SHARED RESOURCE MANAGEMENT FOR EFFICIENT HETEROGENEOUS COMPUTING

A Dissertation
Presented to
The Academic Faculty

by

Jaekyu Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
December 2013

SHARED RESOURCE MANAGEMENT FOR EFFICIENT HETEROGENEOUS COMPUTING

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Richard W. Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: July 11, 2013

To my wife Sijung and our soon-to-be-born son

ACKNOWLEDGEMENTS

First of all, I would like to thank God for allowing me to be in the right place with the right people to complete my Ph.D. study.

I would like to express my deepest love and gratitude to my wife, Sijung. I cannot imagine how my life as a Ph.D. student would have been without her. Her dedication, encouragement, and endurance helped me to complete everything that I have achieved in recent years. Moreover, I thank my wife for sharing with me every moment of the last 12 years. Also, we would like to welcome our soon-to-be-born son. My parents, Hoo Geug Lee and Young Ran Park, and parents-in-law, Jong Hae Ryu and Wey Sook Kang, are always supportive and I would like to express my profound gratitude to my family.

I would like to thank my advisor, Dr. Hyesoon Kim. This dissertation and myself as a computer architect could not have existed without her guidance. When I started graduate school, I was not sure what I wanted to pursue. Then, Dr. Kim gave me an opportunity to work with her. She taught me how to perform academic research, how to write a strong paper, and how to express and present my ideas. She always motivated me with her enthusiasm, guided me to the right direction, and made me interact and collaborate with many other people. Under her guidance, I was able to develop my skills and knowledge as a computer architect.

I thank Dr. Sudhakar Yalamanchili, Dr. Richard Vuduc, Dr. Santosh Pande, and Dr. Moinuddin Qureshi for serving on my dissertation committee and providing me constructive and valuable comments to improve the dissertation. In particular, I thank Dr. Vuduc for advising me on my early prefetching work and thank Dr.

Yalamanchili for my recent on-chip network work.

Many former and current HPArch members contributed to this dissertation. In particular, I thank:

- Nagesh Bangalore Lakshminarayana for sharing this long journey with me. We started our graduate program at the same time and went through all the same processes. We struggled and spent much time together on developing MacSim simulator. I have learned much Linux-related knowledge from him and have enjoyed discussions with him on our work.
- Minjang Kim for giving me advice on programming skills, the job search process, and many other different issues.
- Sunpyo Hong for discussions on many work-related and non work-related topics.
- Jaewoong Sim for intense technical discussions on various topics in recent years.
- Joo Hwan Lee and Hyojong Kim for interacting with me in the late stage of my study.
- Nimit Nigania for helping me on my first project during his internship at Georgia Tech in 2009.

I want to express my sincere gratitude to Dr. Chang Joo Lee, Anwar Rohillah, and Dr. Dong Hyuk Woo for their mentor-ship during my internship at Intel Corporation in Austin, TX and Santa Clara, CA. They helped me on various subjects not only during my internships but also during my job search. Interactions with them helped me to decide on my direction after the school.

Finally, I am grateful to my friends who helped me directly or indirectly on my dissertation. Conversations with them refreshed and energized me to continue my work. I thank Ilhyeon and Seokjun for our trips in the U.S.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 The Problem: Resource Sharing in Heterogeneous CMPs

The demand for more computational power never ends. Traditionally, growth in computational power was carried out by ever-increasing clock frequency until the power wall was hit. To circumvent this barrier, chip multiprocessors (CMPs) were introduced and the number of cores keeps increasing. As the technology scales and manufacturers can put more features in a single chip, the next performance enhancement will be brought by heterogeneous architectures where a certain type of architecture is more power efficient at a subset of tasks. GPU is one such example that is more power efficient for tasks involving massive data and thread-level parallelism. Incorporating a GPU architecture into CMPs is the next logical step, and this architecture is becoming mainstream, as can be seen in a wide spectrum of computing platforms from system-on-chip (SoC) architectures [107, 117] to desktop and low-end server processors, including Intel's Sandy Bridge [52] and Ivy Bridge [49], AMD's accelerated processing units (APU) [5], and NVIDIA's Denver project [106]. In this architecture, GPUs are now integrated on top of the conventional CMPs and their memory hierarchy. Figure 1 depicts an example of such an architecture. In this figure, CPU and GPU cores share last-level caches, on-chip interconnection network, and memory controllers. As a result, this architecture creates new problems and challenges in system resource management because of the sharing between heterogeneous cores. This problem does not exist with discrete GPU systems [6, 105] since CPUs and GPUs have separate physical memory space (cache and off-chip DRAM memory). However, in heterogeneous

CMPs (HCMPs), most system resources are shared between processors.

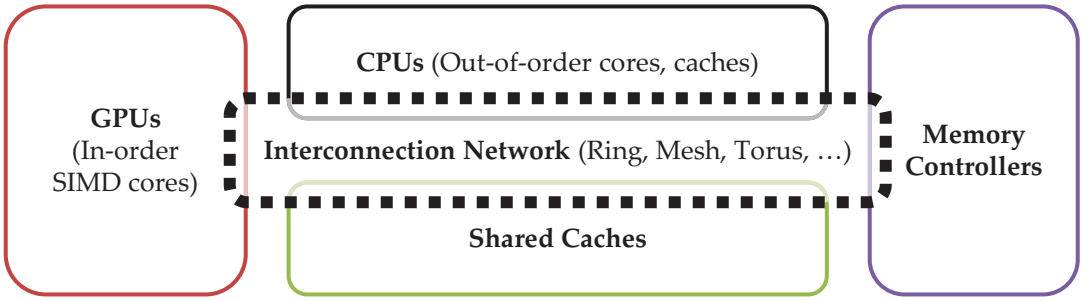


Figure 1: Heterogeneous chip multi-processors (HCMPs).

The resource sharing problem has existed since CMP was introduced. Consequently, many researchers have proposed various resource sharing mechanisms on last-level caches [57, 58, 118, 120, 148, 149], interconnection networks [24, 25, 40, 71], and memory controllers [66, 67, 97, 98]. However, in HCMPs, shared resource management is more challenging due to the different nature of CPU and GPU cores. Typical features of modern high-performance CPU cores include multi-wide superscalar and out-of-order cores. To reduce the penalty of the branch instructions, novel and often power-intensive branch prediction mechanisms are implemented. Large private caches (L1 and L2) as well as aggressive data prefetching mechanisms [62, 99, 129] are often employed to avoid long-latency accesses to off-chip memory. These cores are ideal for serial execution with a small number of threads (1 to 4-way simultaneous multi-threading (SMT)), so they have limited thread level parallelism (TLP).

On the other hand, GPUs use in-order cores and pack more processing elements in each core because integrating more computing units on a given space improves the performance of an overall GPU chip better than allocating a part of the die space to a large cache. With multiple processing elements, each GPU core runs under single-instruction multiple-data (SIMD) execution. As opposed to conventional SIMD processors, multiple threads across cores execute the same

instruction with different data sets in GPU, which is called the single-instruction multiple-thread (SIMT) model. When branch directions within a batch of threads[1] diverge, the execution of each branch path is serialized. Currently, no branch prediction mechanism exists to reduce the penalty of branch instructions. To tolerate memory latencies, GPU cores utilize massive multi-threading. When a thread is stalled due to the long-latency memory instruction, the execution is switched to other available threads. Since GPU cores are designed to pay zero context-switching overhead, this can happen on every instruction issued. To support so many contexts, GPUs have a huge register file, for example a 256 KB register file in NVIDIA Kepler [105] and AMD's GCN [6]. Additionally, GPUs are afforded single-cycle access to massive register files. Due to the high degree of TLP, GPUs coalesce memory requests to reduce memory traffic when possible [109]. Some GPGPU applications have frequent scatter-gather memory operations that hurt performance due to unaligned memory accesses. To mitigate this costly operation, GPUs often have special hardware to support the scatter-gather operation. Table 1 compares the different characteristics between CPU and GPU cores.

Table 1: Comparison between CPU and GPU cores.

	CPU	GPU
Core	out-of-order, superscalar	in-order SIMD
Branch Predictor (BP)	2-level BP, perceptron [60]	no BP
TLP	1-4 way SMT	abundant
Memory	Latency-limited	Bandwidth-limited
Latency tolerance	Caching, prefetching	Caching, multi-threading
Miscellany		Scatter-gather operation

These different characteristics of heterogeneous cores create different aspects of resource sharing problems compared to homogeneous CMPs. First, a significant

[1]This term is called as a warp, wavefront, or EU thread in NVIDIA, AMD, or Intel GPU, respectively.

interference problem occurs in HCMPs. GPUs can produce an excessive number of memory requests at a given period with little processor stalls thanks to GPU's multi-threading capability with very cheap context switching. In addition, SIMD execution of GPU cores often creates unaligned or uncoalesced accesses, which results in multiple transactions from a single memory instruction. As a result, this causes uneven sharing of system resources between cores, and CPUs are unnecessarily penalized.

Moreover, high TLP in GPUs often obfuscates the performance metrics used in the previous mechanisms for conventional CMPs. For example, the cache hit ratio strongly correlates with the performance of CPUs. When the cache hit ratio is improved by using a larger cache or a better cache insertion/replacement policy, the performance of CPU applications improves as well in most cases. Therefore, the cache hit ratio can be a good proxy for performance and many previous mechanisms utilize it. However, this is not the case for GPGPU applications due to the effect of TLP. Even though a GPU core suffers from many cache misses, the core can tolerate extra off-chip memory latencies if it can hold enough threads for continuous execution. Consequently, resource sharing mechanisms in HCMPs, to be effective, should now consider the effect of TLP.

Therefore, in order to solve the resource sharing problem in HCMPs, this thesis presents several efficient resource sharing mechanisms, including shared caches, on-chip network, and dynamic frequency control mechanism, that are aware of the heterogeneity of cores and exploit the different characteristics of CPUs and GPUs for HCMPs.

1.2 *The Solution: Heterogeneity-aware Shared Resource Management*

Among all shared resources, we consider two important resources in this thesis, last-level cache (Chapters IV and VII) and on-chip interconnection network (Chapter V), to tackle resource sharing problems in heterogeneous CMPs. In addition, we present a dynamic frequency regulating mechanism (Chapter VI) that controls the clock frequency of CPU, GPU cores, and the memory (on-chip network and last-level caches) to simultaneously achieve performance improvements and energy efficiency.

TLP-aware shared cache management As described in Section 1.1, cache metrics used in previous mechanisms often mislead the performance behavior of a core or an application due to the effect of TLP. Therefore, in order to see the performance impact of a certain policy, we need to collect performance metrics directly from cores to take into consideration the effect of TLP, instead of relying on indirect and less accurate metrics, such as the cache hit ratio. To this end, a *core sampling* mechanism is proposed. By exploiting the symmetric behavior across GPU cores due to their single-program multiple-data (SPMD) execution model, we can sample cores with different cache policies. If the performance variance of sampled cores is not negligible by different cache policies, we can identify that the cache policy can have a significant impact on performance. In addition, to prevent a significant interference from GPGPU applications, a *cache block lifetime normalization* mechanism is proposed. TAP consists of these two mechanisms and we apply TAP to two previous cache mechanisms, utility-based cache partitioning (UCP) [120] and re-reference interval prediction (RRIP) [58]. These extensions are called TAP-UCP and TAP-RRIP, respectively.

Adaptive virtual channel partitioning for on-chip network In order to provide the quality-of-service (QoS) for network packets, researchers have worked on various aspects of on-chip networks, for example, how to arbitrate packets in routers [24, 25] or how to control injections from source nodes [18, 40, 71]. However, these are not sufficient to resolve the network contention in HCMPs because GPU packets now overflow not only in on-chip router buffers, but also in the injection queues of shared routers (last-level cache tiles and memory controllers). Unless a mechanism manages injection queues and router buffers simultaneously, its effectiveness will be limited. Therefore, an adaptive *virtual channel partitioning (VCP)* mechanism is proposed [73]. A router typically has multiple virtual channels for each port. VCP partitions virtual channels to CPUs and GPUs and controls injections from shared routers based on the VC availability of a corresponding type. VCP utilizes dynamically allocated multiple-queue (DAMQ) [134] for separate injection queues. To find the best partitioning configuration, VCP samples the performance of an application with different partitioning configurations and collects metrics directly from cores, instead of using indirect metrics.

Dynamic frequency control mechanism for efficient resource sharing Although previous mechanisms can be effective for resolving the resource contention problem, they are not designed to improve energy efficiency unless combined with a any power-saving technique. The proposed mechanism, a dynamic frequency regulating mechanism, tries to achieve performance improvement while improving energy efficiency. In HCMPs, CPUs and GPUs have different operating frequencies, i.e., there are separate voltage/frequency domains for different components, and recent processors can dynamically control the frequency of cores based on their utilization to reduce power consumption or improve performance

by taking power from idle cores. Different core frequencies can affect the performance of individual cores as well as resource contention in the system, thereby affecting system throughput. Based on the application type, in particular memory-intensity, performance scalability and performance/power efficiency can be varied. The proposed mechanism tries to find optimal operating frequencies that consider the frequency-scalability of applications and mitigate the interference for cores and memories while not exceeding the chip power budget.

Region-aware energy-efficient cache design for GPU In HCMPs, compared to discrete GPU systems, much larger last-level caches are available to GPU cores due to the GPU integration to CMPs, but the cache is not optimized for GPU cores. Therefore, GPUs may not utilize the cache in an energy-efficient manner. Therefore, we propose a GPU region-aware energy-efficient cache, or GREEN cache. For more efficient parallel execution, GPUs inevitably use stricter and less complex execution and memory models. Also, programmers are asked to provide more information on a program to the device. For example, memory variables in GPU kernels are allocated and mapped from the CPU host code. A programmer should provide the size of the variable as well as other properties such as read-only, write-only, or read-write. From such information, GPU hardware can easily estimate the working set size of a kernel, so unnecessary leakage energy consumption on caches can be reduced by turning-off some cache ways. Moreover, each variable shows distinct cache behavior (for example cache hit ratio) from other variables, while the cache behavior of all instructions that belong to the same variable is near constant. We can exploit this characteristic to save dynamic cache energies by selectively caching (or bypassing) since caching does not help improve performance for a memory variable that does not have any cache hit. In addition, by excluding the size of variables that are set to bypass, we can estimate the

working set size more precisely while preventing interference from those variables.

1.3 Thesis Statement

Efficient shared resource management can improve the performance of the heterogeneous system by considering the heterogeneity of cores and isolating the interference by GPUs.

1.4 Organization

The remainder of this document is organized as follows: Chapter II describes the motivation of the thesis. Chapter III summarizes the related work. Chapter IV presents an efficient cache sharing mechanism. Chapter V describes virtual channel partitioning for on-chip interconnection network. Chapter VI proposes a dynamic frequency regulating mechanism. Chapter VII describes a region-aware energy-efficient cache mechanisms for GPUs. Finally, Chapter VIII concludes the thesis and identifies future research directions.

CHAPTER II

MOTIVATIONS

Sharing system resources in CMPs causes inter-application interference problems. To address these problems, many researchers have been working on this domain, which can be categorized broadly into three topics: shared caches, interconnection networks[1], and memory controllers. Previous mechanisms show effectiveness in CMPs, but they encounter different aspects of problems in HCMPs. This chapter explains those problems and provides the motivation for this thesis.

2.1 Interference Experienced by CPU Applications

The inter-application interference problem has existed even in homogeneous CMPs. However, the problem becomes more complicated and severe due to the heterogeneity of cores. Since GPU cores are capable of running more threads concurrently with SIMD executions, they will generate many more memory requests than CPUs. Also, multi-threading capability enables GPUs to generate continuous memory requests without processor stalls. Since CPU and GPU requests have to compete in shared resources, resource contentions will occur in shared caches, interconnection networks, and memory controllers. As a result, more demanding GPGPU applications will significantly interfere with CPU applications compared to homogeneous CMP workloads.

In order to see the interference of CPU applications in HCMPs, we conduct experiments with a single-threaded CPU application from SPEC 2006 (`perlbench`, `bzip2`, `gcc`, `cactusADM`, and `leslie3d`) along with a GPGPU application

[1]We interchangeably use the term on-chip interconnection network and the network on chip (NoC) throughout the thesis.

(`streamcluster`, `lbm`, and `spmv`) that is running on six SIMD cores. Figure 2 shows the slowdown of CPU applications when they are running with a GPGPU application, where *slowdown* is defined in Eq. (1).

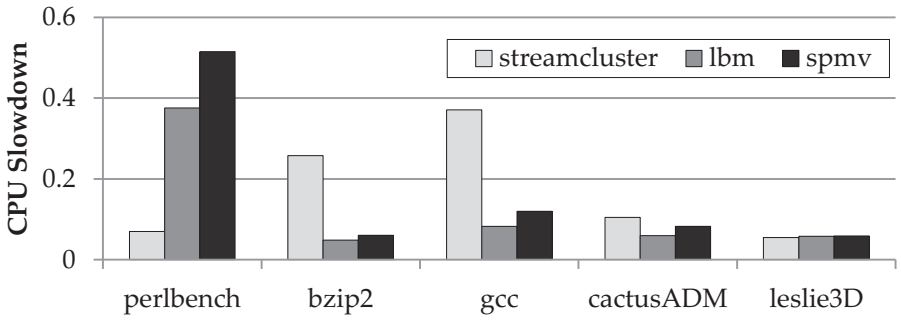


Figure 2: Slowdown of CPU applications caused by a GPU application (x-axis: CPU benchmarks).

$$slowdown = \frac{IPC_{with\_gpu}}{IPC_{alone}} \tag{1}$$

A slowdown of 0.1 indicates that the IPC becomes only 10% compared to when the CPU application is running alone, i.e., a 10 times slowdown. As shown in the figure, we can observe that a significant performance degradation exists due to the interference caused by the GPGPU application. Although we run only one single-threaded CPU application in this experiment, we expect even more slowdown when more CPU applications are running concurrently.

2.2 Cache Sharing between CPUs and GPUs

The baseline hardware cache uses the least recently used (LRU)-approximation replacement policy. Consequently, the cache favors an application that has more cache accesses regardless of cache utilization, i.e., how many hits are serviced from the cache for the application. To solve the contention problem in the cache, two representative approaches exist: one is cache partitioning [120, 131, 132] and the

other is dynamic cache insertions [57,58,118,148,149]. Cache partitioning dedicates a few cache ways to each application so that the cache space for an application cannot be invaded by accesses from other applications. On the other hand, dynamic cache insertions try to identify the best insertion position for applications. By varying the insertion position for each application, they can prevent the cache interference problem.

However, these previous mechanisms might not be effective if they do not consider the different characteristics of GPU cores, in particular rich thread-level parallelism. Figure 3 shows the cycles per instruction (CPI) and misses per kilo instruction (MPKI) changes for conventional CPU applications as the cache size increases. As shown in the figure, CPI strongly correlates to MPKI. In other words, improvements in the cache hit ratio leads to performance improvements. CPU applications are less tolerable to off-chip memory access penalties, so better cache performance tends to yield better system performance.

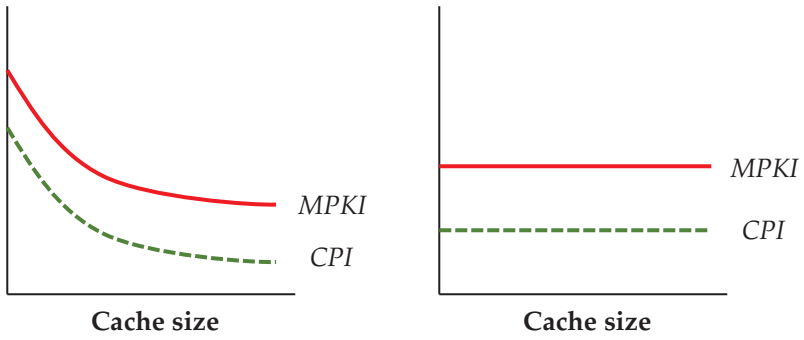


Figure 3: Conventional cache behavior without TLP.

On the other hand, some GPGPU applications show unconventional cache behavior. For example, in Figure 4, even though MPKI decreases as the cache size increases, it does not lead to better performance. Since TLP is so effective in this case, off-chip access latencies can be tolerated.

Considering the interference caused by GPU cores and the effect of TLP in

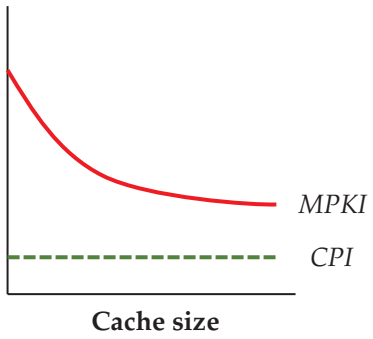


Figure 4: Unconventional cache behavior with TLP.

GPUs, we summarize how previous mechanisms will behave when conventional CPU applications share caches with GPU applications as follows:

- LRU cache - a new cache block is always inserted at the most recently used (MRU) position. Upon a cache hit, the block is again moved to the MRU position. The block will be replaced by the incoming block when it is in the LRU position. As a result, an application with a higher number of cache accesses will occupy more cache space, thereby being favored. In HCMPs, cache accesses from GPU applications will be heavily favored under the LRU cache.

- Dynamic cache partitioning mechanism - most cache partitioning mechanisms are based on the cache related metric, such as the number of cache hits. They give more cache space to applications that tend to have more cache hits. In HCMPs, if a GPU application has a significantly larger number of hits than a CPU application, then it will be highly favored over CPU applications, regardless of how cache affects performance.

- Dynamic insertion policy - most previous mechanisms can identify streaming (or thrashing) applications and isolate accesses from those applications in the limited cache space. However, when both applications have a decent

cache hit ratio, similar to LRU cache, applications with more frequent accesses will be favored, which is GPU applications.

As a result, previous mechanisms that are based on cache-related metrics, e.g., hit ratio, cannot identify the effect of TLP. Thus, we need a mechanism that collects performance metrics directly from cores to identify the effect of TLP.

2.3 Interference in the Network

2.3.1 Importance of On-chip Network

All shared resources, such as last-level caches, interconnection networks, memory controllers and DRAM memories, are important and can be a source of inter-application interference. All these resources are closely related to each other. For example, shared cache affects network pattern and the number of DRAM accesses. Off-chip DRAM accesses consume a significant amount of time, and different DRAM scheduling policies will affect cache hit ratio and network pattern. Finally, how the on-chip network is coordinated will change cache and DRAM access sequences. Among others, the on-chip interconnection network plays a very significant role in the system by connecting all components and governing access sequences in caches and DRAM controllers. Other than private cache accesses, *all communications are made through some kind of the on-chip network*, so memory traffic spends a significant amount of time in the network.

Figure 5 shows the latency distribution of packets of workloads that consist of CPU applications and one memory-intensive GPU application.[2] We estimate latencies in the following categories.

- CACHE: cycles to access the LLC including delays in a queue.
- DRAM: cycles in DRAM controllers to access off-chip DRAM.

[2] *W-HH* and *W-LH* workloads in Table 13.

- NOC_QUEUE: queuing delay in the injection buffer.
- NOC_TRIP: traverse time to reach a destination after injected into the network from the injection buffer.

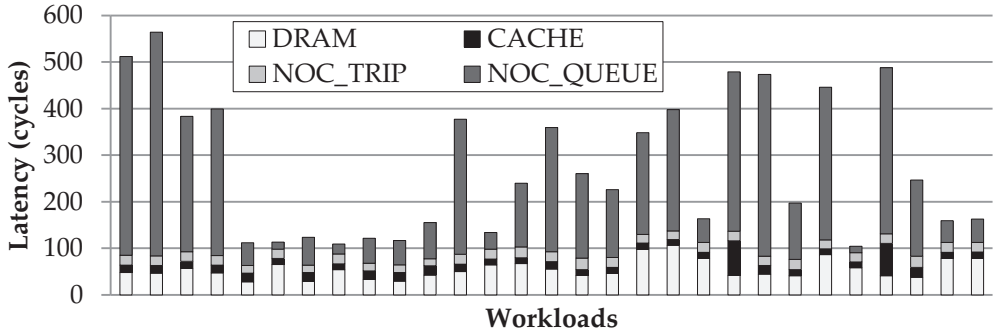


Figure 5: Latency distribution of packets in heterogeneous workloads (x-axis: workloads).

Although the DRAM waiting time accounts for the majority of time in some workloads, the NoC usually consumes most of the time, in particular due to queuing delays in shared routers (LLCs and memory controllers). We can expect that the time spent on the network will increase as the number of cores increases because of increased hop counts and traffic, so the importance of the NoC remains the same in the future.

When we compare the average queuing delays of CPU and GPU packets in heterogeneous workloads, we observe CPU packets experience much longer queuing delays than GPU packets. Figure 6 shows the router buffer occupancy of CPU and GPU packets. As can be seen, GPU packets mostly occupy buffer space. This will cause network interference problem and CPU packets will suffer from the interference.

Consequently, we need a mechanism that isolates the network interference caused by GPU cores in HCMPs.

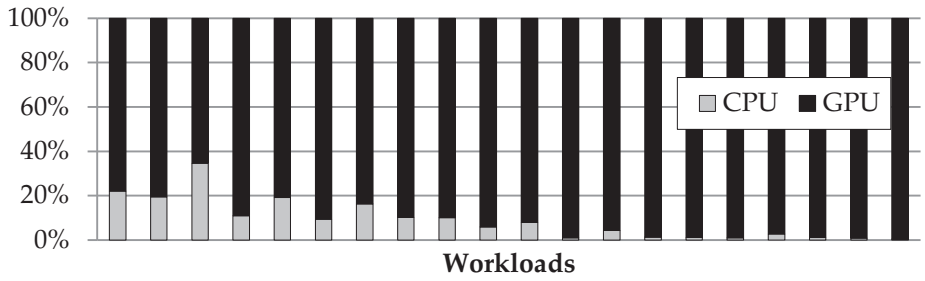


Figure 6: Router buffer occupancy in heterogeneous workloads (x-axis: workloads).

2.3.2 Effectiveness of Previous Mechanisms in HCMPs

Many researchers have conducted research on various aspects of NoC. However, there are a few reasons why previous proposals may be less effective in CPU-GPU heterogeneous architectures compared to homogeneous systems due to the existence of GPU cores. First, most mechanisms only consider arbitrations of packets in a router. This is natural in homogeneous systems because we can expect that a similar number of packets from each application exist in the injection queues. However, due to bursty injections by GPUs, the occupancy of injection queues in shared resources is likely to be skewed such that GPU packets occupy most queue entries. Therefore, the effectiveness of previous mechanisms will be limited. By having separate injection queues for a CPU and GPU or an out-of-order packet scheduler, previous mechanisms can work better, but this will increase the complexity of the scheduler. The scheduler now needs to decide which queue (separate queues) or packet (out-of-order scheduler) to schedule and the decision made by the scheduler should be incorporated with arbitration decisions.

However, even if the previous mechanisms consider separate injection queues, the QoS for NoC needs to consider different characteristics of CPU and GPU cores. Since GPU cores can execute more concurrently running threads, they have higher thread-level parallelism (TLP) and their ability to tolerate latency is

different compared to CPU cores.

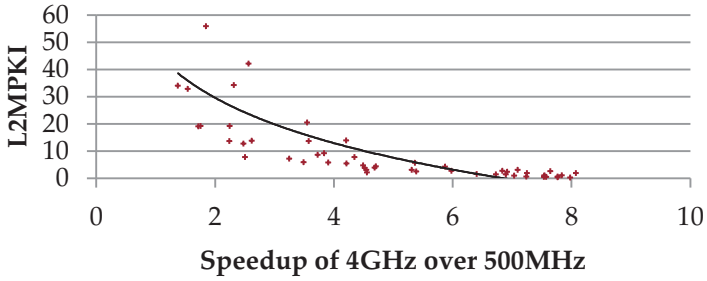
As a result, the different nature of cores makes it difficult to apply previous mechanisms in HCMPs. Therefore, QoS mechanisms for heterogeneous architectures need to have separate queues/out-of-order packet schedulers and to consider the nature of GPU cores to be more effective.

2.4 Motivation for Dynamic Frequency Regulating Mechanism

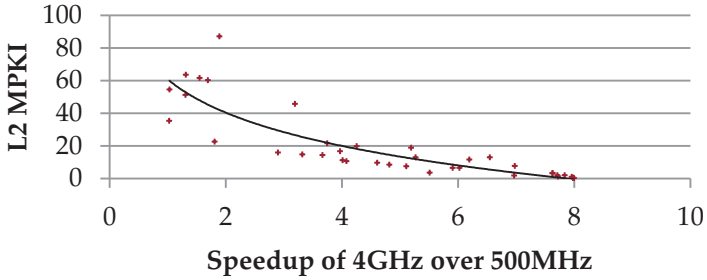
2.4.1 Performance Scalability by Frequency and MPKI

Traditionally, performance improvements were carried by higher clock frequency along with advanced microarchitecture features until the power wall became the main limiter. However, higher frequency does not always guarantee higher performance, i.e., applications have different performance scalability with frequency increase. Based on the characteristics of the application, we may see proportional performance improvements (linear scalability) as well as saturated improvements (log scalability). Figure 7 shows an XY graph that correlates speedup trend as the frequency of cores increases from 500 MHz to 4 GHz and L2 MPKI (misses per kilo instructions) of CPU and GPU applications.

We pick L2 MPKI since it is one of the application characteristics and will not dramatically change with frequency changes and interactions with other applications. From the figure, we can easily observe that L2 MPKI and the speedup have a strong correlation in CPU and GPU applications, where applications with high MPKI show very limited speedup results, while applications with low MPKI show close to linear speedup. Therefore, we can form a simple relation between MPKI and scalability as in Eq. (2). If MPKI is less than a certain threshold, we expect very good scalability. Otherwise, we will observe marginal performance improvement with frequency increase.



(a) CPU applications



(b) GPU applications (L2 MPKI is per-core)

Figure 7: Speedup pattern of applications with frequency increase.

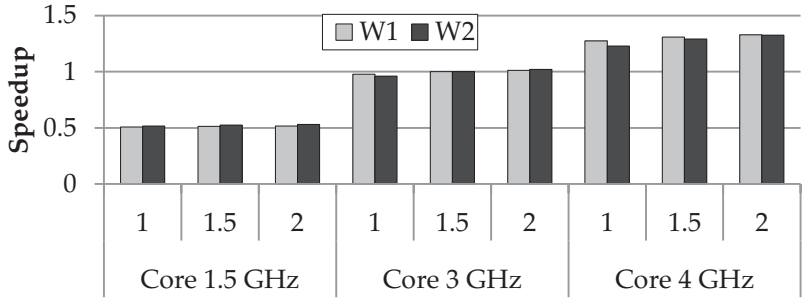
$$\begin{aligned} \textbf{if } MPKI < \theta_{MPKI} &: Performance \propto frequency \\ \textbf{else} &: Performance \propto \log(frequency) \end{aligned} \tag{2}$$

At the same time, L2 MPKI can also be a good proxy for indicating the degree of interference caused by an application. Typically, a network packet is created when a cache miss needs to be serviced from remote places such as shared cache tiles or off-chip memories. A higher MPKI in a given period indicates more memory request injections to the shared resources. Moreover, operating clock frequency is another factor to determine the number of total requests under a system that is capable of performing DVFS. Although MPKI will be similar regardless of frequency change, the number of total memory requests in a given period will be proportional to the frequency unless the system bandwidth is saturated. Therefore, we can formulate the number of memory requests and the interference as in Eq. (3).

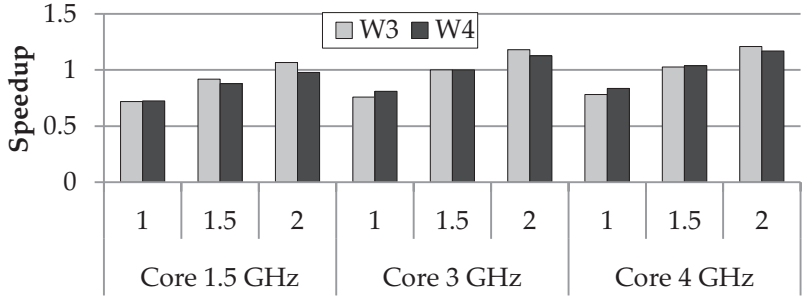
$$interference = total\_req\_per\_time \propto MPKI \times frequency \quad (3)$$

2.4.2 Effect of Core and Memory Frequency

In this section, we examine how different combinations of core and memory system frequency affect system performance based on workloads with different memory intensity. Figure 8 shows the results. Note that `W1` and `W2` consist of four compute-intensive CPU applications and `W3` and `W4` consist of four memory-intensive CPU applications. The detailed evaluation methodology can be found in Section 6.4.



(a) Compute-intensive workloads



(b) Memory-intensive workloads

Figure 8: Performance of different core and memory frequency combinations (memory: 1, 1.5, 2GHz).

We can clearly see two trends from the figures: 1) Memory frequency does not affect the performance of compute-intensive workloads and 2) core frequency does not have a significant impact on the performance of memory-intensive workloads.

When the memory system is busy, the performance bottleneck occurs in the memory, so cores will have more stall cycles by waiting until memory requests are serviced. In this case, lowering the frequency of cores and increasing the frequency of the memory system will yield significant performance improvements while not consuming more energy. In the opposite case, when the memory system is not busy, cores consume more cycles on computations rather than waiting for the memory requests. As a result, faster cores with a slower memory system will yield better performance. Ma et al. [88] have discussed this in the past for CMP workloads, and they utilized the core and memory performance behaviors to partition the power budget between the core and the memory.

2.4.3 Previous Resource Sharing Mechanisms

Previous mechanisms that aim to solve the resource contention problem have some weaknesses. First, previous mechanisms have limited effectiveness on workloads that consist of all compute-intensive applications. The basic intuition of shared resource management is trying to reduce interference caused by memory-intensive applications and prioritize more critical applications. Therefore, they are effective only when a significant resource contention or interference exists. However, with compute-intensive workloads, they perform similarly to the baseline at best. Even, we can observe performance degradation due to the overhead of mechanisms.

Second, previous mechanisms are not designed to optimize the power-efficiency. They focus on improving performance or providing quality-of-service to applications. To achieve the goal, they often introduce new hardware structures, which consumes extra power. Overall energy consumption can be decreased, which is driven by the performance improvement. For example, source throttling has been considered to prevent inter-application interference problem. When a core injects too many packets to the network in a given time, memory requests

from this core will be likely to interfere with other applications. One effective way to solve this problem is limiting the number of packet injections to the network for a period. Chang et al. [18] recently proposed a heterogeneous adaptive throttling mechanism (HAT). They monitored the MPKI of each application and made throttling decisions based on the monitored MPKI value. By utilizing application awareness, HAT showed effectiveness. However, these source-throttling-based mechanisms have limitations. When only source throttling is applied, it can prevent interference, but core resources must idle when the core exceeds packets more than its quota. In order to save energy, DVFS can be applied together with power (or clock) gating idle components. However, improving performance of an application is difficult with source throttling unless core clock frequency is increased.

2.5 Motivation for Energy-Efficient Cache for GPU

Having a large cache is a conventional wisdom for reducing the speed gap between faster cores and slower memory by placing data blocks closer to the processor. The size of on-chip caches continues to increase, for example Intel's latest Haswell architecture [48] put up to 32 MB on-chip caches. On the other hand, GPUs utilize the cache differently. In the discrete GPU system, caches are mainly used to reduce bandwidth to the memory, rather than to decrease memory access latencies. Therefore, cache size is much smaller than in CPUs. For example, the first generation of NVIDIA's GPGPU architecture [104] does not have hardware-managed caches. The cache size increased to 768 KB in the next generation [103] and to 1536 KB in the latest generation [105]. However, as GPUs are now integrated into CMPs, large caches are now available to GPUs, where the cache is optimized for CPUs, not for GPUs.

To utilize caches in a more energy-efficient manner for GPUs, we can exploit

the execution and programming models of GPUs. First, GPUs adopt the single-program multiple-data (SPMD) execution model to better achieve parallelism. Multiple processors execute the same code with different input data. Combined with the massive multi-threading capability of GPUs, we can find very regular behavior across cores, i.e., similar progress is observed among different cores. One interesting behavior that we can observe due to the SPMD model is that cache behavior, such as cache hit ratio, in adjacent memory space shows near constant behavior, while distance memory regions show distinct behaviors. We can exploit this behavior by selectively caching useful data blocks only to save dynamic cache energies.

Second, programmers are asked to provide detailed information on the program in a host code. One type of information is the size of each memory variable in a kernel. Memory variables in a GPU kernel are persistent throughout the kernel execution and the property of the variable (size and readability) is dictated by the programmer and the modification (dynamic allocation and deallocation) is very limited. Using this information, we can precisely estimate the working set size of a GPU kernel, so that we can save leakage energies by turning off unnecessary cache ways.

These two cache optimizations that exploit the execution and programming models of GPUs can be a power-efficient method of utilizing large on-chip caches for GPUs.

CHAPTER III

RELATED WORK

This chapter discusses the related work.

3.1 Related Work on Heterogeneous Architecture

3.1.1 Resource sharing mechanism

In this section, we first discuss resource sharing mechanisms that specifically target CPU-GPU heterogeneous architectures. Lee and Kim [72] studied the cache-sharing behaviors in heterogeneous workloads and proposed TLP-aware cache management schemes, which sample cores with different cache policies to see the performance effects by caches. They also considered the interference problem caused by GPU applications.

Yang et al. [152] proposed a pre-execution mechanism of GPGPU applications on CPU cores. The proposed mechanism automatically extracts memory operations of the GPGPU kernel and dispatches these operations on the CPU when the kernel is launched. Pre-execution from CPU cores brings data blocks of GPGPU kernels in the shared cache, so most off-chip accesses from GPGPU applications are hit in the cache.

Jeong et al. [59] considered quality-of-service (QoS) in a multi-processor system-on-chip when off-chip bandwidth is shared between CPU and real-time constrained graphics applications. The proposed mechanism adaptively prioritizes CPU and GPU requests based on the progress made by graphics applications. Ausavarungnirun et al. [9] proposed the staged memory scheduler (SMS). Due to massive memory accesses by GPU cores, the visibility of the memory requests by the memory scheduler is very limited. SMS attacks this problem with a

multiple-stage memory scheduler. In the first stage, requests from the same source are inserted into the same queue and form a batch based on the row buffer locality. Then, a batch scheduler in the second stage picks an application batch based on the application characteristics and requirements. A scheduler in the final stage issues a ready DRAM command.

3.1.2 Task partitioning mechanism

Many researchers have also focused on how to partition or schedule tasks between heterogeneous cores [8, 34, 38, 56, 61, 82, 85, 115, 143]. As the heterogeneous architecture becomes the mainstream computing platform, frameworks such as Open Computing Language (OpenCL) [111] have been proposed to simultaneously utilize heterogeneous cores for the same program or kernel. Based on the task, running the program on a certain type of core yields better performance, but utilizing more types of cores is beneficial in terms of performance, power, and heat dissipation. As a result, task partitioning becomes a very important yet complex problem to tackle.

3.2 Related Work on Cache Sharing

3.2.1 Dynamic cache partitioning

Suh et al. [131, 132] first proposed dynamic cache partitioning schemes in chip multi-processors that consider the cache utility (number of cache hits) using a set of in-cache counters to estimate the cache-miss rate as a function of cache size. However, since the utility information is acquired within a cache, information for an application cannot be isolated from other applications' intervention.

Utility-based cache partitioning (UCP) [120] addressed this problem by proposing a utility monitor (UMON) that uses separate structures, including ATD (auxiliary tag directory) and way counters. ATD maintains strict LRU-stack per application. Upon hits in ATD, the corresponding way counters will be

incremented. The optimal partition for all applications is determined to maximize the overall number of cache hits.

Kim et al. [65] considered the fairness problem from cache sharing such that slowdown due to the cache sharing is uniform to all applications. Moretó et al. [96] proposed MLP-aware cache partitioning, where the number of overlapped misses will decide the priority of each cache miss, so misses with less MLP will have a higher priority. IPC-based cache partitioning [133] considered performance as the miss rate varies. Even though the cache-miss rate is strongly related to performance, it does not always match the performance. However, since the baseline performance model is again based on the miss rate and its penalty, it cannot distinguish GPGPU-specific characteristics. Yu and Petrov [153] considered bandwidth reduction through cache partitioning. Srikantaiah et al. proposed the SHARP control architecture [128] to provide QoS while achieving good cache space utilization. Based on the cache performance model, each application estimates the cache requirement and central controllers collect this information and coordinate requirements from all applications. Liu et al. [83] considered an off-chip bandwidth partitioning mechanism on top of cache partitioning mechanisms.

