Performance-Correctness Challenges in Emerging Heterogeneous Multicore Processors

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Vineeth Thamarassery Mekkat

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Antonia Zhai

December, 2013

Acknowledgements

Although graduate school is, for the most part, a lonely journey, there are several people who have been instrumental in getting me to this point. Here, I would like to acknowledge those without whom I wouldn't be writing this.

First among equals, I would like to express my sincere gratitude to my advisor Professor Antonia Zhai. Thank you for giving me this wonderful opportunity to pursue topics that interest me. Thank you also for teaching me, from the basics, about conducting quality research, paper writing, and presentation skills.

Next, I would like to thank Professors Mats Heimdahl, David Lilja, and Pen-Chung Yew for serving on my committee. Their comments have greatly helped to improve the final version of my dissertation. Professor Yew's unbridled enthusiasm has been a constant inspiration to me. Thank you for sharing your energy! I would also like to thank Professor Wei-Chung Hsu for giving me the first opportunity to do research at Minnesota, and for re-affirming my interest in computer architecture through his exceptional lectures.

A fellow lab-mate who inspires and supports you is one of the best things that can happen to a graduate student. I was lucky to have three of them. Thank you for the wonderful company: Anup Holey, Ragavendra Natarajan, and Jieming Yin. Thanks are also due to other colleagues: Venkatesan Packirisamy, Guojin He, Yangchun Luo, and Sanyam Mehta.

Several fellow graduate students at the U have been a significant part of my life over the last five years. Many of them have become a friend-for-life to me. The list is long and

I dare not write it all. Besides, they know who they are.

Finally, and most importantly, I would like to express my deepest love and gratitude to my family: my wife Vinitha, our parents and siblings, and our little girl Medha. They have always supported me in my endeavors, and graduate school was no exception. Thank you, Vinitha, for always having the bag packed ready for any crazy plans that I might come up with.

Abstract

We are witnessing a tremendous amount of change in the design of the modern microprocessor. With dozens of CPU cores on-chip recent multicore processors, the search for thread-level parallelism (TLP) is more significant than ever. In parallel, a very different processor architecture has emerged that aims to extract parallelism at an entirely different scale. Originally proposed for accelerating graphical applications, graphics processing units (GPU) are increasingly being employed to improve the performance of general purpose applications.

Advances in process technology and the need for energy efficiency has brought together CPU and GPU cores onto the same die to form on-chip heterogeneous multicore processors. Several industrial designs that follow this philosophy are already part of mainstream computing. The presence of diverse cores on the same die, sharing on-chip resources, presents several challenges in achieving an efficient design. In particular, this thesis addresses two key aspects in designing efficient heterogeneous multicore processors: performance and correctness.

Performance is of paramount concern in the design of a microprocessor, and the last-level cache (LLC) is a critical on-chip component from this perspective. Several techniques have been proposed to efficiently share the LLC among on-chip cores. However, when the on-chip cores show significant diversity in their memory access characteristics, currently proposed techniques face severe challenge in attaining effective LLC sharing. In the first part of this thesis, we address this problem and propose a new policy that improves the management of shared LLC, in the presence of heterogeneous workloads, in terms of performance as well as energy efficiency.

Execution correctness is an important concern in the quest for the extraction of parallelism. Concurrency bugs, such as data race conditions, are severe impediments to the

effectiveness of parallel computing. Although, several techniques have been proposed to identify and rectify data race conditions, their implementation faces several challenges. While software-based mechanisms are cheaper to implement, they inflict severe performance overhead on the monitored application. The high performance of hardware-based mechanisms, on the other hand, comes at the expense of additional hardware support and increased implementation cost. In the second part of this thesis, we propose a technique to utilize available on-chip GPU cores to perform efficient data race detection for the applications executing on the CPU cores.

Overall, with these two techniques, we address two critical challenges in the design of emerging heterogeneous multicore processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The landscape of parallel computing is changing rapidly. It is changing in terms of the processor architecture as well as target applications. On the architecture side, multi-core processors are paving way for many-core processors to extract even higher amount of thread-level parallelism (TLP). Additionally, processor architectures that specialize in accelerating specific applications, such as the graphics processing units (GPU), have gained popularity. Advances in programming support such as CUDA and OpenCL have enabled these accelerator cores to be employed to speed up generic applications. On the other hand, new applications that are of interest to the parallel computing community are also emerging. They include highly parallel applications such as data-parallel applications (scientific applications ported to GPU), gaming/graphics applications, and cloud/data-warehouse applications.

For a long time, the quest for performance improvement was fuelled by the increasing operating frequency of microprocessors. However, ever since this progress has been dampened by thermal and power considerations, processor designers have turned to the extraction of TLP. This has been achieved by increasing the number of threads available on-chip and actively parallelizing applications to make effective use of these threads. Recently, the number of cores/threads available on-chip have come to be counted in dozens,

turning multicore processors into many-core processors.

In parallel to this development, an entirely different processor architecture has been emerging. Originally aimed at improving the performance of graphics applications, thanks to programming models such as CUDA and OpenCL, the GPU is being increasingly applied to improve the performance of general purpose and scientific applications.

