadata and leaves 44 bits for ECC per cache block (e.g., using a (576,532) ECC code) [42, 61, 99].

We implement the “Garbage” (G) state used in KNL [140]. This transient state is used for cases where a DRAM cache line is allocated but the corresponding data is written later [139]. Thus, for every miss to the DRAM cache we must write the block to mark it busy. The KNL designers chose to store the outstanding request state *in the cache* instead of using MSHRs for two reasons. First, to support high bandwidth, 1000s of MSHRs are required, which is costly both in terms of area and power. Second, we find it is very rare for concurrent requests to access the same DRAM cache frame.

The KNL design is a “mostly-inclusive dirty” cache. The cache is allowed to store dirty data, and thus, must write it back to main memory upon eviction. One reason KNL does not use a fully inclusive memory-side design is it requires significant changes to the LLC

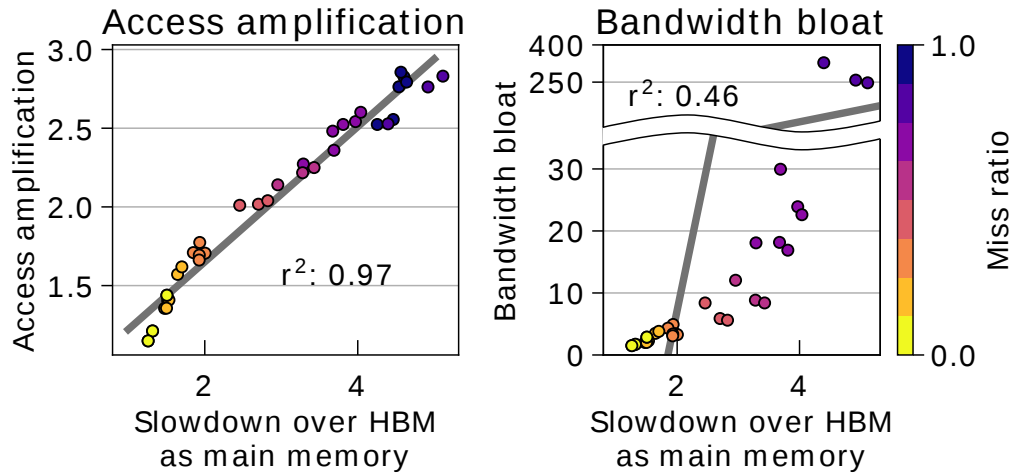


Figure 7.1: Comparison of measured access amplification to bandwidth bloat. Color represents varying miss rates. At a miss rate of 100% the bandwidth bloat is undefined.

and directory controllers to support back-probes on DRAM cache block invalidations. In the case of KNL’s distributed tag directory and distributed LLC [140], supporting back-probes would require adding many transient states to support all possible interleavings of simultaneous requests. Back-probes also increase the network traffic on the shared link between the LLC and the DRAM cache, consuming bandwidth that could be used for demand requests. Additionally, although invalidations may not be common, they can result in pathological behavior in a direct-mapped DRAM cache from repeated conflict misses that can further increasing network traffic and using LLC and directory access bandwidth.

Access amplification

Definition 1 (Access amplification). *The total number of DRAM cache accesses divided by the demand accesses initiated by the SRAM LLC misses.*

In Table 7.1, the actions marked with an asterisk are non-demand requests. For instance, on an LLC-read, the demand part of the access is the initial DRAM cache read to get the tag and data (no asterisk). The *write busy* and *fill* accesses are not servicing the original demand request, but performing auxiliary cache operations (asterisks). Table 7.2 compares the access amplification of the KNL-like cache with a nearly-ideal but infeasible-to-implement

SRAM tag design. Each row in the table shows the counts of each action (e.g., derived from Table 7.1). The demand accesses from the SRAM LCC are all of the requests which are issued to memory in the absence of a DRAM cache.

Access amplification builds off of the ideas of bandwidth bloat as pioneered by Chou et al. [31] and write amplification in flash memories [54]. Both access amplification and bandwidth bloat measure the wasted DRAM cache bandwidth, but bandwidth bloat is defined as the total bytes transferred divided by the useful bytes (i.e., bytes serviced from the DRAM cache). We believe *transactional accesses* are more indicative of the performance overhead than the raw bytes, similar to write amplification (actual number of flash page writes divided by user page writes). Because the cache tags and metadata are encoded in the extra bits provided for ECC, we must read and write the entire cache block on each access. Additionally, DRAM performance is affected more by interference from different accesses on shared resources (e.g., the shared data bus, bank row buffer) than the raw number of bytes read and written. This dichotomy is similar to using input/output operations per second (IOPs) instead of bandwidth to measure the performance of storage devices.

Figure 7.1 quantitatively compares these two metrics showing how the access amplification (left) and bandwidth bloat (right) correlate to the slowdown of the KNL DRAM cache design compared to directly accessing HBM as if it was main memory. We use a microbenchmark with varying DRAM cache miss ratios (color). The key quantitative difference is that bandwidth bloat significantly increases when the hit rate is low. In fact, if the hit rate is 0% (e.g., a streaming workload), the bandwidth bloat is infinite. Access amplification, on the other hand, shows a nearly linear relationship with execution time slowdown (execution time when using HBM as main memory divided by execution time when using HBM as a cache). Thus, we believe the transaction semantics of access amplification provide a better framework for comparing different DRAM cache policies. The goal of access amplification is not to predict performance, but to give designers a model to reason about the performance of different DRAM cache designs.

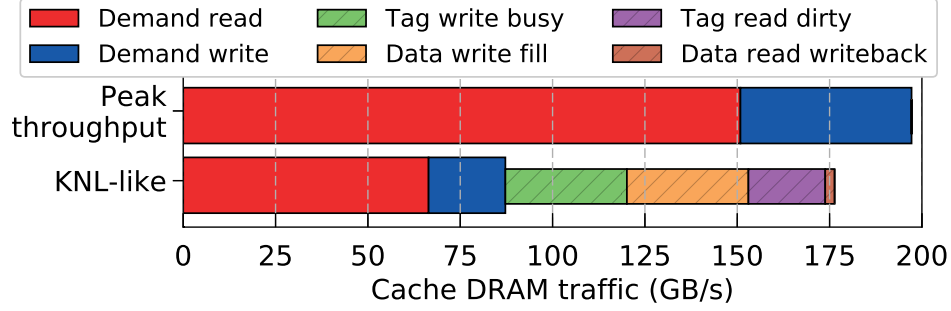


Figure 7.2: Breakdown of DRAM cache traffic between demand accesses and other overheads (hatched). This figure assumes a 50% miss ratio, 50% reads, and 50% dirty blocks. The results are similar with other miss and write ratios.

Design	DRAM cache accesses	MM accesses
Demand accesses	N/A	$R + W$
Unreal. SRAM Tags	$R(1 + m_d) + W(1 + m_d)$	$R(m + m_d) + Wm_d$
KNL-like	$R(1 + 2m) + 2W$	$R(m + m_d) + Wm_d$
Dirty victim	$R + 2W + W_u(1 + m)$	$R * m + m_d(W + W_u)$
Clean victim	$R + W + W_u$	$R * m + W$
Adapt. victim	$R + W(1 + m_d) + W_u(1 + m_d)$	$R * m + m_d(W + W_u)$

Table 7.2: Access amplification for each design. R is the reads. W is the writes. m is the miss rate of the DRAM cache. m_d is the rate of dirty data on each miss. And W_u is the unmodified writebacks. Access amplification is $\text{accesses}/(R + W)$.

Figure 7.2 shows the potential performance impact of access amplification. To illustrate the effects of access amplification, we modeled the HBM based on the high bandwidth memory standard [49, 61]. We used an artificial microbenchmark to send traffic to the HBM. Figure 7.2 shows that over 50% of the DRAM cache traffic is wasted (thin, hatched bars) not servicing demand requests (thick bars). Colors correspond to the actions in Figure 7.1.

7.3 Using the access amplification model

We previously introduced the access amplification of the KNL-like and victim cache designs. In this section, we specifically quantify the access amplification by considering the impacts of two different design parameters. First, the *insertion policy* dictates when to insert new data into the cache. The cache can either implement a mostly-inclusive policy where the

data is inserted at the time of the cache miss, or it can implement a victim cache policy where the data is not inserted until the LLC issues a write request. Second, the *writeback policy* dictates when to write dirty data back to the main memory backing the DRAM cache. The cache can either allow dirty data and writeback the data to main memory when it is evicted from the DRAM cache, or it can disallow dirty data and writeback data to main memory on every LLC-writeback request (i.e., a fully-clean writethrough cache).

Table 7.2 summarizes the access amplification of unrealistic SRAM tags, the KNL-like DRAM cache, and three victim cache designs. This table presents the access amplification in relation to the program characteristics including number of reads, writes, and misses at the DRAM cache.

KNL-like → Dirty victim

Using a victim cache design eliminates the extra DRAM cache write access marking the cache frame as busy while the miss is outstanding. Instead of allocating the cache frame at the time of the miss, a victim cache waits until the LLC evicts the block to allocate a frame. This removes $2m$ (m is the miss ratio) DRAM cache writes on each LLC-read (dirty victim row in Table 7.2).

The victim cache policy also eliminates an extra DRAM cache write access if a block is modified after a DRAM cache miss (part of $2m$ in Table 7.2). In the KNL cache, when an LLC-writeback occurs it overwrites the filled data, and the fill is “useless” as it is never read before it is overwritten.

However, a victim policy requires filling on every LLC eviction which significantly increases the number of LLC-writeback requests (W_u). A traditional victim cache policy [66] would maintain exclusion between the LLC and DRAM cache, requiring every LLC eviction to write back data to the DRAM cache. To reduce the bandwidth requirements, we enforce a non-inclusion property using a “clean-evict” bit [71]. The LLC maintains a per-block clean-evict bit, which is set when a block is fetched from main memory (and not on DRAM

cache hits). When the LLC evicts a block, only blocks with dirty data or a set clean-evict bit are written to the DRAM cache. The clean-evict bit differs from Chou, et al.'s DRAM cache presence (DCP) bit [31] because DCP must be maintained exactly (requiring complicated back probes) while clean-evict is essentially a performance hint. Using the clean-evict bit, $W_u \leq (R \times m)$.

Dirty victim $\rightarrow$ Clean victim cache

A clean cache, such as a conventional writethrough cache, does not need to check the tags before writing the tag and data. Therefore, we can reduce the access amplification of the dirty victim cache by treating it as a writethrough (clean) design.

The “Clean victim” row in Table 7.2 shows there are fewer DRAM cache accesses on LLC-writeback requests for the clean victim design than the other designs. The number of cache accesses reduces from more than $2W$ per access to just W for the clean victim design. This implies that the clean victim cache will perform better for high write traffic workloads. However, writethrough caches have a major drawback: all dirty data must be written to main memory, increasing the main memory traffic.

In Section 7.5 we find that implementing a fully-clean DRAM victim cache eliminates the most access amplification, even more than unrealistic SRAM tags. However, a clean victim cache hurts performance for some applications (Figure 7.8) since it increases the traffic to main memory, which is the bottleneck in write-heavy applications.

7.4 Adaptive victim cache

Ideally, we want an adaptive cache design that behaves like a fully-clean victim cache, without the downsides of squandering main memory bandwidth. Importantly, under a high-write load to the DRAM cache, we want the cache to behave like a writeback cache, not a writethrough cache.

The “Adaptive Victim” row in Table 7.2 shows the goal for our design. By leveraging the victim cache policy, we remove all unnecessary accesses on the LLC-read path, and by falling back on a writeback (dirty) policy, there is no main memory access amplification. In the best case, we only want to have extra reads to the DRAM cache if we are sure the data is dirty, which is the same overheads as the full SRAM tag design.