3.2.2 LLC policies by application level management

TADIP [57] is a dynamic insertion policy (DIP) that dynamically identifies the application characteristic and inserts single-use blocks (dead on fill) in the LRU position to evict as early as possible. PIPP [150] pseudo partitions cache space to each application by having a different insert position for each application, which is determined using a utility monitor as in UCP. Upon hits, each block is promoted toward the MRU by one position. PIPP also considers the streaming behavior of an application. When an application shows streaming behavior, PIPP assigns only one way and allows promotion with a very small probability (1/128).

Pseudo-LIFO [19] mechanisms are a new family of replacement policies based on the fill stack rather than the recency stack of the LRU. The intuition of pseudo-LIFO is that most hits are from the top of the fill stack and the remaining hits are usually from the lower part of the stack. Pseudo-LIFO exploits this behavior by replacing blocks in the upper part of the stack, which are likely to be unused.

Jaleel et al. proposed re-reference interval prediction (RRIP) [58]. Cache replacement policies use some method of future reference prediction. For example, LRU predicts that all caches hits and misses will be re-referenced near-immediate. Dynamic insertion policies detect either a near-immediate or distance re-reference pattern in runtime, but not both at the same time. If an application shows a mixed pattern of temporal and non-temporal data, dynamic policies cannot hold near-immediate blocks in the cache. RRIP solves the problem of this mixed pattern by inserting incoming blocks in the not near-immediate position and promote blocks toward the near-immediate position upon hits. RRIP also uses a dynamic insertion policy to filter out non-temporal accesses. Wu et al. [149] proposed prefetch-aware cache management, which is built on RRIP.

3.3 Related Work on On-chip Interconnection

3.3.1 NoC Research

There has been an extensive amount of work in the past [23, 30] for non-on-chip networks. However, the time scales and the amount of resources available in non-on-chip network environments are much higher than what is permissible/acceptable in an NoC. Therefore, we limit our discussion only to NoC work. A survey paper [14] and a keynote paper [90] laid out practical issues of implementing NoC, their solutions in the literature, and open problems in detail. In this section, we reiterate QoS mechanisms among others and add recent work.

Previous QoS mechanisms can be categorized based on two aspects. First,

based on *whether a mechanism provides guaranteed services*, we can categorize mechanisms into best-effort service (BE) and guaranteed service mechanisms (GS). While hard GS [15, 37, 42, 80, 93, 130, 135, 146] is favorable since it provides predictable outcomes within the tight requirement such as real-time systems, BE [35, 123] can better utilize system resources, thereby improving the system throughput. As a result, many researchers considered hybrid NoC, which combines GS and BE [13, 16, 28].

Second, based on *how QoS is provided*, we can categorize previous mechanisms into 1) resource pre-allocation, 2) prioritization (arbitration), and 3) injection control (source throttling). In resource pre-allocation (or reservation) mechanisms [15, 36, 76, 79, 93], packets are assigned in different traffic classes based on the importance, and NoC resources, including virtual circuits, channels, and buffer space, are reserved for each class.

Priority-based mechanisms [13, 16, 24, 25, 43, 91] are similar to pre-allocation mechanisms since they determine a different priority for each application or packet (i.e., different class in pre-allocation) by estimating criticality from core/application-specific behaviors, including cache misses per instruction and number of miss-predecessor. However, they do not dedicate resources for a certain type and instead rely on arbitrations in various places in the network. In [24, 25], priority is calculated in the centralized logic and each router has the same priority information for all applications. Arbiters of a router schedule packets based on the priority. To prevent the starvation problem, multiple packets often form a batch so that packets in old batches have higher priority than packets in newer batches.

On the other hand, injection (or congestion) control mechanisms [29, 101, 110, 140] try to balance the injections from processing nodes or applications by injecting packets in the pre-defined rate or limiting packet injections. Globally synchronized frame (GSF) allows a limited number of packet injections for each

source in one epoch (or frame) although GSF maintains future frames for handling bursty injections [71]. Each VC is now mapped to a different frame, and router arbiters prioritize packets in older frames. Therefore, GSF can guarantee minimum bandwidth as well as network delay. Grot et al. [40] proposed a preemptive virtual clock (PVC), which uses a virtual clock to track each flow's bandwidth consumption while using frames to reduce the history effect of the virtual clock. PVC also uses the preemption of virtual channels for higher priority packets if lower priority packets occupy the VC so that the priority inversion problem does not occur. A recent proposal by Chang et al. [18] controls injections from each application based on the MPKI (misses per kilo instructions) since MPKI can identify the memory intensity of the application.

3.3.2 Virtual Channel Management Mechanism

In addition to NoC research in the previous section, we also discuss some of the adaptive virtual channel management mechanisms. Virtual channel size and organization can significantly affect the system performance and power consumption [141]. As a result, researchers have tried to find an optimal VC configuration in static or design time based on the characteristics of their target applications and traffic patterns. Also, dynamic buffer management mechanisms are proposed. Choi and Pinkston [21] proposed dynamic VC allocation based on the traffic pattern using virtual channel DAMQs and DAMQs with recruit registers, which are improved DAMQ [134]. Nicopolous et al. [100] proposed ViChaR. The motivation of ViChaR is that the number of virtual channels and the depth of the buffer (based on the size of packet) can significantly affect the performance based on the traffic pattern. Thus, ViChaR optimizes the number of VCs and the depth of the buffer based on the traffic load using a unified buffer. Lai et al. [68] also tried a similar dynamic VC allocation mechanism,

but they considered congestion awareness. Evripidou et al. [31] proposed a VC virtualization mechanism using VC renaming to support arbitrarily large number of VCs. Trivinõ et al. [138] also considered a VC virtualization mechanism as well as NoC resource partitioning mechanism.

3.3.3 Heterogeneous Interconnection Network

We can consider the heterogeneous network to cope with the heterogeneity of cores, so we discuss previous work on heterogeneous on-chip networks in this section.

Mishra et al. [95] proposed HeteroNoC, which asymmetrically allocates resources (buffers and links) to exploit non-uniform demand on a mesh topology. They used two types of routers, small and large, and placed more powerful large routers to congested areas. Grot et al. [39] proposed Kilo-NOC, which isolates shared resources into QoS-enabled regions to minimize the network complexity. While proving QoS for shared resources, Kilo-NOC uses energy-efficient and cost-effective routers for the rest of the network. Bakhoda et al. [10] proposed a throughput-effective NoC for GPU architecture. Due to many cores with a smaller number of memory controllers, a many-to-few traffic pattern is dominant in GPUs. To optimize such traffic, they used a half router, which cannot change the dimension of a packet, to reduce the complexity of the network while increasing the injection bandwidth from the memory controllers to provide burst data read.

3.3.4 NoC Research for GPU Architectures

In this section, we discuss NoC research proposed for GPU architectures. Yuan et al. [154] proposed a complexity-effective memory scheduler for GPU architectures. NoC routers of the proposed mechanism reorder packets to increase row-buffer locality in the memory controllers. As a result, a simple in-order memory scheduler can perform similarly to a much more complex out-of-order scheduler.

Bakhoda et al. [10] proposed a throughput-effective NoC for GPU architectures. Due to many cores with a smaller number of memory controllers, a many-to-few traffic pattern is dominant in GPUs. To optimize such traffic, they used a half router, which cannot change the dimension of a packet, to reduce the complexity of the network while increasing the injection bandwidth from the memory controllers to provide burst data read.

3.4 Related Work on DVFS

Most previous DVFS approaches tried to save power consumption by lowering (or gating) the voltage of idle cores and then they increased the clock frequency of other active cores to improve performance [86, 94, 144]. Modern processors [5, 52, 121, 145] are capable of performing this power optimization.

Li and Martínez [77] proposed power optimization for multi-threaded workloads in CMPs. The power optimization problem for multi-threaded applications is in two dimensions: processor (number of cores) and DVFS level (frequency). Since finding an optimal configuration under different performance requirements for various applications is non-trivial, they tackled the problem by proposing heuristics that try to reduce the search effort while yielding optimal power savings.

Wang et al. [142] proposed a power budget partitioning mechanism for OpenCL applications on a single-chip heterogeneous processor. They proposed a run-time algorithm that determines optimal workload partitioning between CPU and GPU cores, DVFS level, and optimal number of operating cores.

Ma and Wang proposed PGCapping [87], which decouples power-gating and DVFS level while considering the aging of individual cores. By decoupling DVFS and power-gating, the power management algorithm can be less complicated. DVFS often causes a core aging problem when it is intensively applied to a specific

core. PGCapping tackles this problem by monitoring how DVFS was previously applied to cores and trying to apply core power-gating accordingly.

While most approaches have focused on core DVFS only [7, 22, 55, 86, 92, 125, 144, 147], some proposals [32, 41, 74, 88, 126] partition the power budget between cores and uncore (memory and NoC). Ma et al. [88] proposed DPPC, which partitions chip power budget among cores and shared caches. Based on the power-performance model and on-line model estimator, they tried to solve power partitioning as a linear optimization using an on-chip LP solver implementation. Lee et al. [74] analyzed different combinations of clock frequency and number of operating cores to improve the throughput of GPUs within a power budget via dynamic voltage/frequency and core scaling (DVFCS). They also considered adjusting the frequency of the interconnection network based on the application behavior. PEPON [126] is a two-level power budget distribution strategy. In the first level, power is distributed to cores, NoC, and last-level caches based on a regression-based performance model. Then, the power budget is further distributed to individual cores and last-level caches. A performance per watt model is used to assign power budget to individual cores and a utility-based strategy is adopted for caches.

3.5 Related Work on Low-Power Cache

The idea of semantic-aware caching has been studied and even successfully commercialized in various forms. The most common example is having a separate cache for instruction and data, commonly found in the level 1 cache of modern CPUs. On the other hand, a discrete GPU used to have graphics-oriented special-purpose caches including z-cache and color cache. In terms of academic research, Lee et al. proposed a separate cache for stack data and non-stack data [69] to improve instruction-level parallelism of a superscalar processor.

Lee and Ballapuram proposed separate TLBs for stack, global static, and heap data to improve the energy efficiency of a CPU [70]. Ballapuram and Lee also exploited such semantic information to suppress snoop energy consumption in a multi-core processor [11]. Cache bypassing [33, 63, 64, 139] is also widely studied. In these mechanisms, cache blocks that are predicted to be dead on arrival will not be inserted in caches. While not directly related to caching, page-level prefetching [136] for CPU workloads can be considered as well to exploit different characteristics of different regions although it may end up with inaccurate prediction and may require higher overhead due to its fine-grained tracking compared to our memory object-based approach. Note that, due to its extremely high thread-level parallelism, a memory object in a GPGPU program is typically far larger than a page.

Researchers have also proposed various ideas to dynamically resize a cache. To name a few, Albonesi proposed selective cache ways to disable ways when a CPU runs an application with a small memory footprint to save static energy [3]. To maintain data consistency, the author explored two options: flushing the cache or making all ways accessible for coherence requests. Powell et al. proposed Gated-Vdd, in which the authors proposed to gate the supply voltage for unnecessary sets of a cache [116]. Ranganathan et al. proposed reconfigurable caches enabling/disabling ways in a set-associative cache and evaluated their proposals with media workloads [122]. To maintain data consistency, they explored two schemes, cache scrubbing and lazy transitioning. Dhodapkar and Smith proposed detecting the change of program phases, estimating the working set size of a phase, and dynamically reconfiguring the underlying hardware cache [26]. Due to the extreme dynamics of CPU workloads, the authors proposed a rather sophisticated hardware mechanism to estimate the working size of a given application.

CHAPTER IV

AN EFFICIENT CACHE SHARING MECHANISM

4.1 Introduction

This chapter introduces an efficient way of sharing the last-level cache (LLC) between CPUs and GPUs. The LLC is one of the most important shared resources in chip multi-processors (CMP). Managing the LLC significantly affects the performance of each application as well as the overall system throughput. Under the recency-friendly LRU approximations, widely used in modern caches, applications that have high cache demand acquire more cache space. The easiest example of such an application is a streaming application. Even though a streaming application does not require caching due to the lack of data reuse, data from such an application will occupy the entire cache space under LRU when it is running with a non-streaming application. Thus, the performance of a non-streaming application running with a streaming application will be significantly degraded.

To improve the overall performance by intelligently managing caches, researchers have proposed a variety of LLC management mechanisms [19, 57, 58, 65, 120, 128, 150, 151]. These mechanisms try to solve the problem of LRU by either (1) logically partitioning cache ways and dedicating fixed space to each application [65, 120, 128, 151] or (2) filtering out adverse patterns within an application [58, 150]. In logical partitioning mechanisms, the goal is to find the optimal partition that maximizes the system throughput [120, 128, 151] or that provides fairness between applications [65]. On the other hand, the other group of cache mechanisms identifies the dominant pattern within an application and

avoids caching for non-temporal data. This can be done by inserting incoming cache blocks into positions other than the most recently used (MRU) position to enforce a shorter lifetime in the cache.

However, these mechanisms are not likely applicable to CPU-GPU heterogeneous architectures for two reasons. The first reason is that GPGPU applications often tolerate memory latency with massive multi-threading. By having a huge number of threads and continuing to switch to the next available threads, GPGPU applications can hide some of the off-chip access latency. Even though recent GPUs have employed hardware-managed caches [103], caching is merely a secondary remedy. This means that caching becomes effective when the benefit of multi-threading is limited, and increasing the cache hit rate even in memory-intensive applications does not always improve performance in GPGPU applications. The second reason is that CPU and GPGPU applications often have different degrees of cache access frequency. Due to the massive number of threads, it is quite common for GPGPU applications to access caches much more frequently than CPUs do. Since previous cache mechanisms did not usually consider this effect, many policies will favor applications with more frequent accesses or more cache hits, regardless of performance.

To accommodate the unique characteristics of GPGPU applications running on heterogeneous architectures, we need to consider (1) how to identify the relationship between cache behavior and performance for GPGPU applications even with their latency-hiding capability and (2) the difference in cache access rate. Thus, we propose a thread-level parallelism (TLP)-aware cache management policy (TAP). First, we propose *core sampling* that samples GPU cores with different policies. For example, one GPU core uses the MRU insertion policy in the LLC and another GPU core uses the LRU insertion. Performance metrics such as cycles per instruction (CPI) from the cores are periodically compared by the *core sampling*

controller (CSC) to identify the cache friendliness[1] of an application. If different cache policies affect the performance of the GPGPU application significantly, the performance variance between the sampled cores will be significant as well. The second component of TAP is *cache block lifetime normalization* that considers the different degrees in access rate among applications. It enforces a similar cache lifetime to both CPU and GPGPU applications to prevent adverse effects from a GPGPU application that generates excessive accesses.

Inspired by previously proposed utility-based cache partitioning (UCP) and re-reference interval prediction (RRIP) mechanisms, we propose two new mechanisms, TAP-UCP and TAP-RRIP, that consider GPGPU application characteristics in heterogeneous workloads.

4.2 The Problem: Cache Behavior of GPGPU Applications

In this section, we explain the cache behavior of GPGPU applications. First, we classify GPGPU applications based on how the cache affects their performance. Figure 9 shows cycles per instruction (CPI) and misses per kilo instruction (MPKI) variations for all application types as the size of the cache increases. Note that to increase the size of the cache, we fix the number of cache sets (4096 sets) and adjust the number of cache ways from one (256 KB) to 32 (8 MB).

Application types A, B, and C in Figure 9 (a), (b), and (c) can be observed in both CPU and GPGPU applications. We summarize these types as follows:

- Type A has many computations and very few memory instructions. The performance impact of those few memory instructions is negligible since memory latencies can be overlapped by computations. Thus, the CPI of this type is close to the ideal CPI, and MPKI is also very low.

[1] *Cache friendliness* means that more caching improves the performance of an application.

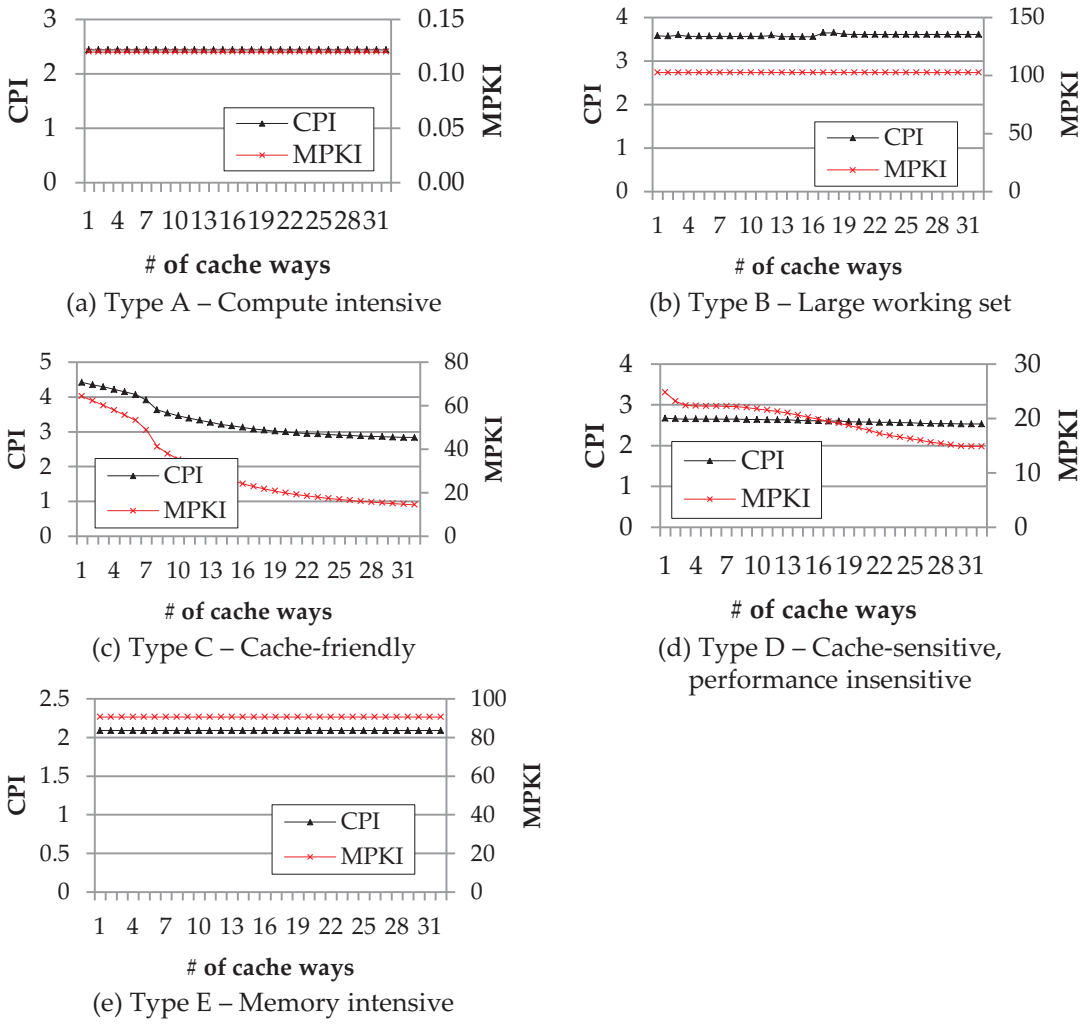


Figure 9: Application types based on how the cache affects performance (Ideal CPI is 2 in the baseline. We fix the number of cache sets (4096 sets) and vary the number of cache ways from one (256KB) to 32 (8MB)).

- Thrashing applications are typical examples of type B. Because there is a lack of data reuse or the working set size is significantly larger than the limited cache size, CPI is high and MPKI is extremely high.

- Type C applications are typical cache-friendly benchmarks. For these benchmarks, more caching improves the cache hit rate as well as performance.

However, Types D and E in Figure 9 (d) and (e) are *unique* to GPGPU applications. These types have many cache misses, but multi-threading is so

effective that *almost all memory latency can be tolerated*. We summarize these types as follows:

- Type D shows that MPKI *is reduced* as the cache size increases (cache-sensitive), but there is *little performance improvement* (performance-insensitive). In this type, multi-threading can effectively handle off-chip access latencies even without any caches, so having larger caches shows little performance improvement.

- Type E is very similar to type B. Due to the thrashing behavior, cache MPKI is very high. However, unlike type B, the observed CPI is very close to the ideal CPI of 2 since the thread-level parallelism (TLP) in type E can hide most memory latencies.

Note that types C and D are almost identical except for the change in CPI. For type C, larger caches are beneficial, but not for type D. Since these two types have identical cache behavior, we cannot differentiate them *by just checking the cache behavior*. Hence, our mechanisms aim to distinguish these two types by identifying the relationship between cache behavior and performance.

4.3 Prior Last-Level Cache Management

Here we provide the background of previous cache mechanisms and explain why they may not be effective for CPU and GPGPU heterogeneous workloads. These mechanisms can be categorized into two groups, namely, *dynamic cache partitioning* and *promotion-based cache management*.

4.3.1 Dynamic Cache Partitioning

Dynamic cache partitioning mechanisms achieve their goal (throughput, fairness, bandwidth reduction, etc.) by strictly partitioning cache ways among applications. Therefore, the interference between applications can be reduced by having

dedicated space for each application. Qureshi and Patt [120] proposed Utility-based Cache Partitioning (UCP), which tries to find an optimal cache partition such that the overall number of cache hits is maximized. UCP uses a set of shadow tags and hit counters to estimate the number of cache hits in each cache way for each application. Periodically, UCP runs a partitioning algorithm to calculate a new optimal partition. In every iteration of the partitioning algorithm, an application with the maximum number of hits will be chosen. The partitioning iterations continue until all ways are allocated to applications.

To minimize the adverse effect from streaming applications, Xie and Loh [151] proposed Thrasher Caging (TC). TC identifies thrashing applications by monitoring cache access frequency and the number of misses and then enforces streaming/thrashing applications to use a limited number of cache ways, called the cage. Since the antagonistic effect is isolated in the cage, TC improves performance with relatively simple thrasher classification logic.

These mechanisms are based on the assumption that high cache hit rate leads to better performance. For example, UCP [120] finds the best partition across applications that can maximize the number of overall cache hits. UCP works well when cache performance is directly correlated to the core performance, which is not always the case for GPGPU applications. They are often capable of hiding memory latency with TLP (types D and E). UCP prioritizes GPGPU applications when they have a greater number of cache hits. However, this will degrade the performance of CPU applications, while there is no performance improvement on GPGPU applications. Hence, we need a new mechanism to identify the performance impact of the cache for GPGPU applications.

4.3.2 Promotion-based Cache Management

Promotion-based cache mechanisms do not strictly divide cache capacity among applications. Instead, they insert incoming blocks into a non-MRU position and promote blocks upon hits. Thus, non-temporal accesses are evicted in a short amount of time and other accesses can reside for a longer time in the cache by being promoted to the MRU position directly [58] or promoted toward the MRU position by a single position [150].

For example, the goal of Re-Reference Interval Prediction (RRIP) [58] is to be resistant to scan (non-temporal access) and thrashing (larger working set) by enforcing a shorter lifetime for each block and relying on cache block promotion upon hits.[2] The conventional LRU algorithm maintains an LRU stack for each cache set. An incoming block is inserted at the head of the stack (MRU), and the tail block (LRU) is replaced. When there is a cache hit, the hitting block will be moved to the MRU position. Thus, the lifetime of a cache block begins at the head and continues until the cache block goes through all positions in the LRU stack, which will be a waste of cache space.

On the other hand, RRIP inserts new blocks *near* the LRU position instead of at the MRU position. Upon a hit, a block is moved to the MRU position. The intuition of RRIP is to give less time for each block to stay in the cache and to give more time only to blocks with frequent reuses. Thus, RRIP can keep an active working set while minimizing the adverse effects of non-temporal accesses. RRIP also uses dynamic insertion policies to further optimize the thrashing pattern using set dueling [118].

Promotion-based cache mechanisms assume a similar number of active threads in all applications, and thereby assume a similar order of cache access rates

[2]Note that we use a thread-aware DRRIP for our evaluations.

among applications. This is a reasonable assumption when there are only CPU workloads. However, GPGPU applications have more frequent memory accesses due to having an order-of-magnitude more threads within a core. Therefore, we have to take this different degree of access rates into account to prevent most blocks of CPU applications from being evicted by GPGPU applications even before the first promotion is performed.

4.3.3 Summary of Prior Work

Table 2 summarizes how previous mechanisms work on heterogeneous workloads consisting of one CPU application and each GPGPU application type. For types A, B, D, and E, since performance is not significantly affected by the cache behavior, having fewer ways for the GPGPU application would be most beneficial. However, previous cache-oriented mechanisms favor certain applications based on the number of cache hits or cache access rate, so the GPGPU application is favored in many cases, which will degrade the performance of a CPU application.

Table 2: Application favored by mechanisms when running heterogeneous workloads (1 CPU + each type of GPGPU application).

Workloads		Favored application type			Ideal
	GPGPU	UCP	RRIP	TC	
CPU+	Type A	CPU	CPU	none	CPU
	Type B	CPU	≈ or GPGPU	CPU	CPU
	Type C	GPGPU	GPGPU	CPU	Fair share
	Type D	GPGPU	GPGPU	CPU	CPU
	Type E	CPU	≈ or GPGPU	CPU	CPU

For type C GPGPU applications, due to excessive cache accesses and a decent cache hit rate, both UCP and RRIP favor GPGPU applications. However, the ideal partitioning will be formed based on the behavior of applications, and usually, giving too much space to one application results in poor performance. On the other hand, TC can isolate most GPGPU applications by identifying them as thrashing.

The *Ideal* column summarizes the ideal scenario of prioritization that maximizes system throughput.

4.4 The Solution: TLP-Aware Cache Management Policy

This section proposes a thread-level parallelism-aware cache management policy (TAP) that consists of two components: core sampling and cache block lifetime normalization.

4.4.1 Core Sampling

As we discussed in Section 4.2, we need a new way to identify the cache-to-performance effect for GPGPU applications. Thus, we propose a sampling mechanism that applies a different policy to each core, called *core sampling*. The intuition of core sampling is that most GPGPU applications show symmetric behavior across cores on which they are running.[3] In other words, each core shows similar progress in terms of the number of retired instructions. Using this characteristic, core sampling applies a different policy to each core and periodically collects samples to see how the policies work. For example, to identify the effect of cache on performance, core sampling enforces one core (Core-POL1) to use the LRU insertion policy and another core (Core-POL2) to use the MRU insertion policy. Once a period is over, the *core sampling controller* (CSC) collects the performance metrics, such as the number of retired instructions, from each core and compares them. If the CSC observes significant performance differences between Core-POL1 and Core-POL2, we can conclude that the performance of this application has been affected by the cache behavior. If the performance delta is negligible, caching is not beneficial for this application. Based on this sampling result, the CSC makes an appropriate decision in the LLC (cache insertion or

[3]There are some exceptional cases; pipelining parallel programming patterns do not show the symmetric behavior.

partitioning) and other cores will follow this decision. Core sampling is similar to set dueling [118]. The insight of set dueling is from *Dynamic Set Sampling* (DSS) [119], which approximates the entire cache behavior by sampling a few sets in the cache with a high probability. Similarly, the symmetry in GPGPU applications makes the core sampling technique viable. Figure 10 shows the framework of core sampling.

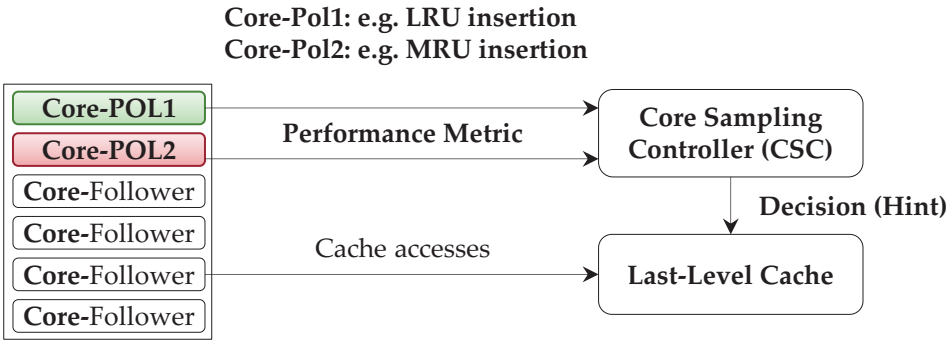


Figure 10: The core sampling framework.

Among multiple GPU cores, one core (Core-POL1) uses policy 1, another core (Core-POL2) uses policy 2, and all others (Core-Followers) follow the decision in the LLC made by the CSC. Inputs to the CSC are performance metrics from Core-POL1 and Core-POL2.

Core Sampling with Cache Partitioning When core sampling is running on top of cache partitioning, the effect of different policies for a GPGPU application is limited to its dedicated space once the partition is set for each application. For example, if a GPGPU application has only one way and CPU applications have the rest of the ways, sampling policies affect only one way for the GPGPU application. In this case, no difference exists between the MRU and LRU insertion policies. Therefore, we set core sampling to enforce Core-POL1 to bypass the LLC with cache partitioning.

Benchmark Classification by Core Sampling Based on the CPI variance between Core-POL1 and Core-POL2, we categorize the variance into two groups using $Threshold_{\alpha}$. If the CPI delta is less than $Threshold_{\alpha}$, caching has little effect on performance. Thus, types A, B, D, and E can be detected. If the CPI delta is higher than $Threshold_{\alpha}$, this indicates that an application is cache-friendly. When an application has asymmetric behavior, core sampling may misidentify this application as cache-friendly. However, we found that there are only a few asymmetric benchmarks and the performance penalty of misidentifying these benchmarks is negligible. Note that we set $Threshold_{\alpha}$ to 5% from empirical data.

Overhead of the core sampling The core sampling mechanism has following overheads:

- Control logic: Since we assume the logic for cache partitioning or promotion-based cache mechanisms already exists, core sampling only requires periodic logging of performance metrics from two cores and the performance-delta calculation between the two. Thus, the overhead of the control logic is almost negligible.

- Storage overhead: The core sampling framework requires the following additional structures. 1) *One counter per core* to count the number of retired instructions during one period: Usually, most of today's processors already have this counter. 2) *Two registers to indicate the ids of Core-POL1 and Core-POL2*: When a cache operation is performed to a cache line, the core id field is checked. If the core id matches with Core-POL1, the LRU insertion policy or LLC bypassing is used. If it matches with Core-POL2, the MRU insertion policy is used. Otherwise, the underlying mechanism will be applied to cache operations.

Table 3 summarizes the storage overhead of core sampling. Since core sampling is applied on top of dynamic cache partitioning or promotion-based cache mechanisms, such as UCP or RRIP, we assume that the underlying hardware already supports necessary structures for them. Therefore, the overhead of core sampling is fairly negligible.

Table 3: Hardware complexity of the core sampling (our baseline has 6 GPU cores and 4 LLC tiles).

Hardware	Purpose	Overhead
20-bit counter per core	Perf. metric	20 $\times$ 6 cores = 120 bits
2 5-bit registers	Ids of Core-POL1 and Core-POL2	10-bit $\times$ 4 LLC tiles = 40 bits
Total		160 bits

Discussions on the core sampling We further discuss possible issues with the core sampling.

1. Worst-performing core and load imbalance - Core sampling may hurt the performance of a sampled core, Core-POL1 or Core-POL2, if a poorly performing policy is enforced during the entire execution. In set dueling, a total of 32 sets will be sampled out of 4096 sets (64B cache line, 32-way 8MB cache). Only 0.78% of the entire cache sets are affected. Since we have a much smaller number of cores than cache sets, the impact of having a poorly performing core might be significant. Also, this core may cause a load imbalance problem among cores. However, these problems can be solved by periodically rotating sampled cores instead of fixing which cores to sample.

2. Synchronization - Most current GPUs cannot synchronize across cores, so core sampling is not affected by synchronization. However, if future GPUs support synchronization such as a barrier across cores, since all cores will

make the same progress regardless of the cache policy, core sampling cannot detect performance variance between cores. In this case, we turn off core sampling and all cores follow the policy of the underlying mechanism after a few initial periods.

3. Handling multiple applications - So far, we assume that GPUs can run only one application at a time. When a GPU core can execute more than one kernel concurrently[4], the following support is needed: (1) We need separate counters for each application to keep track of performance metrics; (2) Instead of Core-POL1 and Core-POL2 being physically fixed, the hardware can choose which core to be Core-POL1 and Core-POL2 for each application, so each application can have its own sampling information.

4.4.2 Cache Block Lifetime Normalization

GPGPU applications typically access caches much more frequently than CPU applications. Even though memory-intensive CPU applications also exist, the cache access rate cannot be as high as that of GPGPU applications due to a much smaller number of threads in a CPU core. Also, since GPGPU applications can maintain high throughput because of the abundant TLP in them, there will be continuous cache accesses. However, memory-intensive CPU applications cannot maintain such high throughput due to the limited TLP in them, which leads to less frequent cache accesses. As a result, there is often an order of difference in cache access frequencies between CPU and GPGPU applications. Figure 11 shows the number of memory requests per 1000 cycles (RPKC) of applications whose RPKC is in the top and bottom five, along with the median and average values from all CPU and GPGPU applications, respectively. The top five CPU applications have

[4]NVIDIA's Fermi now supports the concurrent execution of kernels, but each core can execute only one kernel at a time.

over 60 RPKC, but the top five GPGPU applications have over 400 and two of them have even more than 1000 RPKC. Hence, when CPU and GPGPU applications run together, we have to take into account this difference in the degree of access rates.

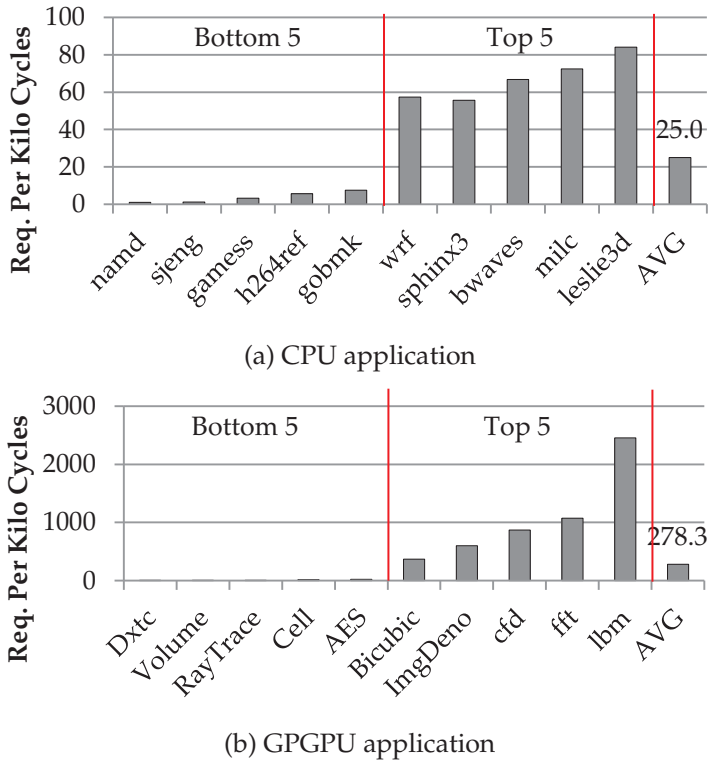


Figure 11: Memory access rate characteristics.

To solve this issue, we introduce *cache block lifetime normalization*. First, we detect access rate differences by collecting the number of cache accesses from each application. Periodically, we calculate the access ratio between applications. If the ratio exceeds the threshold, T_{xs}[5], this ratio value is stored in a 10-bit register, called *XSRATIO*. When the ratio is lower than T_{xs}, the value of XSRATIO will be set to 1. When the value of XSRATIO is greater than 1, TAP policies utilize the value of the XSRATIO register to enforce similar cache residential time to CPU and GPGPU applications. We detail how the XSRATIO register is used in the following sections.

[5]We set T_{xs} to 10, which means a GPGPU application has 10 times more accesses than the CPU application that has the highest cache access rate, via experimental results.

4.5 TAP Extensions

We also propose two new TAP mechanisms: TAP-UCP and TAP-RRIP in this section.

4.5.1 TAP-UCP

TAP-UCP is based on UCP [120], a dynamic cache partitioning mechanism for only CPU workloads. UCP periodically calculates an optimal partition to adapt a run-time behavior of the system. For each application, UCP maintains an LRU stack for each sampled set[6] and a hit counter for each way in all the sampled sets during a period. When a cache hit occurs in a certain position of an LRU stack, the corresponding hit counter will be incremented. Once a period is over, the partitioning algorithm iterates until all cache ways are allocated to applications. In each iteration, UCP finds the marginal utility of each application using the number of remaining ways to allocate and the hit counters of an application.[7] Then, UCP allocates one or more cache ways to the application that has the highest marginal utility.

As explained in Section 4.3.1, UCP tends to favor GPGPU applications in heterogeneous workloads. However, TAP-UCP gives more cache ways to CPU applications when core sampling identifies that a GPGPU application achieves little benefit from caching. Also, TAP-UCP adjusts the hit counters of a GPGPU application when the GPGPU application has a much greater number of cache accesses than CPU applications. To apply TAP in UCP, we need two modifications in the UCP's partitioning algorithm.

The first modification is that only one way is allocated for a GPGPU application

[6]UCP collects the information only from sampled sets to reduce the overhead of maintaining an LRU stack for each set.

[7]Marginal utility is defined as the utility per unit cache resource in [120]. For more details, please refer to Algorithm 1.

when caching has little benefit on it. To implement this, we add one register to each cache, called the *UCP-Mask*. The CSC of core sampling sets the UCP-Mask register when caching is not effective; otherwise the value of the UCP-Mask remains 0. TAP-UCP checks the value of this register before performing the partitioning algorithm. When the value of the UCP-Mask is set, only CPU applications are considered for the cache way allocation.

The second modification is that when partitioning is performed, we first divide the value of the GPGPU application's hit counters by the value of the XSRATIO register, which is periodically set by cache block lifetime normalization, as described in Section 4.4.2. More details about the TAP-UCP partitioning algorithm is described in Algorithm 1.

4.5.2 TAP-RRIP

First, we provide more details about the RRIP mechanism [58], which is the base of TAP-RRIP. RRIP dynamically adapts between two competing cache insertion policies, Static-RRIP (SRRIP) and Bimodal-RRIP (BRRIP), to filter out thrashing patterns. RRIP represents the insertion position as the Re-Reference Prediction Value (RRPV). With an n-bit register per cache block for the LRU counter, an RRPV of 0 indicates an MRU position and an RRPV of 2^n-1 represents an LRU position. SRRIP always inserts the incoming blocks with an RRPV of 2^n-2, which is the best performing insertion position between 0 to 2^n-1. On the other hand, BRRIP inserts blocks with an RRPV of 2^n-2 with a very small probability (5%) and for the rest, which is the majority, it places blocks with an RRPV of 2^n-1. RRIP dedicates few sets of the cache to each of the competing policies. A saturating counter, called a Policy Selector (PSEL), keeps track of which policy incurs fewer cache misses and decides the winning policy. Other non-dedicated cache sets follow the decision made by PSEL.

Algorithm 1 TAP-UCP algorithm (modified UCP)

```
1: balance = N
2: allocation[i] = 0 for each competing application i
3: if XSRATIO > 1    // TAP-UCP begin .....................................
4:     foreach way j in GPGPU application i do
5:         way_counter_i[j] /= XSRATIO    // TAP-UCP end ..................
6: while balance do:
7:     foreach application i do:
8:         // TAP-UCP begin ..............................................
9:         if application i is GPGPU application and UCP-Mask == 1
10:            continue
11:        // TAP-UCP end ................................................
12:        alloc = allocations[i]
13:        max_mu[i] = get_max_mu(i, alloc, balance)
14:        blocks_req[i] = min blocks to get max_mu[i] for i
15:    winner = application with maximum value of max_mu
16:    allocations[winner] += blocks_req[winner]
17:    balance -= blocks_req[winner]
18: return allocations
19:
20: // get the maximum marginal utility of an application
21: get_max_mu(app, alloc, balance):
22:    max_mu = 0
23:    for (ii=1;ii<=balance;ii++) do:
24:        mu = get_mu_value(p, alloc, alloc+ii)
25:        if (mu > max_mu) max_mu = mu
26:    return max_mu
27:
28: // get a marginal utility
29: get_mu_value(app, a, b):
30:    U = change in misses for application p when the number of blocks
31:        assigned to it increases from a-way to b-way (a < b)
32:    return U/(b-a)
```

To apply TAP to RRIP, we need to consider two problems: 1) manage the case when a GPGPU application does not need more cache space and 2) prevent the interference by a GPGPU application with much more frequent accesses. When either or both problems exist, we enforce the BRRIP policy for the GPGPU application since BRRIP generally enforces a shorter cache lifetime than SRRIP for each block. Also, the hitting GPGPU block will not be promoted and GPGPU blocks will be replaced first when both CPU and GPGPU blocks are replaceable. In pseudo-LRU approximations including RRIP, multiple cache blocks can be in LRU positions. In this case, TAP-RRIP chooses a GPGPU block over a CPU block for the replacement.