The natural next step in the evolution of the microprocessor is the integration of these diverse cores onto the same die. Heterogeneous multicore processors, containing both CPU and GPU cores, have become a reality with processors such as Intel Sandy Bridge and AMD Fusion APU. Although this development offers opportunities to improve the performance of diverse applications efficiently, it introduces significant challenges in effectively sharing on-chip resources such as the last-level cache (LLC). Given the differences in the cache sensitivity and memory access rate of the constituent cores, existing cache management policies face difficulty in effectively sharing the LLC among the diverse cores in a heterogeneous multicore processor. However, efficient sharing of on-chip resources such as LLC is paramount to extracting performance in these emerging architectures.

Availability of increased on-chip parallelism introduces further challenges to the extraction of performance. *Correctness* of parallel execution is one such challenging aspect faced by applications that attempt to extract this parallelism. Concurrency bugs, including data race conditions, are one such example of the correctness issue that parallel applications face. While several software and hardware techniques have been proposed to detect such concurrency bugs, they face many shortcomings. While software techniques incur significant performance impact, hardware techniques introduce substantial on-chip modifications that increase the cost of implementation. The ability to utilize available on-chip resources to perform concurrency bug detection would help improve the correctness of the emerging heterogeneous multicore processors.

1.1 Dissertation Objective and Summary

This dissertation explores the challenges in designing the emerging heterogeneous multicore processors. The two critical aspects of parallel computing we focus on in this dissertation are: *performance* and *correctness*. In the presence of diverse cores sharing on-chip resources, extracting *performance* becomes a serious challenge. In addition to the extraction of performance, ensuring *correctness* is of critical importance for the effectiveness of these multicore processors.

In the first half of this dissertation, we propose and evaluate a new cache management policy that targets the shared LLC in a heterogeneous multicore processor. The policy is motivated by the observation that GPU applications are, in general, cache insensitive and that the amount of available TLP in GPU applications is a good indicator of their cache sensitivity. We utilize TLP as a metric to decide when to bypass the shared LLC for the GPU memory accesses so that the cache space is allocated to a more cache sensitive CPU application.

In the second half of this dissertation, we explore techniques to utilize available on-chip resources in a heterogeneous multicore processor to improve the execution correctness of the parallel programs executing on the CPU cores. We design and evaluate a mechanism that achieves the performance of hardware-based data race detection mechanisms at nearly the cost of software-based mechanisms.

1.2 Related Work

This dissertation closely relates to the following areas of computer architecture research: microprocessor architecture, performance characterization, cache management policies and concurrency bug detection.

1.2.1 Architectures & Applications

Several works have studied the move from single-core to multicore designs for microprocessors. What started as simultaneous multi-threading (SMT) [1] support soon extended to full-fledged multicore designs [2]. The count of processing elements in current multicore designs have reached dozens, earning them the title of many-core processors [3, 4]. In addition to such homogeneous multicore processors, heterogeneous multicore processors have also been studied [5]. In parallel to these developments, accelerator processors that improve the performance of specific applications have emerged. A modern graphics processing unit (GPU) [6, 7] that can support thousands of parallel execution threads belongs to this category. Processors that integrate CPU and GPU cores on the same die have emerged [8, 9] and are appearing to become the standard for processor design for the foreseeable future.

These dramatic advancements in processor design has, for the most part, been fuelled by the demand for performance improvement in the diverse applications that have come into spotlight recently. Many of these emerging applications have come into being as techniques to process the enormous amount of data that we generate. Several works have studied these emerging applications. Intel prioritizes three classes of applications among them: recognition, mining and synthesis (RMS) [10]. We have conducted detailed performance characterization of data mining applications [11, 12] in order to better understand the architectural and compiler optimizations required for improving the performance of these emerging applications. In their detailed discussion on the emerging landscape of parallel computing research, Asanovic et al. [13] have made several observations and recommendations to improve parallel computing in the many-core era.

1.2.2 Cache Management Policies

Efficient management of the shared LLC is critical to the extraction of performance, and several policies have been proposed. In general, they can be divided into cache partitioning techniques and cache replacement policies. Cache partitioning techniques [14, 15, 16, 17,

18, 19, 20] aim at reaching an effective partitioning of the LLC among applications so that the performance target is met. The performance target could be overall performance, overall throughput or other criterion such as quality of service (QoS). These techniques could make static partitioning decisions [14] or dynamically adapt the partitioning based on application behavior [15, 16, 17, 18, 19, 20]. Cache replacement policies [21, 22, 23, 24], on the other hand, aim to identify the appropriate position to insert a new cache block, and to identify the right victim for replacement, to achieve their performance goal. They base their cache management decisions on application behavior [21], re-reference interval (reuse) characteristics [22], or hierarchy awareness about other levels of cache [24]. However, these policies face difficulty in providing efficient management of the shared LLC in the presence of diverse on-chip cores.