Our adaptive victim design has about the same access amplification as the unrealistic SRAM tag design. The key idea is that we can get most of the benefits of SRAM tags by only storing tags that reference *dirty data* in SRAM. To limit the overhead of storing these tags, under high write conditions we fall back on the dirty victim design, and to increase performance when the main memory traffic is low we upgrade to the clean victim design. Thus, our design robustly performs as well as either the dirty or clean victim caches, as shown in Section 7.5.

In this section, we explain five insights driving our adaptive victim cache design. We first present a simplified design of our adaptive victim policy. Then, we extend this design to large multi-channel DRAM caches. We then discuss the implementation of the adaptive victim policy, an important optimization, and the area overheads of our proposal.

Simplified adaptive victim design

Figure 7.3 shows a simplified example with a 4-entry DRAM cache to illustrate these insights. Step 0 shows the initial state of the cache with four valid and clean frames.

Insight 1: *If there are no dirty blocks in the cache, then it is safe to overwrite a block without first reading the cache tag.*

Action: Store a count of the number of dirty cache frames in SRAM. If the count is zero, it is safe to overwrite. Otherwise, we must check before writing.

Example: In step 1 of Figure 7.3, the LLC sends an writeback request to the DRAM cache with the clean-evict bit set. The laundry count of zero signifies there is no dirty data in the cache. Thus, we can treat the cache as though it is fully-clean and write the incoming

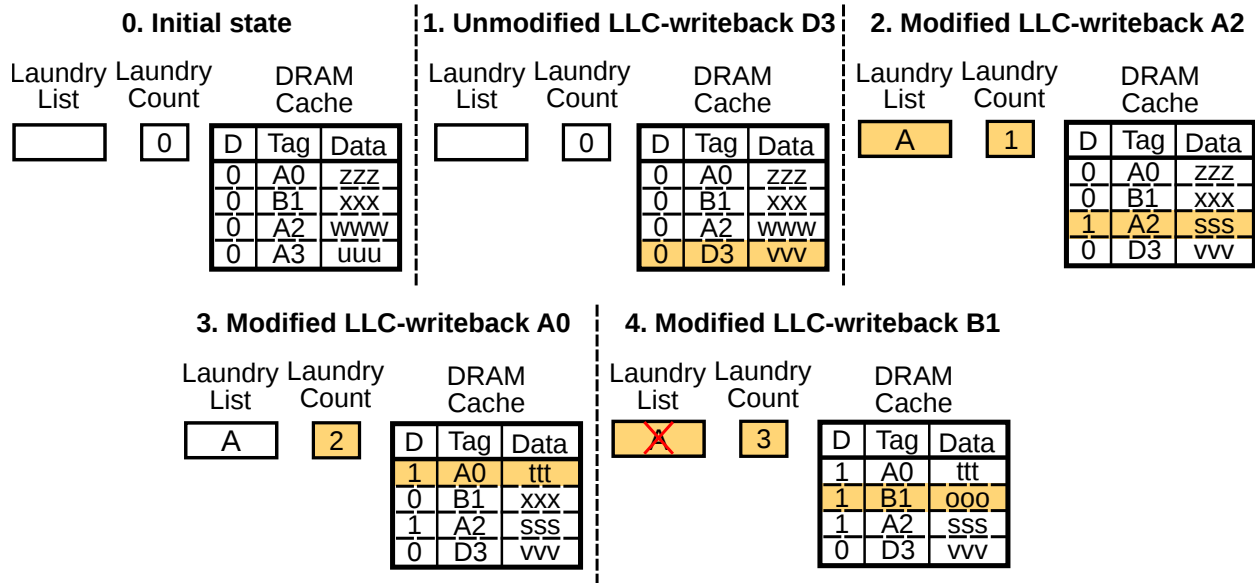


Figure 7.3: Example laundry count and laundry list usage. A simplified four entry cache is shown.

data without first reading the tag and data in the corresponding cache frame since it will only overwrite clean data.

In step 2 of Figure 7.3, the LLC sends a writeback request with modified data. This implies the data in main-memory is stale, and when the line is inserted into the cache, the cache will contain dirty data. Thus, when inserting the line, we increment the laundry count by one. This signifies it is no longer safe to write into the cache without first reading the corresponding frame's tag (like the dirty victim design) since it is possible a new line will overwrite the only copy of the data for a cache block.

Insight 2: *If we know all of the dirty blocks in the cache have the same tag as the current write, then we know it is safe to write the block without first checking.*

Action: Track the tag shared by all dirty blocks. If the tag matches, then it is safe to write. Otherwise, we must invalidate the tag and check before writing.

Example: In step 2 of Figure 7.3, when writing the dirty data for line A2, we also insert the tag (A) into the laundry list structure. A in the laundry list implies that only blocks with a valid tag of A can be dirty in the cache. Blocks with any other tag *must be* clean.

In step 3 of Figure 7.3, the LLC sends a modified writeback for A0. The laundry count is not zero; therefore, it may be unsafe to write this block into the cache. However, since the tag in the laundry list matches the tag of the writeback request, it is safe to overwrite the data in the cache. Either we will overwrite clean data that is found in main memory, or we will safely overwrite stale data with a more up-to-date version. In this step, we also increment the laundry count to match the number of dirty frames in the cache.

Finally, step 4 of Figure 7.3, shows a modified LLC-writeback to a block with a *different* tag (B). In this case, it is unsafe to write the data into the cache, so before writing the data, we must check the tag of the corresponding cache frame (exactly like a dirty victim cache). Additionally, when inserting the block in a dirty state, we must invalidate the laundry list entry for A since it is no longer safe to write blocks from A into the cache and increment the laundry count.

Applying insights 1 & 2 to large caches

Insights 1 and 2 do not directly scale to large cache sizes. These insights quickly break down since even a small percentage of dirty lines cause the cache to fall back on the dirty victim policy.

To extend these insights to large caches, we first define a cache super-frame as an aligned set of cache frames in a direct-mapped cache. Super-frames are similar to super-blocks; however, super-blocks are aligned regions of *memory*, and super-frames describe a region of cache storage. A super-frame can hold data from a single super-block or from multiple super-blocks since multiple super-blocks map to the same super-frame.

Insight 3: *We can partition the cache into super-frames and apply insights 1 and 2 on many small cache regions (super-frames) instead of the entire cache.*

Action: Each super-frame has a dirty count stored in the *laundry counts* and a tag stored in the *laundry list* (see Section 7.4). It is safe to write a block into the cache if the super-frame's laundry counts entry is zero *or* there is a tag match in the laundry list. Else, we

must check the cache before overwriting the cache frame.

Above, we described the adaptive victim design as if it was a centralized structure. However, to support high bandwidth, die-stacked DRAM is often split into many channels (e.g., 16 channels for the HBM standard [61]). These channels operate independently and may not be co-located on chip. Thus, we do not assume a centralized laundry list and laundry counts, but split the structures across the channels into multiple banks. Each bank of the laundry structures tracks the subset of the super-frame assigned to that channel (super-frames are continuous in physical space). Figure 7.4a shows the physical address bits used to access the laundry counts and laundry list structures. Banking the laundry structures across multiple channels increases their area overhead (e.g., the tags in the laundry list may be replicated), but it allows these structures to flexibly support high bandwidth.

Laundry counts and laundry list

The laundry counts is an SRAM-based structure that tracks the number of dirty frames in each cache super-frame (Figure 7.4b). The laundry counts is a directly indexed array with one entry per super-frame in the cache. The count is updated on each DRAM action by incrementing when adding dirty data and decrementing when flushing dirty data to main memory (Figure 7.5 contains more details).

We choose to track super-frames instead of every cache frame to decrease the area overhead of the laundry counts. Tracking dirty information using a bitvector for each 64-byte block requires 2 MB of on-chip SRAM area for a 1 GB DRAM cache. Section 7.5 shows that HPC workloads have high spatial locality in the DRAM cache so a bitvector will not significantly improve performance over coarser super-frame tracking.

We require the laundry count to be consistent with the dirty bits in the cache. Thus, we cannot increment the laundry count if the modified LLC-writeback overwrites already dirty data. To facilitate dirty block tracking, we add another bit to each LLC-entry. On a

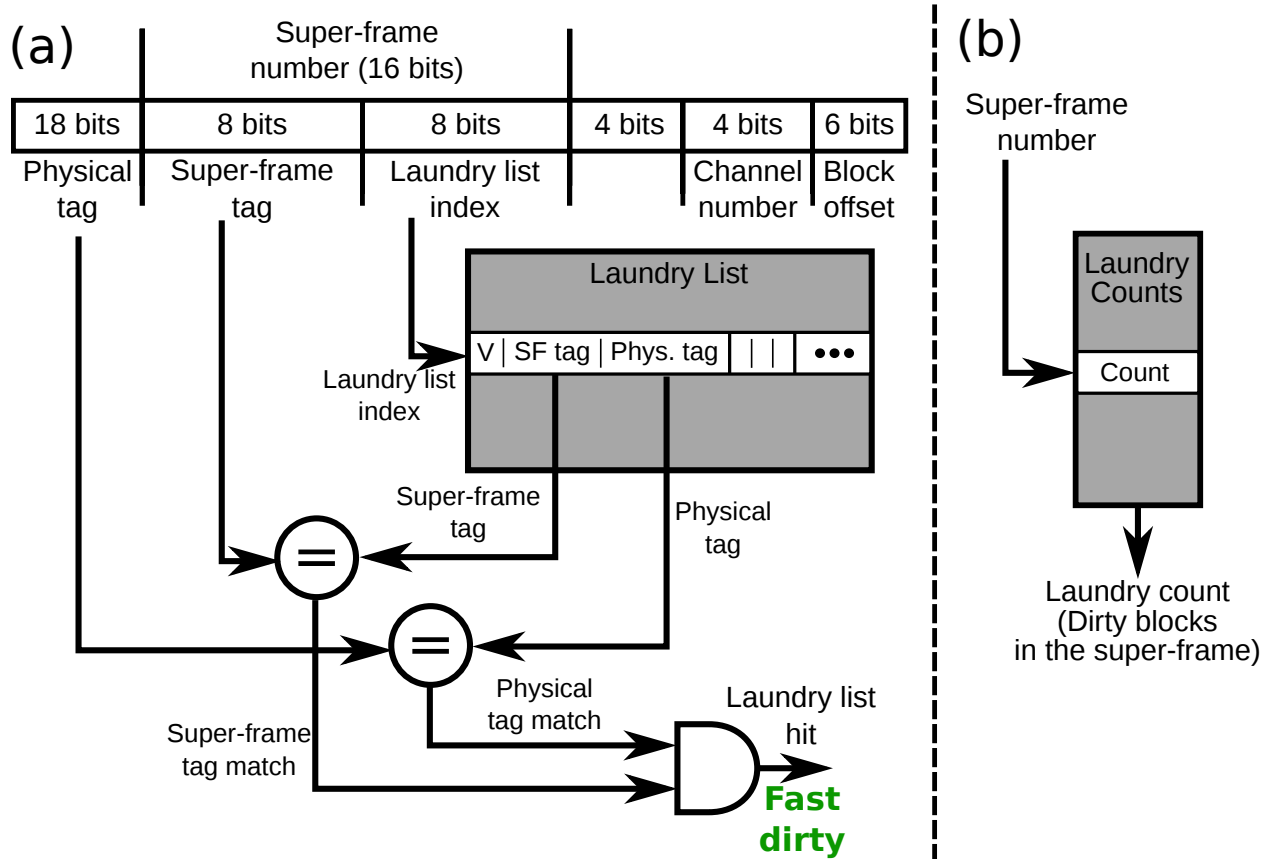


Figure 7.4: Operation of the laundry list (a) and laundry counts (b) structures.

response to an LLC-read hit at the DRAM cache, we include a cache-dirty bit which is set if the block is dirty in the DRAM cache. Like the clean-evict bit, this bit is tracked at the LLC and sent back to the DRAM cache on an LLC-eviction. If the cache-dirty bit is set, the line is already dirty in the cache, and we do not increment the laundry count.