In TAP-RRIP, we add an additional register, called the *RRIP-Mask*. The value of the RRIP-Mask register is set to 1 when 1) core sampling decides caching is not beneficial for the GPGPU application or 2) the value of the XSRATIO register is greater than 1. When the value of the RRIP-Mask register is 1, regardless of the policy decided by PSEL, the policy for the GPGPU application will be set to BRRIP. Otherwise, the winning policy by PSEL will be applied. Table 4 summarizes the policy decision of TAP-RRIP for the GPGPU application.

Table 4: TAP-RRIP policy decisions for the GPGPU application.

RRIP's decision	TAP (RRIP-Mask)	Final Policy	GPGPU type	Note
SRRIP	0	SRRIP	Type C	Base RRIP
BRRIP	0	BRRIP		
SRRIP	1	BRRIP	Types A, B, D, E	Always BRRIP
BRRIP	1	BRRIP		

4.6 Evaluation Methodology

4.6.1 Simulator

We use MacSim simulator [45] for our simulations. As the frontend, we use Pin [84] for the CPU workloads and GPUOcelot [27] for GPGPU workloads. For all simulations, we repeat early terminated applications until all other applications finish, which is a similar methodology used in [57, 58, 120, 150]. Table 5 shows the evaluated system configuration. Our baseline CPU cores are similar to the CPU cores in Intel's Sandy Bridge [52], and we model GPU cores similarly to those in NVIDIA's Fermi [103]; each core is running in SIMD fashion with multi-threading capability.

Table 5: Evaluated system configurations.

CPU	1-4 cores, 3.5GHz, 4-wide, out-or-order gshare branch predictor 8-way 32KB L1 I/D (2-cycle), 64B line 8-way 256KB L2 (8-cycle), 64B line
GPU	6 cores, 1.5GHz, in-order, 2-wide 8-SIMD No branch predictor (switch to the next ready thread) 8-way 32KB L1 D (2-cycle), 64B line 4-way 4KB L1 I (1-cycle), 64B line
L3 Cache	32-way 8MB (4 tiles, 20-cycle), 64B line
NoC	20-cycle fixed latency, at most 1 req/cycle
DRAM	4 controllers, 16 banks, 4 channels DDR3-1333. 41.6GB/s Bandwidth, FR-FCFS

4.6.2 Benchmarks

Tables 6 and 7 show the type of CPU and GPGPU applications that we use for our evaluations. We use 29 SPEC 2006 CPU benchmarks and 32 CUDA GPU benchmarks from publicly available suites, including NVIDIA CUDA SDK, Rodinia [20], Parboil [137], and ERCBench [17]. For CPU workloads, Pinpoint [113] was used to select a representative simulation region with the *ref* input set. Most

GPGPU applications are run until completion.

Table 6: CPU benchmarks classification.

Type	Benchmarks (INT // FP)
Cache-friendly (10)	bzip2, gcc, mcf, omnetpp, astar // leslie3d, soplex, lbm, wrf, sphinx3
Streaming / Large working set (6)	libquantum // bwaves, milc, zeusmp, cactusADM, GemsFDTD
Compute intensive(13)	perlbench, gobmk, hmmer, sjeng, h264ref, xalancbmk // gamess, gromacs, namd, dealII, povray, calculix, tonto

Table 7: GPGPU benchmarks classification.

Type	Benchmarks (SDK // Rodinia // ERCBench // Parboil)
A (4)	dxtc, fastwalsh, volumerender // cell // NA // NA
B (12)	bicubic, convsep, convtex, imagedenoise, mergesort sobelfilter // hotspot, needle // sad // fft, mm, stencil
C (3)	quasirandom, sobolqrng // raytracing // NA // NA
D (4)	blackscholes, histogram, reduction // aes // NA // NA
E (9)	dct8x8, montecarlo, scalarprod // backprop, cfd, nn, bfs // sha // lbm

Table 8 describes all workloads that we evaluate for heterogeneous simulation. We thoroughly evaluate our mechanisms on an excessive number of heterogeneous workloads. We form these workloads by pseudo-randomly selecting one, two, or four CPU benchmarks from cache-friendly and compute-intensive group in Table 6 and one GPGPU benchmark from each type in Table 7. For Stream-CPU workloads, in addition to streaming applications from SPEC2006 (Table 6), we add five more streaming benchmarks from the Merge [81] benchmarks.

4.6.3 Evaluation Metric

We use the geometric mean (Eq. (4)) of the speedup of each application (Eq. (5)) as the main evaluation metric.

Table 8: Heterogeneous workloads.

Type	# CPU	# GPGPU	# of total workloads
1-CPU	1	1	152 workloads
2-CPU	2	1	150 workloads
4-CPU	4	1	75 workloads
Stream-CPU	1	1	25 workloads

$$speedup = geomean(speedup_{(0\ to\ n-1)}) \tag{4}$$

$$speedup_i = \frac{IPC_i}{IPC_i^{baseline}} \tag{5}$$

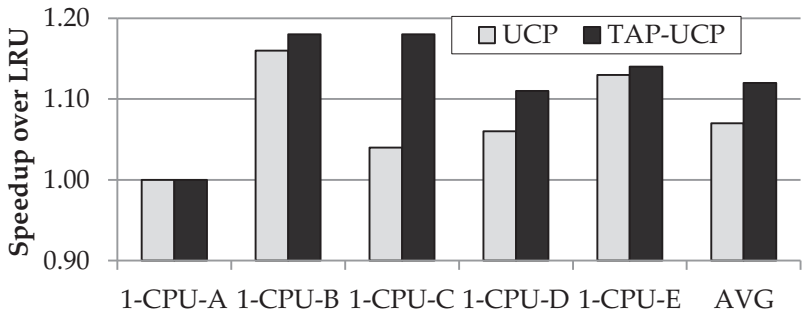
4.7 Experimental Evaluation

4.7.1 TAP-UCP Evaluation

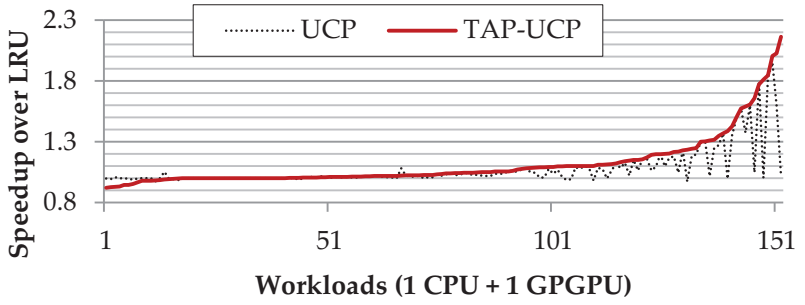
Figure 12 shows the base UCP and TAP-UCP speedup results normalized to the LRU replacement policy on all 1-CPU workloads (one CPU + one GPGPU). Figure 12 (a) shows the performance results for each GPGPU application type. For 1-CPU-A[8], 1-CPU-B, and 1-CPU-E workloads, as explained in Table 2, since these types of GPGPU applications do not have many cache hits, UCP can successfully partition cache space toward being CPU-friendly. Therefore, the base UCP performs well on these workloads and improves performance over LRU by 0%, 15%, and 12% for workloads 1-CPU-A, 1-CPU-B, and 1-CPU-E, respectively. Note that since type A applications are computation-intensive, even LRU works well.

For 1-CPU-C and 1-CPU-D, UCP is less effective than other types. For 1-CPU-C, we observe that the number of cache accesses and cache hits of GPGPU applications is much higher than that of CPUs (at least an order). As a result, UCP strongly favors the GPGPU application, so there is a severe

[8]CPU application with one type A GPGPU application. Same rule applies to other types.



(a) UCP speedup results per type



(b) S-curve for TAP-UCP speedup results

Figure 12: TAP-UCP speedup results.

performance degradation in the CPU application. Therefore, UCP shows only a 3% improvement over LRU. However, by considering the different access rates in two workloads, TAP-UCP successfully balances cache space between CPU and GPGPU applications. TAP-UCP shows performance improvements of 14% and 17% compared to UCP and LRU, respectively, for 1-CPU-C workloads. For 1-CPU-D, although larger caches are not beneficial for GPGPU applications since they have more cache hits than CPU applications, UCP naturally favors the GPGPU applications. However, the cache hit pattern of the GPGPU applications often shows a strong locality near the MRU position, so UCP stops the allocation for GPGPU applications after a few hot cache ways. As a result, UCP performs better than LRU by 5% on average. The performance of TAP-UCP is 5% better than UCP by detecting when more caching is not beneficial for GPGPU applications.

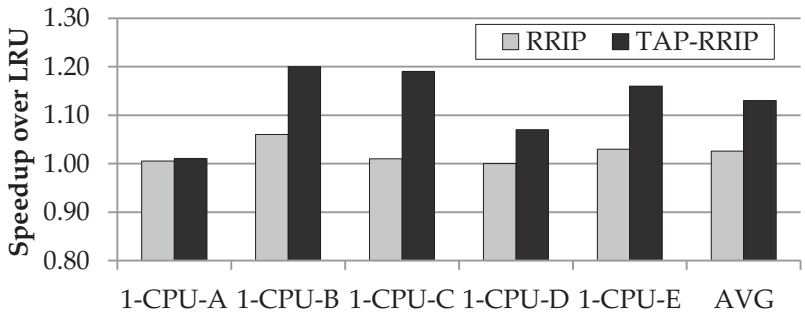
From the s-curve[9] result in Figure 12 (b), TAP-UCP usually outperforms UCP except in a few cases with type C GPGPU applications. When a CPU application is running with a type C GPGPU application, giving very few ways to GPGPU applications increases the bandwidth requirement significantly. As a result, the average off-chip access latency increases, so the performance of all other memory-intensive benchmarks is degraded severely. We see these cases only in seven workloads out of 152. In our future work, we will monitor bandwidth increases to prevent these negative cases. Overall, UCP performs 6% better than LRU, and TAP-UCP improves UCP by 5% and LRU by 11% across 152 heterogeneous workloads.

4.7.2 TAP-RRIP Evaluation

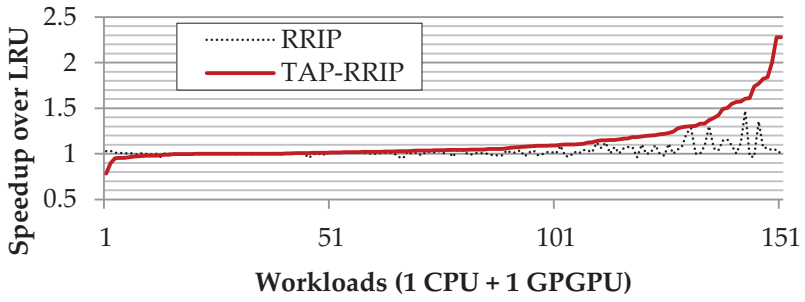
Figure 13 (a) presents the speedup results of RRIP and TAP-RRIP for each GPGPU type. We use a thread-aware DRRIP, which is denoted as RRIP in the figures, for evaluations with a 2-bit register for each cache block. Other configurations are the same as in [58]. The base RRIP performs similarly to LRU. As explained in Section 4.3.2, RRIP favors GPGPU applications because of its more frequent cache accesses. Thus, GPGPU blocks occupy the majority of cache space. On the other hand, TAP-RRIP tries to give less space to GPGPU blocks if core sampling identifies that more caching is not beneficial.

Figure 13 (b) shows the s-curve for the performance on all 152 workloads. Although RRIP does not show many cases with degradation, RRIP is not usually effective and performs similarly to LRU. However, TAP-RRIP shows performance improvement in more than half of the evaluated workloads. Two TAP-RRIP cases show degradation of more than 5%. Again, this is the problem due to type C GPGPU applications (too little space is given to the GPGPU application, so the

[9]For all s-curve figures from now on, we sort all results by the performance of the TAP mechanisms in ascending order.



(a) RRIP speedup results per type



(b) S-curve for TAP-RRIP speedup results

Figure 13: TAP-RRIP speedup results.

bandwidth is saturated).

On average, the base RRIP performs better than LRU by 3% while TAP-RRIP improves the performance of RRIP and LRU by 9% and 12%, respectively.

4.7.3 Streaming CPU Application

When a streaming CPU application is running with a GPGPU application, our TAP mechanisms tend to unnecessarily penalize GPGPU applications even though the streaming CPU application does not need any cache space. Since we have only considered the adverse effect of GPGPU applications, the basic TAP mechanisms cannot effectively handle this case. Thus, we add a streaming behavior detection mechanism similar to [150, 151], which requires only a few counters. Then, we minimize space usage by CPU applications once they are identified as streaming. The enhanced TAP-UCP will allocate only one way to a streaming CPU application

and the enhanced TAP-RRIP will reset the value of the RRIP-Mask register to operate as the base RRIP, which works well for streaming CPU applications. Figure 14 shows the performance results of the enhanced TAP mechanisms (TAP-S) that consider the streaming behavior of CPU applications on the 25 Stream-CPU workloads in Table 8.

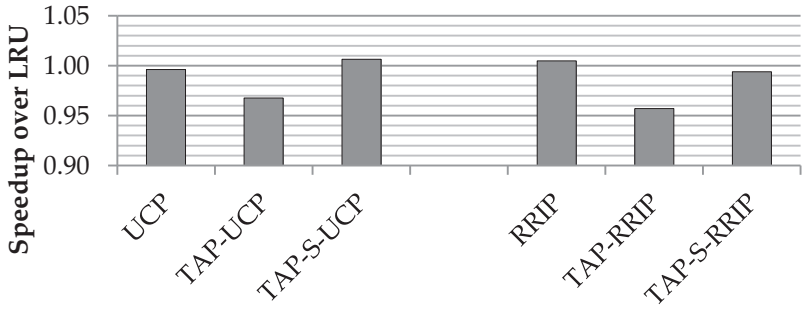


Figure 14: Enhanced TAP mechanism (TAP-S) results.

The basic TAP mechanisms degrade performance by 3% and 4% over LRU, respectively. However, the TAP-S mechanisms solve the problem of basic mechanisms and show a performance similar to LRU. Since all other previous mechanisms, including LRU, can handle the streaming application correctly, the TAP mechanisms cannot gain further benefit. Note that the TAP-S mechanisms do not change the performance of other workloads.

4.7.4 Multiple CPU Applications

So far, we have evaluated the combinations of one CPU and one GPGPU application. In this section, we evaluate multiple CPU applications running with one GPGPU application (2-CPU and 4-CPU workloads in Table 8). As the number of concurrently running applications increases, the interference by other applications will also increase. Thus, the role of intelligent cache management becomes more crucial. Figure 15 shows evaluations on 150 2-CPU and 75 4-CPU workloads. TAP-UCP shows up to a 2.33 times and 1.93 times speedup on 2-CPU

and 4-CPU workloads, respectively. TAP-UCP performs usually no worse than the base UCP except in a few cases. In this case, two or four memory-intensive CPU benchmarks are running with one type C GPGPU application. On average, TAP-UCP improves the performance of LRU by 12.5% and 17.4% on 2-CPU and 4-CPU workloads, respectively, while UCP improves by 7.6% and 6.1%.

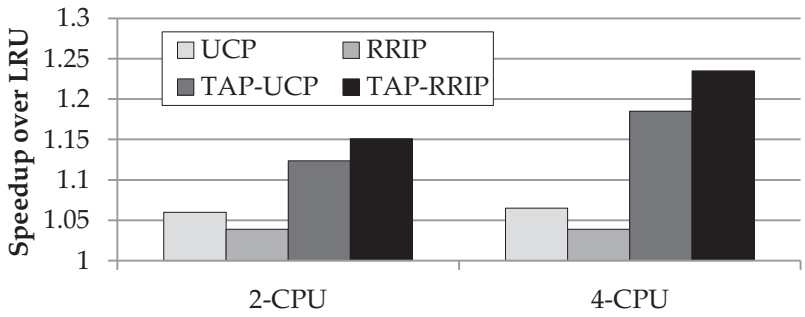


Figure 15: Multiple CPU application results.

TAP-RRIP shows up to a 2.3 times and 2.2 times speedup on 2-CPU and 4-CPU workloads, respectively. On average, RRIP improves the performance of LRU by 4.5% and 5.6% on 2-CPU and 4-CPU workloads, respectively, while TAP-RRIP improves even more, by 14.1% and 24.3%. From multi-CPU evaluations of the TAP mechanisms, we conclude that our TAP mechanisms show good scalability by intelligently handling inter-application interference.

4.7.5 Comparison to Static Partitioning

Instead of using dynamic cache partitioning, a cache architecture can be statically partitioned between CPUs and GPUs, but statically partitioned caches cannot use the resources efficiently. In other words, it cannot adapt to workload characteristics at run-time. In this section, we evaluate a system that statically partitions the LLC between the CPUs and GPUs evenly. All CPU cores (at most 4) share 16 ways of the LLC regardless of the number of concurrently running CPU applications, and the GPU cores (6 cores) share the rest of the 16 ways. Figure 16 shows the TAP

results compared to static partitioning.

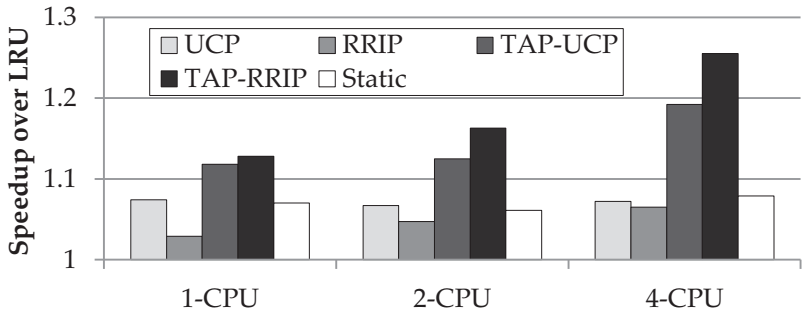


Figure 16: Static partitioning results.

For 1-CPU workloads, static partitioning shows a 6.5% improvement over LRU, while TAP-UCP and TAP-RRIP show 11% and 12% improvements. However, as the number of concurrently running applications increases, static partitioning does not show further improvement (7% over in both 2-CPU and 4-CPU workloads), while the benefit of the TAP mechanisms continuously increases (TAP-UCP: 12% and 19%, TAP-RRIP: 15% and 24% for 2-CPU and 4-CPU workloads, respectively). Moreover, static partitioning performs slightly worse than LRU in many cases (52 out of 152 in 1-CPU, 54 out of 150 in 2-CPU, and 28 out of 75 in 4-CPU workloads, respectively), even though the average is 7% better than that of LRU. We conclude that static partitioning on average performs better than LRU, but it cannot adapt to workload characteristics, especially when the number of applications increases.

4.7.6 Cache Sensitivity Evaluation

Figure 17 shows the performance results with other cache configurations. We vary the associativity and size of caches. As shown, our TAP mechanisms constantly outperform their corresponding mechanisms, UCP and RRIP, in all configurations. Therefore, we can conclude that our TAP mechanisms are robust to cache configurations.

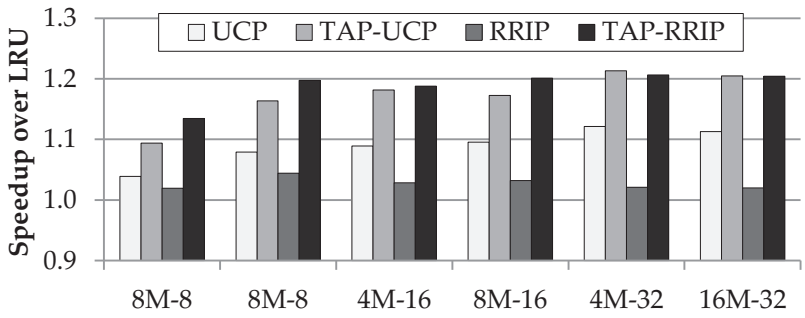


Figure 17: Cache sensitivity results (size-associativity).

4.7.7 Comparison to Other Mechanisms

In this section, we compare the TAP mechanisms with other cache mechanisms, including TADIP [57], PIPP [150], and TC [151] along with UCP and RRIP. TADIP is a dynamic insertion policy (DIP) that dynamically identifies the application characteristic and inserts single-use blocks (dead on fill) in the LRU position to evict as early as possible. PIPP [150] pseudo partitions cache space to each application by having a different insert position for each application, which is determined using a utility monitor as in UCP. Upon hits, each block is promoted toward the MRU by one position. PIPP also considers the streaming behavior of an application. When an application shows streaming behavior, PIPP assigns only one way and allows promotion with a very small probability (1/128). Figure 18 shows the speedup results.

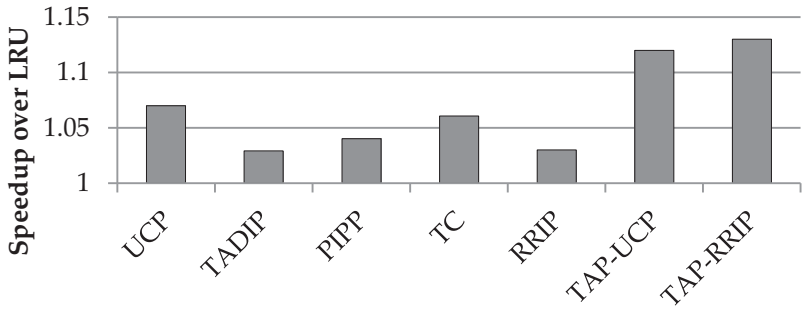


Figure 18: TAP comparison to other policies.

As explained in Section 4.3.2, if cache space is not strictly partitioned, an applications that has more frequent cache accesses is favored. As a result, TADIP also favors GPGPU applications, thereby showing only 3% improvement over LRU. On the other hand, PIPP can be effective by handling GPGPU applications as streaming. Since most GPGPU applications are identified as streaming, PIPP can be effective for types A, B, D, and E GPGPU applications. However, for type C, due to the saturated bandwidth from off-chip accesses by GPGPU applications, PIPP is not as effective as it is for other types. TC has similar benefits and problems as PIPP. On average, PIPP and TC improve performance by 4% and 6%, respectively, over LRU. Our TAP mechanisms outperform these previous mechanisms by exploiting GPGPU-specific characteristics.

4.8 Summary of This Chapter

LLC management is an important problem in today's chip multi-processors and in future many-core-heterogeneous processors. Many researchers have proposed various mechanisms for throughput, fairness, or bandwidth. However, none of the previous mechanisms consider GPGPU-specific characteristics in heterogeneous workloads such as underlying massive multi-threading and the different degree of access rates between CPU and GPGPU applications. Therefore, when CPU applications are running with a GPGPU application, the previous mechanisms will not deliver the expected outcome and may even perform worse than the LRU replacement policy. In order to identify the characteristics of a GPGPU application, we propose core sampling, which is a simple yet effective technique to profile a GPGPU application at run-time. By applying core sampling to UCP and RRIP and considering the different degree of access rates, we propose the TAP-UCP and TAP-RRIP mechanisms. We evaluate the TAP mechanisms on 152 heterogeneous workloads and show that they improve the performance by 5% and 10% compared

to UCP and RRIP and 11% and 12% to LRU. In future work, we will consider bandwidth effects in shared cache management on heterogeneous architectures.

CHAPTER V

ADAPATIVE VIRTUAL CHANNEL PARTITIONING

5.1 Introduction

An on-chip heterogeneous architecture that integrates GPU cores on top of conventional CPU-only chip multiprocessors (CMP) has become a popular architecture trend, as can be seen in Intel's Sandy Bridge [52] and Ivy Bridge [49], AMD's accelerated processing units (APU) [5], and NVIDIA's Denver project [106]. In this architecture, various on-chip resources are shared between CPU and GPU cores, such as last-level cache (LLC), on-chip interconnection networks, memory controllers, and DRAM memories.

The resource sharing problem has existed since CMP was introduced. In CPU-GPU heterogeneous architectures, however, we expect more shared resource contention, especially interference suffered by CPU applications from GPGPU applications due to the different nature of CPU and GPU cores. CPU cores typically employ 1- to 4-ways of simultaneous multi-threading and rely on larger caches to tolerate memory access latencies. On the other hand, GPU cores operate with tens of active threads to minimize the penalty of the off-chip memory latency. The high degree of thread-level parallelism (TLP) in GPU cores leads to much more frequent network injections, which only exacerbates the resource sharing problem.

We tackle the resource sharing problem in the on-chip network (NoC) in this chapter. Sources of interference can be located in any shared resources, from shared last-level caches (LLC) to memory controllers (MC). Nonetheless, the NoC is one of the most important shared mediums because it connects all components and all communication traverses through it. The management of

the NoC significantly affects the performance of each application as well as the system throughput. The baseline on-chip routers are usually maintained under the round-robin or oldest-first arbitration policies, so applications with higher network demands will be favored. Consequently, GPGPU applications are favored naturally and CPU applications will face unfair network resource utilization in heterogeneous architectures.

To solve the resource sharing problem in the NoC, researchers have proposed router arbitration policies [24, 25, 40, 71] in the homogeneous CMP domain. These policies consider different application characteristics and prioritize critical packets or applications. However, these mechanisms may not be used directly for heterogeneous architectures because they do not consider the heterogeneity of cores. GPU cores inject packets much more frequently than CPU cores because they are capable of running many concurrent threads with SIMD executions, which leads to an unbalanced number of packets between the CPU and GPU in the network. These characteristics of GPU cores increase the thread-level parallelism (TLP) of cores, thereby making them more tolerant to latency and bandwidth than CPU cores. Therefore, NoC mechanisms for heterogeneous architectures need to consider different characteristics of GPU cores to be effective.

Here, we propose a virtual channel partitioning (VCP) mechanism, which is simple yet effective, to attack resource sharing problems in the NoC for heterogeneous systems. A router typically has multiple input and/or output virtual channels (VC) that share physical links and thus bandwidth. By dedicating a number of VCs to CPU and GPU applications, we can guarantee a minimum service in the network to each type. Also, VCP naturally arbitrates packets that pass through the router because VCP forces GPU packets to occupy only a part of VCs, so CPU packets can occupy an available VC immediately after they arrive. To provide CPU and GPU packets according to their VC availability, VCP requires

separate injection queues for CPU and GPU packets. Injection queues of shared caches and memory controllers have both types of packets. If the shared queue with a first-come first-serve (FCFS) scheduler is used for the injection, even if available VCs of a certain type exist, packets cannot be injected until they arrive at the head of the queue. VCP uses DAMQ-based injection queues [134] to maintain separate queues so that VCP can supply packets to their corresponding partition with low overhead.

However, VCP may result in significant performance degradation for bandwidth-limited GPGPU applications. Without partitioning, bandwidth-limited GPGPU applications could have utilized more bandwidth, but their performance may be degraded because of the reduced bandwidth with partitioning. Therefore, the performance trade-off between CPU and GPGPU applications should be carefully balanced. Moreover, how different partitioning configurations affect performance varies by the workload characterization. Also, they behave differently even in the same workload if different phases exist. Therefore, partitioning should cope with the run-time behaviors of applications. This naturally leads us to study an adaptive partitioning mechanism. For better adaptation, our proposed feedback-directed VCP uses a sampling technique to dynamically compare different partitioning configurations and enforces the best performing configuration.

We claim our **contributions** to be as follows:

1. We propose a feedback-directed virtual channel partitioning (VCP) mechanism that can arbitrate packets that pass through the local router while providing a more balanced number of packets to the network.

2. VCP considers different characteristics of GPU cores by directly collecting performance metrics from cores, while coarse-grain control of virtual channels in VCP enables us to use very simple hardware.

3. VCP improves system performance by 15% across 39 heterogeneous workloads. More importantly, results show at most a 2.5% performance degradation and only two workloads show negative speedup, while VCP performs better than any static configurations.

The rest of this chapter is organized as follows. Section 5.2 explains the current problems and design space explorations of NoC in heterogeneous architecture. Section 5.3 introduces our proposal, VCP. We present the evaluation methodology and results in Sections 5.4 and 5.5. Section 5.6 summarizes the chapter.

5.2 Problems and Design Space Exploration in NoCs

This section describes the potential problems in designing the on-chip interconnection network in a CPU-GPU heterogeneous architecture.

5.2.1 Routing Algorithm

NoC routers typically employ a simple static routing algorithm to minimize latency and complexity. For example, x-y or shortest-distance algorithms are widely used. However, this may result in link congestion in the heterogeneous architecture. For example, Figure 19 shows a diagram of Intel's Ivy Bridge with the ring network. In the figure, the GPGPU packets are not likely to use the upper link since the lower link offers the shortest distance from the GPU cores to the L3. The lower link is also used between the L3 and the memory controllers. Therefore, only CPU packets use the upper link, which is possibly under-utilized. While studies on other algorithms show improved network performance, they are limited to traffic generated by specialized or CPU-only applications [46, 47, 89].

5.2.2 Resource Contention and Partitioning

CPU and GPU packets compete to acquire resources in various places, especially virtual and physical channels. When the resources are naively shared by both

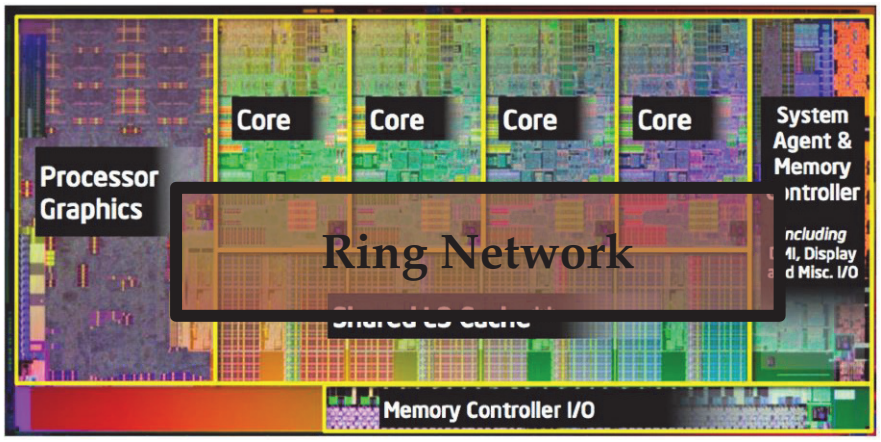


Figure 19: Diagram of Intel's Ivy Bridge die with the ring network.

kinds of cores, higher-demanding cores will acquire the most resources, which are GPU cores. This is the same problem found in the LRU cache-replacement policy in the shared cache. To solve this problem, many researchers have proposed various static and dynamic cache partitioning mechanisms [120, 132]. Similarly, partitioning mechanisms can be applied to on-chip virtual and physical channels. Each port of a router has multiple virtual channels. We can partition these virtual channels to each application. Similarly, if multiple physical channels exist, we can dedicate some channels to CPU cores and the other channels to GPU cores. If the interference exhibited by other applications is significant, resource partitioning would prevent interference and improve performance. However, this can lead to resource under-utilization if partitioning is not balanced with demand. Therefore, partitioning should be carefully applied to on-chip network resources.

5.2.3 Arbitration Policy

Multiple arbiters exist in each router to coordinate packets from different ports. In a CPU-GPU heterogeneous architecture, due to the different network demands, arbitration between CPU and GPU packets is a non-trivial problem. At first glance, statically giving higher priority to CPU applications appears to be a reasonable solution since CPU applications are more latency sensitive. However, when CPU

and GPGPU applications are both bandwidth-intensive, CPUs may be robbed of their fair share of the bandwidth. Therefore, the arbitration policy should also be carefully applied.

5.2.4 Homogeneous or Heterogeneous Link Configuration

A homogeneous router configuration has the practical benefit of easier implementation. If all NoC routers are identical, each router module can be duplicated with little or no individual adjustment. Since the requirements of CPU and GPU cores are very different, routers may require higher bandwidth interconnection in terms of the link width or larger buffers to effectively handle traffic from both applications. However, this may result in under-utilization of resources in a certain core. For example, if a wider link width is used, GPGPU applications may directly benefit from more bandwidth capability, but CPU applications may not because they do not require such a high bandwidth. Therefore, the utilization of CPU links will be low. A heterogeneous link configuration may work better in this situation but requires more complex implementation and may not perform as well in some bandwidth-intense situations. However, a heterogeneous configuration will require more design and implementation efforts compared to the homogeneous network. We leave this discussion to future work since this is beyond the scope of our study.

5.2.5 Placement

As explained in Section 5.2.1, any placement of these components – CPU, GPU, L3, MC – may result in unbalanced utilization of on-chip interconnection resources for some scenarios or under-utilization for all situations. Figure 20 shows four possible examples of placement in the ring network. Among these examples, the placement of memory controllers (Figure 20 (d)) in many-core CMPs is studied by Abts et al. [2].

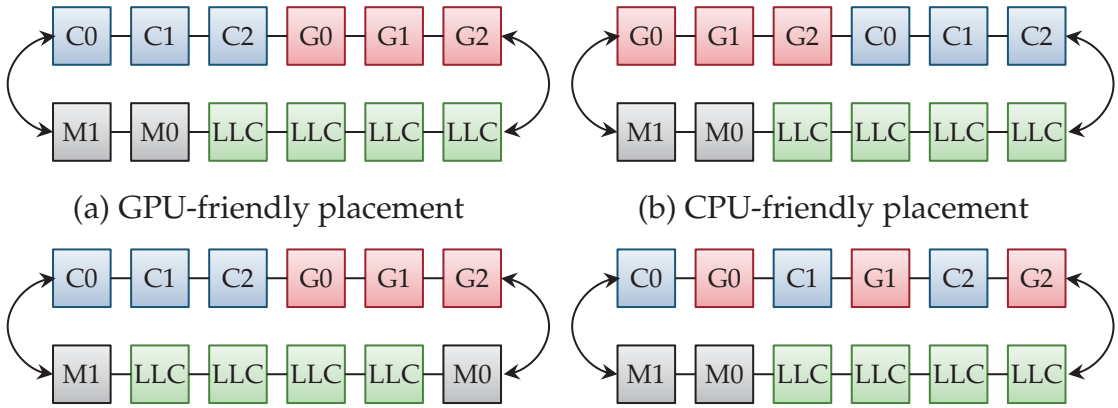


Figure 20: Placement examples in the ring network.

The first two examples are GPU- and CPU-friendly placements. Since all cache misses from a core need to reach L3 caches first, the distance between a core and a target L3 node may have a major impact on performance. Figure 20 (a) shows that the distance between the GPU cores and the L3 caches is shorter than the distance from the CPU cores. If there are more frequent accesses from the GPU cores to the L3 caches, this placement results in better system performance. For the same reason, Figure 20 (b) is more beneficial for CPU applications.

In another configuration, each memory controller is placed at the end of the die in Figure 20 (c). If we can map the disjoint address range of the physical memory for the two types of cores (by the operating system), we can balance the link usage and the latency between each core to the L3 cache and traffic to the memory controllers will be reduced. This setup could effectively divide the chip into two halves, which would be the most beneficial when each half requires the same amount of bandwidth, but would otherwise result in a major imbalance in resource distribution.

Figure 20 (d) shows an interleaved placement, where CPU and GPU cores are interleaved. The possible benefit of this design is that it can balance the traffic in

each direction from each application. In other designs, the traffic from each type of application tends to head in the same direction due to the shortest-distance routing algorithm. When too much traffic is headed in one direction, the application will slow down.

Although some placements are not practical in an actual implementation, this is beyond the scope of our study. We leave this discussion to future work.

5.3 Feedback-Directed Bandwidth Partitioning

In this section, we describe the details of our proposed mechanism: a feedback-directed partitioning-based bandwidth control (VCP) for the NoC in heterogeneous architectures.

5.3.1 Virtual Channel Partitioning

To orchestrate bandwidth more effectively in heterogeneous systems, we propose a *virtual channel partitioning (VCP)* mechanism. VCP dedicates a number of VCs for CPU and GPU cores, similar to cache partitioning mechanisms. By splitting VCs, VCP naturally arbitrates packets that pass through the router (i.e., the current router is an intermediate node, not a destination). Since VCP forces GPU packets to occupy only a part of VCs, CPU packets can occupy an available VC immediately after they arrive. Therefore, CPU packets can be more prioritized compared to unpartitioned VCs. Figure 21 shows the simplified VC arbitration. The VC arbiter is able to identify the type of available VC and tries to select a packet with the same type from input VCs.

To feed CPU and GPU packets based on the corresponding VC availability, VCP requires separate injection queues for CPU and GPU packets in the network interface. Even though dedicated VCs exist for CPU packets by VCP, until a CPU packet arrives at the head of the injection queue, none of the CPU packets can be injected into the NoC under the FCFS scheduling. Under the VCP, a

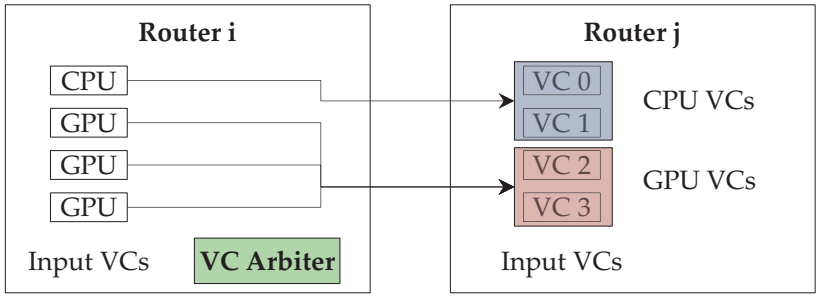


Figure 21: Packet arbitration in VCP

packet can be sent to a router if an available CPU VC exists. To reduce the overhead of having two separate queues, VCP uses dynamically-allocated multi-queue (DAMQ) buffers [134]. DAMQ buffers can maintain separate virtual linked lists for each type of packet with very low overhead, so that the header packet of each type can easily be selected. Therefore, the injection scheduler only needs to check the availability of each VC type and send the corresponding type of VC from its queue, so the scheduler is as simple as the baseline FCFS policy. Figure 22 shows the injection queue and the scheduler of the network interface. DAMQ enables a physically shared, but virtually separate, queue for CPU and GPU packets. The packet scheduler is simple FCFS, but it needs to check the availability of each type of VC, thereby sending a corresponding type of packet.

In this way, VCP can effectively arbitrate packets based on the partitioning configuration, while a more balanced number of packets is provided to the network from the injection queue. Moreover, VCP manages VCs in a very coarse-grain (CPU or GPU partition) manner, which makes the hardware very simple regardless of the number of cores.

5.3.1.1 Where to apply VCP?

Virtual channel partitioning does not have to apply routers that have only CPU or GPU packets since VCP aims to coordinate routers where CPU and GPU packets

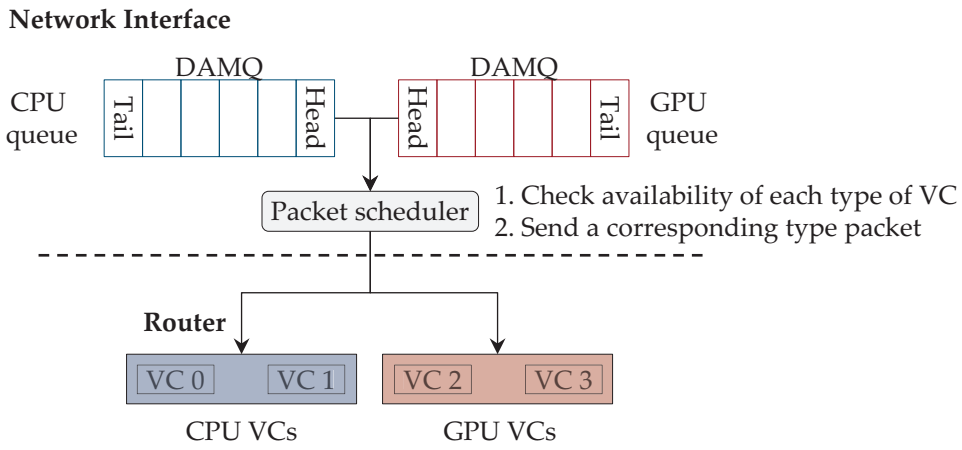


Figure 22: Packet injection from the network interface.

compete. However, in this case, we have to design routers with/without VCP, so design cost may increase. An alternative is for all routers to have enabled VCP by default, but VCP can be disabled by monitoring the number of CPU and GPU packets in a router.