1.2.3 Concurrency Bug Detection

Several concurrency bug detection techniques have been proposed to improve the correctness of parallel execution. They can broadly be divided into hardware [25, 26, 27] or software [28, 29, 30, 31] schemes. While hardware schemes offer high performance data race detection at the cost of implementation overhead, software schemes offer cheaper data race detection capability, albeit at the cost of performance. Techniques [32, 33] have been proposed to achieve low performance overhead at the cost of minimal hardware modifications. However, these techniques usually trade-off on accuracy or coverage to achieve the performance goal. Techniques that perform concurrency bug detection with low performance overhead, at the cost of minimal implementation overhead, is not only desirable, but critical to the effectiveness of increased extraction of parallelism.

1.3 Dissertation Contributions

This dissertation makes several key contributions in improving the understanding of the changing landscape of parallel computing and the emerging heterogeneous multicore processor architectures.

1. This thesis investigates the shortcomings of existing cache management policies for heterogeneous multicore processors. It identifies cache sensitivity and TLP characteristics of GPU applications as metrics that could be utilized to improve the cache management policy.

2. This thesis proposes, implements and evaluates a new cache management policy (HeLM) for the LLC shared by the diverse cores in a heterogeneous multicore processor. HeLM outperforms currently proposed policies in terms of both performance as well as energy efficiency.

3. This thesis studies the increasing importance of providing efficient support for ensuring program correctness in the emerging many-core processors. It also identifies the shortcomings of existing hardware-based and software-based concurrency bug detection tools.

4. This thesis proposes, implements and evaluates a concurrency bug detection tool (GUARD) that utilizes available GPU cores on-chip a heterogeneous multicore processor to efficiently perform data race detection of the parallel application executing on the CPU cores. We also address two key challenges in data race detection schemes: accuracy and scalability.

1.4 Dissertation Outline

The organization of the rest of this dissertation is outlined as follows:

- Chapter 2 discusses the changing landscape of computing, focussing on the evolution of the microprocessor architecture. In particular, it focusses on the emerging heterogeneous multicore processors, and identifies performance and correctness as two key challenges to be addressed.

- Chapter 3 conducts a detailed study of the memory access behavior of CPU and GPU applications, and identifies characteristics that could be utilized to improve cache sharing in heterogeneous multicore processors.

- Chapter 4 presents a new cache management policy for the shared LLC in a heterogeneous multicore processor.

- Chapter 5 addresses the correctness aspect of parallel computing and presents a technique that utilizes available on-chip GPU cores to perform efficient concurrency bug detection for parallel applications executing on the CPU cores.

- Chapter 6 further improves the data race detector by tackling its accuracy and scalability issues.

- Chapter 7 presents our conclusion and discusses directions for future work.

Chapter 2

Emerging Landscape of Computing

The landscape of computing has witnessed drastic changes in the last two decades. The changes have been apparent in the applications being executed, as well as the computational infrastructure being utilized for their execution. This development brings out several challenges in designing efficient computational systems. In this chapter, we summarize the developments in the architecture of microprocessors in the last two decades. We then focus the rest of this thesis in addressing the challenges in designing the emerging microprocessor architectures.

2.1 Microprocessor Architecture

Microprocessor architecture has undergone a tremendous change in the last two decades. In this chapter, we will briefly review the evolution of microprocessor during this period and explore in detail the emerging microprocessor architectures.

2.1.1 Multicore Processors

In addition to the architectural techniques employed to improve performance by extracting higher levels of instruction-level parallelism (ILP), such as branch prediction, out-of-order

execution, prefetching, etc., improvements in process technology has been aiding progress in this direction. This development has improved the operating frequency which has played a significant role in improving performance. However, round about the turn of this century, power and thermal considerations have dampened the growth in operating frequency. Since then, extraction of TLP by increasing the count of on-chip cores/threads has been the preferred way for designers.

Duplicating sections of pipeline to support execution of multiple threads simultaneously was the first instance of on-chip parallelism. Simultaneous Multi-Threading (SMT) [1] supports the issue of multiple independent threads by utilizing a superscalar processor's different functional units. With advances in hardware technology, duplication of the entire processing core became commonplace. Earliest multicore processors included designs such as Power4 [2]. These processors aimed at improving the performance of parallel applications by reducing the cost of off-chip communication that is a characteristic of the multiprocessing systems prevalent at that time.

The very first multicore processor proposals from academia and industry consisted mostly of homogeneous processing elements. However, there also were proposals to include asymmetric processing elements in the multicore design. Such heterogeneous multicore processors aimed at improving the energy efficiency by employing a processing core of suitable size for a particular application or performance target. Most notable among these proposals was the work by Kumar et al. [5].

2.1.2 Many-Core Processors

In recent years, the number of processing elements on the multicore processor has reached many dozens leading them to be termed as many-core processors. Examples include Tilera [3] and Intel Many Integrated Core (MIC) [4]. Utilizing simpler and power efficient constituent cores, they aim to help general purpose applications extract higher levels of

TLP. Increased on-chip memory and bandwidth aid these many-core processors in achieving this target. However, sharing of on-chip resources and scalability of factors such as cache coherence protocols become a significant challenge in realizing the full potential of these designs.