Insight 4: *Insight 2 is not required for correctness. We do not need to track the tag for every super-frame in the cache.*

Action: Reduce the entries in the laundry list to only cover a subset of the super-frames in the DRAM cache and make the laundry list set-associative to reduce the conflicts between dirty super-frames.

Thus, the *laundry list* stores only *some* tags in SRAM to eliminate the DRAM cache reads on LLC-writeback in the case where the tag matches (Insight 2). Now, when checking if an LLC-writeback is safe, if the laundry counts entry is nonzero (i.e., the super-frame frame

contains dirty data), we check the laundry list. If the laundry list contains a tag for that super-frame *and* that tag matches, then the LLC-writeback is safe to insert into the cache. Section 7.5 shows it is common to overwrite dirty data in the cache and find a matching tag in the laundry list. Thus, we find the laundry list significantly decreases the extra reads compared to only tracking the laundry counts.

To reduce the area overhead, we implement the laundry list as a set-associative array with fewer entries than super-frames in the cache. Each laundry list entry holds the physical tag that is found in the cache, the super-frame tag, and a valid bit.

The laundry list is indexed by a subset of the super-frame number (Figure 7.4a). On a laundry list access, the valid super-frame tags from each way are compared with the incoming request's super-frame tag. If any of these super-frame tags match, we compare the physical tag of that entry to the request's physical tag. On a physical tag match, we take the "fast dirty" path in Figure 7.5. If there are no matching super-frame tags or the physical tags mismatch, we take the "slow dirty" path in Figure 7.5 since we are unsure whether the request is safe to write into the super-frame. In this case, the adaptive victim cache behaves like a dirty victim cache.

In Section 7.5, we show that a laundry list that covers only $1/8$ of the DRAM cache performs as well as a full laundry list for most workloads. The key reason a partial laundry list provides sufficient performance is that most super-frames are fully clean and contain no dirty data. Thus, tracking a tag for each super-frame is unnecessary.

The dirty region tracker (DiRT) proposed by Sim et al. [137] has a similar goal to the laundry list and laundry counts. However, DiRT *constrains* the number of dirty pages, falling back on a writethrough (clean) policy under high-write traffic, which can hurt performance (evaluated in Section 7.5). Our adaptive victim design is more robust and falls back on the higher-performance dirty victim design under high write-traffic.

The flowchart in Figure 7.5 shows the detailed DRAM cache state machine for LLC-writeback requests. The shaded rectangles represent accessing the DRAM cache. The

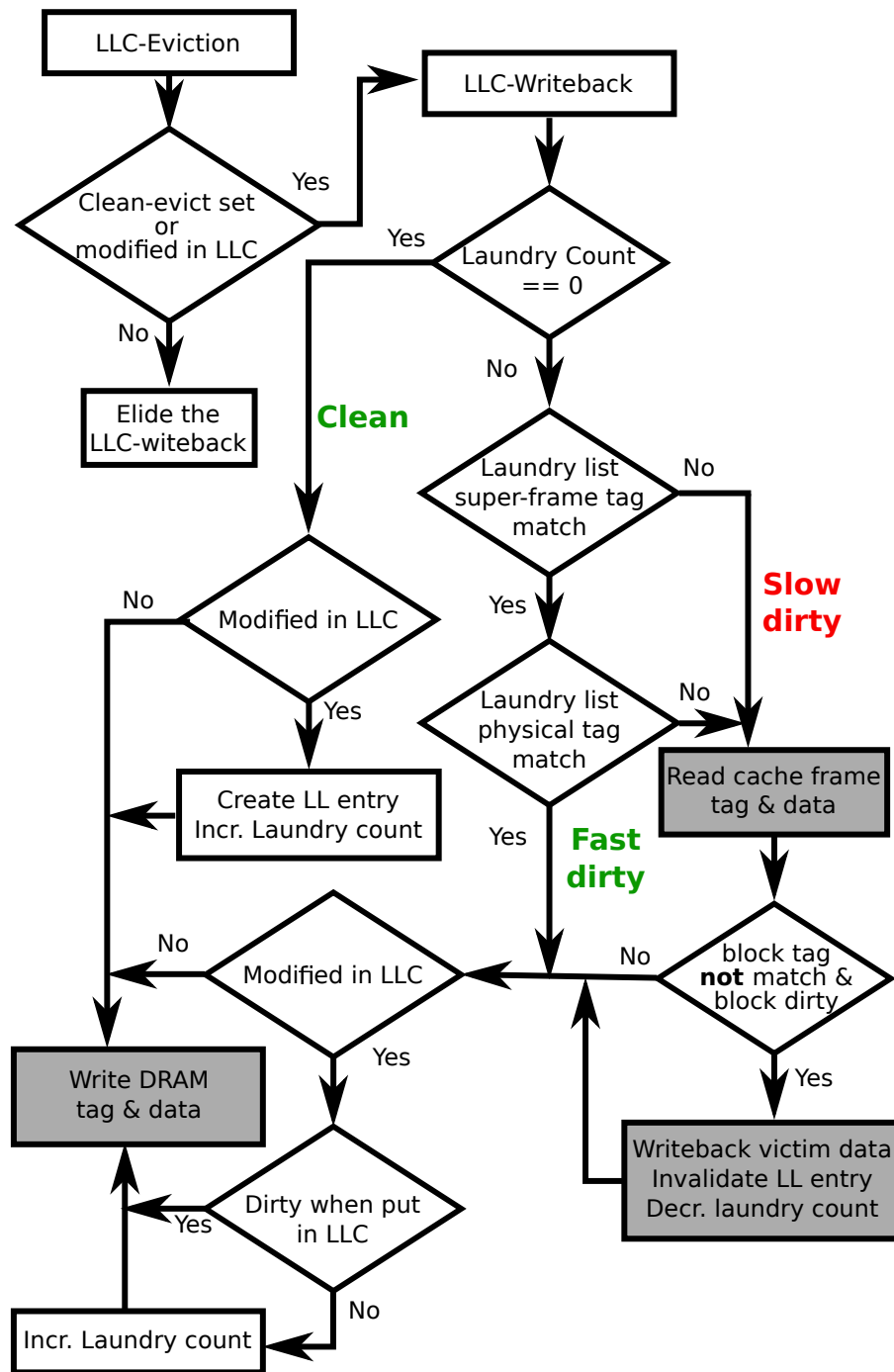


Figure 7.5: Overview of adaptive victim cache operation. The shaded rectangles represent accessing the DRAM cache. Corner cases and error cases are covered in detail in Table 7.3.

“clean” and “fast dirty” path allow the adaptive victim cache to behave like a fully-clean cache and an unrealistic SRAM tag cache, respectively. LLC-read requests are not shown; they simply read the tag and data from the DRAM cache.

Before the LLC-writeback request, the LLC checks the clean-evict bit and LLC’s dirty bit. If neither bit is set, the LLC-writeback is elided since the block is likely already cached and main memory or the DRAM cache contains the most up-to-date data.

Otherwise, the LLC sends an LLC-writeback request with the modified bit, the clean-evict bit, and the cache-dirty bit. On an LLC-writeback, we first check the laundry count for the super-block corresponding to the writeback request. If the count is zero, then it is safe to write the line into the super-frame and the super-frame behaves like a fully-clean cache (without requiring all dirty data to write through). Otherwise, we check the laundry list as shown in Figure 7.4. If there is a laundry list hit, then we take the “fast dirty” since the access is guaranteed to be safe.

The access *may* overwrite data that is stale in main memory if either there is no laundry list entry for the corresponding super-frame or the physical tag of the matching super-frame entry does not match. In this case, we fall back on the behavior of the dirty victim cache and read the tag and data in the DRAM cache. At this point, if the corresponding frame’s data is dirty and the tag does not match the request, we write the data back to main memory.

Finally, we write the data in the LLC-writeback request to the DRAM cache. If we used the “slow dirty” path, we use the canonical information in the DRAM cache and the LLC-writeback request type to update the laundry structures to be consistent with the DRAM cache. Otherwise, we update the laundry structures based on the type of LLC-writeback request since we know the state of the cache frame.

Proactive writeback

Insight 5: *If the cache is mostly clean, we get more benefit from our adaptive victim design.*

Action: Proactively writeback dirty data to main memory *only when it will not hurt performance*.

The adaptive victim DRAM cache will perform best when most of the cache is clean. However, maintaining a fully-clean victim cache hurts performance when the backing main memory bandwidth is saturated. Therefore, we use a proactive writeback design that adaptively cleans blocks in the cache *only when it will minimally affect performance*.

There are two times where we *try* to writeback dirty data from the DRAM cache to the main memory. First, on a modified LLC-writeback we try to write the data through to memory as well as insert it into the cache like the clean victim design. Second, whenever a cache frame is read on an LLC-read request, if it is dirty, we try to clean the frame.

We only issue these proactive writebacks if the main memory bandwidth is not at or near saturation. We track whether or not we are receiving backpressure from main memory and never send a proactive writeback if the main memory port is full or if main memory has been blocked recently (e.g., the last 50 cycles). If we continue to issue writes when the main memory write buffer is full, the writes will take precedence over the main memory reads, hurting performance because main memory reads are the latency-sensitive demand requests from the CPU.

In addition to proactive writeback, we propose a dirty-data scrubber with the DRAM cache controller. This hardware walks the laundry counts structure and issues reads to proactively clean super-frames when there is extra DRAM cache and main memory bandwidth (e.g., when the application is not in a memory-intensive phase). In fact, this scrubber could work together with the DRAM refresh logic to proactively clean DRAM rows that are activated for refresh. We do not present an evaluation of the dirty data scrubber.

Proactive writeback is similar to self balancing dispatch (SBD) proposed by Sim et al. [137] except we only bypass LLC-writeback requests. Additionally, the proactive writeback algorithm is simpler than SBD. Proactive writeback only tracks the backpressure and

number of writes, unlike SBD which calculates the expected queuing delay.

Adaptive victim cache implementation

Table 7.3 shows the detailed implementation of the adaptive victim DRAM cache for LLC-writeback requests. On the left are the states of the LLC blocks when sending an LLC-writeback. On the right are the actions taken by the laundry hardware.

There are five invariants for the laundry hardware.

1. The laundry count for a super-frame is equal to the number of dirty blocks in the super-frame (insight 1).
2. If there is an entry in the laundry list for a super-frame, all dirty blocks in that super-frame have the same tag (insight 2).
3. The laundry count is always incremented on a modified LLC-writeback, unless the data is already dirty in the DRAM cache. There are two ways to detect when the data is dirty: reading the cache block tag and state (row 13) or inferring the frame is dirty when the laundry list entry is valid and the cache-dirty bit is set (row 3).
4. When cleaning a cache frame, the laundry list entry must be invalidated, if it exists. Without removing the laundry list entry in this case, the laundry count could become inconsistent if there is a block cached in the LLC with the cache-dirty bit set.
5. If the tag of the LLC-writeback request matches the tag stored in the cache frame, the data was modified in the LLC (rows 7, 13, 15). Otherwise, there is an error (rows 6, 8, 14, 16). If the block tag matches, the block was inserted into the LLC with the clean-evict bit set, and will only be written back to the DRAM cache if it was modified.