5.3.2 VCP with Different Mixture of Workloads - Adaptability

VCP can have N different partitioning configurations, where N is the number of VCs per port in a router. For example, if a router has six VCs per port, six partitioning configurations are possible: no-partitioning, 1:5 (1 CPU VC and 5 GPU VCs), 2:4, 3:3, 4:2, and 5:1. Since 0:6 or 6:0 configurations will only accept one type of packet, we exclude these configurations. Although VCP can effectively partition on-chip network bandwidth, the exact behavior will be affected by the partitioning configuration as well as by the mixture of workloads running on the heterogeneous system. For example, when CPU applications are running with non-network-intensive GPGPU applications, CPUs will not experience severe interference. Without partitioning, CPUs utilize more network bandwidth, but partitioning will decrease the bandwidth and degrade the performance of CPU applications. Therefore, partitioning should not be used in this case. On the other

hand, with network-intensive GPGPU applications, partitioning must be applied to prevent interference. However, the best performing configuration (for example, 2:2 vs. 1:3 with 4 VCs) can be different for different workloads. To be effective with this type of workload, the performance trade-off between CPU and GPGPU applications should be carefully balanced. Also, the behavior will be affected by the run-time phase as well. Therefore, VCP needs to identify the best performing configuration and adapt run-time behavior.

5.3.3 Feedback-Directed VCP Using Sampling

To estimate the best performing partitioning configuration on heterogeneous workloads, we use a sampling technique for VCP, which we call a feedback-directed VCP (F-VCP), as opposed to static VCP (S-VCP). We sample different VC partitioning configurations across periods and compare the performance of different configurations. F-VCP has the following sampling periods:

- Initial warm-up: To stabilize performance metrics, we idle and disable VCP during this period.
- Training period: To see the performance effect of each configuration, we maintain a configuration for a period. Therefore, the training period consists of N sub-periods with N different configurations. For example, T_1 uses the baseline unpartitioned configuration. T_2 can be a 1CPU-3GPU partitioning configuration. Once a period is over, we collect the number of retired instructions from 1) all CPU applications and 2) a GPGPU application and calculate the speedup over the unpartitioned baseline (T_1) as in Eq. (6) by taking geometric mean of Eq. (7) and Eq. (8).
- Main period: Once the training period is over, if the speedups of all configurations are less than 1, which means partitioning hurts performance,

partitioning will be disabled for the main period. Otherwise, we choose the best performing configuration and apply it for the main period. Once a main period is over, T_1 of the training period will begin.

$$speedup_i = geomean(speedup^i_{CPU}, speedup^i_{GPU}) \tag{6}$$

$$speedup^i_{CPU} = \frac{\#inst\_retired^i_{ALL\_CPU}}{\#inst\_retired^{base(nopartition)}_{ALL\_CPU}} \tag{7}$$

$$speedup^i_{GPU} = \frac{\#inst\_retired^i_{GPU}}{\#inst\_retired^{base(nopartition)}_{GPU}} \tag{8}$$

We set period lengths as in Table 9 from empirical data. With four VCs in the baseline, we test three partitioning configurations (unpartitioned, 1:3, and 2:2), so the training period resumes every 4.6 M cycles.

Table 9: The length of each period in VCP.

Length of initial period	500K cycles
Length of each training period	200K cycles
Length of main period	4M cycles

5.3.3.1 Central Decision Logic

To collect performance metrics from cores, VCP requires a central decision logic (CDL), which is located at the central node of the mesh. We use a similar approach in previous work [24, 25]. When a training period is over, CDL broadcasts to all cores a message that includes the configuration for the next period. Cores maintain the previous policy until they receive the message from CDL. Once they receive the message, they change the policy and send a performance metric during the last period to CDL. Once CDL collects messages from all cores, it will store the results. After all training periods are over, CDL decides the best configuration based on Eq. (6) and sends the decision to all routers. These processes may take up to a few

hundred cycles. Since the length of periods is much longer, the overhead is not so significant.

5.3.3.2 *Why sampling works*

This sampling mechanism can be viable since both CPU and GPGPU applications have shown a similar periodic progress in terms of the number of retired instructions. This is because 1) the GPGPU application is running in a single-program multiple-data (SPMD) manner. Although each thread has a different behavior at a time, this phase behavior becomes blurred by the other hundreds of concurrently running threads and 2) each CPU application has its own phase behavior. However, if we treat all CPU applications as a whole, the phase behavior of each application becomes faint as well. Figure 23 shows a phase example of a heterogeneous workload.

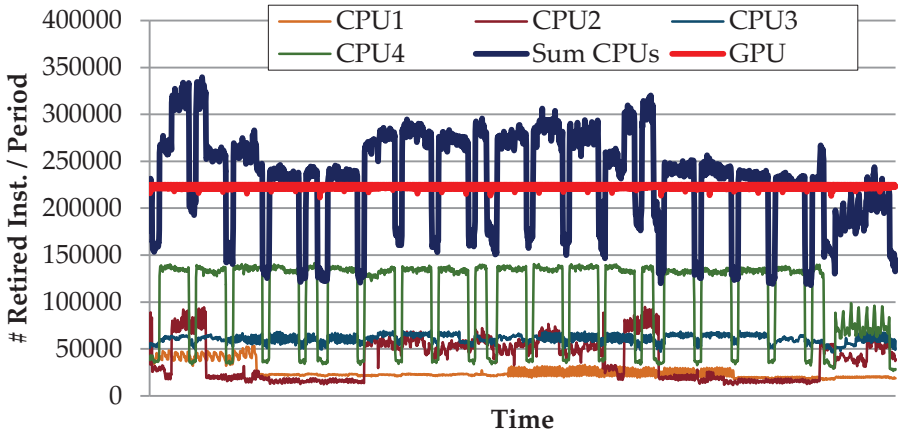


Figure 23: Phases in a heterogeneous workload (Sum: sum of CPU retired instructions).

The GPGPU application (GPU line) shows similar progress across the entire duration. Although CPU applications show some fluctuations, the sum of all CPU applications (Sum line) maintains a similar progress for a sufficient time, so that our sampling mechanism can successfully differentiate the effect of different partitioning configurations.

5.3.3.3 *Drawbacks and Improvements of Sampling*

Sampling may have the following weaknesses: 1) effect of training period length and 2) running non-optimal configurations during training periods.

First, for each training period, our F-VCP requires the collection of performance information as well as network injection information from cores. If the training period is too short, the overhead of communicating information will be too high. Also, we cannot acquire precise performance information due to a performance variation during a short period. On the other hand, if the period is too long, we lose opportunities for improving performance of the system since F-VCP cannot adapt well to run-time behavior changes. To find the optimal period length, we have to identify the phase behavior and adjust the period length accordingly.

Second, during the training period, F-VCP will use N-1 non-optimal configurations, where N is the total number of configurations. However, N is very small in F-VCP since on-chip routers usually have very limited buffer space (only 4 to 6 VCs). Even if many VCs exist, since we observe that the overall system performance is linearly increasing, decreasing, or has the peak in the middle across consecutive configurations, we can employ existing *on-line training techniques* to reduce the overhead of having many non-optimal configurations. Moreover, we can *linearly lengthen the main period* if the same configuration is chosen consecutively after the training periods. If two consecutive decisions are different, then we reset the length of the main period to the original length.

In order to reduce the overhead of sampling, we can also consider on-demand sampling. As explained in Section 5.3.3.2, the performance of the system may show similar progress across periods. Once we find an optimal partitioning configuration through the sampling periods, we can maintain this policy until the performance of the system changes (improves or degrades). Performance changes are due to the run-time phase change and indicate that the current configuration

might not be optimal. However, this approach also has a drawback since it may cause a shorter main period and more communications when frequent phase changes exist.

5.3.4 Hardware Changes and Overhead

Our VCP requires the following hardware changes.

- Router arbiters should store VC partitioning configuration, which requires only a few bits, and be able to check the type of packets (CPU and GPU). Also, an arbitration algorithm needs to be changed such that dedicated VCs enforce that only the packet with the same type can acquire them. The arbiter with these changes is less complex than that of previous mechanisms since it only needs to match the type of packet with a VC.
- We have discussed the need for DAMQ [134] in Section 5.3.1. The overhead of DAMQ is known to be insignificant.
- We have discussed the central decision logic in Section 5.3.3.1.

As a result, our VCP does not require significant changes to the baseline routers and the overhead is almost negligible.

5.3.5 Extension of VCP

VCP can be combined with other NoC mechanisms. Since the goal of VCP is to avoid significant interference by GPU cores, VCP only differentiates CPU or GPU packets. If we want to further differentiate individual applications, other mechanisms can be applied on top of VCP. For example, Aergia [25] can set a different priority for each packet. Within the same VC partition (CPU or GPU), the arbiter can schedule packets based on their priority. We evaluate this VCP extension in Section 5.5.4.

VCP can also coordinate with cache management schemes, such as TAP [72], which tries to find the best cache partitioning configuration between the CPU and GPU in heterogeneous architectures. Our VCP and TAP aim to solve a similar problem. Also, caches and NoCs are not independent and affect each other significantly. Therefore, combining VCP and TAP will yield even better results. However, managing one will affect the other, so combining two and studying their interactions are not trivial, which is beyond the scope of our work.

5.3.6 Discussions

In this section, we discuss possible issues with VCP.

- **Deadlock** will not occur in VCP since CPU and GPU packets will occupy at least one VC and we use the oldest-first policy between the same type of packets.

- If applications always show unpredictable **phase changes**, sampling may misidentify the best performing configuration. Although we detect some workloads with phase changes, our observation is that if partitioning has a significant impact, it can overcome errors. Thus, the negative effective of dramatic phase changes is not so severe.

- No problem will occur during the **transition period** because the VC arbiter defines the allowed type and always searches all input VCs and matches the type.

- Although we consider only two types of heterogeneous cores (CPU and GPU) throughout this chapter, more complex heterogeneous systems exist. For example, most SoC (system-on-chip) architectures, including smart-phones and tablets, have CPUs, GPUs, DSPs, and multiple modems and all these components share the same system resources. Future multi- and

many-core systems may also have several different types of accelerators. In this case, we may need to add more VC types other than CPU and GPU VCs. However, considering the limited number of VCs, we may need to reduce the number of different VC types. We can achieve this by 1) forcing different types of processors to use the same type of VC or 2) letting some processors utilize any type of VCs. This decision should be made by identifying the characteristics of processors and applications running on them, but we do not discuss this further since this is beyond the scope of our work.

- Packets that carry performance metric information (from cores) and decision information (from CDL) are treated as special packets and they can utilize any VC types.

5.4 Evaluation Methodology

5.4.1 Simulator

We use MacSim [45], a trace-driven and cycle-level heterogeneous architecture simulator, for evaluations. For all evaluations, we repeat early terminated applications until all applications have finished at least once. This is to model the resource contention uniformly across the duration of simulation, which is similar to the work in [58, 72, 120, 150]. Table 10 shows the processor configuration. To model a next-generation heterogeneous architecture, we model our baseline CPU similarly to Intel's Sandy Bridge [52], with high-end GPU cores that are similar to the SM (streaming multiprocessor) of NVIDIA Fermi [103].

Table 11 shows the NoC configuration. Although we use a conservative five-stage pipeline model, we include the VCP result with a three-stage pipeline router model in Section 5.5.5. Also, the routers do not use any pipeline bypassing mechanisms, which can reduce latencies by skipping some pipeline stages when switches/links are idle. However, when operating in the regions where congestion

Table 10: Processor configuration.

CPU	4 cores, 3.5GHz, 4-wide, out-of-order (OOO) gshare branch predictor 8-way, 32KB L1 D/I cache, 2-cycle 8-way 256KB L2 cache, 8-cycle
GPU	6 cores, 1.5GHz, in-order, 2-way 16 SIMD width 8-way, 32KB L1 D (2 cycle), 4-way 4KB L1 I (1 cycle) 16KB s/w managed cache
L3 Cache	4 tiles (each tile: 32-way, 2MB), 64B line, LRU
Memory Controller	DDR3-1333, 2 MCs (each 8 banks, 2 channels) 41.6GB/s BW, 2KB row buffer, FR-FCFS scheduler

dominates latency, bypassing provides minimal benefit. As explained, queuing delay, not trip delay, is dominant in our evaluated workloads, so the baseline router model performs similarly to the bypassing router.

Table 11: NoC configuration.

Frequency	1 GHz
Topology	4x4 2D Mesh
Pipeline	4-stage (IB, RC, VCA, SA/ST)
# VCs	4 per port, each VC can hold 5 flits * a packet can have at most 5 flits
# ports	5 per router
Link	128 bits (16 B) with 1-cycle latency
Routing	X-Y
Flow control	credit-based
Placement	Base in Figure 25

5.4.2 Placement

In this section, we discuss the placement of components in the heterogeneous architecture. Several different methods to place CPU/GPU cores, LLC tiles, and memory controllers can exist. For example, Abts et al. [2] discussed how to place memory controllers in a homogeneous mesh network. However, the placement of cores and other components is not discussed to the best of our knowledge.

Although identifying the best placement for a heterogeneous architecture is beyond the scope of our study, instead, we discuss the basic placement ideas and reasoning for our designs.

First, placement must be carefully designed. For example, Figure 24 shows two designs where paths to memory routers (LLCs and MCs) are overlapped from CPU and GPU cores. The assumption here is that all communications between CPU and GPU cores are made only through caches. In these examples, intermediate nodes may suffer from much through traffic due to the overlapped path, which may lead to significant system performance degradations.

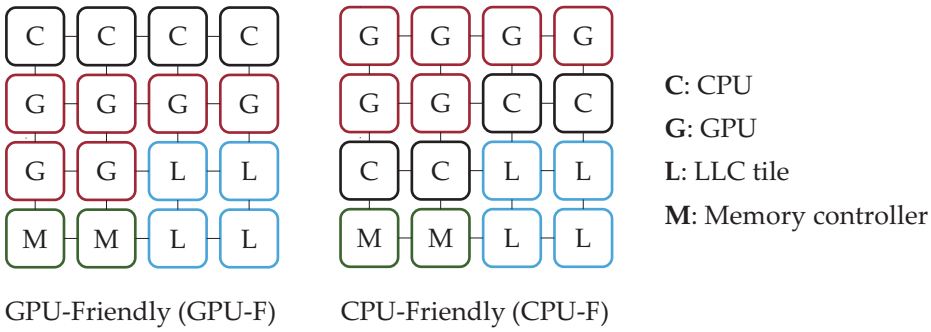


Figure 24: Placement designs with the overlapped path.

Figure 25 shows alternative designs that do not have overlapped path. In all three alternatives, CPU and GPU cores have distinct routes to the memory routers while the placement of LLC tiles and memory controllers varies. The shaded area shows all routers that may have both CPU and GPU packets. Among these placements, we use Baseline (Base) placement in Figure 25 and we evaluate other placements in Figures 24 and 25 in Section 5.5.8.

5.4.3 Benchmarks

We use SPEC 2006 CPU benchmarks and CUDA GPGPU benchmarks from Nvidia CUDA SDK, Rodinia [20], Parboil [137], and ERCBench [17]. For the CPU workloads, Pinpoint [113] was used to select a representative simulation region

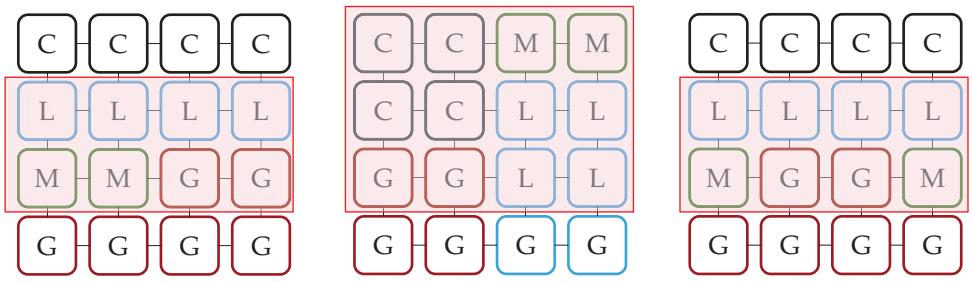


Figure 25: Alternative placement designs (shaded area shows routers that may have both CPU and GPU packets).

with the reference input set. Most GPGPU applications run until completion. Tables 12 shows the CPU and GPGPU benchmarks used for evaluations. We categorize benchmarks into two groups (High and Low network-intensive) based on the packets per kilo cycles (PKC). We use this metric since PKC clearly shows the network intensity of an application. To distinguish low and high groups, we use PKC of 20 and 100 for CPU and GPU applications, respectively.

Table 13 shows evaluated heterogeneous workloads. For all workloads, we run four CPU applications on four CPU cores along with one GPGPU application on six GPU cores. We categorize CPU workloads based on the number of high-type CPU applications (out of four). We choose each application pseudo-randomly.

5.4.4 Evaluation Metric

We use a speedup metric defined in Eq. (9). First, we compute the speedup of each application with a configuration over the baseline unpartitioned configuration (Eq. (11) for CPU and Eq. (12) for GPGPU). Then, we calculate the average speedup of all CPU applications (Eq. (10)). Finally, we take the average of Eq. (10) and Eq. (12).

Table 12: Benchmark characteristics based on the network-intensity (PKC is measured for an entire application. i.e., sum of core PKC).

	High (PKC > 20)		Low (PKC < 20)	
	Bench	PKC	Bench	PKC
CPU	GemsFDTD	58	povray	1
	wrf	63	gamess	2
	bwaves	69	namd	3
	cactusADM	73	sjeng	4
	milc	74	gobmk	6
	leslie3d	84	tonto	10
	lbm	90	perlbench	12
	High (PKC > 100)		Low (PKC < 100)	
	Bench	PKC	Bench	PKC
GPU	nearest-neighbor	166	Dxtc	0.4
	stencil	241	VolumeRender	3.0
	ScalarProd	253	cell	5.3
	bfs	304	raytracing	5.9
	cfd	331	AES	26
	Reduction	417		
	BlackScholes	437		
	SobolQRNG	452		

$$speedup = geomean(speedup_{CPU}, speedup_{GPU}) \tag{9}$$

$$speedup_{CPU} = geomean(speedup_i, where\ 0 \leq i \leq 3) \tag{10}$$

$$speedup_i = IPC_i / IPC_i^{baseline(nopartition)} \tag{11}$$

$$speedup_{GPU} = IPC_{GPU} / IPC_{GPU}^{baseline(nopartition)} \tag{12}$$

Although we use a geometric speedup metric throughout this chapter, our

Table 13: Heterogeneous workloads.

	# High type CPU	GPU type	#	Reference
W-LL	no more than 1	Low	10	5.5.1 only
W-HL	more than 2	Low	13	Entire Section 5.5
W-LH	no more than 1	High	13	
W-HH	more than 2	High	13	

mechanism is not limited by a specific metric. Since our target heterogeneous architecture is an emerging architecture, how to evaluate this architecture is debatable. Regardless, our VCP can easily adapt to any desirable metrics by replacing Eq. (6) and Eq. (9).

5.5 Evaluation Results

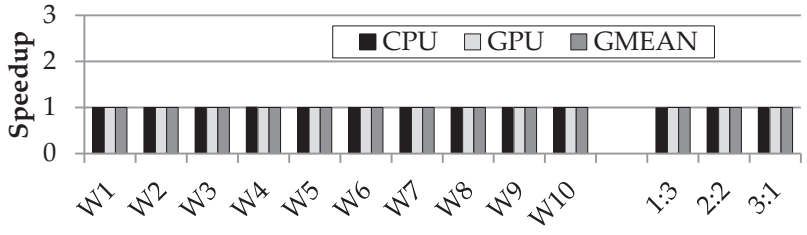
5.5.1 Static VCP Results

First, we show in Figure 26 the VCP results with static configurations (S-VCP) for different workloads to show how VCP affects performance (detailed results of 10 workloads and the average of all configurations). No significant difference exists across all configurations in *W-LL* workloads. Since CPU and GPGPU applications are not network-limited, performance is hardly affected by different configurations. For this reason, we exclude *W-LL* workloads in further evaluations.

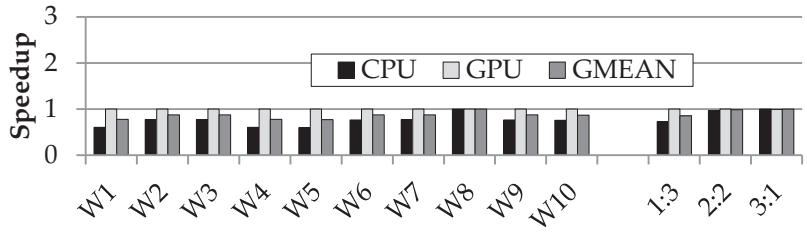
For *W-HL* workloads, since CPU applications can utilize network bandwidth well without partitioning, VCP rather degrades the performance of CPU applications when only a small number of VCs are dedicated for them (1:3 configuration)[1], while the performance of the GPGPU application is not improved at all. As a result, the overall performance is degraded. On average, 1:3, 2:2, and 3:1 static partitioning show 15%, 2%, and 1% degradations, respectively. On the other hand, for *W-LH* workloads, although CPU applications are not network-limited, they experience moderate interference. As a result, dedicating one VC to CPU applications will be sufficient, but too many VCs will degrade the performance of GPGPUs severely. The 1:3 configuration shows an 8.5% improvement, while the 2:2 and 3:1 configurations show 9% and 30% degradations, respectively.

W-HH workloads show very complex behavior. The 2:2 and 1:3 configurations mostly show a benefit, but the always-winning configuration does not exist.

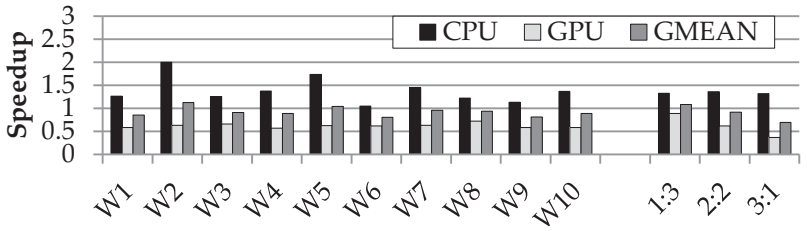
[1]We use #*CPU-VC*:#*GPU-VC* notation for static configurations. For example, 1:3 indicates one CPU VC and three GPU VCs.



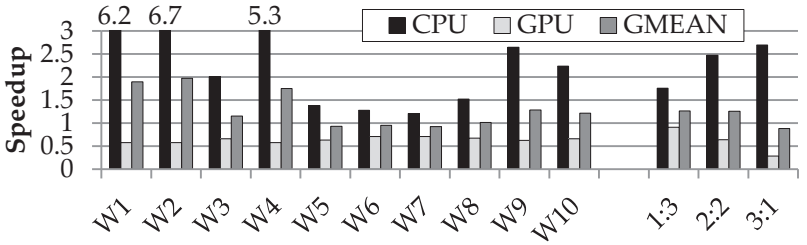
(a) W-LL workload



(b) W-HL workload (first 10 bars with 1:3 configuration)



(c) W-LH workload (first 10 bars with 2:2 configuration)



(d) W-HH workload (first 10 bars with 2:2 configuration)

Figure 26: Static VCP results.

Based on the workloads, the performance variance of the two configurations is significant. Dedicating more VCs for CPU applications will improve performance significantly, but giving more ways diminishes return. Meanwhile, the GPGPU application suffers severe degradation. We have to balance VC partitioning based on the workloads. Overall, 1:3 and 2:2 improve 25%, while 3:1 degrades 13%. The 2:2 configuration shows much better CPU performance, but the performance of GPGPU application is significantly degraded.

5.5.2 Feedback-Directed VCP Results

Figure 27 shows the feedback-directed VCP (F-VCP) result along with static configuration results. As we give more VCs to CPU applications (1:3 to 3:1), the performance benefits of CPU applications diminishes, while performance degradation of the GPGPU application becomes more significant. F-VCP can identify the best static configuration with different workloads, so it can improve CPU applications significantly (46% improvement) while hurting GPGPU very little (9% degradation). F-VCP improves system performance by 15% on average, while the best static configuration (1:3) shows a 9% improvement.

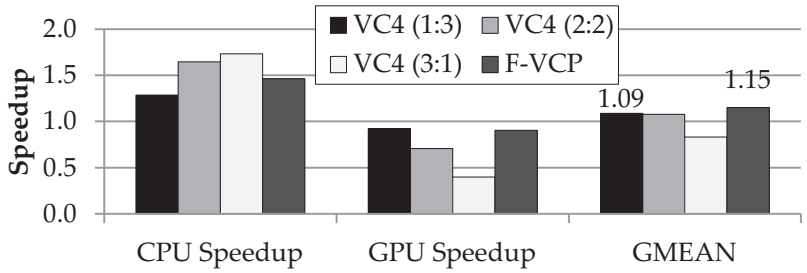


Figure 27: Feedback-directed VCP results.

We show the s-curve of F-VCP in Figure 28 for detailed analysis.[2] F-VCP mostly shows better results than the best of all static configurations. Moreover, across 39 workloads, the maximum performance degradation over the baseline is only 2.5%

[2]We sort workloads by the performance of F-VCP in ascending order.

and only two workloads result in more than a 1% degradation.

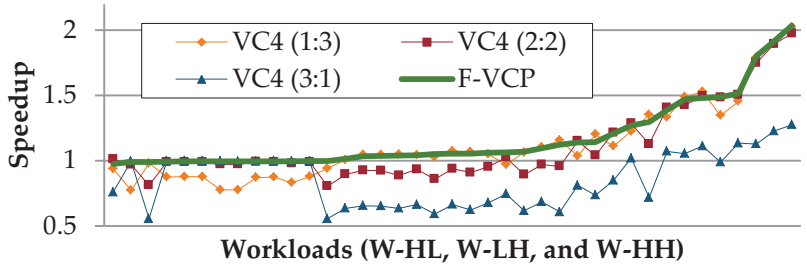


Figure 28: F-VCP s-curve (workloads are sorted by the performance of F-VCP in ascending order).

Also, to show how F-VCP works, we show the average packet latency changes in Figure 29. We can observe that traverse time is almost the same, but the queuing delay of CPU packets decreases significantly, while that of GPU packets increases.

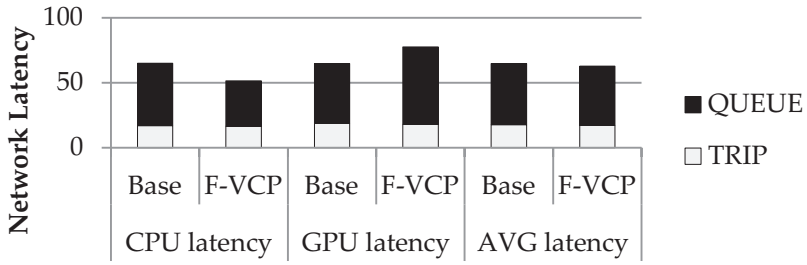


Figure 29: Network latency changes with F-VCP.

Figure 30 shows a policy distribution histogram for each workload. Although F-VCP constantly chooses one configuration in some workloads, many workloads show that F-VCP adapts well to application phase changes if they exist.

5.5.3 Comparison with Different Injection Buffer Scheduling

As mentioned, an unbalanced number of packets between the CPU and GPU exist in the injection buffer when CPU and GPU applications share the network. In order to solve this unbalance problem, we can consider effective packet scheduling, which can be made through out-of-order scheduling. In the in-order injection buffer, a packet should wait until all previous packets get serviced. On the other

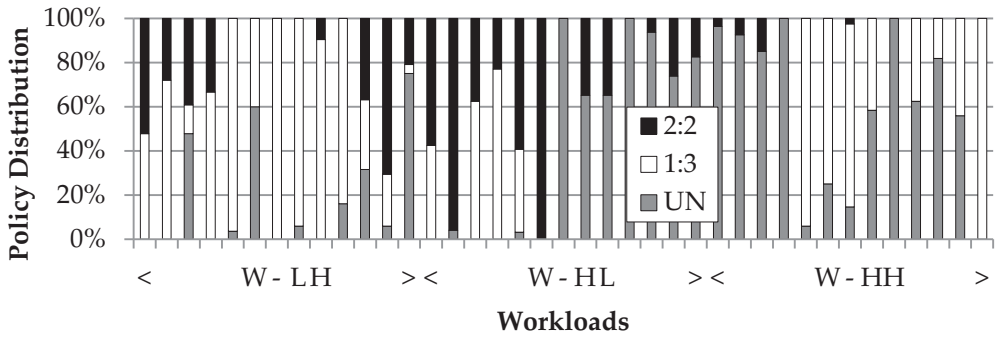


Figure 30: F-VCP policy distribution (UN: unpartitioned).

hand, out-of-order scheduling can prioritize one packet over others. Thus, in this section, we evaluate several packet scheduling policies applied to the injection buffer. In addition, we also evaluate the DAMQ-based injection queue that virtually has separate queues for CPU and GPU packets, and F-VCP.

We evaluate several packet scheduling policies applied to the injection buffer without VC partitioning. Also, we apply ATLAS [66], one of the state-of-the-art memory schedulers, to the packet scheduler. Since ATLAS prioritizes applications that attained the least service during previous periods, ATLAS fits well to prioritize CPU packets that usually attain fewer services than GPU packets in heterogeneous workloads. We summarize all evaluated polices as follows:

- Baseline - first-come first-serve policy
- CPU-first - CPU packets always have higher priority than GPU packets (batching is used for preventing starvation).
- GPU-first - GPU packets always have higher priority.
- ATLAS-A - ATLAS with application granularity.
- ATLAS-C - Similar to ATLAS-A, but we distinguish only two groups: GPGPU or CPU applications.

- MPI - Based on the private cache miss-per-instruction (MPI), we prioritize applications that have lower MPI.
- DAMQ - CPU and GPU packets have virtually separate queues using DAMQ. Scheduling between queues is round-robin.

Figure 31 shows the results. When out-of-order scheduling is applied to the shared injection buffer (other than DAMQ and F-VCP), CPU packets can be prioritized at a moment, but packet occupancy is still very unbalanced with the shared buffer. This limits the benefit of injection buffer scheduling. All evaluated policies show negligible benefits, but only the CPU-first policy shows a 2% improvement. On the other hand, by having separate queues with round-robin scheduling between them, we can mitigate the occupancy unbalance problem. As a result, DAMQ can improve performance by 8%. However, DAMQ cannot outperform F-VCP since the effect of coordination in the injection buffer is limited in virtual channels of routers.

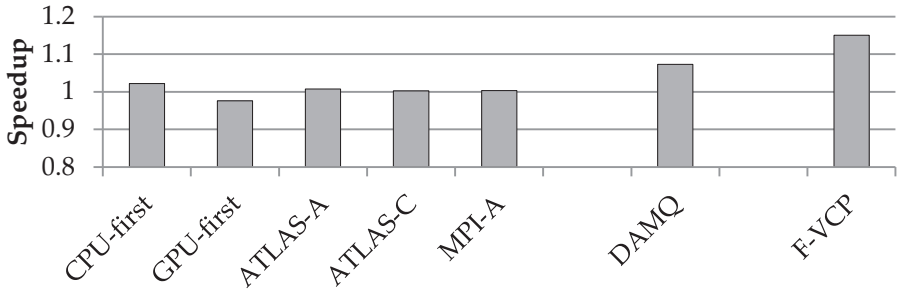


Figure 31: Different injection buffer scheduling results.

From the observations made in this section, we can draw the conclusion that separate queues are favorable to better performance in heterogeneous architectures, but the VC arbitration should be considered at the same time to be more effective, as in VCP.

5.5.4 Comparison with VC Arbitration Policies

In this section, we compare F-VCP with previous VC arbitration mechanisms, application-aware prioritization (denoted as STC) [24] and Aergia [25] along with two static policies, CPU-first and GPU-first.

STC computes the network demand of applications at intervals by looking at a number of metrics such as private cache misses per instruction, average outstanding L1 misses in MSHRs, and average stall cycles per packet. This produces a ranking of applications, and all packets of one application are prioritized over another, resulting in a coarse granularity of control. To prevent application starvation, a batching framework is implemented that prioritizes all packets of one time quantum over another, regardless of source application. Aergia predicts the available latency (slack) of any packet by the number of outstanding L1 misses and prioritizes low-slack (critical) packets over packets with higher slack when they are within the same batching interval. Static policies always give a higher priority to certain types of packets (either CPU or GPU) and form batches to prevent the starvation problem. Moreover, we apply STC and Aergia to the DAMQ-based injection buffer (DAMQ+S and DAMQ+A) and F-VCP (F-VCP+S and F-VCP+A). Figure 32 shows the results.

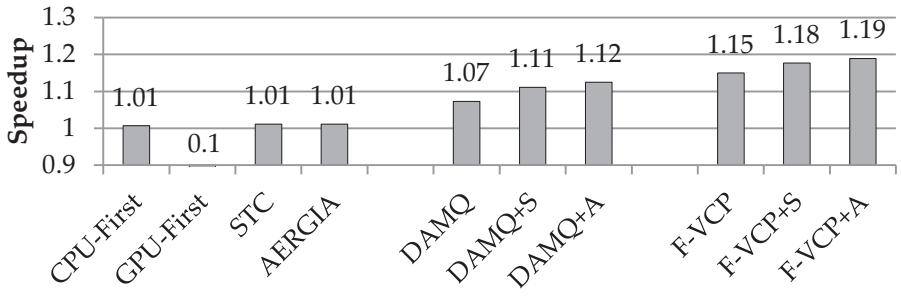


Figure 32: Evaluation of virtual channel arbitration policies.

As explained in Section 2.3.2, NoC mechanisms for heterogeneous architectures should have separate injection queues for CPU and GPU packets. As a result, STC,

Aergia, and CPU-first policies without separate injection queues show around a 1% improvement on average. In a few workloads, Aergia shows up to a 1.55x speedup. These workloads have relatively high CPU-to-GPU packet ratio, so Aergia also can be effective. However, Aergia degrades the performance of almost half of the workloads (10% degradation at most). This is because fine-grain prioritization prevents some CPU applications from being prioritized, but Aergia mostly degrades GPGPU performance by not prioritizing them.

When STC and Aergia are applied along with separate injection queues (DAMQ+S and DAMQ+A), they can be more effective. Since separate injection queues provide a more balanced number of packets between CPU and GPU applications, router arbiters see a similar number of packets, and are thereby effective. STC and Aergia provide 4% and 5% additional performance improvements on top of DAMQ.

On the other hand, our F-VCP successfully manages on-chip routers with almost no degradation cases. Moreover, as discussed in Section 5.3.5, VCP can be extended using previous VC arbitration mechanisms, such as STC and Aergia. We also evaluate these extensions, which are denoted F-VCP+S and F-VCP+A for STC and Aergia, respectively. Even though F-VCP already improves performance by 15%, STC and Aergia provide additional improvements of 3% and 4% by arbitrating packets in finer granularity.

5.5.5 VCP Results with Three-Stage Pipeline Model

As described in Section 5.4.1, we use a conservative five-stage pipeline model in all evaluations so far. This may incur extra latencies in the network. However, as claimed, a shorter-latency router model can improve performance, but it cannot entirely resolve the resource contention problem since the dominant delay occurs in the injection queues. In order to confirm that VCP works well with a faster router design, we re-evaluate the same set of experiments as in Section 5.5.4 with

a three-stage pipeline model. Figure 33 shows results.

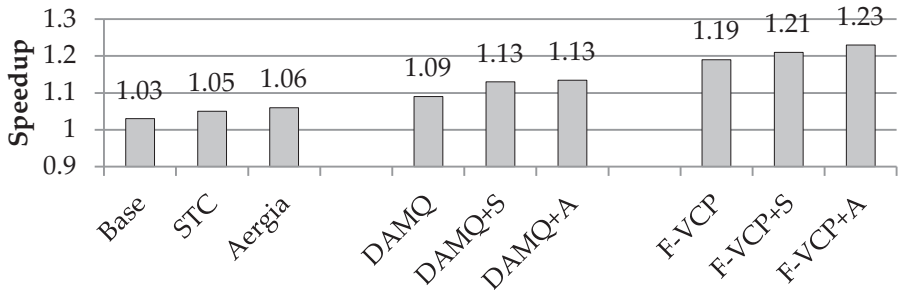


Figure 33: Evaluation of three-stage pipeline router model (normalized to the router model with five-stage pipeline).

Across 39 heterogeneous workloads, the three-stage pipeline model improves performance by 3% over the five-stage pipeline router. Interestingly, the benefit of the shorter-latency router model provides performance improvements of all configurations by 2% to 3%. Moreover, VCP yields an additional 16% performance improvement over the three-stage pipeline model. From this experiment, we can confirm that 1) the shorter-latency router model can improve the performance of the network as well as the system, 2) the network congestion still exists even with a shorter-latency model, and 3) VCP is still an effective solution.

5.5.6 XY/YX Adaptive Routing

In order to optimize the baseline network, we can consider adaptive routing as well. For example, in our baseline placement (Base in Figure 25), although memory routers (L3 and memory controllers) are shared, there are distinct routes from CPU and GPU cores to memory routers. When we use static XY routing only (Figure 34 left), packets must traverse within the shared memory routers. Instead, to reduce the contention in the memory routers, we can use XY/YX adaptive routing (Figure 34 right). When a core sends a request packet, it uses XY routing. When the packet is returned with data, it now uses YX routing. As a result, we can reduce the traversal within the memory routers. Figure 35 shows the result of

XY/YX adaptive routing along with VCP.

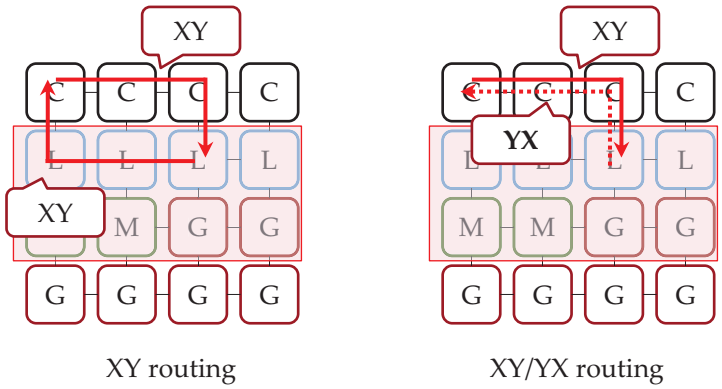


Figure 34: Adaptive XY/YX routing.

As expected, XY/YX routing significantly improves performance by 10%. This is because XY/YX routing reduces the network congestion while improving the network utilization. We can also observe that VCP is still effective and gives an additional 13% performance improvements by mitigating network contentions to the memory routers.[3]

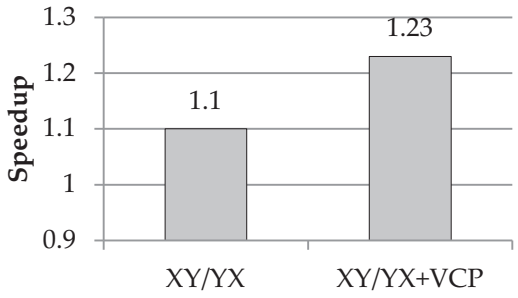


Figure 35: Adaptive XY/YX routing results.

[3]Please note that this optimization is specific to our baseline placement and may not be effective on other configurations.

5.5.7 Sensitivity of VCP

In this section, we evaluate F-VCP with different configurations. Figure 36 shows the F-VCP results with a different number of VCs.[4] In each bar, we compare F-VCP with the baseline router with the same number of VCs (i.e., VC6-F-VCP with VC6). F-VCP performs well with more VCs, but the benefit decreases in VC8 and we expect diminishing improvement with more VCs. Generally, a higher number of VCs perform better by reducing the congestion, so the benefit of F-VCP can also decrease. However, the space for the buffer is limited in on-chip routers, so the number of VCs also has limitations. As a result, we expect that F-VCP will work well within this limitation.

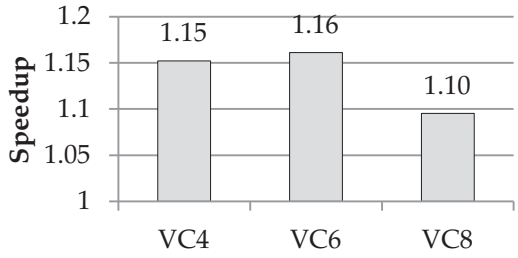


Figure 36: F-VCP with different number of VCs.

We briefly discussed in Section 5.3.3 how different lengths of training periods will affect F-VCP. We perform experiments with different lengths of training periods. Figure 37 shows the results.[5] Generally, different lengths of training periods would not matter on average, but the 800K configuration shows significant variances (from 0.53 to 1.28 speedup). A lengthy period can help reduce the overhead of sampling, but it may fail to adapt run-time behavior.

[4]We fix other configurations the same. Each VC has four buffer entries, so the number of total buffer entries is 4 * # VCs.

[5]We fix the length of the main period 20 times that of the training period.