2.1.3 Graphics Processing Unit

In parallel to these developments in the architecture of general purpose CPU, another processor architecture has emerged. A modern graphics processing unit (GPU) supports thousands of parallel execution threads. Such support is provided by an array of computing cores referred to as the Streaming Multiprocessors (SM). Each SM consists of an array of simple in-order cores referred to as the Streaming Processors (SP). SPs located within a single SM execute the same instruction, but operate on different data, in a given cycle: an execution model known as Single Instruction Multiple Data (SIMD).

Work is allocated to the GPU as *kernels* that contain a large number of threads. Threads within the same kernel are organized into *blocks*, and blocks are mapped as a single unit of work to different SMs. Upon execution, threads within the same thread-block are further partitioned into *warps*. Threads within the same warp are scheduled to run on the SIMD computing engine simultaneously, and thus are executed in lock-step. Initially aimed at improving the performance of graphics applications, thanks to easy to adopt programming models such as CUDA [34] and OpenCL [35], these new processors are increasingly being applied to improve the performance of general purpose and scientific applications.

2.1.4 Heterogeneous Multicore Processors

Given the diversity of target applications in the current landscape of computing, processors that are able to execute diverse applications efficiently is an important design objective. Computer systems that include individual CPU and GPU processors can achieve this, albeit at higher power and communication costs. Integrating diverse cores onto the same

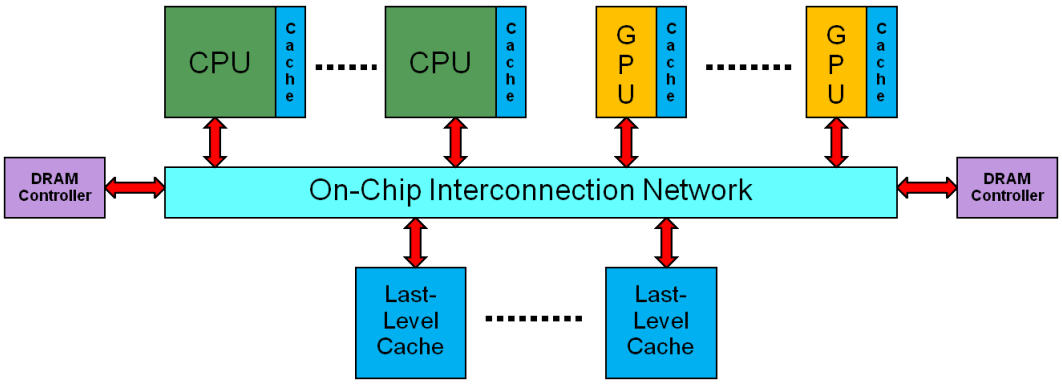


Figure 2.1: A heterogeneous multicore processor with CPU and GPU cores sharing the LLC and other on-chip resources.

die has thus emerged as an attractive alternative.

Designs already in market, such as AMD Fusion APU [8] and Intel Sandy Bridge [9], are advancements in this direction. Already being employed in computing systems at various levels, these designs are in general clever amalgamation of existing CPU and GPU cores. However, designing an efficient heterogeneous multicore processor is a challenging task. In particular, sharing of on-chip resources among cores with diverse characteristics need special attention.

2.2 Challenges in Designing Emerging Processors

We consider a heterogeneous multicore processor design as shown in Figure 2.1. In this design, CPU and GPU cores are integrated onto the same semiconductor die and they share on-chip resources such as the last-level cache, inter-connection network, and memory controllers. In this thesis, we seek to address two important problems concerning these designs: (i) how to improve the performance of these emerging designs; (ii) how to improve the correctness aspect of parallel execution in these designs.

2.2.1 Improving Performance in Heterogeneous Processors

Extracting performance is of paramount concern in the design of a processor. Effective sharing of on-chip resources is critical to achieving this objective. In a heterogeneous multicore processor, this objective becomes challenging due to the diverse characteristics of the constituent cores.

Last-level cache is a shared resource critical to performance. Hence, efficient sharing of LLC is critical to improving performance. In a heterogeneous multicore processor, the differences in cache sensitivity and memory access rate among the constituent cores pose significant challenge to the efficient sharing of the LLC. Under this scenario, existing cache management policies face difficulty in aiding effective sharing of the LLC.

In Chapter 3, we conduct a detailed study of the memory access characteristics and cache sensitivities of the diverse cores on-chip a heterogeneous multicore processor. Based on these observations, in Chapter 4, we propose a new cache management policy for better sharing the last-level cache in a heterogeneous multicore processor.

2.2.2 Improving Correctness of Parallel Execution

With increasing parallelism available on-chip current microprocessors, concerns beyond performance extraction have been gaining importance. *Correctness* is one such issue as increased extraction of parallelism has lead to an increase in issues relate to the correctness of parallel execution [36, 37]. Concurrency bug is an important class of such correctness issues. Detection of concurrency bugs, and recovery from them, is difficult due to the performance and implementation overheads associated with the techniques that have been currently proposed. However, increased availability of parallelism available on-chip increases the need for efficient concurrency bug detection.