Proactive writebacks are not shown in Table 7.3. We try to send a proactive writeback on every “Read tag & data”, including when the writeback of the victim block is not required. We only send a proactive writeback under the conditions detailed in Section 7.4. On a

	Laundry count = 0?	Laundry list super-frame tag match?	Laundry list physical tag match?	Cache block tag match?	Cache block dirty?	Data modified in LC?	Data dirty when inserted into LC?	Read tag & data from DRAM cache	Writeback victim	Create new laundry list entry	Invalidate laundry list super-frame entry	Decrement laundry count	Increment laundry count	Write tag & data into DRAM cache
1	Yes	-	-	-	Yes	-	-							
2	Yes	-	-	-	No	-	-							
3	No	Yes	-	-	Yes	Yes	-							
4	No	Yes	-	-	Yes	No	-							
5	No	Yes	-	-	No	-	-							
6	No	Yes	Yes	Yes	Yes	-	-							
7	No	Yes	Yes	Yes	No	Yes	-							
8	No	Yes	Yes	Yes	No	No	-							
9	No	Yes	No	No	Yes	Yes	-							
10	No	Yes	No	No	Yes	No	-							
11	No	Yes	No	No	Yes	Yes	-							
12	No	Yes	No	No	No	No	-							
13	No	No	-	Yes	Yes	Yes	-							
14	No	No	-	Yes	Yes	No	-							
15	No	No	-	Yes	No	Yes	-							
16	No	No	-	Yes	No	No	-							
17	No	No	-	No	Yes	Yes	-							
18	No	No	-	No	Yes	No	-							
19	No	No	-	No	No	Yes	-							
20	No	No	-	No	No	No	-							

Table 7.3: Details of bandwidth optimized DRAM cache functions. Table 7.4 contains an explanation for each row.

1	First dirty data inserted into a clean super-frame
2	Clean data inserted into a clean super-frame
3	Dirty data inserted into a super-frame that contains dirty data from the same super-block. Can infer the cache frame contains dirty data (invariant 3); do not increment laundry count.
4	Dirty data inserted into a super-frame that contains dirty data from the same super-block.
5	Clean data inserted into a super-frame that contains dirty data from the same super-block. Must invalidate laundry list entry (invariant 4).
6	Error. Violates invariant 2.
7	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
8	Error. Clean-evict filter will elide this writeback (invariant 5).
9	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
10	Clean data inserted into a super-frame that contains dirty data from the same super-block. Must invalidate laundry list entry (invariant 4).
11	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
12	Clean data inserted into a super-frame that contains dirty data from a different super-block.
13	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block. The cache frame was previously dirty; no need to increment the laundry count (invariant 3)
14	Error. Clean-evict filter will elide this writeback (invariant 5).
15	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block. The cache frame was previously clean; increment the laundry count (invariant 3)
16	Error. Clean-evict filter will elide this writeback (invariant 5).
17	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
18	Clean data overwriting a dirty cache frame.
19	Dirty data inserted into a dirty super-frame overwriting clean data.
20	Clean data inserted into a dirty super-frame overwriting clean data.

Table 7.4: Explanation of rows in Table 7.3.

proactive writeback, the laundry count entry is decremented, and if there is a valid laundry list entry for the super-frame, it must be invalidated (invariant 4). Additionally, if we take a proactive writeback action on an LLC-read, we send a write to the DRAM cache to update the dirty bit in the cache to be consistent with the laundry counts. This write is off the critical path.

Adaptive victim cache overheads

The total overhead of the adaptive victim caches structures is 1040 KB split evenly between 16 fully-independent HBM channels (65 KB per channel) compared to 36 MB for full tags for a 1 GB DRAM cache. Although this is a significant area overhead, it is less than 6% of the 16 MB LLC, and due to our decentralized design, each individual structure is small, so there is little added latency and power.

We use a super-frame size of 16 KB, split across 16 HBM channels. Thus, each dirty count array entry tracks 16 64-byte frames (1 KB per channel). The laundry count requires 5 bits for each entry, and so, the laundry count requires 40 KB per channel (640 KB for the whole system). We use a laundry list that can cover $\frac{1}{8}$ of the DRAM cache capacity, assuming high spatial locality. The laundry list has 8192 entries per HBM channel. We analyze the effect of increasing the laundry list size in Section 7.5. The area overhead of the laundry list is 25 K per channel.

7.5 Evaluation

Methodology

DRAM caches make the most sense for workloads that are bandwidth-bound and have very large working set sizes. Therefore, we evaluate the NAS parallel benchmarks (NPB) [1, 9]. These are scientific computing benchmarks with large working set sizes and high memory

Name	Class	Footprint	Description
BT	D	10.8 GB	Block tri-diagonal solver
CG	D	16.3 GB	Conjugate gradient
FT	C	5.1 GB	Discrete 3D FFT
IS	D	33.0 GB	Integer sort
LU	D	9.0 GB	LU Gauss-Seidel solver
MG	D	26.5 GB	Multi-grid on meshes
SP	D	16.0 GB	Scalar pentadiagonal solver
UA	D	9.1 GB	Unstructured adaptive mesh

Table 7.5: NPB version 3.3.1 workloads. Class defines the input size. Footprint is the actual resident memory size.

traffic. Table 7.5 describes the workloads. These workloads run between a few minutes and a few hours, natively.

Section 3.2 in Chapter 3 contains more details of the infrastructure used in this chapter. Briefly, we run the NPB on gem5 [19]. However, NPB are too large to execute to completion in simulation. We use random sampling from the whole execution to ensure capturing all of the workloads’ variation. We use simple random sampling and a method similar to the one used by Sandberg et al. using the virtualization hardware (KVM) to fast-forward to each sampled observation [133]. For each observation, we warm up the caches for 10 ms and run the detailed simulation for 2 ms. We have at least 100 observations for each configuration and application totaling about 200 ms of detailed simulation per workload per configuration (not counting cache warmup) or about 6 billion instructions for each sample.

Performance for NPB are measured in average billion floating point operations per second (GFLOPS) or user-mode instructions (UGIPS) for integer-only workloads. Measuring the performance with GFLOPS or UGIPS provides a fair comparison across configurations when using statistical sampling as the total work accomplished in the observation period is proportional to the number of floating point or user-mode instructions executed.

Table 7.6 describes the system we used to evaluate the NPB. We simulated a large multi-core CPU with abundant thread-level parallelism and a high-bandwidth cache system. The

On-die	Memory system
32 CPU cores	16 HBM channels [61]
32 KB each split L1 I/D	8 GB/s per channel
256 KB private L2	1 GB total capacity
16 MB shared LLC	2 DDR3 channels [60]
16 LLC banks	12.6 GB/s per channel
	64 GB total capacity
10 cycles laundry counts + laundry list latency	

Table 7.6: Simulated system details.

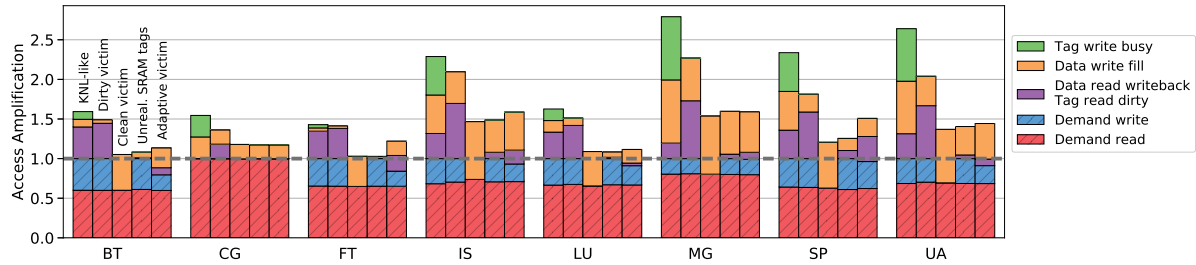
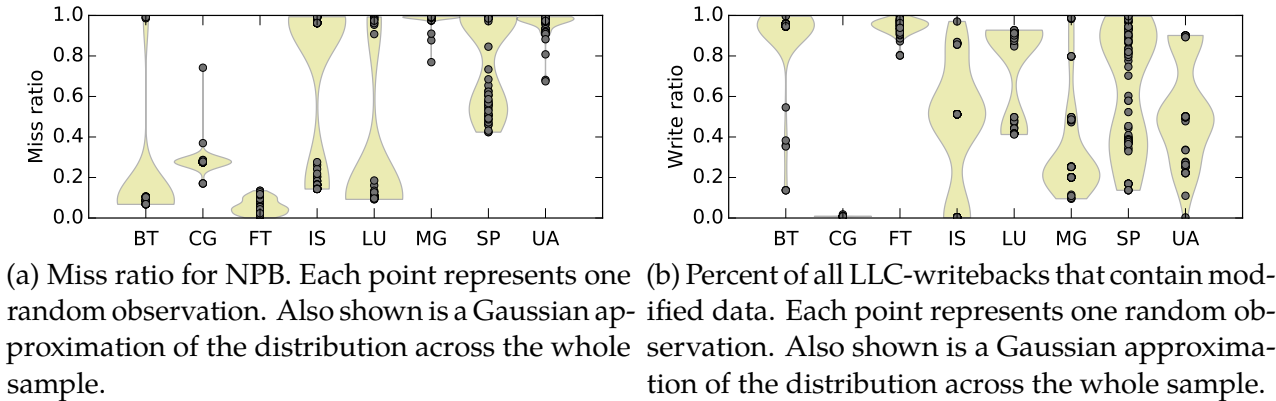


Figure 7.7: Access amplification for the NPB. The dashed line at 1.0 shows the demand requests from the LLC. Any accesses above 1.0 are due to access amplification. Note: there is some noise in the data from the sampling methodology.

on-die CPU system is a proxy for a high-performance compute chip. We use the detailed DRAM models in gem5 to model both HBM and DDR3 [49]. These models are based on the datasheets for each type of memory. When modeling HBM as a DRAM cache, we add one extra BURST cycle to model ECC/tag transfer.

We evaluate four different DRAM cache designs.

KNL-like (Section 7.2) Latency-optimized DRAM cache design.

Dirty victim (Section 7.3) Victim cache that can store dirty data.

Clean victim (Section 7.3) Victim cache that is fully-clean.

Adaptive victim (Section 7.4) Victim cache that is mostly clean, but allows any amount of dirty data.

Workload analysis

The NPB show significant variation both across workloads and within the same workload, which motivates our adaptive victim cache design. Figures 7.6a and 7.6b show the miss ratio and write ratio for each 2 ms observation for each workload. The background area plots are violin plots and show a Gaussian approximation of the distribution. This shows the weights of many observations that show the same value (e.g., most of the BT observations show a miss rate of about 10%, but a few have a miss rate of nearly 100%).

There are a variety of behaviors across the NPB. Figure 7.6a shows the wide variety of DRAM cache miss ratios, even within a single workload. For instance, FT always has a low miss ratio, but IS sometimes has a low miss ratio and sometimes has a miss ratio near 100%.

Figure 7.6b shows the ratio of dirty write requests (roughly the percent of DRAM cache that holds dirty data for a victim cache) for the NPB. This figure shows there is a variety of write behavior, sometimes even within a workload. For instance, CG shows almost no

writes to the DRAM cache, but SP has a broad distribution with phases showing high write proportions and low write proportions. Thus, *it is important for a DRAM cache policy to be robust under many different application characteristics.*

Access amplification

Figure 7.7 shows the access amplification—the number of DRAM cache accesses per demand request from the on-chip LLC (demand reads and writes)—for the eight NPB evaluated. Any accesses above the 1.0 line are “unnecessary” accesses that are not servicing demand requests. Each bar is split by the different types of access amplification described in Section 7.2. The *data read writeback* and *tag read dirty* are combined into a single bar in this figure as each *tag read dirty* access reads both the tag and the data to reduce latency.

Figure 7.7 shows that the KNL-like design makes on average two DRAM cache accesses per demand request. In the worst case (MG), this design has an access amplification of 2.8. MG has a high miss rate, and each miss requires three DRAM cache accesses (Section 7.2).

The dirty victim design reduces the access amplification by removing the *tag write busy* accesses. However, this design increases the relative number of *data read writeback* accesses since the LLC performs must writeback both modified data and clean data if the clean-evict bit is set.