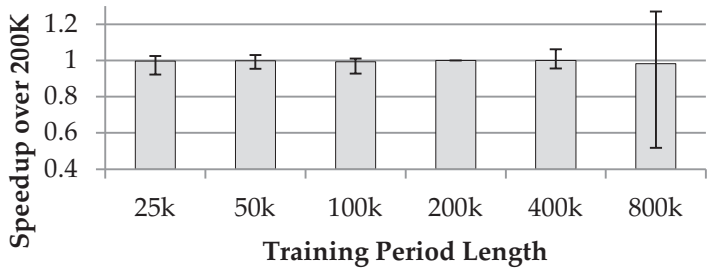


Figure 37: F-VCP with different length of training period (base: 200K).

5.5.8 Different Placement Results

As discussed in Section 5.4.2, we evaluate different placements in this section. Figure 38 shows the results of placements in Figures 24 and 25. All results are normalized to the baseline (Base) placement in Figure 25.

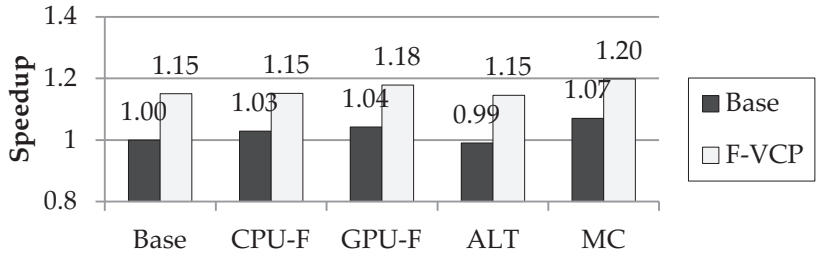


Figure 38: Different placement evaluations.

Even though the designs in Figure 24 have the overlapped paths to the memory routers between CPU and GPU cores, CPU-Friendly and GPU-Friendly designs show 3% and 4% improvements over the baseline, respectively. The trip latency can increase, but dominant delays occur from the injection buffer. By having a shorter distance from memory to CPU (CPU-Friendly) or GPU (GPU-Friendly), queuing delays decrease. On the other hand, the design that distributes memory controllers shows the overall best performance (7%). As discussed in [2], distributing congestion near memory controllers is a key reason for the improvements. With all different placement designs, VCP constantly shows higher than 11% improvement across all alternative designs. From this experiment, we

conclude that different placement affects the performance and VCP can control network bandwidth effectively when network congestion exists regardless of the placement.

5.5.9 Discussions

We discuss F-VCP with other possible configurations that we do not show in this section.

1. Although we do not evaluate larger meshes with more cores, we expect that F-VCP will still be effective. As the size of the network increases, queuing delays can be reduced due to more diverse paths, but it can increase overall traffic from more cores. As long as network congestion exists, F-VCP can successfully arbitrate between CPU and GPU packets.

2. We have treated CPU and GPGPU applications with equal weight so far. When a user or a system wants to have a different weight for CPUs and GPUs, F-VCP requires a very minor change. It only requires changing the feedback metric, which is defined in Eq. (6). Changes in the rest of the system are not necessary.

5.6 Summary of This Chapter

How the NoC for heterogeneous architectures is handled has significant importance. Due to the heterogeneity of CPU and GPU cores, more specifically much higher network injections, CPU applications often suffer from severe interference. Previous mechanisms proposed for homogeneous CMPs have limitations to solve the network resource sharing problem in this architecture. In this work, we propose feedback-directed virtual channel partitioning (F-VCP). On-chip network bandwidth can be controlled by the proposed VCP, which arbitrates packets that pass through the router while providing a more balanced number of packets to

the NoC using DAMQ-based separate injection queues. Across 39 heterogeneous workloads, our VCP shows a 15% improvement compared to the unpartitioned router. We perform thorough evaluations with many different configurations and VCP shows robustness. For future work, we will develop a performance model with on-chip bandwidth partitioning to improve the sampling-based technique in VCP.

CHAPTER VI

DYNAMIC FREQUENCY REGULATING MECHANISM

This chapter proposes a dynamic clock frequency regulating mechanism called DyFR that aims to solve resource contention problem while considering the scalability characteristic of applications. DyFR uses a DVFS technique to reduce the clock frequency of interference-causing application and to improve the performance of application when linear speedup is expected with the frequency increase.

6.1 Introduction

As the technology scales, more features can be implemented on the chip. As a result, in recent processors, we can easily find that a heterogeneous mixture of processing units is packed together on the same chip. For example, recent system-on-chip (SoC) architectures, such as smartphones and tablets, include CPUs, GPUs, DSPs, and other units in the same chip. Recent desktop processors [5, 49, 106] integrate on-chip GPUs along with CPU chip multiprocessors (CMP). This trend is inevitable since general-purpose processors cannot perform well on all kinds of workloads. With the technology scaling, we expect more diverse accelerators can be integrated in future processors.

In this architecture, many system resources are shared among different processing units, for example shared last-level cache, on-chip interconnect network, memory controllers, and DRAM memories. This sharing provides cost-effective implementation and efficient communications among processing elements. However, due to the sharing, the resource contention problem occurs

between cores and applications. Although this problem has existed since the CMP was introduced, as reported in [72], a heterogeneous mixture of cores exerts more pressure on shared resource management. In particular, applications running on GPU cores severely interfere with CPU applications in this architecture.

A rich body of literature exists on previous mechanisms targeting a variety of shared resources to address the resource sharing problem, for example last-level shared cache [58,120], on-chip interconnection network [24,25], and memory controllers [66, 67]. These mechanisms try to minimize the inter-application interference and prioritize more critical cores or applications based on their characteristics. These mechanisms show effectiveness, but they may not be optimal in terms of the power aspect. When resource contention of a system is severe, although quality-of-service (QoS) and fairness can be improved by previous mechanisms, some cores may suffer from processor stalls while waiting for previous memory requests to be serviced. In this case, operating cores more slowly than the base clock frequency does not affect the performance of the core or system throughput but does reduce power consumption. In addition, the benefit of previous mechanisms is limited when no resource contention exists in the system, i.e., a workload consists of all compute-intensive applications.

To overcome the weakness of previous mechanisms, thereby improving performance and power efficiency simultaneously, we utilize dynamic voltage and frequency scaling (DVFS) to solve the resource sharing problem, as opposed to previous approaches. DVFS is a well-known power control technique. DVFS-based mechanisms try to save power by decreasing the voltage/frequency of idle components and improve performance of active components using increased frequency [4,53]. Also, recent proposals try to identify the optimal number of active cores and clock frequency of cores in CMPs and GPUs [74,77]. These mechanisms try to maximize system throughput within the power budget by considering the

performance-power model of individual cores or applications, but they do not consider the resource contention problem. On the other hand, our approach tries to mitigate interference by decreasing clock frequency of interference-causing cores or applications. Unless the network and memory bandwidth are saturated, the number of memory requests from a core is proportional to the clock frequency.

At the same time, we also consider the power partitioning between cores and the memory (interconnection network and shared last-level caches) while considering the frequency scalability of applications. Based on the memory intensity and clock frequency, the power and energy efficiency of an application can vary. For example, higher frequency does not always improve performance since the performance bottleneck may be in the relatively slow memory system and cores cannot make progress even with higher frequency while waiting for memory requests to be serviced. On the other hand, most compute-intensive applications show linear scalability with regard to frequency increases and memory performance does not affect the system throughput much.

To this end, we propose *DyFR* , dynamic frequency regulating mechanism. We observe that the degree of interference can be measured by monitoring cache misses per time (MPT), while the scalability of an application can be determined by misses per kilo instructions (MPKI). MPKI is a property of application, but MPT is a function of MPKI and the operating clock frequency of a core. The main algorithm of DyFR initially tries to isolate the interference caused by GPU applications by lowering the voltage and clock frequency of GPU cores. Then, based on the scalability of CPU applications, the clock frequency of individual CPU cores is dynamically adjusted as well. After evaluating the current power budget of all cores, we adjust different DVFS level to the memory based on the importance of memory.

We claim our contributions to be as follows:

1. We propose a DVFS technique, called *DyFR* , to tackle the resource sharing problem in heterogeneous architectures.

2. *DyFR* considers the property of application and the degree of interference caused by the application.

3. *DyFR* improves the system throughput by 14 % while reducing energy consumption by 23%

The remainder of this chapter is organized as follows. Section 6.2 discusses the motivation for a DVFS mechanism to solve the resource sharing problem, and Section 6.3 describes DyFR. The evaluation methodology and results are presented in Sections 6.4 and 6.5. Section 6.6 concludes this chapter.

6.2 Dynamic Voltage and Frequency Scaling

DVFS (dynamic voltage and frequency scaling) is a well-known and commonly used power technique to adjust the frequency of a core or a system [55]. Most processors employ a variant of the DVFS technique to improve performance or save power. To name a few, Intel Turbo Boost [53] and AMD Turbo CORE [4] increase the core clock frequency to improve performance by taking power from idle cores. In recent proposals [74, 77], researchers have proposed dynamic voltage/frequency and core scaling (DVFCS), which try to combine DVFS with dynamic core sampling (DCS) that tries to identify the optimal number of operating cores.

6.2.1 Voltage and Frequency (VF) Domain

For efficient power management, recent processors have a separate voltage and frequency (VF) domains (or power planes) for different processing cores. For example, Intel's Sandy Bridge has three different VF domains [124]: 1) CPU cores,

ring, and L3 caches, 2) GPU cores, and 3) system agent (PCI-e, display, and memory controllers). In addition, each core has embedded power gates so that it can be turned off individually. Each component has the power management agent (PMA) to collect power and temperature information and control the power of the individual component. The packet control unit (PCU) that locates in the system agent communicates with PMAs and optimizes various power-management functions.

6.2.2 Target Architecture

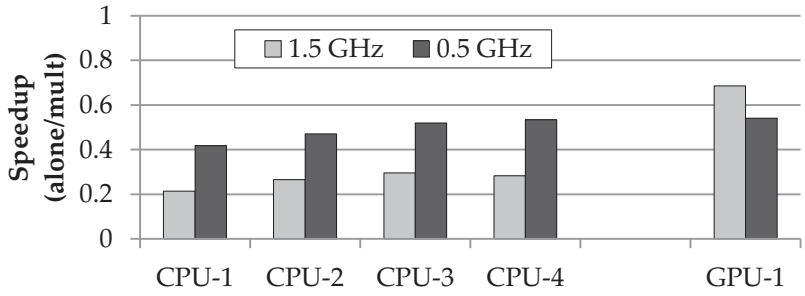
In our target architecture, we assume that there are three VF domains: 1) CPU cores, 2) GPU cores, 3) interconnection network/L3 caches, and memory controllers. CPU cores can run in different frequency, but all GPU cores will run in the same frequency. Note that we intend to use multi-program workloads and assume a busy system that enables the same number of cores as applications. As a result, DCS is not applicable.

6.3 DyFR: Dynamic Frequency Regulating Mechanism

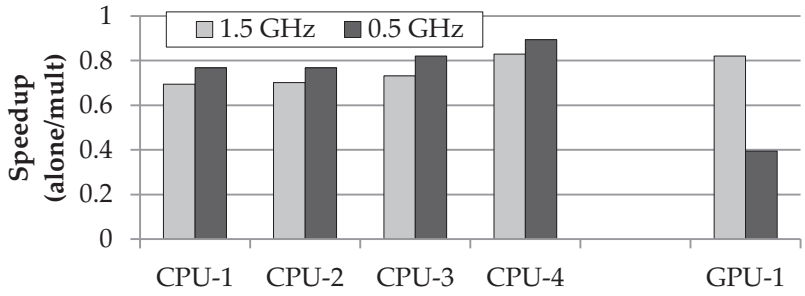
In this section, we describe the proposed mechanism called DyFR (dynamic frequency regulating). The main goal of DyFR is to mitigate inter-application interference. At the same time, DyFR also considers how clock frequency changes affect the performance of applications. To achieve these goals, DyFR performs clock frequency throttling of CPU, GPU, and memory systems synchronously. DyFR tries to find an optimal frequency combination of each component by applying three steps: 1) GPU throttling, 2) CPU throttling, and 3) memory throttling. We detail these steps in the following sections.

6.3.1 Step 1. Mitigating Interference Through GPU Throttling

The first step of DyFR is GPU throttling. GPU cores are capable of running many threads and maintaining a high throughput using massive multi-threading. As a result, GPU applications severely interfere with CPU applications, as reported in [72]. The interference can be identified by monitoring the misses per time (MPT) metric. If the MPT of a GPU application exceeds the threshold, interference is highly likely to exist. Consequently, GPU throttling will be applied. However, the level of interference experienced by CPU applications is different based on the characteristics of concurrently running GPU applications. For example, Figure 39 shows the speedup results (Eq. (17)) of two different workloads (the same CPU workloads with different GPU application) with GPU cores running on 0.5 GHz and 1.5 GHz frequencies.



(a) soplex, leslie3d, libquantum, GemsFDTD, blackscholes



(b) soplex, leslie3d, libquantum, GemsFDTD, gaussian

Figure 39: Speedup results with different GPU clock frequency.

Although `blackscholes` and `gaussian` benchmarks are fairly memory-intensive

applications, as shown in Table 17, we can clearly see that the slowdown of CPU applications is significantly different when GPU cores operate with 1.5 GHz clock frequency. For the workload with the `blackscholes` GPU application (Figure 39 (a)), the speedup of CPU applications is between 0.2 and 0.3, i.e., a 3.3x to 5x slowdown. However, with the `gaussian` application, we can observe less severe slowdown for CPU applications.

In order to mitigate the interference, lowering the clock frequency of GPU cores can be an alternative solution. Compared to 1.5 GHz, operating GPU cores with a 0.5 GHz clock frequency improves the performance of CPU applications due to the reduced interference. However, we can also observe the performance degradation of the GPU application. When we measure the speedup of the system (Eq. (17)) in 0.5 GHz over 1.5 GHz for these two workloads, there is a 20% improvement for `blackscholes`, but a 27% degradation for `gaussian`. Consequently, GPU throttling must be carefully applied based on the characteristics of GPU applications. In particular, we have to compare 1) CPU application performance improvements due to the reduced interference and 2) GPU application performance degradation. Interestingly, we discover that the benefit of CPU applications can be inferred from the performance degradation of GPU application. Less degradation by the GPU application indicates that the application is more memory-intensive and more severe interference exists with the GPU application. Therefore, when we observe a significant performance degradation by GPU applications after lowering the clock frequency, we have to stop decreasing the frequency of GPU cores.

To check the performance variation of GPU applications, we can compare the performance metric of the GPU application across two periods. Due to the single-program multiple-data (SPMD) execution model of GPUs, we can observe similar progress by the GPU application throughout the execution periods. When we apply two different frequencies over periods, we can compare the performance

of the two periods. If we observe near-linear performance degradation with the frequency decrease (when Eq. (13) is greater than threshold), we stop decreasing the frequency of the GPU cores.

$$power\_perf\_efficiency = \frac{\Delta perf}{\Delta power} \tag{13}$$

In this way, GPU throttling considers and mitigates the interference caused by the GPU application. However, if GPU throttling identifies that the interference caused by the GPU application is not severe, it then considers the frequency scalability. Based on the monitored MPKI value, the frequency of CPU cores is adjusted accordingly. This approach is similar to CPU throttling, which is detailed in the following section.

6.3.2 Step 2: CPU Throttling

The second step of DyFR is CPU throttling, which can be independently applied as GPU throttling. CPU throttling is applied solely based on the scalability property of CPU applications. As explained in Section 2.4.1 the MPKI metric is a property of applications (i.e., different clock frequency will not significantly affect the MPKI of applications) and can be a good proxy for identifying the scalability. Therefore, we increase or decrease the frequency of CPU cores based on the MPKI of the application. The intuition of CPU throttling comes from power/performance efficiency of the application, which is defined in Eq. (13). Based on the monitored MPKI during the last period, if MPKI is less than the threshold, we increase the clock frequency. This is because we expect linear performance improvements while maintaining similar power efficiency. If MPKI is greater than threshold, we decrease the clock frequency. In this case, power-efficiency can be greatly improved since we maintain similar performance with less power budget.

6.3.3 Step 3. Memory Throttling

After we apply GPU and CPU throttling, we can measure how much of the power budget remains for the memory. Then, the frequency of the memory system (last-level caches, on-chip networks, and memory controllers) can be determined automatically. The intuition of memory throttling is described in Section 2.4.2. If cores consume more power budget based on their scalability, this indicates that the workload consists of many compute-intensive applications, so the importance of the memory is low. Therefore, lower memory frequency will not harm the system throughput, while the system can operate within the power budget. In the opposite case, if cores consume less power, this indicates that many memory-intensive applications are currently running and we can improve system throughput by increasing the frequency of the memory. When more power budget is available to the memory, we have two different options based on how we utilize it:

1. *power-saving* mode: if less power consumption is more favorable, for example embedded systems, we can keep the base frequency. We can save more power while maintaining similar performance.

2. *high-performance* mode: When the performance is more important, we can use this extra power to improve memory performance, which eventually leads to better system performance.

Note that we assume the mode can be controlled by the system or user and we evaluate both modes separately.

Since memory throttling is applied after core throttling, the chip power budget never exceeds the given power budget under DyFR. At the same time, since core throttling captures the workload characteristics well, DyFR can achieve better performance with the improved energy efficiency.

6.3.4 Central Control Logic

As described in Section 6.2.1, DyFR requires a central control logic (CCL) similar to package control unit (PCU) in Intel's Sandy Bridge [124] to control the power budget for each component. Although CPU and GPU throttling can be applied locally, we have to measure the available power budget for the memory. Thus, each core, in particular the power management unit (PMU), sends its decision to the CCL. Then, CCL calculates the remaining budget and regulates the operating frequency of the memory system. Note that we assume the CCL is located in the central location of our baseline mesh network.

6.3.5 DyFR: Putting It All Together

This section describes the entire DyFR, which combines core and memory throttling mechanisms. Algorithm 2 shows the algorithm of DyFR. As explained, GPU throttling (line 3:8) will be applied first and then CPU throttling (line 11:19) is applied. Based on the remaining power budget, memory throttling is performed accordingly (line 21:22).

We summarize in Table 14 how DyFR works based on the workload. If the workload is none of the above cases, the detailed decision is based on the characteristics of each application and memory throttling may or may not be applied based on the remaining power budget. We present in Section 6.5.1.1 detailed case studies based on the workload.

6.3.5.1 Overhead Analysis

In order to cope with the dynamic behavior, Algorithm 2 is performed periodically. A shorter period can better adapt to run-time behavior, but the overhead becomes significant. Two types of overhead exist with DyFR.

1. First, when the DVFS level of a component is changed, operations of the

Algorithm 2 DyFR algorithm.

```
// cpu_budget + gpu_budget + mem_budget = 1

// GPU throttling
if MPT(GPU)  T_MPT^GPU:
  if MPKI(GPU)  T_MPKI-H^GPU && freq(GPU)  MIN_f^GPU:
    freq(GPU) -= 300 MHz
else if MPKI^GPU  T_MPKI-L^GPU && freq(GPU)  MAX_f^GPU:
  freq(GPU) += 300 MHz
total_saving = (1.5 - freq(GPU))/1.5 * gpu_budget

// CPU throttling
for (int ii = 0; ii  num_cpu_core; ++ii):
  if MPT(CPU_i)  T_MPT^CPU:
    if MPKI(GPU)  T_MPKI-H^GPU && freq(GPU)  MIN_f^GPU:
      freq(CPU_i) -= 500 MHz
  else if MPKI(CPU_i)  T_MPKI-L^CPU && freq(CPU_i)  MAX_f^CPU:
    freq(CPU_i) += 500 MHz
  total_saving += (1 - freq(CPU_i)/3.0) * cpu_budget
                / num_cpu_core

// Memory throttling
freq(MEM) = (int)(1.5 * (1+total_saving)/0.3) * 0.3
```

component during the transition period are halted. Modern processors have a phase lock loop (PLL) to control the clock signal. Since the output clock is jittering during the PLL lock time, all operations are halted during this period. PLL lock time typically lasts tens of microseconds in a digital PLL [12, 75]. Note that the penalty of the DVFS mechanism is well discussed in [112].

2. The second overhead comes from the communication cost between cores and the CCL. Since we use memory throttling to balance the power budget across the system, some information is collected from cores to the CCL, which in turn sends to the memory. Although most systems have this overhead, too frequent collection of information will incur a significant communication overhead in particular the size of network increases. We set the length of

Table 14: DyFR results based on the workload (Comp: compute-intensive, Mem: memory-intensive).

Workload		Freq. Change			Note
GPU	CPU	GPU	CPU	MEM	
Comp	Comp	+	+	-	
Comp	Mem	+	-	=	
Mem	Comp	-	+	=	
Mem	Mem	-	-	=	Power-saving
Mem	Mem	-	-	+	High-performance

period as in Table 16 from the empirical data. We also show the period sensitivity result in Section 6.5.5.

When it comes to the hardware overhead (storage, combinatorial logic), DyFR needs to utilize a few performance counters such as the number of retired instructions and cache misses. Since these performance counters are already included in current processors, DyFR does not incur extra overhead.

6.3.5.2 Discussions

We discuss possible issues or improvements of DyFR in this section.

- As explained, two steps (GPU and CPU throttling) of DyFR are performed locally and memory throttling is performed in CCL. This approach can be compared with an all-global decision, i.e., all cores send their performance metric to CCL and CCL makes an appropriate decision and sends the decision back to cores and the memory. This can reduce the functionality of the power management agent (PMA) since it is delegated to CCL. Moreover, this can reduce the design cost of PMA. However, when we compare the outcome of the two approaches, we expect similar performance benefits and the all-global approach has slightly higher communication overhead.

6.4 Evaluation Methodology

6.4.1 Simulator

We use MacSim [45], a trace-driven, cycle-level heterogeneous architecture simulator, for evaluations. For all evaluations, when an application completes its execution, we re-execute the application until all applications have finished at least once to model uniform system-level resource contention throughout the simulation, which is similar to the work in [72, 120, 150]. We try to model a next generation heterogeneous architecture that consists of high performance CPU and GPU cores. CPU cores are out-of-order and superscalar with large caches and branch predictor. GPU cores run under massive multi-threading and wide SIMD execution units. Table 15 shows the system configuration.

Table 15: Processor configuration.

CPU	4 cores, 3GHz, 4-wide, out-of-order (OOO) gshare branch predictor 8-way, 32KB L1 D/I cache, 2-cycle 8-way 256KB L2 cache, 8-cycle
GPU	6 cores, 1.5GHz, in-order, 2-way 16 SIMD width 8-way, 32KB L1 D (2 cycle), 4-way 4KB L1 I (1 cycle) 16KB s/w managed cache
L3 Cache	1.5GHz, 4 tiles (each tile: 32-way, 2MB), 64B line, LRU
NoC	1.5GHz, 4x4 2D Mesh, 3-stage pipeline x-y routing, 16B link width, 1-cycle latency 4 VCs per port, each VC can hold 5 flits
Memory Controller	DDR3-1600, 2 MCs (each 8 banks, 2 channels) 41.6GB/s BW, 2KB row buffer, FR-FCFS scheduler

Table 16 shows the configuration used in DyFR. We set the minimum and maximum CPU and memory clock frequencies based on the assumed power distribution. Due to the lack of public data regarding the power distribution data, we statically assign 30%, 30%, and 40% for CPU cores, GPU cores, and the memory (last-level cache and NoC without DRAM memory power), respectively.

Table 16: DyFR configuration.

Configuration	Value
Period length	100 us
T_{MPT}	2500
T_L	10
T_H	5
CPU freq.	1.5 - 4 GHz (500 MHz unit)
GPU freq.	600 MHz - 2.1 GHz (300 MHz unit)
L3/NoC freq.	600 MHz - 2.1 GHz (300 MHz unit)
PLL lock time	10 us

6.4.2 Benchmarks and Workloads

We use a part of SPEC 2006 CPU benchmarks and GPU benchmarks from various suites [20, 102, 137]. For CPU workloads, Pinpoint [113] was used to select a representative simulation region with the ref input set. Most GPGPU applications are run until completion. Then, we categorize benchmarks into two groups, linear-scalable and log-scalable, based on the MPKI value. Table 17 lists all benchmarks used in evaluations.

Then, we form heterogeneous workloads by pseudo-randomly choosing benchmarks from each group. Table 18 shows all types of workloads.

6.4.3 Metric

IPC (instruction per cycle) or CPI (cycle per instruction) is a common metric to measure the performance of an application. However, since our proposal aims to dynamically change the frequency configuration, one cycle indicates a different time unit. As a result, we use the execution time metric. Using this metric, we calculate the performance of the heterogeneous system. For heterogeneous configuration c, we first calculate the performance of CPU applications (Eq. (14)) and GPU applications (Eq. (15)). For the performance of the configuration c, we measure the geometric mean of CPU and GPU performance (Eq. (16)). Finally, the speedup of a configuration over the baseline configuration (without DyFR) is

Table 17: Benchmark characteristics based on the frequency-scalability (MPKI for GPU applications is measured for one core).

	Log (MPKI > 10)		Linear (MPKI < 5)	
	Bench	MPKI	Bench	MPKI
CPU	gcc	42.0	gobmk	0.96
	mcf	108.2	sjeng	0.4
	libquantum	33.9	gamess	0.08
	bwaves	19.0	povray	0.47
	milc	19.1	xalancbmk	1.78
	leslie3d	19.0	h264ref	1.31
	GemsFDTD	13.6		
	lbm	34.2		
	soplex	55.8		
	Log (MPKI > 20)		Linear (MPKI < 5)	
	Bench	MPKI	Bench	MPKI
GPU	backprop	14.22	cell	1.06
	cfd	61.5	lavaMD	1.79
	lbm	63.3	leukocyte	0.76
	spmv	60.0	tpacf	0.03
	gaussian	45.6	mri-q	0.19
	blackscholes	22.4	cutcp	0.31
	streamcluster	35.1		

calculated using Eq. (17).

Table 18: Heterogeneous workloads.

	# Log type CPU	GPU type	#
W-LL	no more than 1	Linear	10
W-HL	more than 2	Linear	13
W-LH	no more than 1	Log	13
W-HH	more than 2	Log	13

$$Perf^{c}_{CPU} = \sum_{cid=0}^{n-1} exe\_time^{alone}_{cid} / exe\_time^{c}_{cid} \quad (14)$$

$$Perf^{c}_{GPU} = exe\_time^{alone}_{GPU} / exe\_time^{c}_{GPU} \quad (15)$$

$$Perf^{c}_{sys} = geomean(Perf^{c}_{CPU}, Perf^{c}_{GPU}) \quad (16)$$

$$Speedup_{c} = Perf^{c}_{sys} / Perf^{base}_{sys} \quad (17)$$

Other metrics can be used such as the weighted speedup [127]. However, if we treat the GPU application the same as one CPU application, where the GPU application is running on multiple GPU cores and the CPU application that is running on a single CPU core, always penalizing the GPU application will yield the best outcome. For instance, if we can get 50% performance improvement for each CPU application while degrading the performance of the GPU application by 50%, the weighted speedup will be improved significantly, but the system throughput will be dramatically reduced. As a result, we calculate the performance of CPU cores (using the weighted speedup metric) and GPU cores (speedup) separately, then take the geometric mean of the two. How to evaluate the heterogeneous system is debatable, but we do not discuss it in this chapter.

6.5 Results

6.5.1 DyFR Evaluation Results

We evaluate DyFR in this section. As described in an earlier section, DyFR aims to improve the performance and energy efficiency simultaneously through a DVFS technique by considering the characteristics of each application. Figure 40 shows the result of DyFR with heterogeneous workloads in Table 18. We analyze the result from three aspects: 1) performance of CPU and GPU applications with their geometric mean, 2) power distribution across CPU, GPU cores, and the memory, and 3) energy reduction.

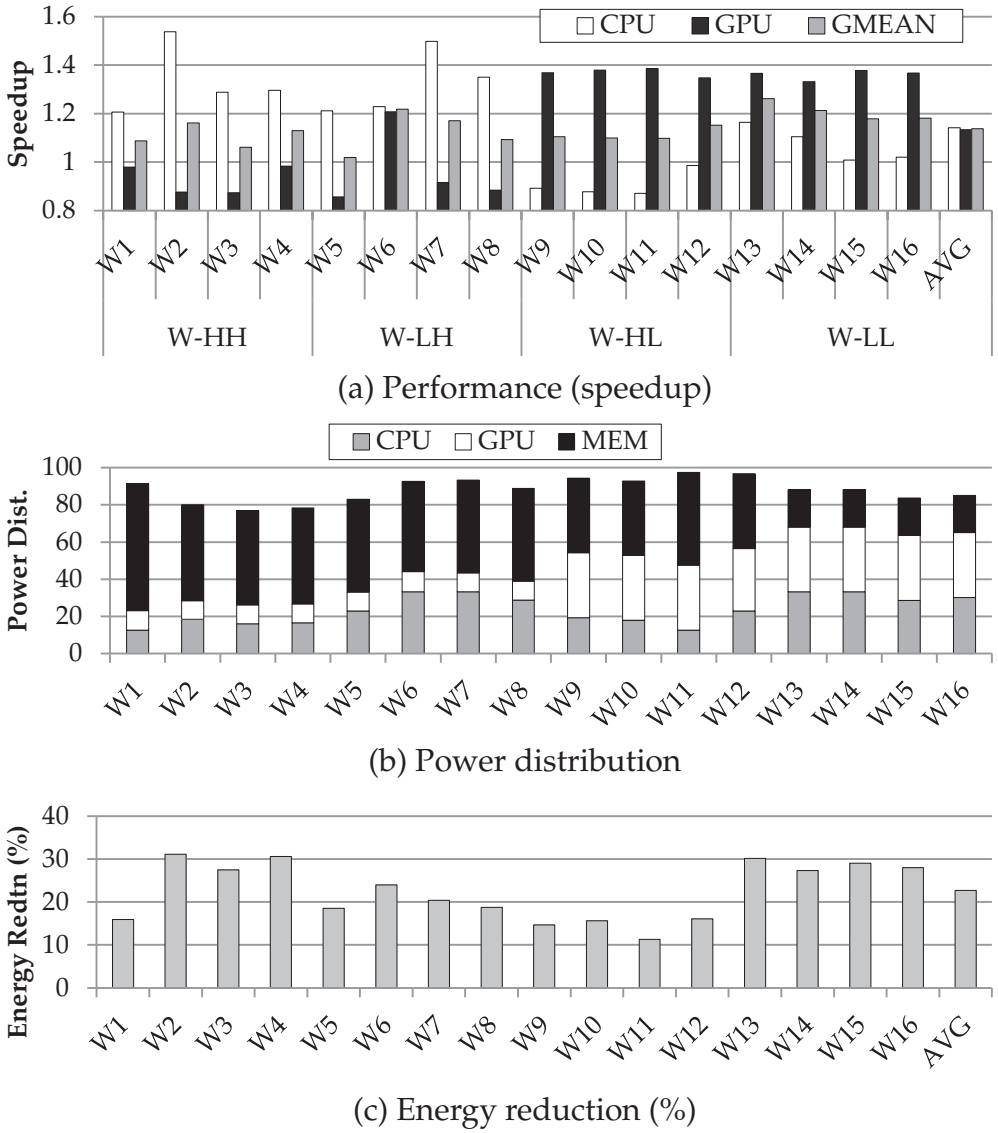


Figure 40: DyFR evaluation results.

Figure 40 (a) shows the performance result of DyFR. If memory-intensive applications exist in a workload (i.e., `W-HH`, `W-LH`, and `W-HL` workloads), DyFR decreases the clock frequency of memory-intensive applications through CPU and GPU throttling. We find that the performance of some CPU or GPU applications are degraded (below one), but degradation is marginal (no greater than 15%) compared to the benefit of its counterpart. As a result, overall speedup (Eq. (17)) for all 16 workloads is never below one and DyFR improves performance by 14%

on average.

We observe one interesting case in W6 that consists of gobmk, sjeng, gamess, povray, and streamcluster benchmarks. Since all CPU applications are compute-intensive, performance improvement of CPU applications is expected, but we can observe a 27% performance improvement for the GPU application as well even though DyFR lowers the clock frequency of GPU cores. We can confirm the behavior of streamcluster from its speedup result. As Figure 41 shows, increasing frequency does not help improve performance. We study streamcluster further and discover that thread-level parallelism is very limited and MPKI is so high that it cannot tolerate long memory access latency. Consequently, performance improvement by higher frequency is not expected.

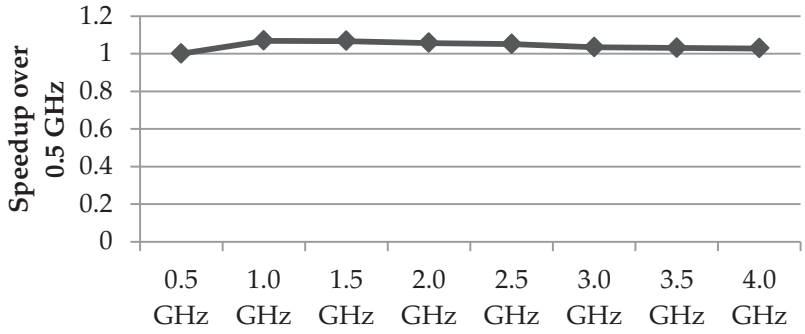


Figure 41: Speedup result of streamcluster.

To see how the power budget is distributed to the CPU, GPU, and memory, Figure 40 (b) shows the power distribution. We can observe a clear trend across different workloads. Less power will be allocated to memory-intensive applications since lowering the clock frequency will not degrade performance much. Moreover, it can also reduce the interference caused by memory-intensive GPU application. On the other hand, more power budget will be given to compute-intensive applications because we expect linear speedup with regard to the frequency increase. Based on the workload characteristics, the available power budget to the memory is automatically determined. For example, W1

workload consists of four memory-intensive CPU and one memory-intensive GPU applications. Consequently, the frequency of all cores is lowered and more power budget is available to the memory. On the other hand, `W16` is composed of all compute-intensive applications. Thus, DyFR increases the frequency of all cores close to the maximum frequency configuration and a very small portion of budget is available to the memory.

We also analyze the energy consumption by DyFR. With better performance and less power consumption by DyFR, energy efficiency is greatly improved. Across all workloads, we observe no less than 10% energy reduction, while the highest reduction is close to 35%. Overall, DyFR decreases energy consumption by 23%.

From these results, we can conclude that DyFR can achieve both performance and energy efficiency improvements for various kinds of workloads without incurring negative effects.

6.5.1.1 Case Study

For deeper understanding, we present detailed case studies for different workload combinations in Table 18.

Case Study 1: compute-intensive GPU and compute-intensive CPU The first case study is the workload that consists of compute-intensive GPU and CPU applications. These applications have low MPKI (all benchmarks are in the `linear` group in Table 17) and show good performance scalability with regard to clock frequency. In this case, DyFR tries to increase the operating frequency of all cores while lowering that of the memory. Figure 42 shows performance, power distribution, and energy reduction results.

Figure 42 (a) shows that the performance of both CPU and GPU applications is significantly improved due to the increased clock frequency. The increased

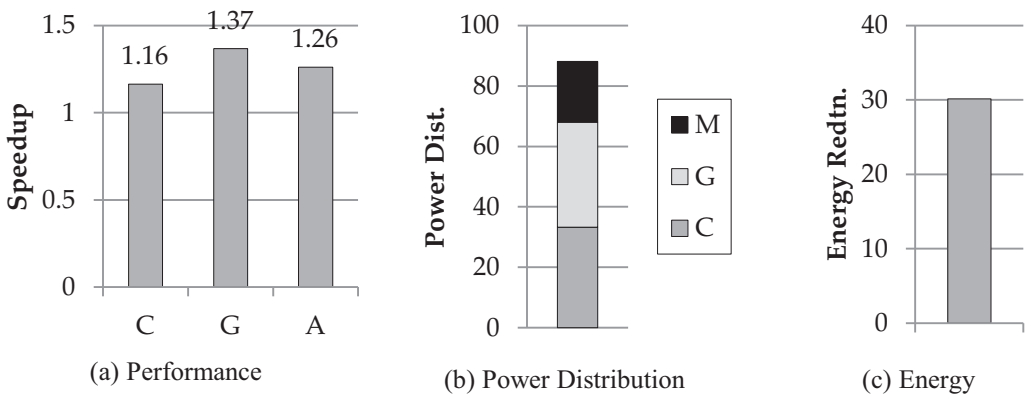


Figure 42: Compute-Intensive GPU and CPU workloads (gobmk + sjeng + gamess + povray / mri-q, In (a), C: CPU, G: GPU, A: geometric mean. In (b) M: memory, G: GPU, C: CPU).

power consumption by cores can be offset by the decreased frequency in the memory, as shown in Figure 42 (b), while the slower memory does not affect performance much. Performance improvement with the power consumption reduces the energy consumption of this workload by 30%, as shown in Figure 42 (c).

Case Study 2: memory-intensive GPU and compute-intensive CPU The second case is when the memory-intensive GPU application is running with compute-intensive CPU applications. In this case, GPU throttling is first applied to reduce the interference. Then, the frequency of CPU applications will be increased based on their MPKI. Since the change in the memory does not affect system performance much, it remains in the base frequency. Figure 43 shows the result.

Although the GPU application is running very low clock frequency by GPU throttling, its performance is degraded by only 8%, while DyFR improves the performance of CPU applications by 50% (Figure 43 (a)). We can also check how DyFR works from its power distribution: CPUs are running higher than their base frequency, GPUs are running lower than their base, and memory is running on its base frequency (Figure 43 (b)), which leads to overall less power consumption.

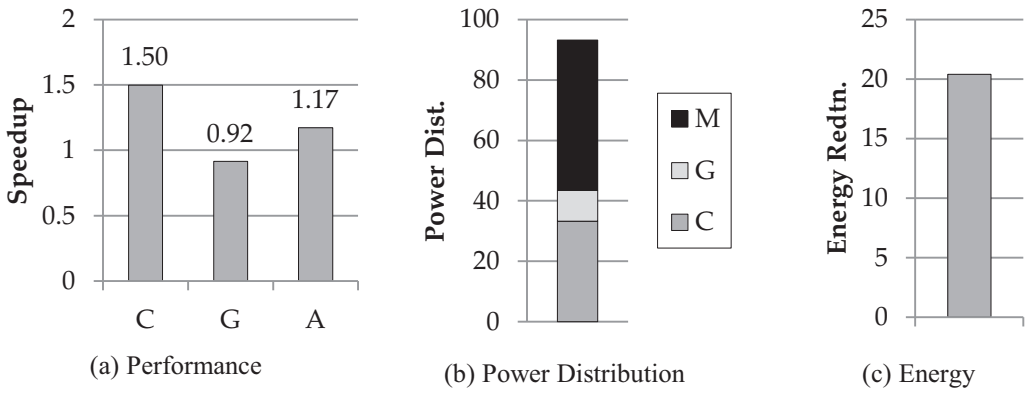


Figure 43: Memory-Intensive GPU and Compute-Intensive CPU workloads (h264 + sjeng + povray + gamess / lbm).

Figure 43 (c) shows a 21% energy reduction by DyFR in this workload.

Case Study 3: Compute-Intensive GPU and Memory-Intensive CPU With compute-intensive GPU and memory-intensive CPU applications, we can observe the opposite result from the previous case study. CPU cores are throttled down while GPU cores are throttled up. Although the memory frequency has been lowered, it depends on the workload characteristics. Figure 44 shows the result.

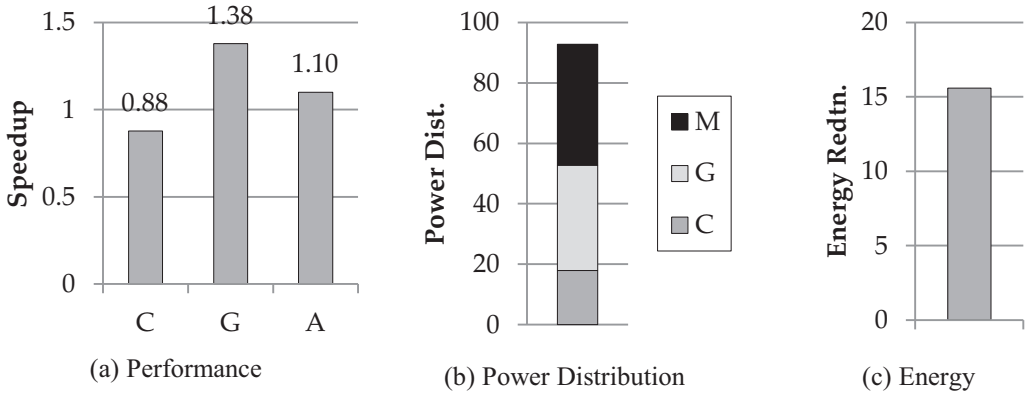


Figure 44: Compute-Intensive GPU and Memory-Intensive CPU workloads (gobmk + gcc + leslie3d + bwaves / lavaMD).