In Chapter 5, we propose a technique to utilize resources available on-chip the emerging heterogeneous multicore processors to performance concurrency bug detection efficiently.

Chapter 3

Shared Last-Level Cache Management

From a performance perspective, the last-level cache (LLC) is one of the most important units on-chip a microprocessor. Several design alternatives have been proposed and implemented for the LLC. Among these, the primary concern is about the sharing nature of the LLC. Both private and shared architecture for the LLC have been proposed for multicore processors. Shared LLC benefits from several factors, including: (i) shared design enables dynamic allocation of the cache space between cores, improving the effectiveness of the cache; (ii) space utilization is improved as there is no need to replicate data shared among cores; (iii) overhead of maintaining coherence state is low compared to private LLC where the tags have to be replicated [38].

These characteristics have lead to the adoption of shared LLC in almost all the multicore processors available in market now. However, efficient sharing of the LLC among cores is still a challenging task.

3.1 Considerations in LLC Management

Shared LLC management mechanisms aim at improving the performance of applications by efficiently sharing the LLC. They utilize application characteristics to adapt insertion and replacement of cache blocks, and/or to partition the cache among the sharing applications, with the objective of improving cache hits. The primary concerns in shared LLC management include: (i) victim selection; (ii) block insertion; and (iii) block promotion/demotion. Application characteristics that are considered while making these decisions include: (i) memory access pattern; (ii) data reuse characteristics; and (iii) impact of cache hit on actual performance.

3.2 Prior Work on LLC Management

Based on these characteristics, several shared LLC management schemes have been proposed. In general, they can be divided into two categories: (i) cache partitioning techniques; and (ii) cache replacement policies.

Partitioning Techniques Stone et al. [14] conducted one of the first studies on optimal cache partitioning, where, they proposed a static partitioning based on miss rate information for various applications with varying cache sizes. Dynamic cache partitioning mechanisms, on the other hand, aim to achieve their performance goal by dividing the cache ways among the applications at runtime. Suh et al. [16] introduce dynamic cache partitioning among threads executing on the same chip by utilizing hardware performance counters to maximize cache hit among threads. Moreto et al. [19] propose a dynamic cache partitioning mechanism that considers the memory-level parallelism of an application and the impact of cache misses on its performance. Quality-of-service (QoS) considerations were addressed for multicore cache partitioning by Chang et al. [39], while fairness was considered by Kim et al. [40]. Utility-based cache partitioning (UCP) [17] tries to find

the optimal cache partitioning by prioritizing applications on the basis of benefit from cache over the demand for cache. Thrasher caging [20] re-evaluates cache partitioning mechanisms in the presence of one or more thrashing applications. Recently, PriSM [41] has introduced a probabilistic shared cache management framework that considers various aspects such as cache hit ratio, fairness, and QoS.

Replacement Policies Cache replacement policies aim to identify the appropriate position to insert a new cache block, and to identify the right victim for replacement, to achieve their performance goal. Qureshi et al. propose the dynamic insertion policy (DIP) [21] that overcomes the impact of thrashing behavior of certain applications on other applications in the workload. DIP achieves this by inserting cache blocks from thrashing workloads at LRU position to minimize their cache lifetime. Jaleel et al. [22] utilize re-reference interval prediction (RRIP) to develop a replacement policy that is both thrashing and scan resistant. PIPP [23] is a cache management technique that combines insertion and promotion policies to utilize the benefits of cache partitioning and adaptive insertion. Both RRIP and PIPP show that inserting cache blocks at MRU position (near-immediate re-reference) is not optimal, while insertion of cache blocks at non-MRU position and promoting them on cache hits improves the cache utilization significantly. Pseudo-LIFO [42] proposes a new family of cache replacement policies that is based on fill stack as opposed to the recency stack followed by previous policies. Recently, hierarchy awareness about other levels of cache has been introduced by CHAR [24] to improve the replacement policy applied at the LLC.

3.3 Memory Access Characteristics of Diverse Cores

Memory access characteristics of applications play a critical role in the design of the LLC management policies. However, diversity in the characteristics make designing a shared

LLC management policy challenging as the LLC would be shared by these diverse applications.