The clean victim design further reduces the access amplification. This design trades the demand write requests for *data write fill* accesses, since for every LLC-writeback this design fills a clean copy of the data. Although the clean victim design eliminates most access amplification, for some workloads it does not perform well because it increases main memory traffic (Section 7.5).

Next, the unrealistic SRAM tag design shows the minimum access amplification while still achieving good performance. This design significantly reduces the access amplification compared to the dirty victim design because, the cache controller knows exactly which accesses are required before accessing the DRAM cache.

The rightmost bar shows the access amplification of the adaptive victim design is only slightly higher than the unrealistic SRAM tag design. Like the SRAM tag design, the adaptive victim cache limits the unnecessary accesses and, in most cases, only accesses the DRAM cache when required. The adaptive victim is within 6% of the amplification reduction of unrealistic SRAM tags.

Like the clean victim design, the adaptive victim design trades some demand write (dirty write) accesses for *data write fill* accesses. In the cases where there is ample main memory bandwidth, the adaptive victim cache uses proactive writebacks perform a clean fill instead of a dirty write. For some workloads like SP and MG, there is not any spare main memory bandwidth, so the adaptive victim design rarely issues proactive writebacks (the number of demand writes is the same between the dirty victim design and adaptive victim design). BT and FT have many proactive writebacks and some of their demand writes are replaced with *data write fill* accesses (clean writes).

NPB performance

Figure 7.8 shows the overall performance for the eight NPB evaluated. The bars represent the average performance (either GFLOPS or UGIPS) of each application across the observations taken. The error bars show a 95% confidence interval for the mean performance.

Some applications see no performance improvement or only a minor performance improvement with a 1 GB DRAM cache (IS, UA, MG) compared to no DRAM cache. For MG and UA, the miss rate is very high: an average of over 90% (Figure 7.6a). For IS, some iterations fit in the 1 GB cache and show low miss rates, but most do not and these iterations that do not fit in the cache dominate the execution time. For UA, even if all of the memory is HBM, there is not much performance improvement. This workload is not bandwidth-bound, but latency-bound due to pointer-chasing through the unstructured grid datastructure.

Across the NPB, some workloads show higher performance with a dirty victim DRAM

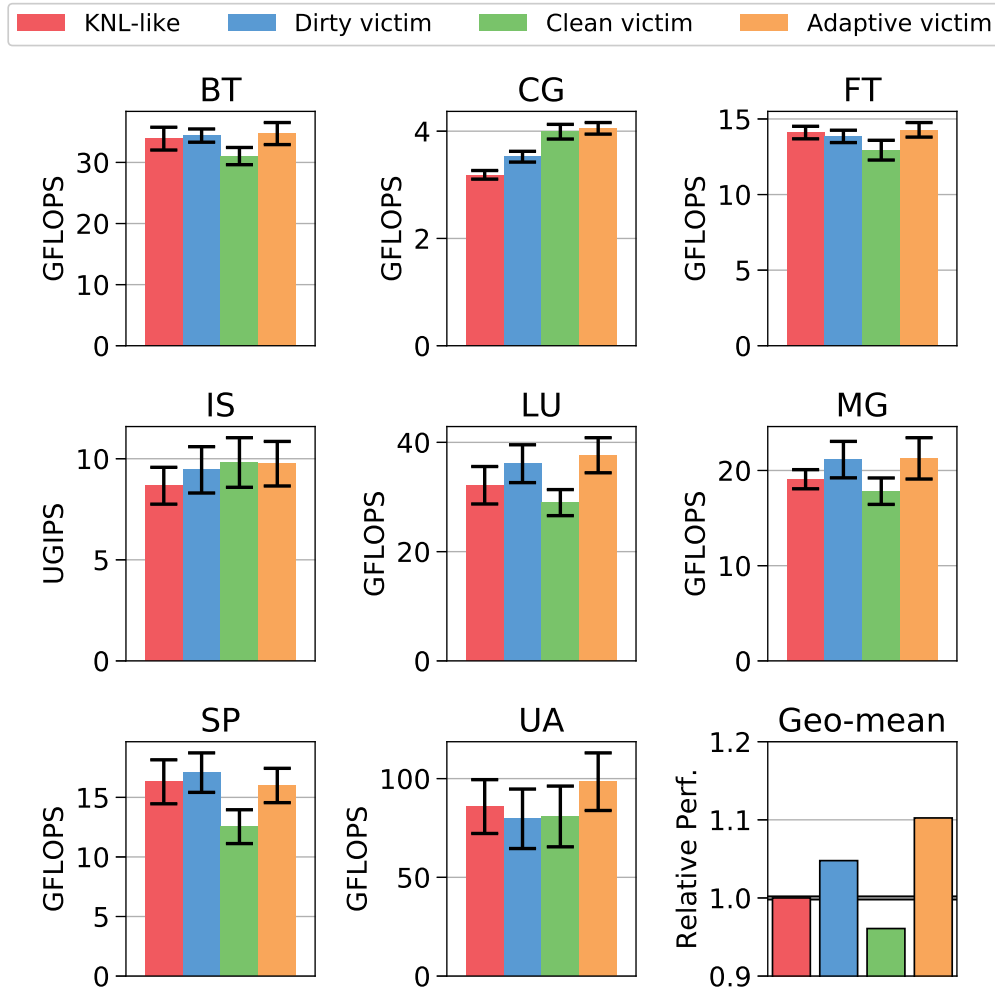


Figure 7.8: Performance of NPB. All workloads measured in GFLOPS, except for IS with UGIPS. The error bars represent a 95% confidence interval for the mean based on the Student's t-Test. The average assumes equal weight for each of the workloads.

cache (BT, FT, LU, MG, SP) and some perform better with a clean victim design (IS, CG). The adaptive victim DRAM cache design performs at least as well as the best of either victim design. In most cases, the adaptive victim design performs as well as the unrealistic SRAM tag design (not shown). The only case the unrealistic SRAM tags outperforms the adaptive victim is FT, where there is a very high write-ratio. In this case, the adaptive victim performs the same as the dirty victim design.

The lower-right graph shows the average relative performance of the four DRAM cache designs with equal weight between each application. Our adaptive victim design shows an average of about 10% performance improvement compared to the baseline KNL-like

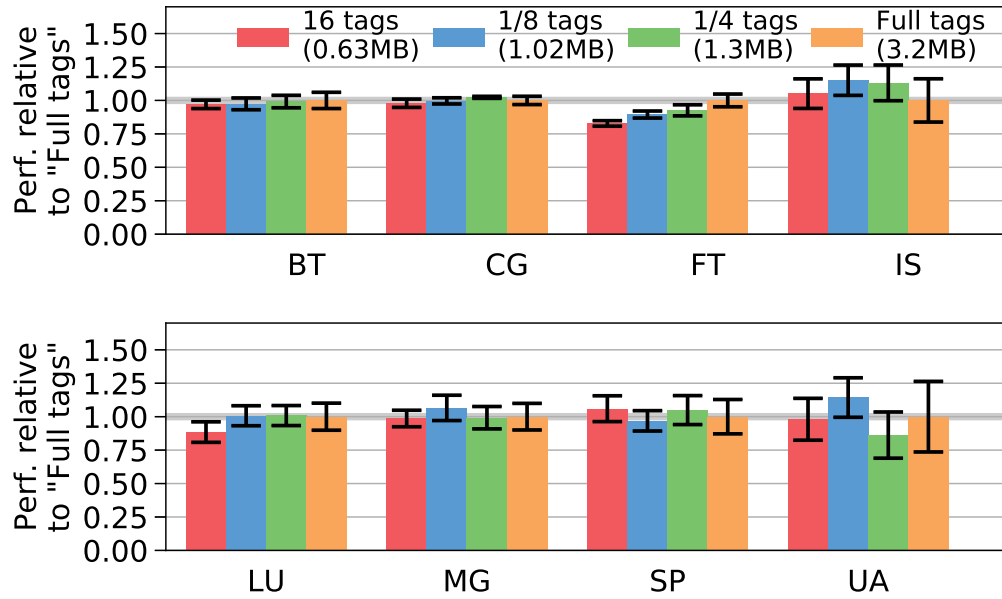


Figure 7.9: Relative performance of different tag array sizes for the laundry list. Error bars show a 95% confidence performance interval for the mean, and all bars are normalized to the average performance with “full tags”.

design. This also shows that the KNL-like design performs worse than a simple victim cache design, and the victim clean design hurts performance, on average. Our adaptive victim cache design is only 2.3% slower than an impractical design with all tags stored in SRAM.

Laundry list size

Recall that an unrealistic SRAM tag designs requires at least 36 MB of on-chip SRAM to store the cache tags. Our adaptive victim design requires about 1 MB of on-chip SRAM. We find that increasing the on-chip storage does not improve the performance of our adaptive victim cache.

Figure 7.9 shows the relative performance of each application for varying sizes of the laundry list. The “full tags” is provisioned with an entry for every super-frame, this would have an overhead of 3.2 MB. “16 tags” represents the minimal number of tags (one per channel) for the laundry list. The 0.63 MB area overhead of this design are the laundry counts. The “ $\frac{1}{8}$ tags” is the design used in the previous results (1040 KB), and the “ $\frac{1}{4}$ tags”

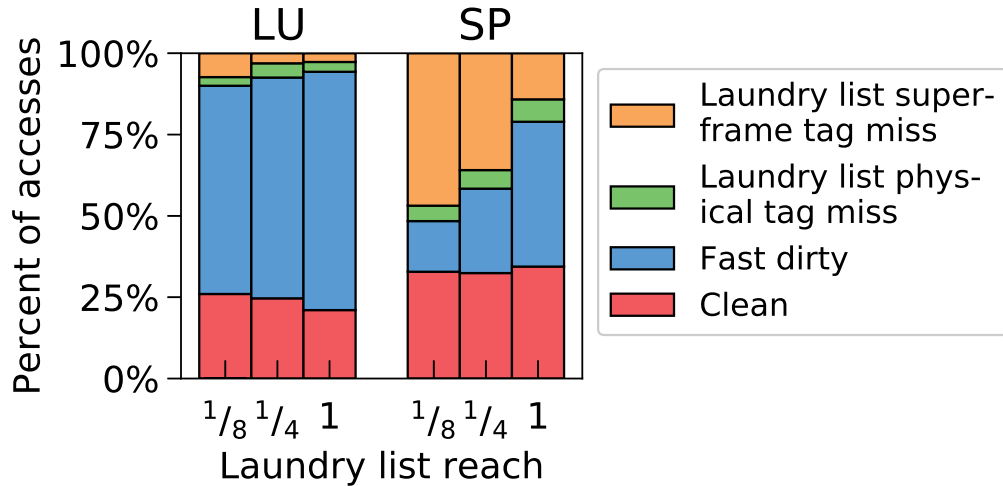


Figure 7.10: Breakdown of laundry list and laundry counts lookups for all write requests for two representative applications, FT and LU. Compares a laundry list with $1/8$ reach, $1/4$ reach, and large enough to cover the entire cache (impractically large).

shows the performance if the tag array is provisioned so at most $1/4$ of the cache is allowed to be dirty. The “ $1/4$ tags” design has an area overhead of about 1.3 MB.

The laundry counts structure is most important for providing high performance with the adaptive victim cache. Figure 7.9 shows that for most applications the size of the laundry list does not significantly affect performance. Although some bars for smaller laundry list designs appear higher, they are within the noise of our sampling methodology. FT and LU show a clear effect when the tag array size is increased. However, even for FT, the “ $1/8$ tags” performs as well as the dirty victim design (Figure 7.8) demonstrating the robustness of our design. In the worst case, it falls back on the dirty victim performance, not the clean victim design.