CPU throttling on memory-intensive CPU applications results in a 12% performance degradation, but the GPU application gains a 38% performance improvement. As a result, system performance is improved by 10% as shown

in Figure 44 (a). CPU cores consume less power, but GPU cores consume more power. We also see that there is a slight power reduction in memory (Figure 44 (b)). Consequently, there is a 15.5% energy savings (Figure 44 (c)).

Case Study 4: Memory-Intensive GPU and Memory-Intensive CPU The last case study is when both GPU and CPU applications are memory-intensive. In general, the core clock frequency is less important with this workload, but that of the memory is much more important. Therefore, the clock frequency of both CPU and GPU cores will be lowered while we have two modes (power-saving and high-performance) for the memory, as explained in Section 6.3.3. Figure 45 shows the results for both modes.

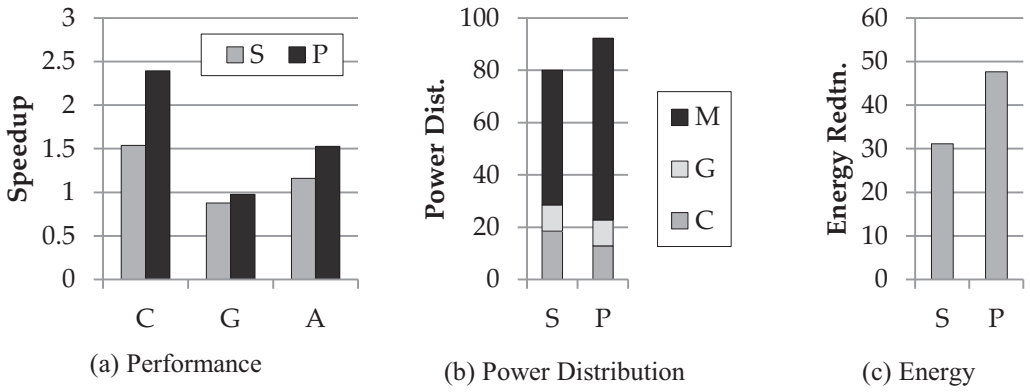


Figure 45: Memory-intensive GPU and Memory-intensive CPU workloads (milc + gcc + libquantum + lbm / blackscholes, S: power saving, P: high-performance).

In the power-saving mode (denoted by S), we can see that CPU and GPU power consumption is decreased due to the lowered clock frequency, while memory power consumption remains similar (Figure 45 (b)). However, there is a significant performance improvement for CPU applications with a small GPU performance degradation (Figure 45 (a)). This is due to the reduced interference caused by the GPU application. As a result, DyFR yields 16% performance improvement with more than a 30% energy reduction.

On the other hand, in the high-performance mode (denoted by P), we can observe a huge performance improvement by 53% because the power given to the memory is increased by 18%. As a result, the high-performance mode reduces energy consumption further by 48% in this case study.

6.5.2 Power-saving and High-Performance Modes

In this section, we evaluate two DyFR modes, power-saving and high-performance, which are described in Section 6.3.3. When we have more power budget available to the memory, we have two options: 1) maintain the same memory frequency to save power or 2) increase voltage/frequency for better performance. Which mode is more preferable is based on the system requirements. Figure 46 shows the performance improvement and energy reduction results. Note that we only show W1-W8 workloads since other workloads have less power budget due to increased frequency in cores, as shown in Figure 40 (b).

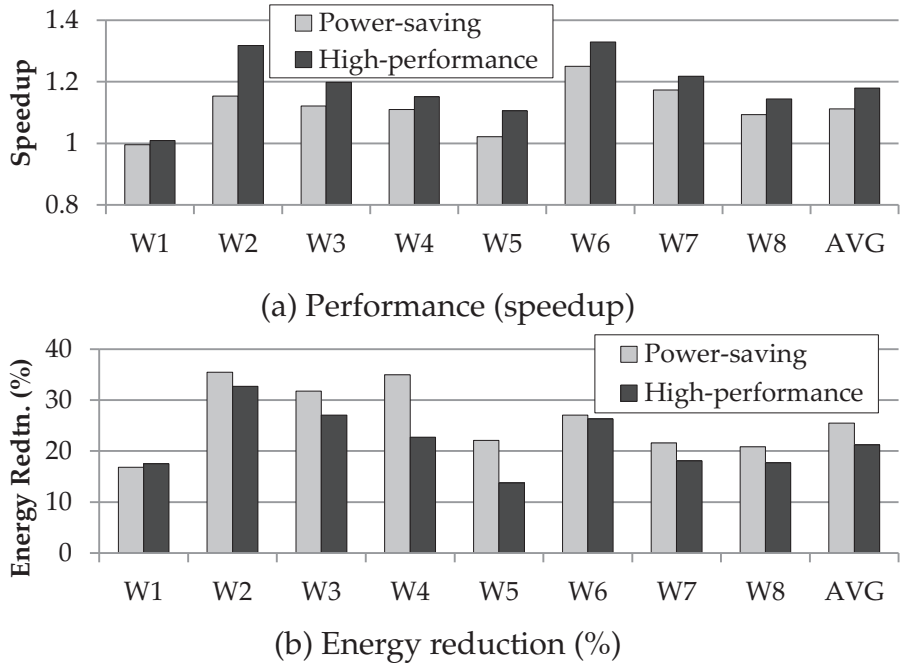


Figure 46: Evaluation of power-saving and high-performance modes in DyFR.

We can observe that the high-performance mode always yields better performance than the power-saving mode. The benefit over power-saving mode is up to 37% and 6.8% on average. How much benefit a workload can get is proportional to the increased power budget for the memory (maximum available power budget is 100 - stacked bar in Figure 40 (b)).

However, we can also see that high-performance mode consumes more energy except in the `W2` workload due to its huge performance gain. The energy consumption is increased by 3% without `W2` and by 1% on average. This is a rather expected outcome since power increase does not always lead to proportional performance improvements. However, when we measure other energy efficiency metrics, such as EDP (energy-delay product) or ED^2P (energy-delay square product), the high-performance mode shows better efficiency since they put more importance on the performance. Nonetheless, the energy consumption metric is directly related to the battery life in SoC devices, so we have to carefully decide which metric to use.

6.5.3 DyFR Results with CPU-only CMP Workloads

Although HCMPs can execute CPU and GPU applications together, it is also important that DyFR is able to cope well with CPU-only CMP workloads. Thus, we perform a DyFR experiment with CMP workloads in this section. Figure 47 shows the performance, power distribution, and energy reduction results. Note that we assume the power budget for CPU cores is 45% and the remaining 55% is for the memory.

Our first observation with the CPU-only workload is that severe inter-application interference does not exist as in the heterogeneous workload since we model the network and memory bandwidth to be sufficient to handle the heterogeneous workload. Consequently, decreasing the clock frequency

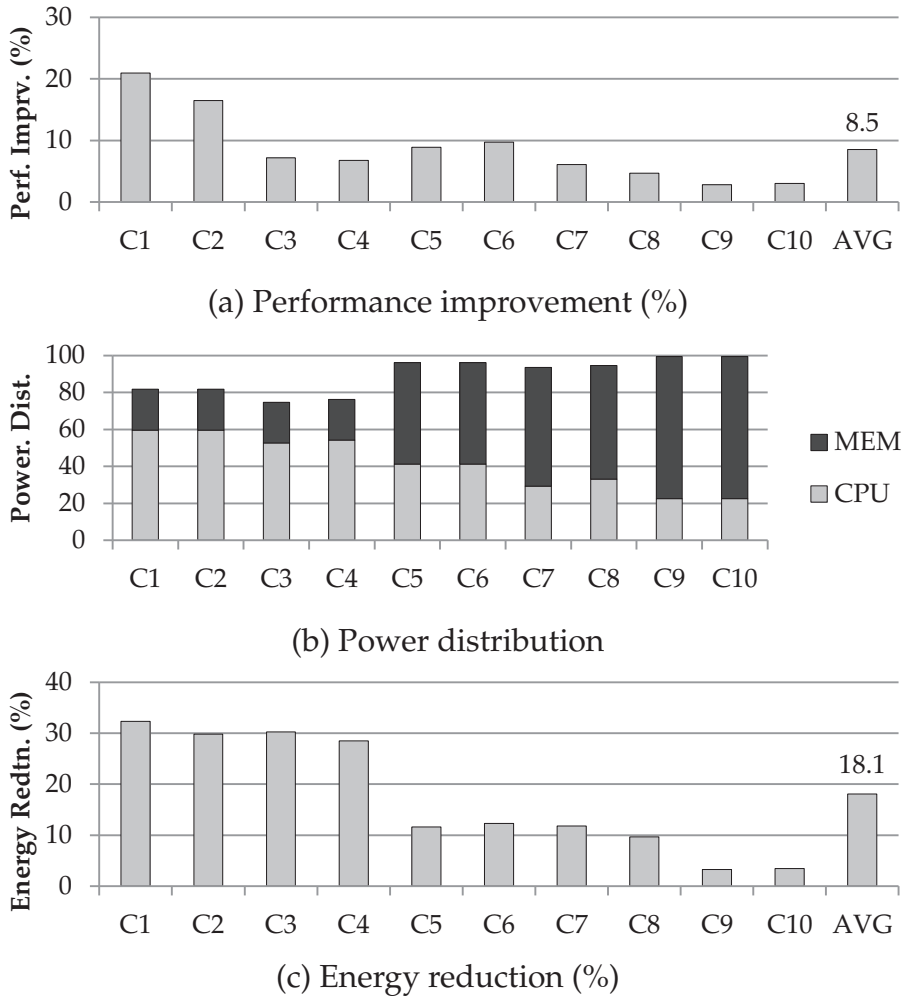


Figure 47: DyFR results with CPU-only workloads (from left to right, the memory-intensity of the workload increases).

of a memory-intensive application does not improve the performance of other applications. We also observe that the effectiveness of DyFR is higher with a workload with more compute-intensity (bars in the left side) applications. For example in `C1` to `C4` workloads, we observe a significant performance improvement with near 30% energy reductions. This is because frequency increase leads to proportional performance improvements for compute-intensive applications, but the increase in the memory frequency cannot match the benefit of core throttling. On average, DyFR improves the performance of 10 CMP workloads by 8.5% while

reducing the energy consumption by 18.1%. Thus, we can conclude that DyFR works well with CMP workloads.

6.5.4 Comparison with Other Mechanisms

Since the goal and target architecture of DyFR are different with other DVFS-based mechanisms, it is hard to directly compare DyFR with them. Instead, we compare it with mechanisms that are proposed to solve the interference problem in the heterogeneous architecture. In this section, we compare DyFR with TAP (TLP-aware cache management schemes) and adaptive VCP (virtual channel partitioning). TAP [72] is a cache sharing mechanism in heterogeneous architectures to exploit the unique characteristic of GPU applications, in particular abundant thread-level parallelism (TLP). The authors applied their TAP mechanism and extended two previous mechanisms, UCP [120] and RRIP [58], which are called TAP-UCP and TAP-RRIP. VCP [73] is a resource partitioning mechanism applied to NoC in heterogeneous architectures. Each router generally has multiple virtual channels that are shared by applications for input ports. VCP partitions virtual channels to CPU and GPU cores so that the interference caused by GPU cores is isolated. Figure 48 shows the result for memory-intensive workloads (`W1-W8`) and all workloads (`W1-W16`).

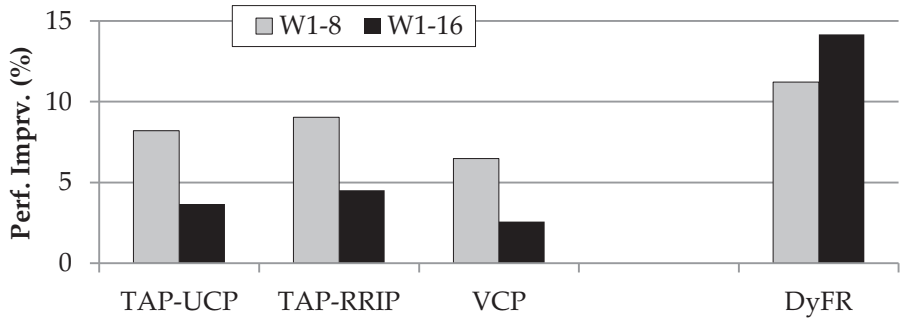


Figure 48: Comparison with other mechanisms.

As explained in Section 2.4.3, the benefit of previous mechanisms is inevitably

limited when no severe resource contention exists. If applications well utilize the shared resources without the interference, no previous resource sharing mechanisms can be effective. However, DyFR does not have such limitations. Consequently, DyFR consistently outperforms other mechanisms. In particular, the overall benefits including all workloads (W1-W16) of other mechanisms decrease since their benefit is no more than 1% on compute-intensive workloads.

In terms of energy efficiency, since no other power-saving technique is applied to TAP and VCP, energy efficiency is solely based on performance. However, as shown in Figure 40 (c), DyFR yields a great energy efficiency improvement due to the combination of performance improvement and power savings through core and memory throttling. From this experiment, we can conclude that DyFR can be a good solution for the resource contention to improve performance and energy efficiency simultaneously in heterogeneous architecture.

An interesting follow-up question is whether TAP and VCP can be combined with DyFR. However, from our initial experiment, it is not a trivial task to combine them for several reasons. First, all mechanisms should be synchronized. TAP and VCP rely on a kind of sampling mechanism to collect performance metrics to see the effect of a configuration. If DyFR changes the configuration in the middle of the sampling period, sampling is likely to fail. Second, the outcome of DyFR will affect the degree of interference in the shared resources and the effectiveness of their metrics. As a result, we leave this to the future work.

6.5.5 Sensitivity Results of DyFR

We discussed the effect of sampling period length in Section 6.3.5. To see the effect, we perform period sensitivity experiments. Figure 49 shows the performance with error bars and energy reduction results. We vary the length of period from 25 us to 1.6 ms.

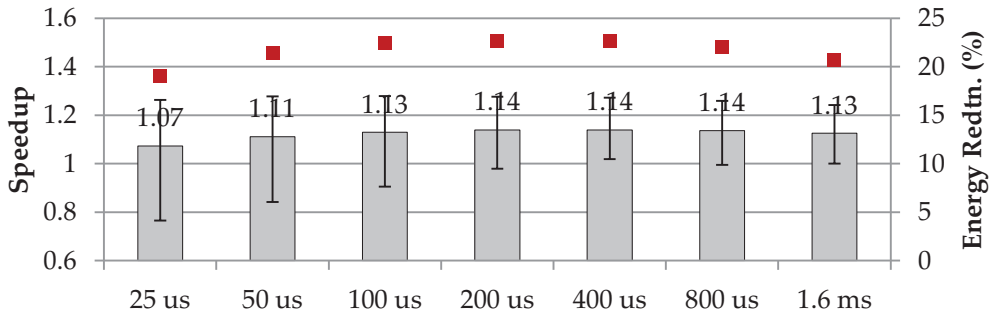


Figure 49: DyFR period sensitivity results with min-max error bars.

All configurations show performance improvement (7% to 14%) with a significant energy reduction (18% to 23%), but we can see the performance variance as well. Since we set the PLL lock time as 10 us (Table 16), the penalty in shorter period configurations (50 us and 100 us) is significant. As a result, we can observe some workloads with significant performance degradation with these configurations. However, the overall performance improvement is still 7% and 11% for 50 us and 100 us, respectively. This is because PLL lock occurs when voltage/frequency is changed. In other words, once the frequency configuration reaches an optimal point, there will be infrequent PLL lock times, so not much penalty will be incurred. From the result, longer period configuration is more preferable, but the system will be running for a longer time under non-optimal configurations. Adapting the length of the period can be an improvement to DyFR. In the earlier period, we can begin with a shorter period to reach the optimal configuration quickly; then we gradually increase the length of the period. In this way, we can minimize the penalty of DVFS level change and improve the effectiveness.

6.6 Summary of This Chapter

In this chapter, we proposed DyFR, which dynamically controls the clock frequency of the CPU, GPU cores, and the memory (L3 and NoC). Computing

on heterogeneous architectures will be more prevalent in coming years due to their performance and power efficiency, but shared resource management is still challenging. DyFR consists of three steps: GPU throttling tries to mitigate the interference caused by GPU applications; CPU throttling considers the scalability of individual applications with respect to the frequency changes; and memory throttling balances the power budget not to exceed the chip-wise power budget. We also examined the two different modes, power-saving and high-performance, based on system requirements. We showed that DyFR is a viable solution to successfully reduce interference while improving performance and energy efficiency. Across 16 heterogeneous workloads, DyFR shows a 14% performance improvement, with a 23% energy reduction on average.

CHAPTER VII

GPU REGION-AWARE ENERGY-EFFICIENT CACHE

7.1 Introduction

As heterogeneous computing becomes the mainstream computing paradigm across a wide computing spectrum from a smartphone application processor to a low-end server processor, many system components are being integrated into a main CPU die and programming language, such as OpenCL [111], is being developed to utilize heterogeneous processors. Not surprisingly, such an unprecedented integration provides novel, interesting research opportunities and challenges for computer architects. Among those opportunities, one of the most interesting problems, we believe, is how GPUs can utilize a large on-chip cache (*e.g.*, 128MB L4 cache in Intel's Haswell products) energy efficiently. Note that OpenCL enables an application to be running on CPU-only, GPU-only, or both types of cores, but we focus on running an OpenCL application on only GPUs in this paper.

In the past, a discrete GPU did not have much cache implemented, mainly because integrating more GPU cores, i.e., computing units, on a given space improves the performance of an overall GPU chip better than allocating a part of the die space to a large cache. As a result, a conventional, discrete GPU had a few, small, dedicated, special-purpose caches for different graphics pipeline stages, including texture, color, and z-caches. These dedicated caches worked well by exploiting different levels of locality across different graphics operations. In other words, for some operations that do not have much locality, a corresponding architectural block reads data directly from off-chip DRAM. Otherwise, data is

brought from the dedicated, special-purpose cache for other operations that have a certain level of locality. However, in the integrated platform where a CPU and its associated cache hierarchy are integrated with a GPU, a large, general-purpose shared cache, primarily designed for improving the average performance of various, general-purpose CPU applications, is avilable to the GPU thanks to the integration, which might be sub-optimal for an integrated GPU.

In this work, we explore how GPGPU applications can energy efficiently exploit the large on-chip cache. We specifically study how cache hit rate varies across an address space of GPGPU workloads, analyze why we observe these behaviors, and determine how we can exploit these characteristics with a very cost effective hardware solution to maximize the benefit of an on-chip cache without consuming unnecessary energy. These optimization opportunities were limited in the conventional CPU environment due to its extreme freedom of memory manipulation, but we found that the uniqueness of the OpenCL programming model[1], in particular, a disciplined memory model, which follows the rules/agreements faithfully, allows us to perform semantic-aware optimizations easily and correctly.

As a showcase to demonstrate the benefit of our findings, we propose a programming model/architecture collaborative optimization scheme called GPU Region-aware Energy-Efficient Non-inclusive cache hierarchy, or GREEN cache. In particular, we apply our findings to well-known low-energy cache techniques, selective caching, and dynamic cache resizing. *Region-aware caching (RAC)* selectively caches a subset of memory objects[2] used by a GPGPU kernel, and *region-aware cache resizing (RACR)* turns off a subset of a large on-chip cache if the GPGPU kernel turns out not to utilize the entire cache capacity. This demonstrates that our findings are very practical and cost-effective for implementation with existing

[1]CUDA programming model has a similar memory model as OpenCL.

[2]In this work, we define a region as a linear memory space allocated for a memory object of a GPU kernel, which is mapped from the host code.

systems.

The contributions of our work include the following:

1) We first analyze how cache hit rate varies across different regions in the address space of a GPGPU workload.
2) We then demonstrate how well this memory behavior correlates with the OpenCL semantic information, in particular different memory objects.
3) We propose two cache optimization techniques, region-aware caching and region-aware cache resizing, to show the benefit of our findings.
4) Finally, we propose a few extensions of the GREEN cache so that it can cope with existing cache partitioning schemes to support concurrent GPU kernel executions.

The rest of this chapter is organized as follows: Section 7.2 explains the baseline GPU architecture and its programming model to help readers understand our proposal easily. Section 7.3 proposes the GREEN cache, demonstrating the motivational data and detailing the proposals. Section 7.4 explains the simulation methodology, simulated machine configurations, and the evaluated workload. Section 7.5 shows the simulation results. Section 7.6 concludes the chapter.

7.2 GPU Model

In this section, we describe the execution, programming, and memory models of GPUs.

7.2.1 Disciplined Memory Model in GPUs

A CPU typically offloads a large chunk of computation to the GPU to improve the performance or energy efficiency of a certain kernel. Due to this offload computation model, running existing legacy CPU applications without any modification is not beneficial. Instead, a GPU program is typically written in a GPU-specific programming model such as OpenCL [111]. Interestingly, these

programming models ask programmers to provide more information about a specific memory region than a legacy CPU programming model. For example, unlike a CPU programming model in which any legitimate memory location was accessible from a function, OpenCL restricts a kernel function from freely accessing memory space other than linear memory space that is explicitly passed through its input arguments. This is partly because a CPU and a GPU do not fully share their memory space. For example, a discrete GPU and a CPU clearly have their own dedicated memory, e.g., GDDR memory on a discrete GPU card and DDR system memory. Moreover, historically, a graphics kernel allocated separate memory objects into different regions, e.g., one memory object for each texture, which naturally evolved into the disciplined memory model of the current GPU programming model. For these reasons, a GPU programming model 1) requires programmers to explicitly express the properties of memory objects that a kernel will use and 2) follows the disciplined memory model.

7.2.2 Memory Objects and Kernel Arguments

An example of the disciplined GPU memory model is shown in Figure 50 (a), which is a snippet of the OpenCL kernel definition in `hotspot` from Rodinia suite [20]. As shown in the figure, *hotspot_c* kernel takes 13 arguments: the first three arguments are pass-by-pointer arguments and the other 10 arguments are pass-by-value arguments. Among these arguments, we focus on the first three pointer arguments. These pointers are actually pointers to a memory object created by an OpenCL API function, *clCreateBuffer*. Then, they are explicitly mapped to a kernel through another OpenCL API function, *clSetKernelArg*, as part of host code. An example host code that creates a memory object and maps the object into a kernel for the first argument, *power*, is shown in Figure 50 (b). In this code, *clCreateBuffer* allocates a contiguous memory space with the size defined in line

number 3 (third argument) and marks this memory object as a read-only region. Furthermore, in line number 6, a programmer explicitly declares that a pointer to this memory object will be given to *hotspot_c* as the first argument, *power*, by explicitly declaring the argument index 0 (the second argument of *clSetKernelArg*).

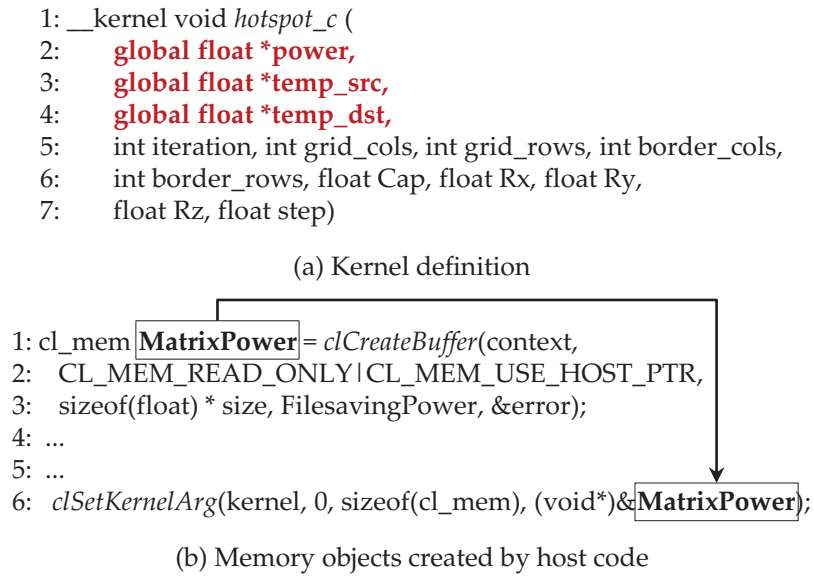


Figure 50: Memory variable example in `hotspot` benchmark.

In reality, this information is passed to the GPU hardware through the OpenCL library and GPU device driver, and a GPU core recognizes each region by looking up the region ID encoded in a memory instruction. For example, in Intel's GPU, each memory (Send) instruction has an immediate field for the region ID [51], which is an index to the address (or region) binding table [50].

In this example, while each memory object looks similar to a memory array in the conventional c/c++ programming language, a GPU kernel code can manipulate each object in a very restricted manner. Unlike memory variables in a conventional CPU programming model, which can be dynamically allocated and deallocated frequently, each memory object is persistent throughout a GPU kernel execution. Furthermore, as opposed to having many small function calls that can freely access the entire heap memory space in a CPU programming model, an

OpenCL programming model requires programmers to access their data structures in a very disciplined way and even recommends programmers access memory in a coalesced manner within the memory object.

7.3 GREEN Cache

7.3.1 Disciplined Memory Model and GPU Hardware

As explained in the previous section, OpenCL has very unique properties, such as explicit information on each memory object and a very disciplined memory model within a GPU kernel. The reason a GPU programming model prefers a memory object, *i.e.*, a linear memory space, and why a coalesced access pattern within the memory object is important are highly correlated with the fundamental characteristics of a GPU as follows:

1) Coalesced access patterns allow GPU hardware to fetch a cache line once and to fully consume the cache line. For example, a scalar code that reads a 4B float array within a loop of 16 iterations will fetch the same cache line (64B) 16 times because the cache datapath width is fixed to 64B. On the other hand, a 16-way SIMD hardware can fetch a cache line once to perform one SIMD instruction while fully utilizing cache bandwidth and saving energy significantly in the cache.

2) Due to hardware overhead, GPU hardware cannot afford many ports in its L1 cache. As a result, non-coalesced access patterns (or scatter-gather patterns) will end up accessing the single- or dual-ported cache multiple times to fulfill a single SIMD load operation. In other words, such serialization will severely degrade performance and efficiency, not to mention poorly utilizing cache bandwidth.

3) Coalesced access within a warp is very likely to be achieved by the code that accesses memory space with a linear indexing function of a thread ID. This pattern will make sure that memory address space accessed across neighboring warps is also well-coalesced across a linear memory region. This feature is very helpful

for the back-end memory controller to maximize the DRAM row buffer locality because the memory controller can collect many requests that are mapped to the same DRAM row buffer from many warps and serve these requests by opening a DRAM row only once. Such a pattern improves DRAM bus protocol efficiency, utilizes the DRAM internal bandwidth effectively, and minimizes the over-fetch inefficiency of a DRAM row, which is a by-product of cost-optimized DRAM array design.

7.3.2 Exploiting the Different Behavior of Memory Objects

For the aforementioned reasons, OpenCL has very unique semantic information, which, we believe, provides us very novel, interesting opportunities to perform cross-layer optimization across a programming model and hardware. Among the potential opportunities, in this work, we focus on energy-efficiently utilizing on-chip cache by exploiting the semantic information.

To demonstrate these opportunities, we first profiled how cache hit rate varies across the address space of a GPU kernel. The memory behavior is well captured in Figure 51, which shows the cache hit rate of each virtual page address of `hotspot`. In this figure, we profiled every single memory request, whether it generates a cache hit or miss either in the L1 or L2 cache, and incremented a corresponding performance counter for a virtual page that the address of this request belongs to. The cache hit rate varies significantly across address space, while the cache hit rate stays almost constant within a neighboring address. More interestingly, we found that the region of the neighboring address space is *well correlated with that of each memory object*, as shown in the top of the figure. For example, the L2 cache hit rate of memory object 1 is close to 60% while that of memory object 2 is close to 40%.[3]

Initially, we were excited to find this behavior, but at the same time, we

[3]Table 21 shows the memory variable information of all evaluated benchmarks, such as the input size and the number of variables for a kernel.

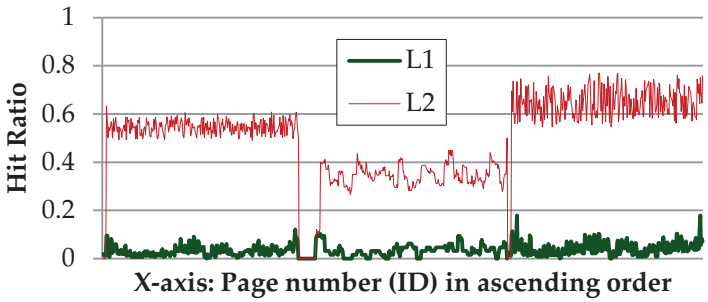


Figure 51: Cache hit rate across address space of `hotspot`.

wondered why we failed to exploit this behavior in the past since this kind of behavior must exist in the conventional CPU model, as data parallel kernels are a subset of various CPU workloads. Consequently, we seek to answer why it was difficult for us to exploit such a property in the CPU environment but why it is easier in the GPU environment. After studying the properties of GPU kernels, we realized that this behavior can be better exploited, especially in the GPU context, for the following uniqueness of the GPU execution model compared to the CPU execution model:

1) Threads spawned from the same GPU kernel execute the same set of code due to its single-program multiple-data (SPMD) execution model. Consequently, we have a large degree of similarity across different threads. In other words, a kernel offloaded to a GPU will have a very stable, repeated compute pattern until the kernel finishes its computation, which opens up an easy learning opportunity. Also, due to the explicit kernel boundary, we can detect the change of program phases very easily.

2) Moreover, GPU kernels generally have more predictable locality than CPU applications since GPGPU programs are heavily optimized to store the data in the scratch-pad memory, which is usually specified *at the beginning of kernel execution* for each thread to increase the locality behavior. This leads to highly predictable locality behavior in the L2 cache as well.

3) On the contrary, the CPU workload clearly has a wide variety of characteristics due to the general-purpose nature of CPUs. Thus, applying optimization techniques for such a niche opportunity, *e.g.*, an application with well-managed memory access pattern, does not always work for all types of CPU workloads and potentially comes with huge hardware overhead to guarantee efficient, but correct, execution. If we were to exploit the well-managed memory behavior in a CPU environment, we would be able to track it in a page granularity, but this fine-grained tracking will require high hardware overhead (per-page entry) while having a marginal benefit (savings gained only within a page for each entry).

4) Unlike the CPU workload, a GPU kernel has very high data-level parallelism. This is why it can benefit from the GPU despite the relatively high kernel offloading overhead from a CPU to a GPU. Due to the high data-level parallelism, an offloaded GPU kernel traverses larger data structures in a given period than CPU functions. Thus, we can exploit this property in a GPU programming model much more effectively with cost-effective hardware than in a CPU programming model.

5) As described in Sections 7.2.1 and 7.2.2, the existing OpenCL programming model already provides sufficient, guaranteed semantic information, unlike a CPU model in which a function can access the entire heap space freely.

Based on these observations, we envisioned that we could perform interesting optimization by treating different memory objects or regions differently. We believe that we have various opportunities to exploit the semantic information about different memory regions, but, in this work, we particularly focus on how to efficiently use an on-chip cache space. In particular, we propose GPU Region-aware Energy-Efficient Non-inclusive cache[4], or GREEN cache, as an example of

[4]Currently, caches in discrete GPUs and integrated GPUs are non-inclusive between the L1 and L2 hierarchy.

utilizing the unique GPU semantic information. In the rest of this section, we propose techniques to improve the energy efficiency of a large on-chip cache for GPUs.

7.3.3 Region-Aware Caching

The first optimization technique we propose is selective caching. The rationale behind this optimization is that, as shown in the previous section, the cache hit rates of different regions of memory differ significantly. Clearly, caching is very helpful for the group of memory regions with a decent cache hit rate, while caching for the group of memory regions that do not have cache hits just consumes energy and evicts useful data. For example, as shown in Figure 51, we can exploit locality in the L2 cache very well for all memory objects, but all regions do not have many L1 hits, so the L1 cache can be a good target to save energy in this case.

Thus, instead of blindly caching all data, we want to selectively cache data to save dynamic energy. We call this optimization *region-aware caching (RAC)*. To achieve this, first, we need to monitor the cache behavior of each region either by 1) compiler static analysis or 2) dynamic hardware training. Conventional compiler-based static-time profiling might be a cheaper solution, but it has limited knowledge of the runtime behavior. Furthermore, a compiled binary can be used over multiple iterations with different sizes of inputs. As is widely known, depending on the size of the data structure and the size of a cache memory, the effectiveness of caching varies greatly. Therefore, we use hardware-based training. This training mechanism can be performed with a very cost-effective table, an example of which is shown in Figure 52. The table consists of six fields: region ID, number of cache accesses, number of L1 and L2 hits, and L1 and L2 bypass decisions. For each region, we train the table for L1 and L2 cache hits. Note that, unlike other hardware approaches, our approach is extremely cost effective thanks

to very detailed, accurate semantic information given by the programming model of OpenCL, where this information includes not only the region ID and the starting address of the region but also the size of a region and whether it is a read-only, write-only, or read-write region. This semantically guaranteed information makes it impossible for the hardware to become confused between different memory regions, which makes this solution free of functional incorrectness.

Region ID (4b)	# Accesses (4B)	L1 Hit (4B)	L2 Hit (4B)	L1 bypss (1b)	L2 bypass (1b)
0	158,210	32,021	45	0	1
1	221,520	52,013	3,790	0	0

Figure 52: Per-region training table example (b: bit, B: byte).

Using this table-based training, RAC operates as follows: 1) RAC monitors L1/L2 cache hit rates during a training period[5] and determines in which region to bypass a cache (L1 and/or L2) based on monitored behavior. 2) Given this information, when a memory request is about to be issued to the cache hierarchy, the request can be tagged with two-bit bypass fields (whether to bypass L1 and/or L2 caches) so that the underlying cache hierarchy can utilize this information. This process is repeated *for every new kernel invocation* because 1) each kernel may operate on a different set of data and 2) they may have distinct behaviors.

It is worthwhile to note that these training results are much more accurate and stable in GPU architectures than in CPU architectures. While different kernels in a GPGPU program may have different behaviors, each GPU kernel runs in an SPMD manner, so a kernel usually does not show significant phase changes within a kernel. As a result, cache behavior information acquired during the training period will be similar throughout the execution of the kernel. Figure 54 shows an

[5]We set the length of the training period to 100,000 cycles based on empirical data. We show sensitivity data in Section 7.5.1.1.

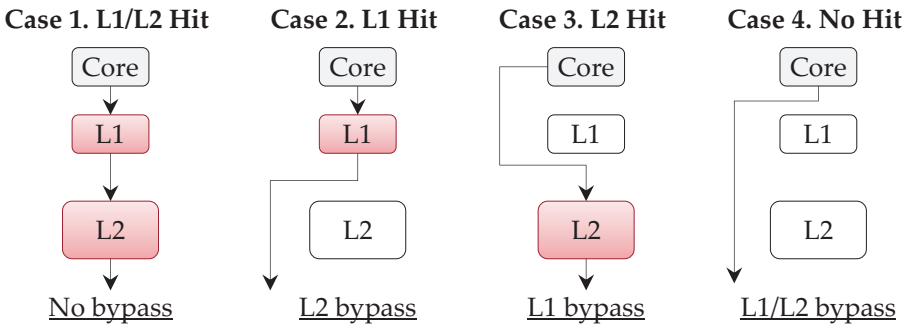


Figure 53: Cache behavior and bypassing decision.

example of the `hotspot` benchmark with the L1 and L2 cache hit rate of each region throughout the simulation duration. After the initial training period (shaded region in the figure), each region shows a near-constant hit rate. Therefore, the trained information can be a good proxy of the entire kernel execution and we do not need further training. Also note that the virtual-to-physical address translation is done through the existing binding table as described in Section 7.2.2. During translation, memory requests can be marked with caching hints using information from the region table, carrying hints throughout cache hierarchy.

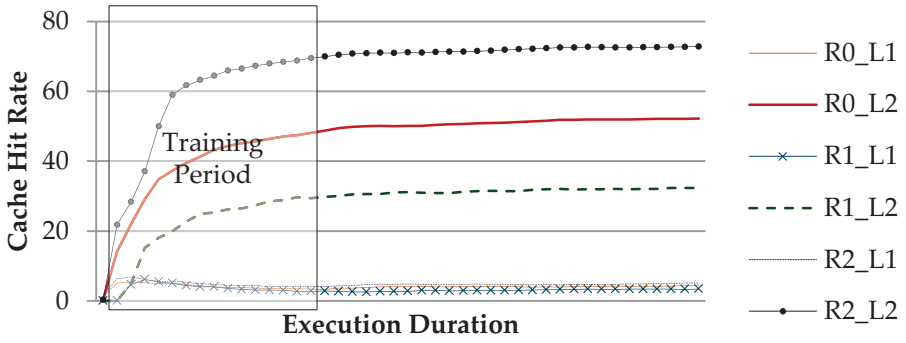


Figure 54: Cache hit rate for each region in the `hotspot` benchmark (R0_L1: L1 cache hit rate of region 0. The shaded region indicates the training period).

7.3.4 Region-Aware Cache Resizing

In addition to region-aware caching, which exploited the different cache hit rate of different memory regions, we also propose exploiting the existing knowledge of

the size of each memory region. As explained in Section 7.2.2, the size information is provided by the OpenCL library, which can directly update a hardware register dedicated for the memory regions. Based on this information, hardware can easily detect the sum of the size of all regions, which can be a proxy for the working set size for a given kernel. Once we know the working set size, we can determine *how many ways of a given set-associative cache should be active* for this kernel. If the working set size is smaller than the cache capacity, we can turn off a subset of the cache to save leakage energy. We call this policy region-aware cache resizing (RACR). In particular, we call the proposal of naïvely applying cache resizing to the L2 cache RACR-Naïve, compared to a more intelligent policy that we discuss later in this section. Equation (19) shows how RACR-Naïve calculates a new effective associativity, $assoc_{naive}$, which is at least one and cannot be greater than the original associativity.

$$total\_region\_size = \sum_{i=1}^{N} size(region_i) \tag{18}$$

$$assoc_{naive} = \left\lceil \frac{total\_region\_size}{L2\_size} \times L2\_assoc \right\rceil \tag{19}$$

, where N is the number of regions

Note that previous proposals for turning off some ways of a cache based on prediction for CPU workload may incur various side-effects such as performance loss and/or higher energy consumption, especially upon incorrect prediction, whereas our approach can easily estimate the working set size for a GPU kernel since the programming model provides all region information. By utilizing this direct information, it is very unlikely that our approach will get confused between different regions or incorrectly predict the working set size.

On top of this, we can further reduce the leakage energy by using a more intelligent policy, excluding the sum of the size of bypassed regions proposed

in the previous section, as in Eq. (20). We call this policy RACR-Bypass. Unlike RACR-Naïve, which cannot turn off a subset of a cache when the working set size is larger than the cache capacity, RACR-Bypass mitigates this limitation by exploiting the outcome of RAC. In particular, once a region is marked as a bypass region, we exclude the size of the region when we estimate the working set size of a given kernel. As a result, the size of an effective working set of the kernel will be smaller than the mere sum of the size of all regions. Equation 22 shows how we calculate a new effective associativity, $assoc_{bypass}$, with the support of RAC.

$$bypass\_region\_size = \sum_{i=1}^{M} size(region\_to\_bypass_i) \tag{20}$$

$$new\_size = total\_region\_size - bypass\_region\_size \tag{21}$$

$$assoc_{bypass} = \left\lceil \frac{new\_size}{L2\_size} \times L2\_assoc \right\rceil \tag{22}$$

, where M is the number of regions to bypass

One weakness of RACR-Bypass compared to RACR-Naïve is that, while RACR-Naïve can be immediately applied upon kernel launch, RACR-Bypass relies on information learned during the training period; thus, RACR-Bypass may be applied for a shorter time than RACR-Naïve. To overcome this weakness, we propose RACR, which combines RACR-Naïve and RACR-Bypass. In RACR, we apply RACR-Naïve immediately upon kernel launch, monitor the cache behavior during the training period, and then apply RACR-Bypass once training is done.

Unfortunately, similar to other low-power proposals on disabling a subset of a cache [3, 122], our scheme also needs to address potential data consistency issues upon changing the size of an active cache. Previous studies usually employed two approaches: 1) flushing the entire cache or dirty lines or 2) performing a lazy eviction. The first approach, flushing, can achieve cache resizing more quickly, but it requires bursty write-backs and other cache operations are stopped during this

period. On the other hand, a lazy eviction does not have the same problems, but a transition period can be much longer and the opportunity for power savings can be reduced.