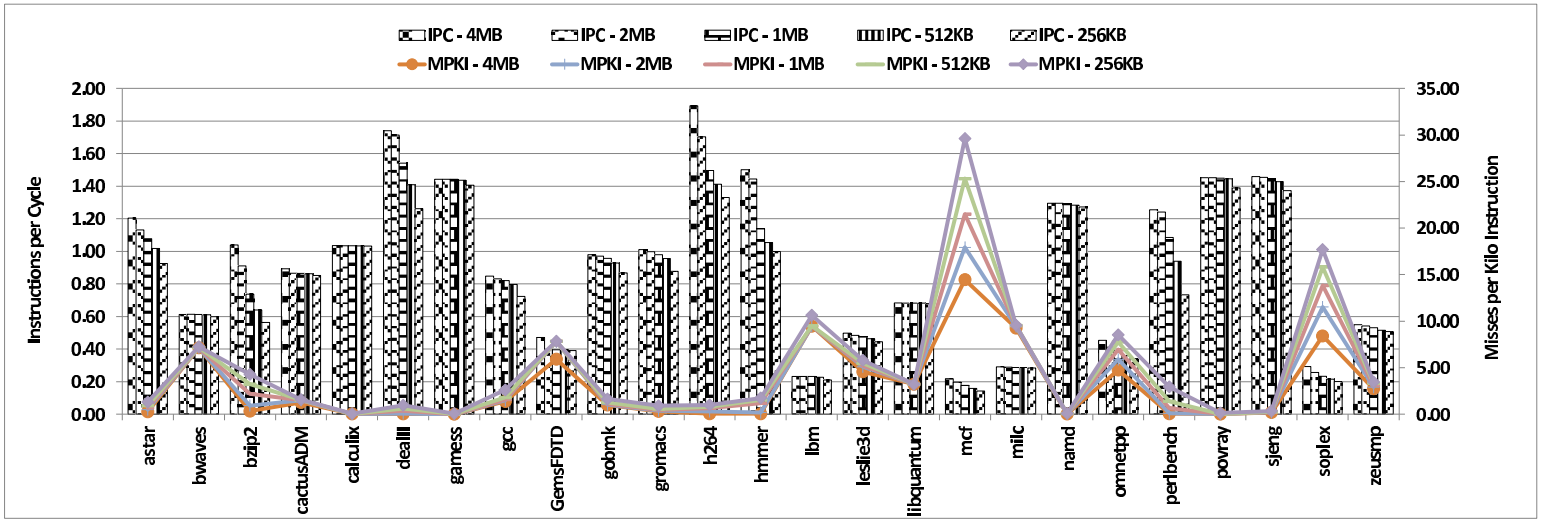
Figure 3.1(a) shows the LLC characteristics of a variety of CPU applications taken from the SPEC CPU 2006 benchmark suite [43]. Cache behavior with varying LLC sizes are presented in terms of LLC misses per kilo instructions (MPKI). The graphs also shows the impact of this cache behavior on performance as instructions per cycle (IPC). It is clear that CPU applications, in general, show a cache sensitive behavior where their performance suffers with decreasing LLC sizes.

With the popularity of programming models such as CUDA [34] and OpenCL [35], GPU cores are being used to accelerate general purpose applications. This has increased the diversity in the LLC characteristics of GPU applications. Figure 3.1(b) shows the LLC characteristics of GPU applications taken from AMD APP benchmark suite [44]. Similar to Figure 3.1(a), this figure also shows LLC characteristics (MPKI) and its impact on performance (IPC). GPU applications display a cache sensitivity behavior that is different from CPU applications. GPU applications are in general less cache sensitive than CPU applications.

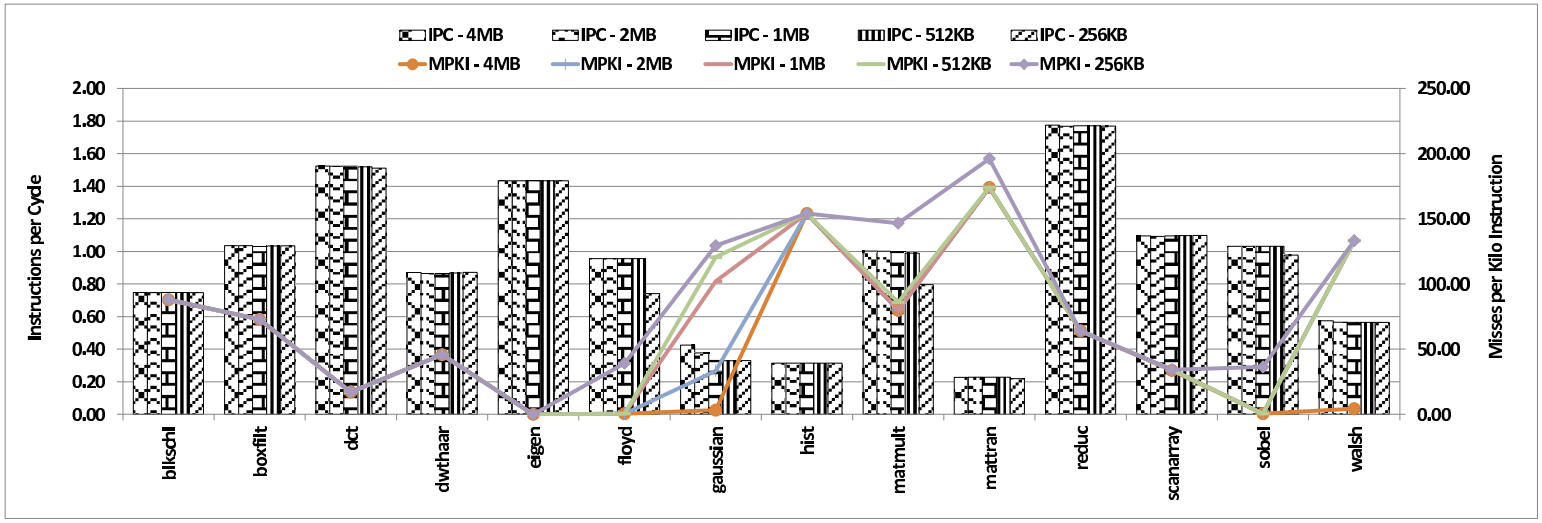
In addition to the diversity in cache sensitivity, there is a significant difference in the memory access rate among CPU and GPU applications. GPU cores support thousands of threads and the applications are highly parallel. This leads to an order of magnitude higher or more memory access rate from GPU applications. Under these circumstances, LLC management becomes very challenging in a heterogeneous multicore processor where both CPU and GPU cores share the LLC.