Figure 7.10 shows the breakdown of the result of consulting the laundry structures for each write request as a percentage of all writes. The labels correspond to the flowchart in Figure 7.5. This figure shows the importance of tracking tags in the laundry list instead of just tracking the whether the super-frame is dirty (i.e., just the laundry counts). The goal of the laundry list is to avoid the “slow dirty” path which reduces the number of DRAM cache reads before writing new data, since reading before writing incurs extra latency and occupancy overhead on the DRAM bus [60]. The sum of the red and blue bars (bottom

two) show the effectiveness of the laundry list at removing these accesses.

For LU, most of the write requests are to data that already exists in the DRAM cache and the laundry list effectively elides the extra write requests. This is why the access amplification for LU is low for our adaptive victim cache design (Figure 7.7). For LU, a larger laundry list allows more of the cache to contain dirty data (lower red bar). This reduces the main memory traffic but does not significantly affect performance since most of the accesses are on the “fast dirty” or “clean” paths.

For SP, increasing the size of the laundry list increases its effectiveness. With a larger laundry list we take the “fast dirty” path more often. However, as Figure 7.9 shows, this does not have a significant effect on performance for this workload. SP and FT (not shown) have the largest effect of increasing the laundry list size. If we instead only used the laundry counts or a bitvector tracking the dirty frames, all of the blue parts of the bars in Figure 7.10 would generate extra DRAM cache accesses increasing access amplification.

As shown by Figure 7.10 we find that tag mismatches in the laundry list are rare. This implies that tracking super-block tags instead of 64-byte block tags does not cause significant false sharing. It is rare that there is more than one dirty super-block within each super-frame in the DRAM cache even for large super-blocks (i.e., 16 KB).

Tracking dirty data

Figure 7.6b showed that some applications have mostly modified write requests to the DRAM cache. Thus, for these applications, a large fraction of the cache is dirty. Therefore, we should not use solutions like DiRT [137] that constrain the amount of dirty data in the cache as this solution reverts to the clean victim cache performance. Our adaptive victim cache is robust under different write traffic conditions.

Figure 7.11 shows the performance of victim designs and a DiRT-like design. To model DiRT, we use the laundry list but do not use the laundry counts. Instead of modeling exactly 1024 pages in the tag array (DiRT design), we allow up to $\frac{1}{4}$ of the cache (256 MB)

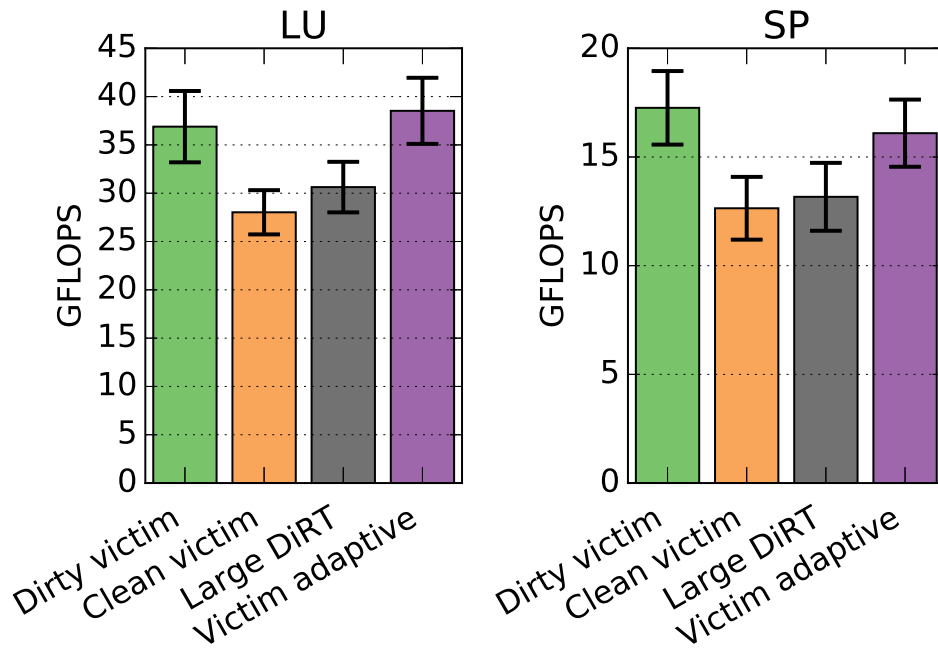


Figure 7.11: Performance for two workloads with a very aggressive DiRT-like design [137].

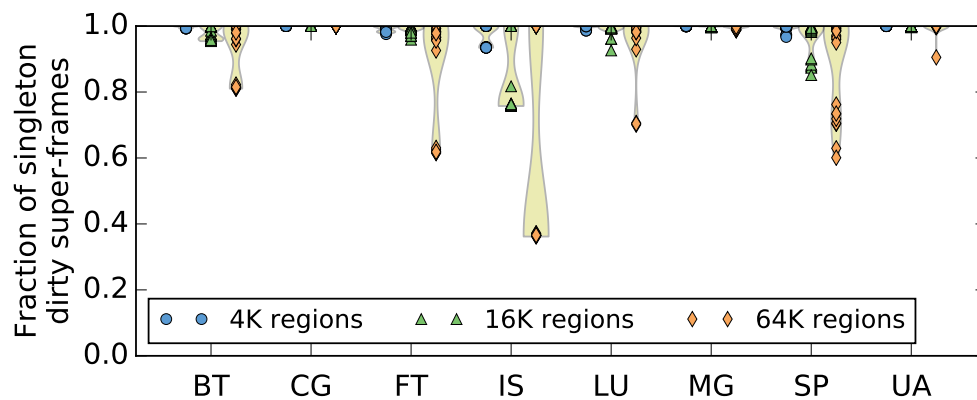


Figure 7.12: Percent of super-frames that have a single super-block cached. Each point represents one random observation. Also shown is a Gaussian approximation of the distribution across the whole sample.

to be dirty, compared to only 4 MB in Sim et al. [137]. This is a very aggressive model of the DiRT.

Figure 7.11 shows that for the workloads where the clean victim design performs poorly DiRT also performs poorly. In this case, DiRT performs about the same as a clean victim cache because more than $\frac{1}{4}$ of the active blocks in the cache are dirty. Therefore, the constraining bandwidth is the main memory bandwidth, not the DRAM cache bandwidth. In both of these cases, the adaptive victim performs as well as the dirty victim design because it changes to a writeback mode instead of writethrough.

Figure 7.12 shows the spatial locality of the NPB. This figure shows the number of super-frames for which only a single super-block is cached for three different super-frame sizes. A super-frame size of 1 KB is the same as directly tracking single cache frames since we have a 16-channel HBM.

Figure 7.12 shows that for most observations for most workloads almost all super-frames have a single super-block. However, if we increase the super-frame size to 64 KB, the spatial locality breaks down for some workloads. Thus, we choose 16 KB super-frames. This size provides a good tradeoff between area overhead and spatial locality.

Results summary

The adaptive victim DRAM cache design gets the best of the clean victim and dirty victim designs. The adaptive victim design is robust, and performs well under both high and low miss rate and high and low fractions of write traffic, unlike the other victim designs. The adaptive victim DRAM cache sees these benefits because it does not squander its bandwidth or the bandwidth to main memory. By tracking a small amount of information in SRAM just for dirty frames, our design limits the access amplification and outperforms the alternatives.

7.6 Related work

There is significant recent work treating HBM as a DRAM cache. We build on previous research that reduces the latency when accessing DRAM caches, and while it is not the focus of this work, our ideas can be used in conjunction with techniques to increase the hit rates of DRAM caches.

Much of the initial work on DRAM caches focused on reducing the latency when storing the cache tags in DRAM [55, 87, 121]. Loh and Hill proposed storing the cache tags in the same DRAM row as the data to reduce the hit latency [87]. Qureshi and Loh developed a latency-optimized DRAM cache design (Alloy cache) that stores the tag-and-data together in a single entity in the DRAM cache [121]. Additionally, Qureshi and Loh advocate for a direct-mapped cache instead of an associative cache. A direct-mapped design reduces the latency, and the increased hit rate of set-associativity is limited due to the large capacity.

Loh and Hill introduce a MissMap to predict whether the line is in the DRAM cache or main memory [87]. The MissMap must always predict conservatively; it cannot predict the line is *not* in the DRAM cache, if it actually is. Our laundry list and laundry counts is similar in spirit to the MissMap, but instead of tracking all cache lines, it only stores information for dirty cache lines, reducing the required area.

BEAR's (Bandwidth efficient architecture) approach to reducing the bandwidth overheads of DRAM caches is to filter most misses and fills to the DRAM cache [31]. In contrast, our adaptive victim design does not use any filters and works synergistically in a system that implements filters like the BAB and NTC from BEAR to further reduce the access amplification.

Sim et al. [137] propose a DRAM cache design which limits the dirty data in the DRAM cache to allow requests to be satisfied by main memory instead of the DRAM cache. However, their cache *constrains* the dirty data. In a high-miss rate cases, the cache behaves like a writethrough cache (clean victim), hurting performance for some workloads. Stuecheli et al. use the SRAM LLC as a write queue for DRAM and proposes some similar hardware. [142].

The “cache cleaner” can be used in conjunction with our ideas to help keep the DRAM cache clean, and the set state vector tracks the oldest dirty lines in the cache. Our laundry list and counts track *all* of the dirty lines in the DRAM cache for correctness.

Many proposals track the DRAM cache tags at a coarse granularity, like our laundry list and laundry counts [30, 62–64, 82]. The footprint cache tracks the data in the cache at a page granularity, but manages the cache on a block granularity [63]. The footprint cache improves the DRAM cache hit rate by prefetching only the likely to be accessed data from main memory. The Unison cache combines the ideas of the footprint cache with Alloy cache by storing the metadata in the DRAM cache providing the benefits of the footprint cache without the SRAM area overheads [62]. However, Unison cache has access amplification for managing the metadata that is now stored in the DRAM cache.

The Bi-Modal cache also uses a dual-granularity design that leverages page-based tracking to reduce the overheads of block-based cache [48]. In this design, instead of storing the tags with the data, the Bi-Modal cache reserves a DRAM cache bank/channel for the tags. This may reduce the interference between unnecessary accesses, but not the access amplification.

Tag tables stores the cache tags in main memory in a page-table-like structure and caches frequently accessed tags in the on-chip SRAM LLC [40]. Tag tables does not have any unnecessary accesses to the DRAM cache, but does increase the traffic to main memory, if there is low spatial locality.

Dirty-block index is similar to the laundry list and laundry counts in that it tracks just the dirty blocks in the cache [135]. However, as described in Section 7.4, the adaptive victim cache tracks which *cache frames*—the specific part of the cache—that is dirty, not the physical addresses that are dirty.

7.7 Conclusions

Off chip bandwidth has long been a bottleneck in computing systems. Increased heterogeneity and logically integrated accelerators (e.g., GPUs 4) are exacerbating this issue, as we show in Chapter 5. 3D-stacked DRAM can provide significant performance improvements, as we show in Chapter 6; however, it come with high power costs.

In this chapter, we investigate a heterogeneous memory system with both 3D stacked DRAM and DRAM DIMMs. We present an adaptive victim DRAM cache design that robustly shifts from behaving as a fully-clean cache to behaving as a dirty cache depending on the program behavior. This design limits the access amplification (total DRAM cache accesses per LLC-miss demand request) to the DRAM cache, improving performance over other designs.

7.8 Artifacts

In the process of writing this chapter, the following artifacts were generated.

- Changes to gem5 to add a DRAM cache model and support the sampling methodology described in Section 3.2. This code is available in the following git repository with the “chtc-micro-1.3” tag. <https://github.com/powerjg/gem5/releases/tag/chtc-micro-1.3>.
- The NAS parallel benchmarks were modified with region of interest annotations and compiled to run in gem5.
- Other supporting files used for full-system simulation.
- Data generated from running the simulator with specific configurations.
- Scripts to process the data and generate the graphs.