In this work, we employ the flush-based approach for more energy savings since we expect that not many cache lines are in the dirty state for GPU applications from their programming model. The rationale behind this reasoning is that many GPGPU applications are already optimized to minimize the number of costly off-chip accesses. As a result, they highly utilize local and scratch-pad memory for temporary writes and then store the final result of all computations to the memory nearly at the end of kernel execution. To examine this hypothesis, we performed experiments to measure the average number of cache sets that contain dirty lines when RACR is applied. Figure 55 shows results. As expected, except for three benchmarks, only a few sets contain dirty lines. The number of sets that contain dirty lines accounts only for 0.59% of the total number of cache sets across 32 benchmarks. As a result, the total number of flushed cache lines is very small, so the length of a transition period can be very short.

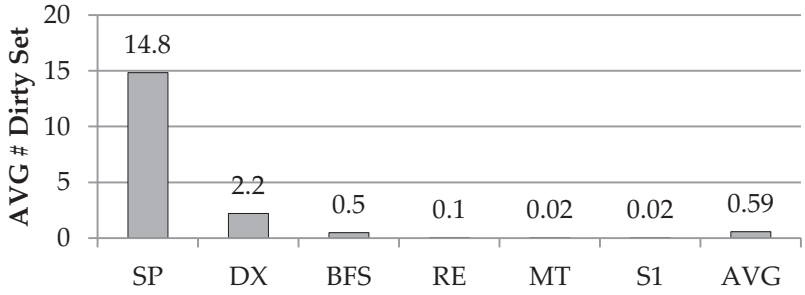


Figure 55: Average # sets that have dirty lines upon resizing.

Although flush-based cache resizing does not incur significant overhead, we still need to walk through all cache sets to check for the existence of dirty lines, which consumes extra energy and time. We can reduce this overhead by using a signature from a cost-effective bit vector. When a new write operation is performed

or an eviction occurs, we update this bit vector by ORing the dirty bits of all the ways of the accessed set. Upon cache resizing, we can query this bit vector to see if a certain cache set includes dirty lines. Since the bit vector is updated whenever a line becomes dirty or evicted, we can precisely detect sets that do not have any dirty lines. Once we identify whether a set includes any dirty block, then we walk through all blocks of the set to write-back the data. Compared to walking through the entire cache, this approach can be more effective while guaranteeing correct execution. Moreover, due to its small size, e.g., 4k bit for a 16-way set-associative 4MB cache with 64B lines, a banked bit vector can be implemented with very low hardware cost, which can further accelerate the lookup process upon resizing thanks to parallel lookup offered by the multi-banked bit vector structure.

7.3.5 Putting It All Together - GREEN Cache

In this section, we provide a summary of the GREEN cache and discuss the benefits and overhead. Table 19 describes mechanisms that consist of the GREEN cache. The biggest strength of the GREEN cache comes with *collaborative interaction between a programming model and hardware-based training*. Since the GREEN cache entrusts complex hardware training to semantic information of the OpenCL programming model, we can avoid complicated hardware logic, while information directly acquired from the programming model is precise, freely given, and independent to program/input/hardware changes. Therefore, this cooperative approach has a huge advantage over software-only or hardware-only approaches.

7.3.5.1 Benefits of GREEN Cache

The benefit of the GREEN cache is twofold. First, by saving power consumption in caches, we can use this saved power to improve the performance of the GPU cores. For example, Intel's Turbo Boost [53] enables cores to run at higher than the base frequency when higher performance is needed and when chip power budget

Table 19: GREEN Cache - putting it all together.

RAC	To save dynamic cache energies - Per-region cache behavior training (# accesses, hits) - Bypass L1/L2 caches based on the behavior.
RACR-Naïve	To save leakage cache energies - Calculate the working set size using Eq. (18). - Disable a few cache ways based on Eq. (19). - Resizing applies when a new kernel is launched.
RACR-Bypass	To save leakage cache energies - Resizing applies after a training of RAC is over. - Recalculate the effective working as in Eq. (22).

is left unused due to other idle cores. Similarly, the GREEN cache can enable a higher operating frequency of cores.

Second, the GREEN cache can be effective even when both CPUs and GPUs are active in the heterogeneous architecture. Lee and Kim [72] recently reported that massive cache accesses from GPU cores can easily break the cache locality of CPU applications when the last-level cache is shared by CPUs and GPUs. The GREEN cache can be combined with the previous mechanism by analyzing the effective working set size of a GPU application so that the interference caused by GPU cores can be minimized.

7.3.5.2 Overhead of GREEN Cache

We analyze possible overhead of the GREEN cache in this section. First, RAC does not incur significant hardware or training overhead. Since the GPU device driver provides the region information to the hardware, extra logic for collecting per-region information is not necessary. In addition, the number of regions is usually small (less than five in Table 21), so total hardware overhead will be less than 125B since region information requires around 100 bits. Regarding the region training, most processors already have the capability to count architectural events such as the number of cache hits and misses, so we can largely reuse the existing circuit

and store the outcome in a table indexed with the region ID.

Second, RACR-Naïve and RACR-Bypass do not require any extra hardware. In RACR-Naïve, the working set size can be estimated using information provided by the device driver, and RACR-Bypass acquires bypassed region information from the RAC mechanism. However, to reduce the overhead of flushing cache lines upon resizing, RACR uses the 4k bit-vector.

Finally, the selective caching in the GREEN cache may affect the cache coherence protocol, which is complicated and error-prone. Fortunately, it turns out that we do not have any side-effect on cache coherence by using region-aware caching. Regardless of cache bypassing, all memory requests need to access a tag directory first to check for the existence of the same line. Furthermore, most modern processors, including CPUs and GPUs, natively support different caching operators for cache affinity [54, 108]. As a result, our proposal can benefit from these existing cache control designs.

7.3.6 GREEN Cache with Multiple Applications

While the discussion so far has been limited to building an energy-efficient cache hierarchy for a GPU kernel, in this section we expand the discussion to GPUs with multiple kernels or applications. Recent GPUs [105] can support concurrent execution of multiple kernels on the same device. In this case, the L2 cache will be shared by multiple applications or kernels. This is analogous to sharing the last-level cache across different CPU applications. To prevent inter-application interferences for CPU workloads, many cache partitioning or replacement/insertion policies have been proposed in the past [57, 58, 118, 120]. These mechanisms aim to improve the cache hit rate, thereby improving performance. To reduce energy while benefiting from previous performance-oriented mechanisms, here we propose to integrate the GREEN cache in these mechanisms. Moreover, we

found various advanced ways to utilize the GREEN cache, which we explain below.

GREEN cache (RAC + RACR) First, we can apply the GREEN cache to the multiple-kernel environment. Since we collect per-region cache information in the core, RAC will not be affected by the existence of other kernels. However, for RACR, the aggregated working set should now account for the working set of all kernels, so the cache resizing controller should aggregate information from all GPU cores and determine how many cache ways should be active.

GREEN cache + Dynamic cache partitioning Moreover, we can utilize the information collected by the GREEN cache to partition cache space between kernels. The partitioning strategy can be different in two cases:

1) When the aggregated working set size is smaller than the L2 size, the number of cache ways for each application will be set by Eq. (22), and we can disable a few ways if available. Each application should acquire at least one cache way.

2) When the aggregated working set size is greater than the L2 size, we partition the cache ways proportionally to the working set size of each application. We first calculate the total region size from all kernels, excluding bypassed regions, as in Eq. (23), which is derived from Eq. (21). Then, we set the number of cache ways for each application based on Eq. (24). Each application will have at least one cache way, and the sum of all allocated cache ways from kernels cannot be greater than the L2 associativity.

$$total\_size = \sum_{i}^{N} new\_size_i \tag{23}$$

$$num\_way_i = \left\lceil L2\_assoc \times \frac{new\_size_i}{total\_size} \right\rceil \tag{24}$$

, where N is the number of kernels

RAC + Previous mechanisms We can integrate RAC in previous mechanisms to save dynamic energy since RAC is orthogonal to the underlying cache mechanism. Clearly, cache accesses that always miss in the caches will not affect the behavior of the underlying cache mechanisms. As a result, we can benefit from performance-oriented prior proposals while saving dynamic cache energy. However, we cannot directly apply any RACR mechanisms since cache resizing will significantly affect the behavior of the underlying cache mechanism. In particular, in this work, we integrate RAC in utility-based cache partitioning (UCP) [120], a dynamic cache partitioning mechanism that uses an auxiliary tag directory to track cache hit information, and re-reference interval prediction (RRIP) [58], a dynamic cache insertion policy that uses a set dueling technique to identify the best insertion position.

7.3.7 Discussions

In this section, we discuss a few interesting issues with GREEN cache.

Applicability to future programming model While GREEN cache can be an energy-efficient solution with the current OpenCL model, the next question would be whether our proposal will be useful in the future, especially when a future GPU programming model can be more flexible and easy to program. For example, CUDA starts to support a more flexible programming model such as pointer support.

While we believe that an advanced programming model can improve the programmability of a GPU, we also strongly believe that such a relaxed access pattern should be used only when really needed since the fundamental nature of a GPU, SIMDness, prefers the coalesced access pattern in a linear memory region. As explained in Section 7.3.1, too frequent uncoalesced accesses within a warp will generate lots of scatter-gather patterns, which will be serialized due to the

limited port counts throughout the entire cache and memory hierarchy. As a result, the relaxed access pattern will significantly waste cache/memory bandwidth and degrade performance and energy efficiency. In other words, for applications with many irregular memory accesses, we cannot run them efficiently on a GPU, and it is better to run them on a general-purpose CPU.

On the other hand, a future programing model may not force programmers to use an explicit API such as *clCreateBuffer*. Even in this case, our proposal demonstrates that such semantic information or hint is very useful in optimizing cache hierarchy; thus we may still want to maintain such information in an alternative form such as pragmas to enjoy the benefit of programming model/hardware collaborative optimization.

Input size One might wonder why cache resizing can be effective given that the typical input size is greater than the cache size. While the input size itself is larger than the cache capacity, we observed that not all these input data are brought to the on-chip cache at the same time. In particular, we found that a hardware scheduling policy of existing GPUs has an interesting, positive impact on cache efficiency as follows: Many work-items (or CUDA threads) are grouped into a unit called a workgroup (or CUDA thread block), one or more of which are assigned to one compute unit. Only after a workgroup finishes its entire kernel execution, next available workgroup is dispatched to the compute unit. Due to this unique scheduling, the cache needs to be as large as a working set for workgroups that can be executed in parallel at the same time on a given GPU hardware, not the entire input set. Moreover, as far as many memory regions being set to bypass, RACR-Bypass can be effective since the size of all these variables will not be counted.

Cache set sampling Cache set sampling [119, 120] can be an alternative way of estimating the working set size. Cache set sampling maintains a true LRU stack

and counters for each position for the stack to estimate cache hit pattern from a few sampled sets. Then, based on how many cache hits occur in how many cache ways from the most recently used (MRU) position, it can decide the number of ways to be active. This approach has advantages and disadvantages over RACR approach. Since it monitors the actual run-time behavior, it can more precisely estimate the real working set size, while RACR estimates the maximum working set size. However, it has run-time overhead for maintaining a stack and counters per cache accesses in sampled sets. In sum, cache sampling can be more precise, but it has more overhead than RACR.

Virtual memory We assumed that GPUs use a future fully shared virtual memory system in the heterogeneous architecture. We believe that our proposal works well in a fully shared virtual memory space, as our "region" is identified and tagged during the virtual-to-physical address translation.

7.4 Evaluation Methodology

7.4.1 Simulator

We evaluate the GREEN cache by extending MacSim simulator [45]. Also, we model a GPU core that is similar to a modern GPU core. Detailed configurations of our baseline design are shown in Table 20. Note that we conservatively model a small, 4MB L2 cache, while a state-of-the-art integrated GPU in Intel's Haswell products has a 128 MB L4 cache [48].

Table 20: Evaluated GPU configurations.

GPU Core	1.2 GHz, 12 cores, in-order, 32 SIMD width
Cache	4-way 32KB L1 cache with 64B lines (write-back)
	16-way 4MB L2 cache with 64B lines (write-back)
	Shared memory, texture memory, constant cache
DRAM	1600 MHz, 4 memory controllers

7.4.2 GPU Power Model

In addition to functional and timing models, we need a power model to show the benefit of the proposal. For this reason, we have developed a GPU power model using Energy Introspector [1], which is based on McPAT [78], and extend MacSim with this model. We faithfully considered all possible GPU architectural components and performed very detailed parameter space explorations and validations by comparing it with real graphics cards. We also attempted to validate leakage by selectively turning on a small subset of GPU cores and varying the frequency. We did our best to correlate power numbers against the measured data on real hardware.

7.4.3 Benchmarks

To quantify the performance and power results of the proposals, we use a total of 32 benchmarks from NVIDIA SDK, Rodinia [20], and Parboil [137] suites. To help readers understand the property of these applications, we listed in Table 21 the number of regions, summation of all region sizes, number of misses per thousand SIMD instructions (MPKI), and both full and abbreviated names for each benchmark.

7.4.4 Evaluation Metric

In this section, to help readers to understand the simulation results clearly, we clarify some metrics used in the result. First, *cache access* is defined in Eq. (25). The number of cache accesses, which is highly correlated with the dynamic cache energy consumption, consists of two types: access and insertion. One cache hit requires one access, but one cache miss requires two accesses: one access (*miss_access*) to check the existence of a cache line and the other to fill a cache line

Table 21: Benchmark list.

Benchmark	Abr.	Avg. Region size per kernel (MB)	MPKI	# Region
backprop	BP	6.88	15.8	3
bfs	BFS	37.19	10.0	4
cfd	CFD	29.26	165.8	4
hotspot	HS	3.00	11.5	3
lud	LUD	0.25	3.5	1
pathfinder	PF	38.53	40.6	3
srad-v1	S1	7.02	73.0	10
srad-v2	S2	96.00	69.2	6
streamcluster	SC	66.82	220.0	5
cutcp	CC	38.34	0.22	2
lbm	LBM	370.3	391.6	2
mri-q	MQ	3.36	0.58	5
sad	SAD	8.59	26.65	3
sgemm	SG	3.36	6.6	3
spmv	SP	29.49	209.6	5
stencil	ST	128	53.2	2
blackscholes	BS	76.29	134.5	5
fdtd3d	FD	128.00	44.5	2
mersennetwister	MT	91.63	51.3	2
montecarlo	MC	1.05	0.43	2
sobolqrng	SQ	38.16	76.8	2
binomialoptions	BO	8.09	0.16	2
convolutionseparable	CS	108.00	86.7	2
convolutiontexture	CT	38.69	45.8	2
dxtc	DX	1.13	0.07	3
eigenvalues	EV	0.10	0.01	10
fastwalshtransform	FW	64.00	177.8	1
histogram	HI	72.00	54.1	2
mergesort	MS	96.50	21.6	6
quasirandomgenerator	QG	12.00	18.3	1
radixsort	RS	16.03	118.5	6
reduction	RE	64.00	307.8	2

(*miss_fill*) from the lower-level cache or off-chip memory.[6]

$$cache\_access = hit + miss\_access + miss\_fill \tag{25}$$

Second, we define *active cache size* as shown in Eq. (27). Average active cache size is same as the average cache usage, which is a good proxy for the leakage power of a cache. To calculate this metric, we first measure $ratio_i$ as in Eq. (26), the normalized execution time in which only i ways (out of N total ways) are enabled. Based on this metric, we calculate active cache size by calculating the weighted average as in Eq. (27).

$$ratio_i = \frac{cycles\ with\ i\ ways}{total\ simulation\ cycles} \tag{26}$$

$$active\_cache\_size = \sum_{i=1}^{N} \frac{ratio_i \times i}{N} \times 100 \tag{27}$$

, where N is the number of cache ways

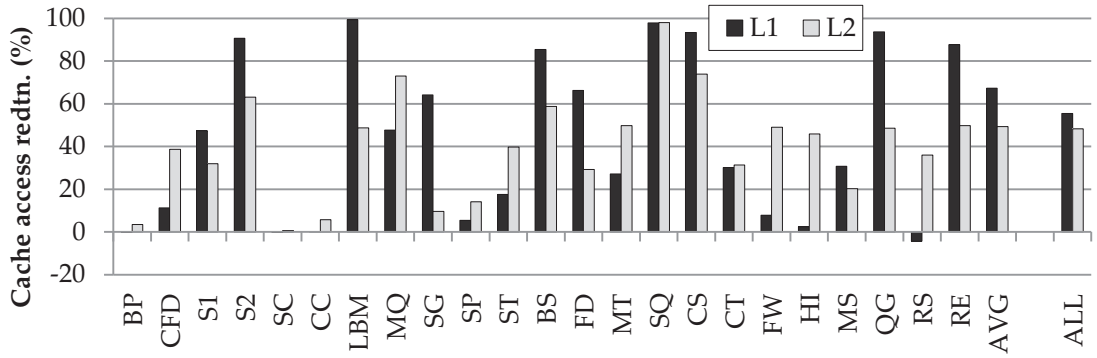
7.5 Evaluation Results

7.5.1 Region-Aware Caching Results

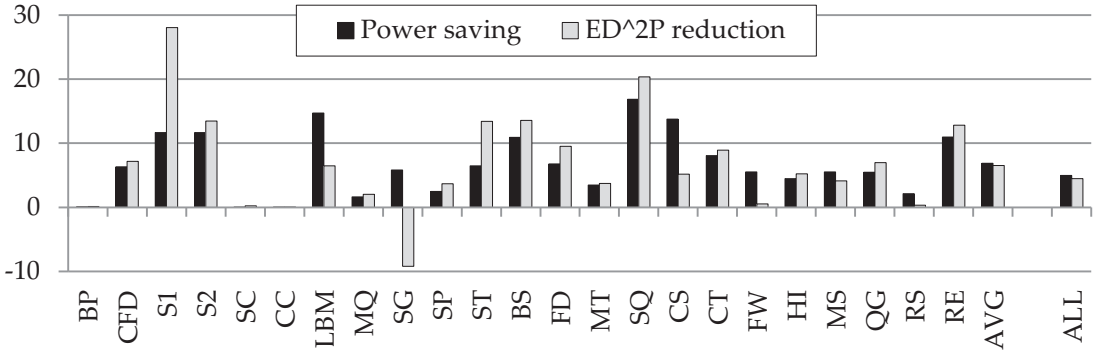
In this section, we evaluate RAC from various aspects: performance, the number of cache accesses, and energy consumption. Before explaining the result, we first want to clarify that some benchmarks (BFS, HS, LUD, PF, SAD, MC, BO, DX, EV, and FW) do not show any difference with RAC because these applications have a decent cache hit rate in both L1 and L2 caches. As a result, no region or memory object is selected to bypass any cache hierarchy. Note that we do not show the result of these benchmarks in detail due to space constraints, but we do include these results when we calculate the average.

[6]We found that energy consumptions of cache read and write are very similar from the output of Cacti [44].

First, we analyze the number of cache accesses, which is defined in Eq. (25). Figure 56 (a) shows that we can remove a significant number of unnecessary cache accesses across a wide range of applications. For example, we can save almost all the dynamic energy of L1 and L2 caches in SQ. Although its working set size is huge (38.16 MB in Table 21), RAC identifies that no region has a high hit rate and removes unnecessary energy consumption on cache lookups and insertions. On average, RAC reduces L1 and L2 accesses, thereby reducing dynamic energy consumption, by 55% and 44%, respectively, across 32 benchmarks.



(a) Cache access reduction (%) defined in Eq. (25)



(b) Total system energy savings (%) and ED^{2}P reduction (%)

Figure 56: The evaluation of RAC (AVG: the average of 23 benchmarks shown in the figure, ALL: all 32 benchmarks).

However, if RAC degrades performance or increases the number of off-chip accesses, the energy efficiency of a system may decrease. Therefore, RAC should not lead to these side-effects. To understand whether RAC incurs any negative effect, we measured the overall performance and the number of off-chip accesses.

We found that only four out of 32 benchmarks (LBM, SG, CS, and FW) show greater than a 2% performance degradation. Among them, RAC increased the number of off-chip accesses when a GPU runs LBM or CS. As a result, their performance was also degraded. However, we found that the performance degradation for the other two benchmarks was not caused by the increased number of off-chip accesses but rather was caused by the altered memory access pattern, which led to more bank conflicts in caches and bank/row conflicts in DRAM memory. Note that although the number of off-chip accesses in the MQ benchmark is significantly increased by 39.6%, the absolute number of off-chip accesses is much smaller than that of L1 and L2 cache accesses (MPKI of MQ in Table 21 is very small). Thus, it does not affect the overall energy consumption and performance.

Figure 56 (b) shows the overall system energy savings and energy-delay^2 product (ED^{2}P) reduction by RAC. Note that this result includes the energy consumption of all components including proposed mechanisms and off-chip accesses. Although we found some benchmarks with performance degradation, only SG shows increased energy consumption and ED^{2}P. On average, RAC shows a 4.8% of total energy savings and a 4.5% ED^{2}P reduction. In particular, the S1 benchmark shows the most benefits of 16.9% and 28.4% total energy savings and ED^{2}P reduction, respectively.

7.5.1.1 Training Period Sensitivity of RAC

Another question that we were interested in was how long we should train the mechanism. The length of the training period in RAC may have a significant impact on performance and energy consumption. If the training period is too short and an application requires a fairly long time to warm up the cache, training with limited information can lead to a wrong decision. As a consequence of the wrong decision, RAC may increase the number of off-chip accesses, resulting in

degraded performance and increased energy consumption. On the contrary, if the training period is too long, RAC could capture the right cache behavior, but it may lose more opportunity to save energy. Due to these motivations, we performed a sensitivity study in this section. The results of the sensitivity study are well captured in Figure 57.

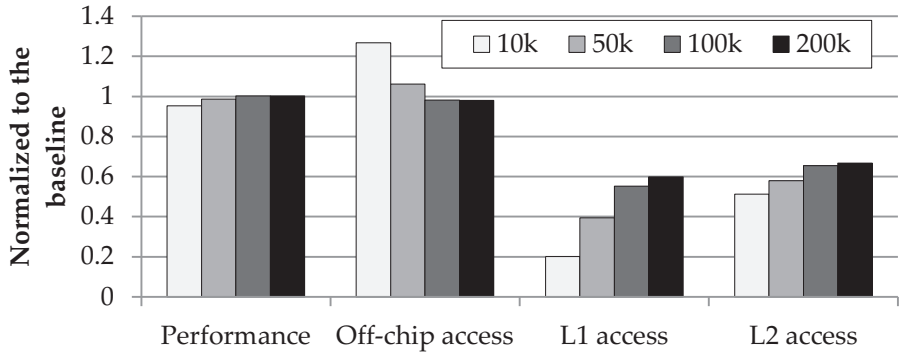


Figure 57: Training period sensitivity of RAC .

This result clearly shows the following trends: 1) When we have a training period that is too short (10,000 cycles), misprediction is more likely to happen. As a result, we can observe that the number of off-chip accesses has significantly increased, which leads to a performance degradation; 2) When we have a training period that is too long (200,000 cycles), we lose opportunities for dynamic cache energy savings. From this study, we concluded that the training period of 100,000 cycles is reasonable considering these trade-offs.

7.5.2 Region-Aware Cache Resizing Results

We evaluate RACR in this section. As explained in Section 7.3.4, we have two types of RACR technique: RACR-Naïve and RACR-Bypass. RACR-Naïve can be effective only if the original working set size of an application is less than the L2 cache size, but RACR-Bypass provides additional leakage energy savings on top of the dynamic energy savings provided by RAC. If many regions have not shown cache-friendly behavior and the aggregated working set size is less than the L2

cache size, we can further reduce the leakage energy of the L2 cache. RACR results are shown in Figure 58. Note that some benchmarks, BP, PF, S2, SC, CC, SAD, SG, SC, FD, BO, and CT, do not benefit from RACR-Naïve or RACR-Bypass since these applications have a working size greater than the L2 size and RAC cannot reduce a working set size smaller than the cache size. We do not show the result of these benchmarks due to space constraints, but we do account for these applications for the average numbers.

First, we analyze the effectiveness of RACR. Figure 58 (a) shows the normalized active cache size. Because we normalized active cache size to a fully enabled cache as in Eq. (27), the result will be one when there is no benefit to using RACR and zero when all cache ways are turned off. In general, energy savings by RACR-Bypass are more effective than by RACR-Naïve. Interestingly, we found four applications, HS, LUD, MC, and EV, with better benefits using RACR-Naïve than RACR-Bypass. We analyzed these cases and made the following observation: These applications consist of multiple short kernels. Unfortunately, the training period for RACR-Bypass is similar or longer than the entire kernel execution time. Consequently, the amount of time that these applications run under RACR-Bypass is very short, thereby limiting the benefit of RACR-Bypass. On the other hand, because RACR-Naïve is applied from the beginning of the kernel execution, RACR-Naïve ends up saving more leakage energy. In addition, RACR, which combines RACR-Naïve and RACR-Bypass, turns out to be effective in 21 benchmarks and saves leakage energy in the L2 cache by 38% on average for all 32 benchmarks.

We also examined whether cache resizing ends up degrading overall performance. We found that none of the applications shows greater than a 5% performance degradation, and only four benchmarks (BFS, S1, CS, and FW) suffer from more than 2% performance degradation. It turns out that the performance degradation was caused by the increased conflict cache misses. For example, if

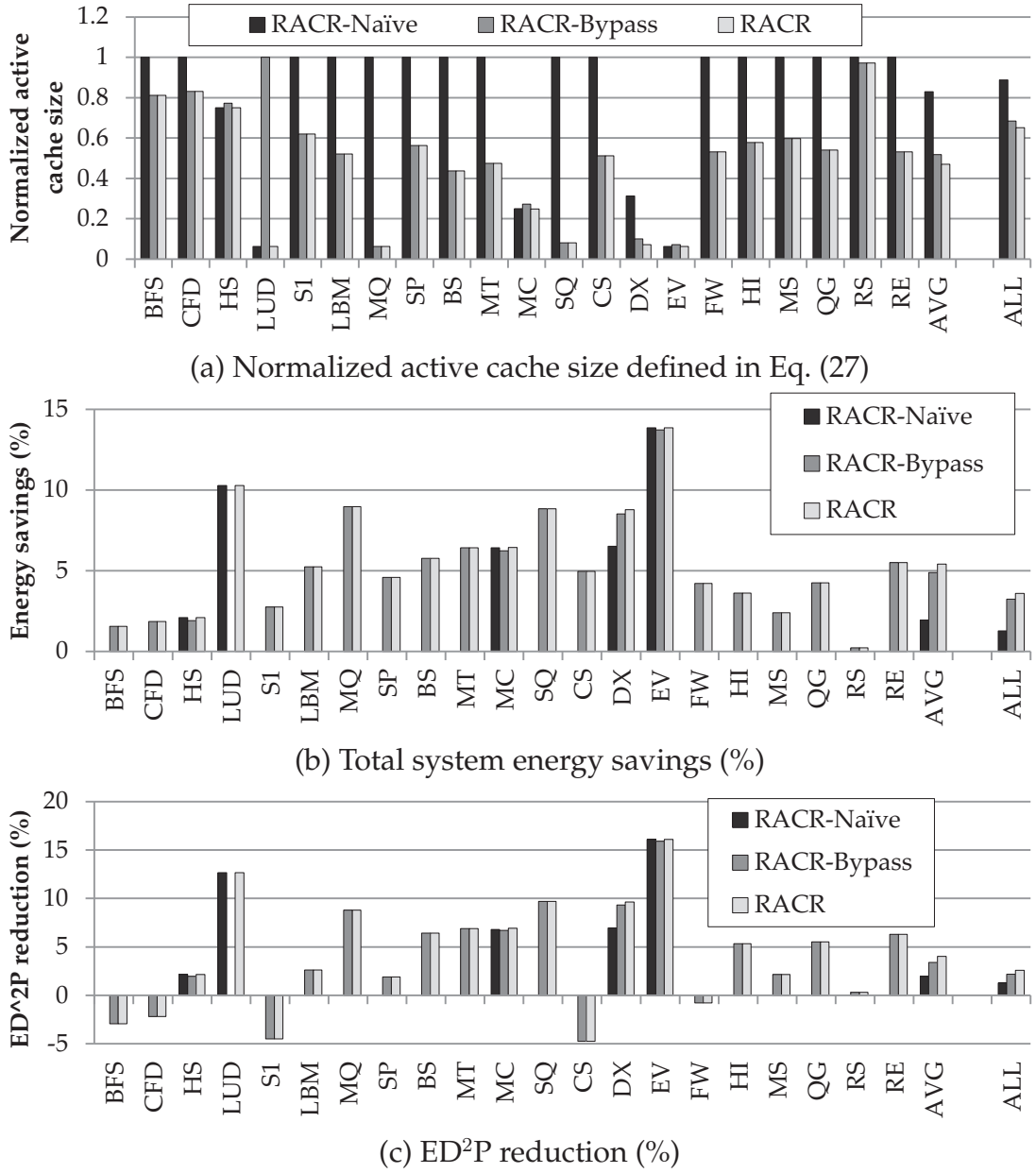


Figure 58: The evaluation of RACR (AVG: the average of 21 benchmarks shown in the figure, ALL: all 32 benchmarks).

an application utilizes only a few hot cache sets, the reduced associativity will increase the number of conflict misses. However, note that these benchmarks do not increase energy consumption due to greater leakage energy savings, as shown in Figure 58 (b). As a result, the ED^2P result in Figure 58 (c) for these benchmarks is decreased by at most 4%, while other benchmarks do not show negative cases. Overall, RACR can save a total of 3% system energy and improve ED^2P by 2.6% on average across 32 benchmarks.

7.5.3 Putting It All Together

In this section, we now show the result of the GREEN cache, which combines RAC and RACR, and evaluate the benefit of the GREEN cache using the following metrics:

- L1 dynamic power savings (L1 Dyn) in % by RAC
- L2 dynamic power savings (L2 Dyn) in % by RAC
- L2 leakage power savings (L2 Leak) in % by RACR
- Total cache (Cache) and system (Total) power savings in % by the GREEN cache
- Performance improvements (Perf) and # off-chip accesses increased (Off-chip) in % by the GREEN cache

Note that we apply the dynamic cache resizing mechanism only in the L2 cache as explained, so the mechanism does not reduce L1 leakage energy. Figure 59 shows the result. Across a wide range of applications (total 32 GPGPU applications), the GREEN cache shows a huge benefit in each energy-savings category while not hurting performance and not incurring extra off-chip accesses. On average, the GREEN cache can save 56% of L1 dynamic energy (L1 Dyn),

39% of L2 dynamic energy (L2 Dyn), and 50% of L2 leakage energy (L2 Leak), which eventually leads to a 29.5% (Cache) and 8.5% (Energy) energy savings in caches and total system, respectively. Meanwhile, we observe a 0.6% performance degradation (Perf.) and 0.4% extra off-chip accesses (Off-chip).

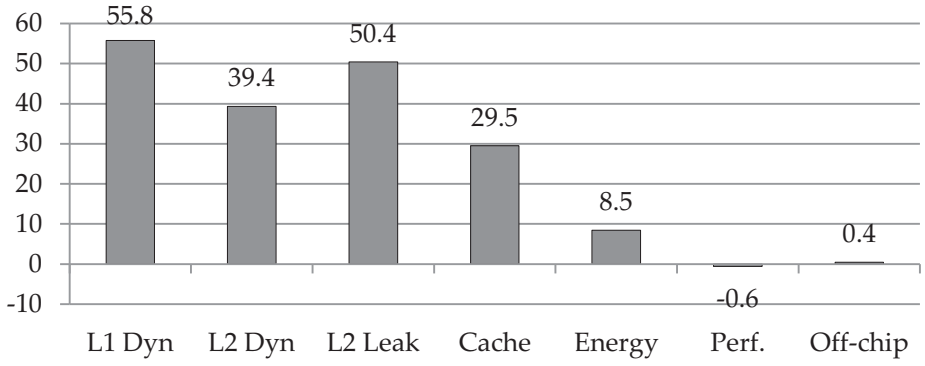


Figure 59: GREEN cache - putting it all together.

7.5.4 Multiple GPU Applications

In this section, we evaluate the extensions of the GREEN cache that support GPUs with multiple applications (Section 7.3.6). For evaluations, we use 20 pairs of GPU applications where each pair was randomly chosen, and we use the weighted speedup metric [127] as defined in Eq. (28). For clarity, we list all evaluated mechanisms with their abbreviated names in Table 22.

$$wspeedup = \sum_{i}^{N} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{28}$$

, where N is the number of applications

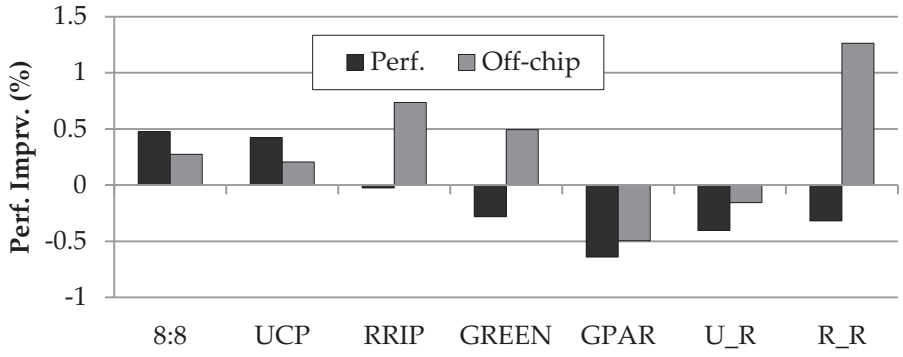
We first show the average performance improvements of these 20 workload pairs with different mechanisms in Figure 60 (a). Interestingly, overall performance turns out to be insensitive to cache partitioning schemes. Clearly, this is very different from observations made by cache partitioning studies for CPU workloads, but we realize that these results are consistent with the

Table 22: List of mechanisms for multi-app experiments.

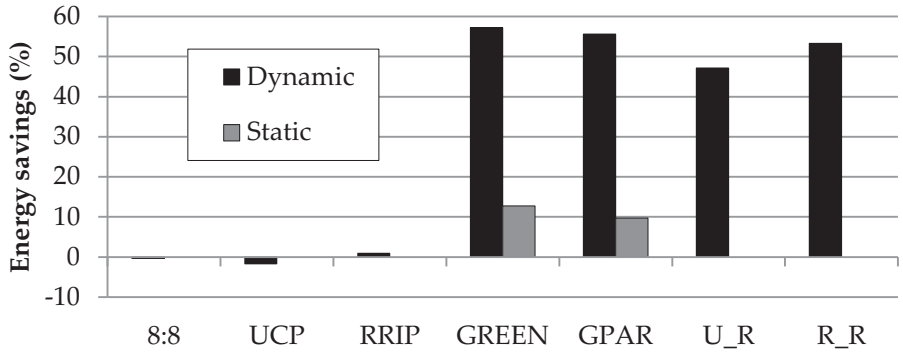
Abr.	Mechanism	Ref.
Base	Baseline LRU cache	
8:8	8:8 static cache partitioning	
UCP	Utility-based cache partitioning	[120]
RRIP	Re-reference interval prediction	[58]
GREEN	Original GREEN cache	Sec. 7.3
GPAR	GREEN cache with dynamic cache partitioning	Sec. 7.3.6
U_R	UCP with RAC	Sec. 7.3.6
R_R	RRIP with RAC	Sec. 7.3.6

previous observations from Figure 58: We can resize the cache without affecting performance much. As long as the cache can provide enough capacity for those cache-friendly regions, the GPGPU workload is less sensitive to the underlying cache mechanism. However, we found that UCP and RRIP show 11% and 7% maximum performance improvement, respectively, in a few cases although the average performance improvements are less than 1%. With the RAC-integrated mechanisms (U_R and R_R), these extensions show a similar maximum and average benefits with their counterpart. On the other hand, GREEN and GPAR show slightly less overall and maximum improvements than other mechanisms.

Regarding the number of off-chip access changes, overall changes in all mechanisms are negligible, but we found that some cases reduce the off-chip traffic by more than 20% compared to the baseline. Although changes in performance and the number of off-chip accesses are negligible, the dynamic and static cache energy savings from the GREEN cache is significant, as shown in Figure 60 (b). Previous mechanisms can save dynamic energy only if they can reduce the number of cache misses and off-chip accesses, while extensions of the GREEN cache can save energy by selectively caching memory accesses. As shown, static partitioning (8:8), UCP, and RRIP do not lead to significant energy savings. On the contrary, all GREEN extensions show a significant (around a 50%) dynamic energy savings.



(a) Performance improvements (Weighted speedup in %) and off-chip access decrease (%) compared to Base



(b) Cache energy saving (%) compared to Base

Figure 60: Multiple application evaluations.

However, the GREEN cache shows less effectiveness for the static energy savings due to the increase in aggregated working set size from multiple applications. This is an obvious limitation of RACR-Naïve, but may not be for RACR-Bypass since it can be effective if many memory regions are set to bypass due to unfruitful cache activities. In summary, as shown in Figure 60 (b), GREEN and GPAR, which use RACR, show 12.7% and 9.8% average static cache energy savings, respectively, across 20 workload pairs.

From this experiment, we conclude that the GREEN cache can be an energy-efficient solution even in multiple-application environment. Furthermore, we found that the GREEN cache works synergistically with previously proposed, performance-oriented cache partitioning methods.

7.6 Summary of This Chapter

In this chapter, we propose utilizing existing semantic information of the OpenCL programming model for building an energy-efficient cache hierarchy, especially in the heterogeneous architecture where CPUs and GPUs share a large on-chip cache. To demonstrate good opportunities for the cross-layer optimization, we first profiled GPGPU applications and found that cache hit rate stays almost constant within a neighboring region of memory while it widely varies across different regions of memory. Then, we found that different levels of cache hit rates highly correlate with different memory objects of the GPGPU application. Based on these findings, we proposed two techniques: *region-aware caching (RAC)* and *region-aware cache resizing (RACR)*. RAC selectively caches a subset of memory objects, and RACR disables a subset of a set-associative cache so that it is just large enough to hold the aggregate working set of cache-friendly memory regions. With these dynamic and leakage energy saving techniques, we found that our proposal, the GREEN cache, can save 56% and 39% of dynamic energy in the L1 and L2 caches, respectively, and 50% of leakage energy in the L2 cache with practically no performance degradation. We also proposed several extensions of the GREEN cache for GPUs with multiple applications, and these extensions show effectiveness.

CHAPTER VIII

CONCLUSION AND FUTURE RESEARCH DIRECTION

8.1 Conclusion

The need for better performance, and power/energy efficient computing brought the advent of heterogeneous chip-multiprocessors (HCMPs) and they become the mainstream computing platform. However, due to the heterogeneity of cores in HCMPs, different aspects of resource sharing problems appeared, in particular the effect of thread-level parallelism and significant interference caused by GPU cores. In order to tackle the problem, we present four resource sharing mechanisms: (1) thread-level parallelism-aware cache management, (2) adaptive virtual channel partitioning, (3) dynamic frequency regulating mechanism, and (4) region-aware energy-efficient GPU cache mechanism, that exploit the different characteristics of CPU and GPU cores.

- Chapter IV presents a cache-sharing mechanism called TLP-aware cache management policy (TAP). Due to the abundant thread-level parallelism (TLP) of GPU cores, cache related metrics, such as misses per kilo instructions (MPKI), can often be misleading. Also, due to the excessive cache accesses from GPU applications, CPU applications are often unnecessarily penalized. In order to identify the effect of TLP on caches, a core sampling technique is proposed. Moreover, cache block lifetime normalization is also proposed to consider the different degree of cache accesses to isolate the interference caused by GPUs. These two TAP mechanisms are applied to two previous mechanisms, utility-based cache partitioning (UCP) and re-reference interval prediction (RRIP).

- Chapter V describes adaptive virtual channel partitioning for the on-chip interconnection network. We observe that memory requests consume a significant amount of time in the network, in particular in the injection queues. In order to solve the network contention problem, we apply a resource partitioning technique to the virtual channels of the router. Routers usually have multiple virtual channels and all cores (or applications) share the VCs in routers attached to the memory system (caches and memory controllers). Under VCP, multiple VCs are partitioned to CPU and GPU cores, so that packets can only use the corresponding type of VC to isolate the interference. In addition, we claim that separate injection queues for CPU and GPU cores should be used and DAMQ is used to implement them. VCP enables simple and cheap packet arbitrations in a router while supplying a more balanced number of packets from CPU and GPU cores to the network.