3.4 Challenges in Heterogeneous Multicore Processors

In general, CPU applications are more cache sensitive than GPU applications. However, when CPU and GPU applications share the cache, existing cache management policies tend to favor GPU applications. The order of magnitude higher memory access from the GPU



(a) CPU cache sensitivity characteristics. CPU applications are taken from the SPEC CPU 2006 benchmark suite [43].



(b) GPU cache sensitivity characteristics. GPU applications are taken from the AMD APP benchmark suite [44].

Figure 3.1: Cache sensitivity characteristics of CPU and GPU applications.

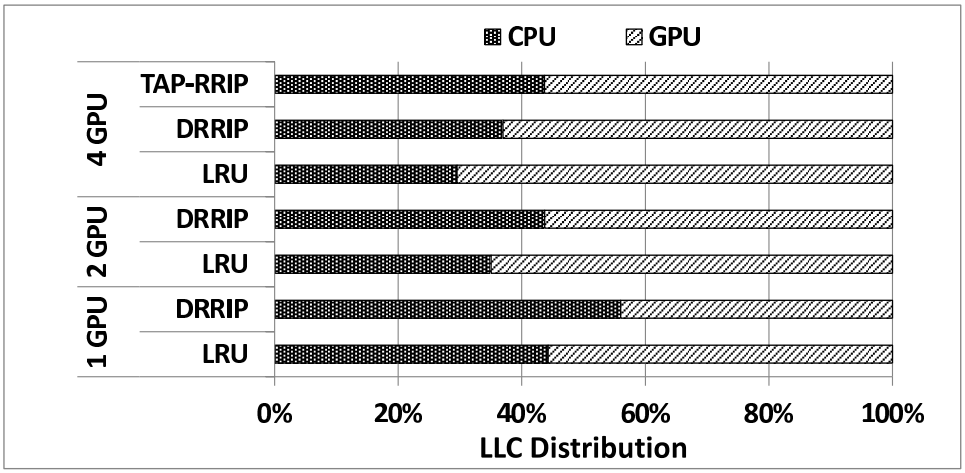


Figure 3.2: Distribution of space in an LLC shared by CPU and GPU cores. SPEC benchmark 401.bzip2 executes on the CPU core. The performance impact is measured across the set of GPU benchmarks shown in Table 4.4. TAP-RRIP [45] results are shown only for 4 GPU configuration as TAP-RRIP needs atleast four GPU cores for full functioning.

cores skew the judgement of current policies to end up assigning majority of the cache space to GPU applications.

Figure 3.2 shows the distribution of space in a shared LLC in a heterogeneous multicore processor containing CPU and GPU cores. We execute a cache sensitive benchmark (401.bzip2 from SPEC CPU2006 benchmark suite [43]) on the CPU core. The GPU cores execute a GPU application from the AMD APP benchmark suite [44] as listed in Table 4.4. The shared LLC is 2MB in size and the detailed specifications of the CPU and GPU cores are given in Table 4.2. This figure evaluates three cache management policies. The most basic policy we consider is the Least Recently Used (LRU) policy. The other policies (DRRIP and TAP-RRIP) are discussed next.

DRRIP: Dynamic Re-Reference Interval Prediction (DRRIP) [22] is a cache management policy developed primarily for homogeneous multicore processors. DRRIP predicts re-reference (reuse) interval of cache lines to be either *intermediate* or *distant*, and inserts

lines at non-MRU (Most Recently Used) position based on the re-reference prediction. If a line is re-used after insertion into the LLC, it is promoted by increasing its age to improve its lifetime in the cache. Non-MRU insertion of cache lines performs better than MRU insertion because most of the lines do not observe immediate re-reference.

TAP-RRIP: TLP-Aware Cache Management Policy (TAP) [45], addresses the diversity of on-chip cores while designing the LLC management policy. TAP identifies the cache sensitivity of the GPU application using a technique called *Core Sampling*. In this technique, two GPU cores (sampling cores) are made to follow LRU and MRU cache replacement policies respectively. The performance difference between these sampling cores indicate the cache sensitivity of the GPU application. This information, along with the difference in LLC access rate between the CPU and GPU cores, is then used to influence the decisions made by the underlying cache management policy. When these metrics indicate a cache sensitive GPU application, both the cores are given equal priority. Whereas, when GPU application is found to be cache insensitive, GPU core is given lower priority in the underlying policy.