— 8 —

CONCLUSIONS AND FUTURE WORK

Technology is trending towards heterogeneous systems due to the breakdown of Dennard Scaling and the slowdown of Moore’s Law. This thesis explores two different kinds of heterogeneity. First, in Chapter 4, we show specialized processing components, namely GPUs, can be extended with the same shared virtual memory semantics as traditional processors. Sharing the same virtual memory subsystem between the CPU and GPU provides three benefits. First, it makes programming the GPU easier since developers are not required to reason about two address spaces. Second, it reduces the overhead of using the GPU for large applications. Third, it simplifies cooperation between the CPU and GPU since each can execute a subset of the application.

We take advantage of these benefits in Chapter 5 and develop new database algorithms for scan and aggregate operators on integrated GPUs that have a shared address space. Chapter 5 further shows an example of how tightly-integrated components can synergistically work together. In a full database application, the GPU is inappropriate for most of the computations. However, for a subset of these computations, scans and some aggregates, the integrated GPU can provide a performance improvement and energy reduction.

In Chapter 6, we explore system designs specifically for bandwidth-constrained workloads, like the database workload in Chapter 5. By applying a simple back-of-the-envelope

analytic model, we show there are tradeoffs between using many small die-stacked systems (best performance) and one large monolithic system (lowest cost if real-time performance is not required).

Finally, in Chapter 7 we consider a real system with heterogeneous memory, the Intel Knights Landing platform with both DRAM DIMMs and 3D stacked DRAM. We show that current DRAM cache policies waste both the stacked DRAM and the backing DRAM bandwidth. We propose a new cache policy, adaptive victim caching, that achieves the best performance of other caching policies. Chapter 7 shows one way to use heterogeneous memory to increase system performance.

Future work

In this section, we give outlines of future research directions based on the work in this thesis.

Accelerator integration

In Chapter 4 we show how to extend virtual memory to GPUs. However, there are many more accelerators, both already on chip and proposed, that lack support for cache coherence and shared virtual memory. Hardware support for cache coherence and shared virtual memory is important for application developers. These are two characteristics of multicore CPUs that all programmers assume are present and breaking this assumption puts significant burden on developers.

Many of the ideas in Chapter 4 can be extended to other types of accelerators. GPUs evaluated in Chapter 4 put high bandwidth pressure on the memory system, and the ideas in Chapter 4 apply straightforwardly to other high bandwidth accelerators.

However, some accelerators have different constraints. For instance, always-on sensors, used for simple applications like step counting and activity detection, are power-

constrained. Opening these systems to all developers could significantly increase their usefulness. A compelling question for low-power sensors is how to efficiently transfer the control from the “always-on” ultra-low power device to another more computationally powerful processor in the system. In addition to transferring control, programmers will also want to transparently transfer their data into and out of these processors. The key difficulty is enabling coherence and consistency without resorting to traditional high-power mechanisms.

Heterogeneous memory systems

In Chapter 7 we optimize one way to integrate heterogeneous memory into the system by using it as a hardware managed cache. With emerging non-volatile memory technologies like Intel and Micron’s 3D XPoint heterogeneous memory is going to be more prevalent. However, using each level as a hardware managed cache of the larger memory is unlikely to be sustainable. Already with HBM and DRAM DIMMs there are some applications that get no benefit from the high-bandwidth memory.

Looking forward, the application developer and the compiler have information about the data reuse. However, requiring the programmer or the system to manually move the data between different types of memory has the same downsides as discrete GPUs as discussed in Chapter 4! Therefore, providing a mechanism to transfer this information to the hardware to guide the policies seems like a fruitful direction for future research.

Evaluation methodology

Not only is hardware technology changing rapidly, but the workloads that execute on this hardware are also evolving. A few examples include machine learning, augmented reality, big-data analytics, and intelligent personal assistants. These applications are end-to-end solutions, consisting of many interacting kernels of computation, and they cannot easily or accurately be represented as a single microbenchmark. Optimizing these applica-

tions requires changes across the entire hardware-software stack from new accelerators and emerging programmable processors to system integration and new programming interfaces.

However, current architecture evaluation infrastructure is not easily adapted to studying end-to-end applications. Architectural simulation today takes workloads out of their native environment and places them in a simulation “Petri dish”. For instance, only a certain Linux kernel and a certain compiler are known to work with the simulation infrastructure, and this environment may be different from the native platform for the application.

Like in other science disciplines, new simulation infrastructure can allow researchers to investigate hardware changes while executing the application natively. We call this new method of simulation “*in situ* simulation”.

It is currently possible to use *in situ* simulation to study CPUs by leveraging ubiquitous virtualization technology by building off of the infrastructure described in Section 3.2. One way to further extend this support is to expose the native hardware performance counters to gem5 increasing the kinds of analysis possible during the program analysis phase. For instance, rather than picking n random locations, the program analysis phase can apply clustering techniques to choose the “best” simulation locations to gain the most information.

This virtualization can be extended to other accelerators, programmable processors, and even to novel devices via fast emulation (e.g., with FPGAs). With this infrastructure, everyone in the computer architecture community will be able to investigate previously unaddressable problems. This evaluation methodology can allow the community to push the boundaries of knowledge more quickly.

Technology trends such as the end of Dennard scaling, the slowing of Moore’s Law, and the big-data revolution are pushing the limits of our computing platforms. These are unique conditions for the computer architecture community, which has seen computational systems grow exponentially in efficiency for the past 50 years. Looking forward, there are exciting

opportunities for novel research. The next solutions to increasing computational efficiency are not going to come from small optimizations of current systems, these have already been wrung mostly dry, but from creative new architectures and cross-stack collaborations.

BIBLIOGRAPHY

- [1] NAS Parallel Benchmarks. <https://www.nas.nasa.gov/publications/npb.html>.
- [2] 3D-ICs. <http://www.jedec.org/category/technology-focus-area/3d-ics-0>, 2012. Accessed: 2014-6-24.
- [3] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak R. Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. *PVLDB*, 6(11):1057–1067, 2013.
- [4] AMD. AMD Opteron X2100 Series APU. <http://www.amd.com/en-us/products/server/x2100>. Accessed: 2014-5-28.
- [5] AMD. AMD’s most advanced APU ever. <http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx>. Accessed: 2014-1-23.
- [6] AMD. Graphics Card Solutions. <http://products.amd.com/en-us/GraphicCardResult.aspx>. Accessed: 2014-1-23.
- [7] AMD. AMD Graphics Cores Next (GCN) Architecture. Technical report, 2012.
- [8] Mehdi Asnaashari. RRAM enabling new Era of Solid State Storage Arrays. In *Flash Memory Summit 2015*, 2015.

- [9] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, H D Simon, V Venkatakrishnan, and S K Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.
- [10] Ali Bakhoda, G.L. Yuan, W.W.L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 2009.
- [11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, page 48, New York, New York, USA, 2010. ACM Press.
- [12] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, Jul 2013.
- [13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13*, pages 237–248, New York, NY, USA, 2013. ACM.
- [14] Bradford Beckmann. AMD's gem5 APU Simulator. In *Second gem5 User Workshop*, 2015.
- [15] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*, page 26, New York, New York, USA, 2008. ACM Press.

- [16] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, pages 383–394, New York, New York, USA, 2013. ACM Press.
- [17] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 62–63, feb 2011.
- [18] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40. IEEE, sep 2009.
- [19] 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. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.
- [20] Bryan Black. MICRO 46 Keynote: Die Stacking is Happening, 2013.
- [21] Bryan Black, Murali Annavaram, Ned Brekelbaum, John Devale, Jiang Lei, Gabriel H. Loh, Don McCauley, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, Shen John, and Clair Webb. Die stacking (3D) microarchitecture. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 469–479, 2006.
- [22] Haran Boral and David J. DeWitt. Database machines: An idea whose time passed? A critique of the future of database machines. In *IWDM Workshop*, pages 166–187, 1983.

- [23] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.
- [24] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture - ISCA '96*, pages 78–89, New York, New York, USA, 1996. ACM Press.
- [25] Bryan Casper. Reinventing DRAM with the Hybrid Memory Cube. <http://blogs.intel.com/intellabs/2011/09/15/hmc/>, 2011.
- [26] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195, Sept 2013.
- [27] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, oct 2009.
- [28] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA Conference*, pages 105–115, 2007.
- [29] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *SIGMOD Conference*, pages 271–282, 2001.
- [30] Chia Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of

- Hardware-Managed Cache. *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2014.
- [31] 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, New York, USA, 2015. ACM Press.
- [32] Wu chun Feng and Shucui Xiao. To GPU synchronize or not GPU synchronize? In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3801–3804, May 2010.
- [33] HMC Consortium. Hybrid Memory Cube Specification 1.0, 2013.
- [34] Radoslav Danilak. System and method for hardware-based GPU paging to system memory, 2009.
- [35] Dell. PowerEdge R930. <http://www.dell.com/us/business/p/powerededge-r930/pd>.
- [36] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA Conference*, pages 365–376, 2011.
- [37] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *PVLDB*, 3(1-2):670–680, September 2010.
- [38] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
- [39] Ziqiang Feng and Eric Lo. Accelerating aggregation using intra-cycle parallelism. In *Data Engineering (ICDE), 2015 IEEE 31th International Conference on*, 2015.

- [40] Sean Franey and Mikko Lipasti. Tag tables. *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 514–525, 2015.
- [41] Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay Patel, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10*, New York, New York, USA, Mar 2010. ACM Press.
- [42] Kourosh Gharachorloo, Luiz André Barroso, and Andreas Nowatzky. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *12th Symposium on Computer Architecture and High-Performance Computing (HPCA 12)*, 2000.
- [43] Gabriela Gligor and Silviu Teodoru. Oracle Exalytics: Engineered for Speed-of-Thought Analytics. *Database Systems Journal*, 2(4):3–8, December 2011.
- [44] Brian T. Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating database operations using a network processor. In *DaMoN Workshop*, 2005.
- [45] Juan Gómez-Luna, Izzat El Hajj, Victor Chang Li-Wen Garcia-Flores, Simon 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.
- [46] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUteraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD Conference*, page 325, 2006.
- [47] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD Conference*, page 215, 2004.

- [48] Nagendra Gulur, Mahesh 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*, number i, pages 38–50. IEEE, dec 2014.
- [49] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha 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. IEEE, mar 2014.
- [50] Rajeeb Hazra. Accelerating insights in the technical computing transformation. In *International Supercomputing Convergence*, 2014.
- [51] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *SIGMOD Conference*, page 511, 2008.
- [52] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB*, 6(10):889–900, 2013.
- [53] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD Conference*, pages 389–400, 2007.
- [54] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference on - SYSTOR '09*, page 1, 2009.
- [55] Cheng-Chieh Huang and Vijay Nagarajan. ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 51–60, 2014.

- [56] IBM. IBM to produce Micron’s hybrid memory cube in debut of first commercial, 3D chip-making capability.
<https://www-03.ibm.com/press/us/en/pressrelease/36125.wss>, 2011.
- [57] Intel. Intel® Xeon® Processor E7-8860 v3.
http://ark.intel.com/products/84680/Intel-Xeon-Processor-E7-8860-v3-40M-Cache-2_20-GHz.
- [58] Intel. Volume 3A: System Programming Guide Part 1. In *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, pages 4.35–4.38. 2013.
- [59] Intel. Intel and Micron Produce Breakthrough Memory Technology.
<https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/>,
2015.
- [60] JEDEC. DDR3 SDRAM Standard. Technical report, JEDEC, 2012.
- [61] JEDEC. High Bandwidth Memory (HBM) DRAM. Technical Report November, JESD235A, 2015.
- [62] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. *47th Annual IEEE/ACM International Symposium on Microarchitecture*, (Micro):25–37, 2014.
- [63] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 404–415, 2013.
- [64] Xiaowei Jiang, N. Madan, and Li Zhao. CHOP: Adaptive filter-based dram caching for CMP server platforms. *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA 16)*, pages 1–12, 2010.