- Chapter VI describes a dynamic frequency regulating mechanism (DyFR). The DVFS technique is typically employed to save power or optimize performance within a power budget. On the contrary, DyFR uses the DVFS technique to mitigate the inter-application interference caused by memory-intensive applications. In addition, DyFR also considers the frequency-scalability of applications. Based on the interference and application characteristics, core clock frequencies are dynamically adjusted. Then, the memory clock frequency is automatically controlled based on the available power budget. We introduce two different DyFR modes: power-saving and high-performance based on how the remaining power for the memory should be utilized. DyFR overcomes the limitations of previous mechanisms: 1) when resource contention does not exist and 2) when core resources waste power and energy due to idling. DyFR achieves both performance improvement and energy efficiency.

- Chapter VII proposes a GREEN (GPU region-aware energy-efficient non-inclusive) cache to effectively utilize a large shared last-level cache by GPU cores in HCMPs. Conventional discrete GPU systems have only a small capacity of last-level caches, but much larger caches are now available for the GPU since GPU cores are integrated into the HCMPs. The last-level caches in this architecture are not optimized for GPU applications, so the energy-efficiency of caches used by GPU cores can be decreased. Consequently, we propose two mechanisms: RAC (region-aware caching) and RACR (region-aware cache resizing), to save static and dynamic cache energies. The intuition of GREEN cache is that the GPUs employ disciplined programming and memory models for achieving better parallel performance, so it allows us to perform semantic-aware optimizations easily.

8.2 Future Research Direction

8.2.1 Future Work for TAP

In Chapter 4, we consider only the workload that consists of one GPU application and multiple multi-programmed CPU workloads. However, more diverse workloads can be running on HCMPs, for example OpenCL [111] like applications that utilize both CPU and GPU cores, GPU applications with multi-threaded CPU applications, and multiple GPU applications running on GPU cores. Since it is important that TAP can adapt well to various workloads, we will discover other behaviors in different workloads in future work.

8.2.2 Future Work for VCP

As discussed in Sections 5.3.3.3 and 5.3.6, sampling-based mechanisms can provide a simple and cheap solution, but they also have some drawbacks. A better solution could be to rely on a statistical or analytical model-based approach. There is a rich body of work on previous mechanisms that model traffic patterns

using statistical models. If we can combine the traffic model with the GPU performance model, we can completely remove the sampling and acquire more benefits. However, modeling such behavior for the heterogeneous architecture is not trivial and we have to construct a different performance model for the CPU, GPU, and the network.

In addition, as shown in Sections 5.5.4 (different router arbitration policies) and 5.5.6 (adaptive routing mechanism), VCP can be combined with other NoC mechanisms because it is orthogonal to others. We can consider adaptive routing mechanisms, packet arbitrations, source throttling, congestion control mechanisms, and many others to combine with VCP to be more effective.

8.2.3 Future Work for DyFR

Temperature is also very important factor that affects power. Each component has thermal headroom and the frequency will be throttled down when the temperature exceeds the headroom. Higher voltage/frequency increases the temperature and higher temperature in turn increases power consumption. A recent proposal by Paul et al. [114] considers both thermal and performance coupling. In HCMPs, the heat dissipation is exchanged between cores since they share the same die. As a result, even if one type does not consume much power, its voltage/frequency can be scaled down due to the heat dissipation by other cores. Consequently, DyFR will consider the thermal effect in future work to be more accurate.

8.2.4 Future Work for GREEN Cache

In Chapter 7, we consider only how to utilize the shared last-level cache in an energy-efficient manner for GPU cores. Moreover, we can apply the semantic-aware caching mechanisms to partition cache space between CPU and GPU applications, similar to the TAP mechanism. Once we identify the required cache amount for the GPU using region-aware caching schemes, we can apply other

cache partitioning mechanisms such as UCP to partition cache space between CPU applications. Since the semantic information can be easily given to the hardware without extra cost, this approach can be a very inexpensive yet effective solution for sharing the last-level cache between CPU and GPU cores.

8.2.5 Coordinated Resource Sharing

The eventual goal of this thesis is to design a coordinated system that combines cache sharing (TAP), interconnection network (VCP), and frequency regulating (DyFR) mechanisms. TAP and VCP are applied to different shared resources, while DyFR controls the frequency of all components in HCMP. At a glance, these mechanisms seem to be orthogonal to each other, but this is not true. First, the cache and on-chip network are connected, so the behavior of one component will affect that of the other component. For example, having better shared cache management can reduce the number of cache misses, which in turn reduces the number of network packets. Also, based on how packets are scheduled/traversed from the network, cache sharing behavior will be affected as well. Second, synchronization is necessary for all mechanisms. A sampling technique is used in both TAP and VCP mechanisms and they rely on it to collect performance metrics to determine the effect of a configuration. If voltage/frequency change is applied in the middle of the sampling period, the outcome of sampling will be inaccurate. As a result, sampling is likely to fail. Also, shared resource contention and inter-application interference within them are affected by DyFR.

These reasons inspire the need for a coordinated (or synchronized) framework for efficient resource sharing in heterogeneous chip multiprocessors that can effectively consolidate the benefit of all proposed mechanisms.

REFERENCES

[1] "Energy Introspector," Georgia Tech. http://manifold.gatech.edu/projects/energy-introspector/.

[2] ABTS, D., JERGER, N. D. E., KIM, J., GIBSON, D., and LIPASTI, M. H., "Achieving predictable performance through better memory controller placement in many-core cmps.," in *Proc. of the 31st annual Int'l. Symp. on Computer Architecture, ISCA-31*, (New York, NY, USA), pp. 451–461, ACM, 2009.

[3] ALBONESI, D., "Selective cache ways: On-demand cache resource allocation," in *Proc. of the 32nd Int'l. Symp. on Microarchitecture, MICRO-32*, (Washington, DC, USA), pp. 248–259, IEEE Computer Society, 1999.

[4] AMD, "Phenom II key architectural features." http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx.

[5] AMD, "Fusion." http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx, 2011.

[6] AMD, "Graphics Core Next (GCN)." http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx, 2012.

[7] ANNAVARAM, M., GROCHOWSKI, E., and SHEN, J., "Mitigating amdahl's law through epi throttling," in *Proc. of the 27th annual Int'l. Symp. on Computer Architecture, ISCA-27*, (Washington, DC, USA), pp. 298–309, IEEE Computer Society, 2005.

[8] AUGONNET, C., CLET-ORTEGA, J., THIBAULT, S., and NAMYST, R., "Data-aware task scheduling on multi-accelerator based platforms," in *Proc. of the 2010 IEEE 16th Int'l Conf. on Parallel and Distributed Systems, ICPADS'10*, (Washington, DC, USA), pp. 291–298, IEEE Computer Society, 2010.

[9] AUSAVARUNGNIRUN, R., LOH, G., CHANG, K., SUBRAMANIAN, L., and MUTLU, O., "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *Proc. of the 34th annual Int'l. Symp. on Computer Architecture, ISCA-34*, (Piscataway, NJ, USA), pp. 416–427, IEEE Press, 2012.

[10] BAKHODA, A., KIM, J., and AAMODT, T. M., "Throughput-effective on-chip networks for manycore accelerators.," in *Proc. of the 43rd Int'l. Symp. on Microarchitecture, MICRO-43*, (Washington, DC, USA), pp. 421–432, IEEE Computer Society, 2010.

[11] BALLAPURAM, C. S., SHARIF, A., and LEE, H.-H. S., "Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors," in *Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XIII*, (New York, NY, USA), pp. 60–69, ACM, 2008.

[12] BASHIR, A., LI, J., IVATURY, K., KHAN, N., GALA, N., FAMILIA, N., and MOHAMMED, Z., "Fast lock scheme for phase-locked loops," in *Proc. of IEEE Custom Integrated Circuits Conference, CICC'09*, (Washington, DC, USA), pp. 319–322, IEEE Computer Society, 2009.

[13] BEIGNÉ, E., CLERMIDY, F., VIVET, P., CLOUARD, A., and RENAUDIN, M., "An asynchronous noc architecture providing low latency service and its multi-level design framework," in *Proc. of the 11th IEEE Int'l Symp. on Asynchronous Circuits and Systems, ASYNC'05*, (Washington, DC, USA), pp. 54–63, IEEE Computer Society, 2005.

[14] BJERREGAARD, T. and MAHADEVAN, S., "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, June 2006.

[15] BJERREGAARD, T. and SPARSØ, J., "A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'05*, (Washington, DC, USA), pp. 1226–1231, IEEE Computer Society, 2005.

[16] BOLOTIN, E., CIDON, I., GINOSAR, R., and KOLODNY, A., "Qnoc: Qos architecture and design process for network on chip," *Journal of Systems Architecture (JSA)*, vol. 50, no. 2–3, pp. 105–128, 2004.

[17] CHANG, D. W., JENKINS, C. D., GARCIA, P. C., GILANI, S. Z., AGUILERA, P., NAGARAJAN, A., ANDERSON, M. J., KENNY, M. A., BAUER, S. M., SCHULTE, M. J., and COMPTON, K., "Ercbench: An open-source benchmark suite for embedded and reconfigurable computing.," in *20th Int'l Conf. on Field Programmable Logic and Applications, FPL'10*, (Washington, DC, USA), pp. 408–413, IEEE Computer Society, 2010.

[18] CHANG, K. K.-W., AUSAVARUNGNIRUN, R., FALLIN, C., and MUTLU, O., "Hat: Heterogeneous adaptive throttling for on-chip networks," in *Proc. of the 24th Int'l Symp. on Computer Architecture and High Performance, SBAC-PAD'12*, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2012.

[19] CHAUDHURI, M., "Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 401–412, ACM, 2009.

[20] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., and SKADRON, K., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the 2010 IEEE Int'l. Symp. on Workload Characterization, IISWC'10*, (Washington, DC, USA), pp. 44–54, IEEE, 2009.

[21] CHOI, Y. and PINKSTON, T. M., "Evaluation of queue designs for true fully adaptive routers," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 64, no. 5, pp. 606–616, 2004.

[22] COCHRAN, R., HANKENDI, C., COSKUN, A. K., and REDA, S., "Pack & cap: adaptive dvfs and thread packing under power caps," in *Proc. of the 44th Int'l. Symp. on Microarchitecture, MICRO-44*, (New York, NY, USA), pp. 175–185, ACM, 2011.

[23] DALLY, W. and TOWLES, B., *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[24] DAS, R., MUTLU, O., MOSCIBRODA, T., and DAS, C. R., "Application-aware prioritization mechanisms for on-chip networks," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 280–291, ACM, 2009.

[25] DAS, R., MUTLU, O., MOSCIBRODA, T., and DAS, C. R., "Aérgia: exploiting packet latency slack in on-chip networks," in *Proc. of the 32nd annual Int'l. Symp. on Computer Architecture, ISCA-32*, (New York, NY, USA), pp. 106–116, ACM, 2010.

[26] DHODAPKAR, A. S. and SMITH, J. E., "Managing multi-configuration hardware via dynamic working set analysis," in *Proc. of the 24th annual Int'l. Symp. on Computer Architecture, ISCA-24*, (Washington, DC, USA), pp. 233–244, IEEE Computer Society, 2002.

[27] DIAMOS, G., KERR, A., YALAMANCHILI, S., and CLARK, N., "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proc. of the 19th Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'10*, (New York, NY, USA), ACM, 2010.

[28] DOBKIN, R. R., VISHNYAKOV, V., FRIEDMAN, E., and GINOSAR, R., "An asynchronous router for multiple service levels networks on chip," in *Proc. of the 11th IEEE Int'l Symp. on Asynchronous Circuits and Systems, ASYNC'05*, (Washington, DC, USA), pp. 44–53, IEEE Computer Society, 2005.

[29] DUATO, J., JOHNSON, I., FLICH, J., NAVEN, F., JAVIER, G. P., and FRINÓS, T. N., "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks," in *Proc. of the 11st Int'l. Symp. on High Performance Computer Architecture, HPCA-11*, (Washington, DC, USA), pp. 108–119, IEEE Computer Society, 2005.

[30] DUATO, J., YALAMANCHILI, S., and NI, L., *Interconnection Networks: An Engineering Approach*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1st ed., 1997.

[31] EVRIPIDOU, M., NICOPOULOS, C., SOTERIOU, V., and KIM, J., "Virtualizing virtual channels for increased network-on-chip robustness and upgradeability.," in *Proc. of IEEE Computer Society Annual Symp. on VLSI, ISVLSI'2012*, (Los Alamitos, CA, USA), pp. 21–26, IEEE Computer Society, 2012.

[32] FELTER, W., RAJAMANI, K., KELLER, T., and RUSU, C., "A performance-conserving approach for reducing peak power consumption in server systems," in *Proc. of the 19th Annual Int'l Conf. on Supercomputing*, (New York, NY, USA), pp. 293–302, ACM, 2005.

[33] GAUR, J., CHAUDHURI, M., and SUBRAMONEY, S., "Bypass and insertion algorithms for exclusive last-level caches," in *Proc. of the 33rd annual Int'l. Symp. on Computer Architecture, ISCA-33*, (New York, NY, USA), pp. 81–92, ACM, 2011.

[34] GHIASI, S., KELLER, T., and RAWSON, F., "Scheduling for heterogeneous processors in server systems," in *Proc. of the 2nd Conf. on Computing Frontiers, CF'05*, (New York, NY, USA), pp. 199–210, ACM, 2005.

[35] GOOSSENS, K., WIELAGE, P., PEETERS, A., and VAN MEERBERGEN, J., "Networks on silicon: Combining best-effort and guaranteed services," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'02*, (Washington, DC, USA), pp. 423–425, IEEE Computer Society, 2002.

[36] GOOSSENS, K., DIELISSEN, J., GANGWAL, O. P., PESTANA, S. G., RADULESCU, A., and RIJPKEMA, E., "A design flow for application-specific networks on chip with guaranteed performance to accelerate soc design and verification," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'05*, (Washington, DC, USA), pp. 1182–1187, IEEE Computer Society, 2005.

[37] GOOSSENS, K., DIELISSEN, J., and RADULESCU, A., "æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test*, vol. 22, pp. 414–421, Sept. 2005.

[38] GREWE, D., WANG, Z., and O'BOYLE, M., "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proc. of the 2013 Int'l. Symp. on Code Generation and Optimization, CGO-10*, (Los Alamitos, CA, USA), pp. 1–10, IEEE Computer Society, 2013.

[39] GROT, B., HESTNESS, J., KECKLER, S. W., and MUTLU, O., "Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service

guarantees.," in *Proc. of the 33rd annual Int'l. Symp. on Computer Architecture, ISCA-33*, (New York, NY, USA), pp. 401–412, ACM, 2011.

[40] GROT, B., KECKLER, S. W., and MUTLU, O., "Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip.," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 268–279, ACM, 2009.

[41] HANSON, H., FELTER, W., HUANG, W., LEFURGY, C., RAJAMANI, K., RAWSON, F., and SILVA, G., "Processor-memory power shifting for multi-core systems," in *Fourth Workshop on Energy-Efficient Design, WEED 2012, in conjunction with ISCA.*, 2012.

[42] HANSSON, A., SUBBURAMAN, M., and GOOSSENS, K., "aelite: a flit-synchronous network on chip with composable and predictable services," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'09*, (3001 Leuven, Belgium, Belgium), pp. 250–255, European Design and Automation Association, 2009.

[43] HARMANCI, M., ESCUDERO, N., LEBLEBICI, Y., and IENNE, P., "Quantitative modelling and comparison of communication schemes to guarantee quality-of-service in networks-on-chip," in *Proc. of 2005 IEEE Int'l Symp. on Circuits and Systems, ISCAS 2005*, vol. 2, (Piscataway, NJ, USA), pp. 1782–1785, IEEE Press, 2005.

[44] HP LABS, "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model." `http://www.hpl.hp.com/research/cacti/`.

[45] HPARCH RESEARCH GROUP, "MacSim." `http://code.google.com/p/macsim/`, 2012.

[46] HU, J. and MARCULESCU, R., "Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures.," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'03*, (Washington, DC, USA), pp. 688–693, IEEE Computer Society, 2003.

[47] HU, J. and MARCULESCU, R., "DyAD: smart routing for networks-on-chip.," in *Proc. of the 41st annual Design Automation Conference, DAC'04*, (New York, NY, USA), pp. 260–263, ACM, 2004.

[48] INTEL, "Haswell." `http://www.intel.com/content/www/us/en/processors/core/4th-gen-core-processor-family.html`.

[49] INTEL, "Ivy Bridge." `http://www.intel.com/content/www/us/en/silicon-innovations/intel-22nm-technology.html`.

[50] INTEL, "OpenSource HD Graphics Programmer's Reference Manual Volume 4 Part 1," https://01.org/linuxgraphics/sites/default/files/documentation/ivb_ihd_os_vol4_part1.pdf.

[51] INTEL, "OpenSource HD Graphics Programmer's Reference Manual Volume 4 Part 3," https://01.org/linuxgraphics/sites/default/files/documentation/ivb_ihd_os_vol4_part3.pdf.

[52] INTEL, "Sandy Bridge." http://software.intel.com/en-us/articles/sandy-bridge/.

[53] INTEL, "Turbo boost tecnhology." http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[54] INTEL, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2012.

[55] ISCI, C., BUYUKTOSUNOGLU, A., CHER, C.-Y., BOSE, P., and MARTONOSI, M., "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. of the 39th Int'l. Symp. on Microarchitecture, MICRO-39*, (Washington, DC, USA), pp. 347–358, IEEE Computer Society, 2006.

[56] JABLIN, T. B., PRABHU, P., JABLIN, J. A., JOHNSON, N. P., BEARD, S. R., and AUGUST, D. I., "Automatic cpu-gpu communication management and optimization," in *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation*, (New York, NY, USA), pp. 142–151, ACM, 2011.

[57] JALEEL, A., HASENPLAUGH, W., QURESHI, M., SEBOT, J., STEELY, JR., S., and EMER, J., "Adaptive insertion policies for managing shared caches," in *Proc. of the 17th Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'08*, (New York, NY, USA), pp. 208–219, ACM, 2008.

[58] JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., and EMER, J., "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of the 32nd annual Int'l. Symp. on Computer Architecture, ISCA-32*, (New York, NY, USA), pp. 60–71, ACM, 2010.

[59] JEONG, M. K., EREZ, M., SUDANTHI, C., and PAVER, N., "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proc. of the 49th annual Design Automation Conference, DAC'12*, (New York, NY, USA), pp. 850–855, ACM, 2012.

[60] JIMÉNEZ, D. A. and LIN, C., "Dynamic branch prediction with perceptrons," in *Proc. of the 7th Int'l. Symp. on High Performance Computer Architecture, HPCA-7*, (Washington, DC, USA), pp. 197–206, IEEE Computer Society, 2001.

[61] JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., GIL, M., FURSIN, G., and NAVARRO, N., "Predictive runtime code scheduling for heterogeneous architectures," in *Proc. of the 4th Int'l Conf. on High Perf. Embedded Architecture and cCompilers, HiPEAC'09*, (Berlin, Heidelberg), pp. 19–33, Springer-Verlag, 2009.

[62] J.NESBIT, K. and E.SMITH, J., "Data cache prefetching using a global history buffer," in *Proc. of the 10th Int'l. Symp. on High Performance Computer Architecture, HPCA-10*, (Washington, DC, USA), pp. 96–105, IEEE Computer Society, 2004.

[63] JOHNSON, T. L. and MEI W. HWU, W., "Run-time adaptive cache hierarchy management via reference analysis.," in *Proc. of the 19th annual Int'l. Symp. on Computer Architecture, ISCA-19*, (New York, NY, USA), pp. 315–326, ACM, 1997.

[64] KHARBUTLI, M. and SOLIHIN, Y., "Counter-based cache replacement and bypassing algorithms.," *IEEE Trans. Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[65] KIM, S., CHANDRA, D., and SOLIHIN, Y., "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. of the 13rd Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'04*, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[66] KIM, Y., HAN, D., MUTLU, O., and HARCHOL-BALTER, M., "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *Proc. of the 16th Int'l. Symp. on High Performance Computer Architecture, HPCA-16*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[67] KIM, Y., PAPAMICHAEL, M., MUTLU, O., and HARCHOL-BALTER, M., "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. of the 43rd Int'l. Symp. on Microarchitecture, MICRO-43*, (Washington, DC, USA), pp. 65–76, IEEE, 2010.

[68] LAI, M., WANG, Z., GAO, L., LU, H., and DAI, K., "A dynamically-allocated virtual channel architecture with congestion awareness for on-chip routers," in *Proc. of the 45th annual Design Automation Conference, DAC'08*, (New York, NY, USA), pp. 630–633, ACM, 2008.

[69] LEE, H. and TYSON, G., "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *Proc. of the 2000 Int'l Con.f on Compilers, architecture, and synthesis for embedded systems, CASES'00*, (New York, NY, USA), pp. 120–127, ACM, 2000.

[70] LEE, H.-H. S. and BALLAPURAM, C. S., "Energy efficient d-tlb and data cache using semantic-aware multilateral partitioning," in *Proc. of the 2003*

Int'l Symp. on Low Power Electronics and Design, ISLPED'03, (New York, NY, USA), pp. 306–311, ACM, 2003.

[71] LEE, J. W., NG, M. C., and ASANOVIC, K., "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *Proc. of the 30th annual Int'l. Symp. on Computer Architecture, ISCA-30*, (Washington, DC, USA), pp. 89–100, IEEE Computer Society, 2008.

[72] LEE, J. and KIM, H., "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture.," in *Proc. of the 18th Int'l. Symp. on High Performance Computer Architecture, HPCA-18*, (Washington, DC, USA), pp. 91–102, IEEE Computer Society, 2012.

[73] LEE, J., LI, S., KIM, H., and YALAMANCHILI, S., "Adaptive virtual channel partitioning for network-on-chip in heterogeneous architecture," *ACM Trans. Design Autom. Electr. Syst. (TODAES)*, vol. 1, no. PrePrints, 2013.

[74] LEE, J., SATHISHA, V., SCHULTE, M., COMPTON, K., and KIM, N. S., "Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling," in *Proc. of the 20th Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'11*, (Washington, DC, USA), pp. 111–120, IEEE Computer Society, 2011.

[75] LEE, S. and SAKURAI, T., "Run-time voltage hopping for low-power real-time systems," in *Proc. of the 37th annual Design Automation Conference, DAC'00*, (New York, NY, USA), pp. 806–809, ACM, 2000.

[76] LEUNG, L.-F. and TSUI, C.-Y., "Optimal link scheduling on improving best-effort and guaranteed services performance in network-on-chip systems," in *Proc. of the 43rd annual Design Automation Conference, DAC'06*, (New York, NY, USA), pp. 833–838, ACM, 2006.

[77] LI, J. and MARTÍNEZ, J. F., "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *Proc. of the 12nd Int'l. Symp. on High Performance Computer Architecture, HPCA-12*, (Washington, DC, USA), pp. 77–87, IEEE Computer Society, 2006.

[78] LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., and JOUPPI, N. P., "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, pp. 469–480, ACM, 2009.

[79] LIANG, J., LAFFELY, A., SRINIVASAN, S., and TESSIER, R., "An architecture and compiler for scalable on-chip communication," *IEEE Trans. VLSI System (TVLSI)*, vol. 12, no. 7, pp. 711–726, 2004.

[80] LIANG, J., SWAMINATHAN, S., and TESSIER, R., "asoc: A scalable, single-chip communications architecture," in *Proc. of the 9th Int'l. Conf. on Parallel*

Architectures and Compilation Techniques, PACT'00, (Washington, DC, USA), pp. 37–46, IEEE Computer Society, 2000.

[81] LINDERMAN, M. D., COLLINS, J. D., WANG, H., and MENG, T. H., "Merge: a programming model for heterogeneous multi-core systems," in *Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XIII*, (New York, NY, USA), pp. 287–296, ACM, 2008.

[82] LIU, C., LI, J., HUANG, W., RUBIO, J., SPEIGHT, E., and LIN, X., "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proc. of the 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'12*, (New York, NY, USA), pp. 23–32, ACM, 2012.

[83] LIU, F., JIANG, X., and SOLIHIN, Y., "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *Proc. of the 16th Int'l. Symp. on High Performance Computer Architecture, HPCA-16*, (Washington, DC, USA), pp. 1 –12, IEEE Computer Society, 2010.

[84] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., and HAZELWOOD, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of the ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, (New York, NY, USA), pp. 190–200, ACM, 2005. `http://www.pintool.org`.

[85] LUK, C.-K., HONG, S., and KIM, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 45–55, ACM, 2009.

[86] MA, K., LI, X., CHEN, M., and WANG, X., "Scalable power control for many-core architectures running multi-threaded applications," in *Proc. of the 33rd annual Int'l. Symp. on Computer Architecture, ISCA-33*, (New York, NY, USA), pp. 449–460, ACM, 2011.

[87] MA, K. and WANG, X., "Pgcapping: exploiting power gating for power capping and core lifetime balancing in cmps," in *Proc. of the 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'12*, (New York, NY, USA), pp. 13–22, ACM, 2012.

[88] MA, K., WANG, X., and WANG, Y., "DPPC: Dynamic power partitioning and capping in chip multiprocessors," in *Proc. of the 2011 IEEE 29th Int'l Conf. on Computer Design, ICCD 2011*, (Washington, DC, USA), pp. 39–44, IEEE Computer Society, 2011.

[89] MA, S., JERGER, N. D. E., and WANG, Z., "Dbar: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip.," in *Proc. of the 33rd annual Int'l. Symp. on Computer Architecture, ISCA-33*, (New York, NY, USA), pp. 413–424, ACM, 2011.

[90] MARCULESCU, R., OGRAS, U., PEH, L.-S., JERGER, N., and HOSKOTE, Y., "Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 1, pp. 3–21, 2009.

[91] MARESCAUX, T. and CORPORAAL, H., "Introducing the supergt network-on-chip; supergt qos: more than just gt," in *Proc. of the 44th annual Design Automation Conference, DAC'07*, (New York, NY, USA), pp. 116–121, ACM, 2007.

[92] MENG, K., JOSEPH, R., DICK, R. P., and SHANG, L., "Multi-optimization power management for chip multiprocessors," in *Proc. of the 17th Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'08*, (New York, NY, USA), pp. 177–186, ACM, 2008.

[93] MILLBERG, M., NILSSON, E., THID, R., and JANTSCH, A., "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'04*, (Washington, DC, USA), pp. 890–895, IEEE Computer Society, 2004.

[94] MISHRA, A. K., SRIKANTAIAH, S., KANDEMIR, M., and DAS, C. R., "Cpm in cmps: Coordinated power management in chip-multiprocessors," in *Proc. of the 2010 ACM/IEEE Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis, SC'10*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[95] MISHRA, A. K., VIJAYKRISHNAN, N., and DAS, C. R., "A case for heterogeneous on-chip interconnects for CMPs," in *Proc. of the 33rd annual Int'l. Symp. on Computer Architecture, ISCA-33*, (New York, NY, USA), pp. 389–400, ACM, 2011.

[96] MORETÓ, M., CAZORLA, F. J., RAMÍREZ, A., and VALERO, M., "MLP-aware dynamic cache partitioning.," in *Proc. of the Int'l Conf. on High-Performance Embedded Architectures and Compilers, HiPEAC'08*, vol. 4917, (Berlin, Heidelberg), pp. 337–352, Springer-Verlag, 2008.

[97] MUTLU, O. and MOSCIBRODA, T., "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. of the 40th Int'l. Symp. on Microarchitecture, MICRO-40*, (Washington, DC, USA), pp. 146–160, IEEE Computer Society, 2007.

[98] MUTLU, O. and MOSCIBRODA, T., "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proc. of the 30th annual Int'l. Symp. on Computer Architecture, ISCA-30*, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2008.

[99] NESBIT, K. J., DHODAPKAR, A. S., and SMITH, J. E., "AC/DC: An adaptive data cache prefetcher," in *Proc. of the 13rd Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'04*, (Washington, DC, USA), pp. 135–145, IEEE Computer Society, 2004.

[100] NICOPOULOS, C. A., PARK, D., KIM, J., VIJAYKRISHNAN, N., YOUSIF, M. S., and DAS, C. R., "Vichar: A dynamic virtual channel regulator for network-on-chip routers," in *Proc. of the 39th Int'l. Symp. on Microarchitecture, MICRO-39*, (Washington, DC, USA), pp. 333–346, IEEE Computer Society, 2006.

[101] NILSSON, E., MILLBERG, M., ÖBERG, J., and JANTSCH, A., "Load distribution with the proximity congestion awareness in a network on chip," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'03*, (Washington, DC, USA), pp. 11126–11127, IEEE Computer Society, 2003.

[102] NVIDIA, "CUDA SDK 4.2." `https://developer.nvidia.com/cuda-toolkit-42-archive`.

[103] NVIDIA, "Fermi: Nvidia's next generation cuda compute architecture." `http://www.nvidia.com/fermi`.

[104] NVIDIA, "Geforce 8800 graphics processors." `http://www.nvidia.com/page/geforce_8800.html`.

[105] NVIDIA, "Kepler compute architecture." http://www.nvidia.com/object/nvidia-kepler.html.

[106] NVIDIA, "Project denver." `http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/`.

[107] NVIDIA, "Tegra APX Application Processors." `http://www.nvidia.com/object/product_tegra_apx_us.html`.

[108] NVIDIA, *PTX: Parallel Thread Execution ISA Version 2.3*, 2011.

[109] NVIDIA Corporation, *CUDA Programming Guide, V4.0*.

[110] OGRAS, U. Y. and MARCULESCU, R., "Analysis and optimization of prediction-based flow control in networks-on-chip," *ACM Trans. Design Autom. Electr. Syst. (TODAES)*, vol. 13, no. 1, 2008.

[111] OPENCL, "The open standard for parallel programming of heterogeneous systems." `http://www.khronos.org/opencl`.

[112] PARK, J., SHIN, D., CHANG, N., and PEDRAM, M., "Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors," in *Proc. of the 2010 Int'l Symp. on Low Power Electronics and Design, ISPLED'10*, (New York, NY, USA), pp. 419–424, ACM, 2010.

[113] PATIL, H., COHN, R., CHARNEY, M., KAPOOR, R., SUN, A., and KARUNANIDHI, A., "Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation," in *Proc. of the 37th Int'l. Symp. on Microarchitecture, MICRO-37*, (Washington, DC, USA), pp. 81–92, IEEE Computer Society, 2004.

[114] PAUL, I., MANNE, S., ARORA, M., BIRCHER, W. L., and YALAMANCHILI, S., "Cooperative boosting: Needy versus greedy power management," in *Proc. of the 35th annual Int'l. Symp. on Computer Architecture, ISCA-35*, (New York, NY, USA), pp. 1–12, ACM, 2013.

[115] PHOTHILIMTHANA, P. M., ANSEL, J., RAGAN-KELLEY, J., and AMARASINGHE, S., "Portable performance on heterogeneous architectures," in *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XVIII*, (New York, NY, USA), pp. 431–444, ACM, 2013.

[116] POWELL, M., YANG, S.-H., FALSAFI, B., ROY, K., and VIJAYKUMAR, T. N., "Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. of the 2000 Int'l Symp. on Low Power Electronics and Design, ISPLED'00*, (New York, NY, USA), pp. 90–95, ACM, 2000.

[117] QUALCOMM, "Snapdragon s4 mobile processors." `https://developer.qualcomm.com/download/qusnapdragons4whitepaperfnlrev6.pdf`.

[118] QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., and EMER, J., "Adaptive insertion policies for high performance caching," in *Proc. of the 29th annual Int'l. Symp. on Computer Architecture, ISCA-29*, (New York, NY, USA), pp. 381–391, ACM, 2007.

[119] QURESHI, M. K., LYNCH, D. N., MUTLU, O., and PATT, Y. N., "A case for MLP-aware cache replacement," in *Proc. of the 28th annual Int'l. Symp. on Computer Architecture, ISCA-28*, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.

[120] QURESHI, M. K. and PATT, Y. N., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of the 39th Int'l. Symp. on Microarchitecture, MICRO-39*, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[121] RAJAMANI, K., RAWSON, F., WARE, M., HANSON, H., CARTER, J., ROSEDAHL, T., GEISSLER, A., SILVA, G., and HUA, H., "Power-performance

management on an ibm power7 server," in *Proc. of the 2010 Int'l Symp. on Low Power Electronics and Design, ISPLED'10*, (New York, NY, USA), pp. 201–206, ACM, 2010.

[122] RANGANATHAN, P., ADVE, S., and JOUPPI, N. P., "Reconfigurable caches and their application to media processing," in *Proc. of the 22nd annual Int'l. Symp. on Computer Architecture, ISCA-22*, (New York, NY, USA), pp. 214–224, ACM, 2000.

[123] RIJPKEMA, E., GOOSSENS, K. G. W., RADULESCU, A., DIELISSEN, J., VAN MEERBERGEN, J., WIELAGE, P., and WATERLANDER, E., "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'03*, (Washington, DC, USA), pp. 10350–10355, IEEE Computer Society, 2003.

[124] ROTEM, E., NAVEH, A., ANANTHAKRISHNAN, A., WEISSMANN, E., and RAJWAN, D., "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, pp. 20–27, Mar. 2012.

[125] SASAKI, H., IMAMURA, S., and INOUE, K., "Coordinated power-performance optimization in manycores," in *Proc. of the 22nd Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'13*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2013.

[126] SHARIFI, A., MISHRA, A. K., SRIKANTAIAH, S., KANDEMIR, M., and DAS, C. R., "Pepon: performance-aware hierarchical power budgeting for noc based multicores," in *Proc. of the 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'12*, (New York, NY, USA), pp. 65–74, ACM, 2012.

[127] SNAVELY, A. and TULLSEN, D. M., "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. of the 9th Int'l. conference on Architectural support for programming languages and operating systems, ASPLOS-IV*, (New York, NY, USA), pp. 234–244, ACM, 2000.

[128] SRIKANTAIAH, S., KANDEMIR, M., and WANG, Q., "Sharp control: Controlled shared cache management in chip multiprocessors," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 517 –528, ACM, 2009.

[129] SRINATH, S., MUTLU, O., KIM, H., and PATT, Y. N., "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. of the 13rd Int'l. Symp. on High Performance Computer Architecture, HPCA-13*, (Washington, DC, USA), pp. 63–74, IEEE Computer Society, 2007.

[130] STEFAN, R., MOLNOS, A., and GOOSSENS, K., "daelite: A tdm noc supporting qos, multicast, and fast connection set-up," *IEEE Transactions on Computers (TC)*, vol. 99, no. PrePrints, 2012.

[131] SUH, G. E., RUDOLPH, L., and DEVADAS, S., "Dynamic partitioning of shared cache memory," *Journal of Supercomputing*, vol. 28, pp. 7–26, Apr. 2004.

[132] SUH, G., DEVADAS, S., and RUDOLPH, L., "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. of the 8th Int'l. Symp. on High Performance Computer Architecture, HPCA-8*, (Washington, DC, USA), pp. 117–128, IEEE Computer Society, 2002.

[133] SUO, G., YANG, X., LIU, G., WU, J., ZENG, K., ZHANG, B., and LIN, Y., "IPC-based cache partitioning: An IPC-oriented dynamic shared cache partitioning mechanism," in *Proc. of the Int'l Conf. on Convergence and Hybrid Information Technology, ICHIT'08*, (Washington, DC, USA), pp. 399–406, IEEE Computer Society, 2008.

[134] TAMIR, Y. and FRAZIER, G. L., "Dynamically-Allocated Multi-Queue buffers for VLSI communication switches.," *IEEE Trans. on Computers (TC)*, vol. 41, no. 6, pp. 725–737, 1992.

[135] TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., and AGARWAL, A., "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, Mar. 2002.

[136] TEMAM, O. and JEGOU, Y., "Using virtual lines to enhance locality exploitation," in *Proc. of the 8th Int'l Conf. on Supercomputing, ICS-8*, (New York, NY, USA), pp. 344–352, ACM, 1994.

[137] THE IMPACT RESEARCH GROUP, UIUC, "Parboil benchmark suite." `http://impact.crhc.illinois.edu/parboil.php`.

[138] TRIVINÕ, F., SÁNCHEZ, J. L., ALFARO, F. J., and FLICH, J., "Exploring noc virtualization alternatives in cmps," in *Proc. of the 2012 20th Euromicro Int'l Conf. on Parallel, Distributed and Network-based Processing, PDP'12*, (Washington, DC, USA), pp. 473–482, IEEE Computer Society, 2012.

[139] TYSON, G., FARRENS, M., MATTHEWS, J., and PLESZKUN, A. R., "A modified approach to data cache management," in *Proc. of the 28th Int'l. Symp. on Microarchitecture, MICRO-28*, (Los Alamitos, CA, USA), pp. 93–103, IEEE Computer Society Press, 1995.

[140] VAN DEN BRAND, J. W., CIORDAS, C., GOOSSENS, K., and BASTEN, T., "Congestion-controlled best-effort communication for networks-on-chip,"

in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'07*, (San Jose, CA, USA), pp. 948–953, EDA Consortium, 2007.

[141] VARATKAR, G. and MARCULESCU, R., "Traffic analysis for on-chip networks design of multimedia applications," in *Proc. of the 39th annual Design Automation Conference, DAC'02*, (New York, NY, USA), pp. 795–800, ACM, 2002.

[142] WANG, H., SATHISH, V., SINGH, R., SCHULTE, M. J., and KIM, N. S., "Workload and power budget partitioning for single-chip heterogeneous processors," in *Proc. of the 21st Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'12*, (New York, NY, USA), pp. 401–410, ACM, 2012.

[143] WANG, J., RUBIN, N., WU, H., and YALAMANCHILI, S., "Accelerating simulation of agent-based models on heterogeneous architectures," in *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6*, (New York, NY, USA), pp. 108–119, ACM, 2013.

[144] WANG, Y., MA, K., and WANG, X., "Temperature-constrained power control for chip multiprocessors with online model estimation," in *Proc. of the 31st annual Int'l. Symp. on Computer Architecture, ISCA-31*, (New York, NY, USA), pp. 314–324, ACM, 2009.

[145] WARE, M. S., RAJAMANI, K., FLOYD, M. S., BROCK, B., RUBIO, J. C., III, F. L. R., and CARTER, J. B., "Architecting for power management: The power7 approach," in *Proc. of the 16th Int'l. Symp. on High Performance Computer Architecture, HPCA-16*, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[146] WEBER, W.-D., CHOU, J., SWARBRICK, I., and WINGARD, D., "A quality-of-service mechanism for interconnection networks in system-on-chips," in *Proc. of Design, Automation, and Test in Europe Conference and Exhibition, DATE'05*, (Washington, DC, USA), pp. 1232–1237, IEEE Computer Society, 2005.

[147] WINTER, J. A., ALBONESI, D. H., and SHOEMAKER, C. A., "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proc. of the 19th Int'l. Conf. on Parallel Architectures and Compilation Techniques, PACT'10*, (New York, NY, USA), pp. 29–40, ACM, 2010.

[148] WU, C.-J., JALEEL, A., HASENPLAUGH, W., MARTONOSI, M., STEELY, JR., S. C., and EMER, J., "Ship: signature-based hit predictor for high performance caching," in *Proc. of the 44th Int'l. Symp. on Microarchitecture, MICRO-44*, (New York, NY, USA), pp. 430–441, ACM, 2011.

[149] WU, C.-J., JALEEL, A., MARTONOSI, M., STEELY, JR., S. C., and EMER, J., "Pacman: prefetch-aware cache management for high performance caching," in *Proc. of the 44th Int'l. Symp. on Microarchitecture, MICRO-44*, (New York, NY, USA), pp. 442–453, ACM, 2011.

[150] XIE, Y. and LOH, G. H., "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. of the 31st annual Int'l. Symp. on Computer Architecture, ISCA-31*, (New York, NY, USA), pp. 174–183, ACM, 2009.

[151] XIE, Y. and LOH, G. H., "Scalable shared-cache management by containing thrashing workloads.," in *Proc. of the Int'l Conf. on High-Performance Embedded Architectures and Compilers, HiPEAC'10*, vol. 5952, (Berlin, Heidelberg), pp. 262–276, Springer-Verlag, 2010.

[152] YANG, Y., XIANG, P., MANTOR, M., and ZHOU, H., "CPU-assisted GPGPU on fused CPU-GPU architectures," in *Proc. of the 18th Int'l. Symp. on High Performance Computer Architecture, HPCA-18*, (Washington, DC, USA), pp. 103–114, IEEE Computer Society, 2012.

[153] YU, C. and PETROV, P., "Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms," in *Proc. of the 47th annual Design Automation Conference, DAC'10*, (New York, NY, USA), pp. 132–137, ACM, 2010.

[154] YUAN, G. L., BAKHODA, A., and AAMODT, T. M., "Complexity effective memory access scheduling for many-core accelerator architectures," in *Proc. of the 42nd Int'l. Symp. on Microarchitecture, MICRO-42*, (New York, NY, USA), pp. 34–44, ACM, 2009.