- [65] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.
- [66] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA '90*, pages 364–373, New York, New York, USA, 1990. ACM Press.
- [67] Toni Juan, Tomas Lang, and Juan J. Navarro. Reducing TLB power requirements. In *Proceedings of the 1997 international symposium on Low power electronics and design - ISLPED '97*, pages 196–201, New York, New York, USA, 1997. ACM Press.
- [68] I. Kadayif, a. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction TLB energy. *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 185–196, 2002.
- [69] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. In *DaMoN Workshop*, pages 55–62, 2012.
- [70] Gokul B Kandiraju and Anand Sivasubramaniam. Characterizing the d -TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '02*, page 129, New York, New York, USA, 2002. ACM Press.
- [71] Hari S. Kannan, Brian P. Lilly, Perumal R. Subramoniam, and Pradeep Kanapathipillai. Delaying cache data array updates, 2016.
- [72] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*, page 24, New York, New York, USA, 2011. ACM Press.

- [73] Steven W. Keckler. Life after Dennard and How I Learned to Love the Picojoule. In *MICRO 44 Keynote*, 2011.
- [74] Taeho Kgil, Shaun D’Souza, Ali Saidi, Nathan Binkert, Ronald Dreslinski, Trevor Mudge, Steven Reinhardt, and Krisztian Flautner. PicoServer: Using 3D Stacking Technology To Enable A Compact Energy Efficient Chip Multiprocessor. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, page 117, New York, New York, USA, 2006. ACM Press.
- [75] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD Conference*, pages 339–350, 2010.
- [76] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.
- [77] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A cpu-gpu heterogeneous simulation framework user guide. 2012.
- [78] Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). *Synthesis Lectures on Computer Architecture*, 7(2):1–96, November 2012.
- [79] Joonyoung Kim and Younsu Kim. HBM: Memory Solution for Bandwidth-Hungry Processors. In *Hot Chips 26*. SK hynix Inc., 2014.

- [80] Jung Pill Kim, Taehyun Kim, Wuyang Hao, Hari M Rao, Kangho Lee, Xiaochun Zhu, Xia Li, Wah Hsu, Seung H Kang, Nowak Matt, and Nick Yu. A 45nm 1Mb embedded STT-MRAM with design techniques to minimize read-disturbance. In *2011 Symposium on VLSI Circuits - Digest of Technical Papers*, pages 296–297, June 2011.
- [81] Marcin Kościelnicki. envytools.
<https://github.com/pathscale/envytools/blob/master/hwdocs/memory/nv50-vm.txt>,
 2013.
- [82] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A fully associative, tagless DRAM cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pages 211–222, New York, New York, USA, 2015. ACM Press.
- [83] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, number c, pages 469–480, 2009.
- [84] Yinan Li and Jignesh M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD Conference*, pages 289–300, 2013.
- [85] Yinan Li and Jignesh M Patel. WideTable: An Accelerator for Analytical Data Processing. *PVLDB*, 7(10):907—918, 2014.
- [86] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE Conference*, pages 1111–1120, April 2008.

- [87] Gabriel H. Loh and Mark D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. *44th IEEE/ACM International Symposium on Microarchitecture*, page 454, 2011.
- [88] Jason Lowe-Power, Mark D. Hill, and David A. Wood. When to use 3D Die-Stacked Memory for Bandwidth-Constrained Big Data Workloads. In *The Seventh Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (BPOE 7)*, 2016.
- [89] Andrew McAfee and Erik Brynjolfsson. Big Data: The Management Revolution. *Harvard Business Review*, oct 2012.
- [90] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 95–104. IEEE, apr 2008.
- [91] Jiayuan Meng and Kevin Skadron. A reconfigurable simulator for large-scale heterogeneous multicore architectures. In *ISPASS 2011*, 2011.
- [92] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU. *ACM SIGARCH Computer Architecture News*, 40(3):72, sep 2012.
- [93] Justin Meza, Jichuan Chang, Hanbin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Computer Architecture Letters*, 11(2):61–64, 2012.
- [94] Microsoft. C++ amp: Language and programming model version 1.0, 2012.
- [95] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the mellanox

- infiniband software stack. In *Proceedings of the 12th International Conference on Parallel Processing, Euro-Par'06*, pages 124–133, Berlin, Heidelberg, 2006. Springer-Verlag.
- [96] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, Hewlett Packard Labs, 2009.
- [97] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 208–219, 2006.
- [98] John Nickolls and William J Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, mar 2010.
- [99] Andreas Nowatzky, Gunes Aybay, Michael C. Browne, Edmund J. Kelly, Michael Parkin, Bill Radke, and Sanjay Vishin. The s3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champaign, Illinois, USA, August 14-18, 1995. Volume I: Architecture.*, pages 1–10, 1995.
- [100] NVIDIA. NVIDIA’s Next Generation CUDA Compute Architecture: Fermi, 2009.
- [101] NVIDIA. NVIDIA CUDA C Programming Guide Ver. 4.0, 2011.
- [102] NVIDIA. NVIDIA’s Next-Gen Pascal GPU Architecture.
<http://blogs.nvidia.com/blog/2015/03/17/pascal/>, 2015.
- [103] NVIDIA. Tesla P100 Data Center Accelerator.
www.nvidia.com/object/tesla-p100.html, 2017.
- [104] Mike O’Connor. Highlights of the High- Bandwidth Memory (HBM) Standard. In *The Memory Forum*, 2014.
- [105] Oracle. Big Data Analytics. Technical Report March, Oracle, 2013.

- [106] Oracle. Oracle's SPARC T7 and SPARC M7 Server Architecture for the Real-Time Enterprise. Technical Report October, 2015.
- [107] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, may 2008.
- [108] J Thomas Pawlowski. Hybrid Memory Cube (HMC). In *Hot Chips 23*, 2011.
- [109] Jon Peddie. Market Watch. Technical Report Q1'04, Jon Peddie Research, 2014.
- [110] Stephen Phillips. M7: Next generation SPARC, 2014.
- [111] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on GPUs. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, pages 743–758, New York, New York, USA, 2014. ACM Press.
- [112] Orestis Polychroniou and Kenneth A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN Workshop*, page 6, 2013.
- [113] Orestis Polychroniou and Kenneth A. Ross. Vectorized bloom filters for advanced SIMD processors. In *DaMoN Workshop*, page 6, 2014.
- [114] 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, New York, USA, 2013. ACM Press.
- [115] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*, 13(1):1–4, 2014.

- [116] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *The 20th IEEE International Symposium On High Performance Computer Architecture*, 2014.
- [117] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. Implications of emerging 3D GPU architecture on the scan primitive. *SIGMOD Rec.*, 44(1), 2015.
- [118] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. Toward GPUs being mainstream in analytic processing: An Initial Argument Using Simple Scan-aggregate Queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware - DaMoN'15*, pages 1–8, New York, New York, USA, 2015. ACM Press.
- [119] Steven Przybylski. SDRAMs Ready to Enter PC Mainstream. *Microprocessor Report*, 10(6):17–23, 1996.
- [120] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.
- [121] 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 *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 235–246, dec 2012.
- [122] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

- [123] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *ICDE Conference*, pages 60–69, 2008.
- [124] Parthasarathy Ranganathan. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer*, 44(1):39–48, jan 2011.
- [125] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. Memory access scheduling. In *27th International Symposium on Computer Architecture (ISCA 2000)*, June 10-14, 2000, Vancouver, BC, Canada, pages 128–138, 2000.
- [126] Phil Rogers. The programmer’s guide to the apu galaxy, 2011.
- [127] Phil Rogers. Heterogeneous System Architecture Overview. In *Hot Chips 25*, 2013.
- [128] Phil Rogers, Joe Macri, and Sasa Marinkovic. AMD heterogeneous Uniform Memory Access, 2013.
- [129] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, jan 2010.
- [130] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 233–248, 2011.
- [131] Kazuta Saito, Richard J. Webb, and Blake R. Dronen. Advances of 3M wafer support system. In *2011 IEEE International 3D Systems Integration Conference (3DIC)*, 2011 *IEEE International*, pages 1–4. IEEE, jan 2012.

- [132] Samsung. Samsung Starts Mass Producing Industry’s First 3D TSV Technology Based DDR4 Modules for Enterprise Servers.
<http://www.samsung.com/semiconductor/about-us/news/13602>, 2014.
- [133] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. *2015 IEEE International Symposium on Workload Characterization*, pages 183–192, 2015.
- [134] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious SIMD sort. In *SIGMOD Conference*, pages 351–362, 2010.
- [135] Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. The Dirty-Block Index. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 157–168. IEEE, jun 2014.
- [136] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002.*, pages 45–57, 2002.
- [137] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike O’Connor, and Mithuna 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. IEEE, dec 2012.
- [138] Avinash Sodani. Race to Exascale: Opportunities and Challenges Intel Corporation, 2011.
- [139] Avinash Sodani. MEGI cache states. Private communication, 2016.

- [140] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, mar 2016.
- [141] Yoon-jong Song, Gitae Jeong, In-gyu Baek, and Jungdal Choi. What Lies Ahead for Resistance-Based Memory Technologies? *Computer*, 46(8):30–36, aug 2013.
- [142] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The virtual write queue: coordinating DRAM and last-level cache policies. In *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, New York, New York, USA, 2010. ACM Press.
- [143] Liwen Sun, Sanjay Krishnan, Reynold S. Xin, and Michael J. Franklin. A partitioning framework for aggressive data skipping. *PVLDB*, 7(13):1617–1620, 2014.
- [144] Michal Switakowski, Peter A. Boncz, and Marcin Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12):1759–1770, 2012.
- [145] P.J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, jun 1990.
- [146] Peter C. Tong, Sonny S. Yeoh, Kevin J. Kranzusch, Gary D. Lorensen, Kaymann L. Woo, Ashish Kishen Kaul, Colyn S. Case, Stefan A. Gottschalk, and Dennis K. Ma. Dedicated mechanism for page mapping in a GPU, 2008.
- [147] Transaction Processing Performance Council. TPC Benchmark H. Revision 2.14.3, nov 2011.
- [148] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for CPU-GPU computing. In *PACT '12*, 2012.
- [149] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual

- memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, April 2016.
- [150] Scott Wasson. Chip problem limits supply of quad-core Opterons, 2007.
- [151] Don Weiss, Michael Dreesen, Michael Ciraula, Carson Henrion, Chris Helt, Ryan Freese, Tommy Miles, Anita Karegar, Russell Schreiber, Bryan Schneller, and John Wu. An 8MB level-3 cache in 32nm SOI with column-select aliasing. In *2011 IEEE International Solid-State Circuits Conference*, pages 258–260. IEEE, feb 2011.
- [152] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. Vectorizing database column scans with complex predicates. In *AMDS Workshop*, pages 1–12, 2013.
- [153] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.
- [154] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: the architecture and design of a database processing unit. In *ASPLOS Conference*, pages 255–268, 2014.
- [155] Roland E. Wunderlich, Tom F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 84–95. IEEE Comput. Soc, 2003.
- [156] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. Reducing GPU Address Translation Overhead with Virtual Caching. Technical Report Tech Report TR-1842, Computer Science Dept., University of Wisconsin–Madison, Dec. 2016.

- [157] Vitaly Zakharenko, Tor Aamodt, and Andreas Moshovos. Characterizing the Performance Benefits of Fused CPU/GPU Systems Using FusionSim. In *DATE '13*, 2013.
- [158] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1–1, 2015.
- [159] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD Conference*, pages 145–156, 2002.