We consider three configurations of GPU cores for these experiments: configurations with 1, 2, and 4 GPU cores (each supporting up to 1024 threads as mentioned in Table 4.2) sharing the 2MB LLC with the CPU core (architecture described in Table 4.2). The higher the GPU core count, larger the pressure from the GPU application on the shared LLC. For configurations with 1 and 2 GPU cores, we can see that LRU policy end ups giving majority of the cache space to the GPU application. DRRIP being an advanced policy compared to LRU is able to improve the cache occupancy of the cache sensitive CPU application. However, it still gives equal priority to both CPU and GPU cores, and hence GPU still occupies majority of the LLC space. Also, the effectiveness of the policy reduces with increasing GPU core count.

TAP needs more than 2 GPU cores for its *Core Sampling* technique to work. Hence, in the configuration with 4 GPU cores, we consider a variant of TAP built on the DRRIP

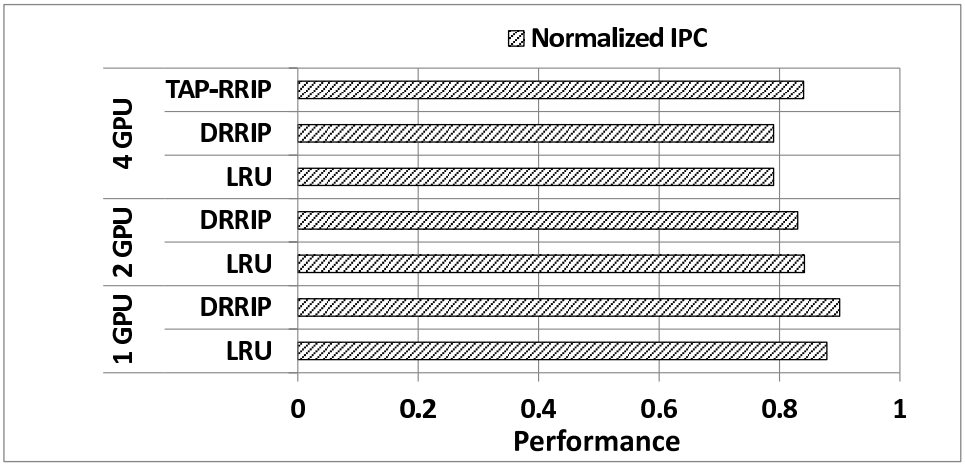


Figure 3.3: Performance impact of GPU applications on CPU applications. IPC of 401.bzip2 is normalized to its IPC when executing on the heterogeneous processor without interference from the GPU cores. TAP-RRIP [45] results are shown only for 4 GPU configuration as TAP-RRIP needs atleast four GPU cores for full functioning.

policy called TAP-RRIP. For LRU and DRRIP, we see that the cache distribution get further skewed against the CPU application. TAP-RRIP, being aware of the core diversity, is able to improve the cache occupancy of the cache sensitive CPU application. However, the improvement is not by a large margin. This is due to a side-effect of the core sampling technique used by TAP-RRIP, which leaves significant amount of zero-reuse blocks in the MRU position. We discuss the details of the performance of TAP-RRIP in Section 4.3.2.

Overall, this experiment shows that management of cache shared by diverse cores is challenging, and currently proposed techniques favor the lesser cache sensitive GPU cores due to its higher memory access rate. This leads to performance penalty for the cache sensitive CPU application. Figure 3.3 shows the performance impact of introducing GPU cores that share the LLC on the cache sensitive CPU application. The normalized IPC shown here is the IPC of the CPU application, executing along with the GPU application (sharing a 2MB LLC), relative to its IPC when executing alone with a 1MB LLC. The performance of all the three policies are worse than the performance of LRU without

interference from GPU cores. Also, this performance degradation increases with increasing GPU core count for each policy.

In addition to the performance aspect, presence of diverse cores could change the energy consumption profile, both on-chip as well as off-chip, for the heterogeneous multicore processor under existing policies. This could result in a significant increase in energy consumption at the LLC module if the order of magnitude higher access rate of GPU is not efficiently handled by the cache replacement policy. Also, since this increase in energy consumption does not imply performance improvement in a typical cache insensitive GPU application, the energy efficiency of the processor is impacted. Bandwidth utilization is another characteristic that would also be significantly impacted by the high memory access rate from GPU cores. Since energy consumption and bandwidth utilization have already turned into first-order constraints in processor design, these aspects could be significant challenge to the viability of cache management policies in future processor designs.

3.5 Exploiting the GPU Memory Access Characteristics

Previous experiment shows that the management of shared LLC in a heterogeneous multicore processor needs a reevaluation. We next evaluate in detail, the memory access characteristics of GPU applications and aim to identify characteristics that we can utilize to improve the LLC management policies.

3.5.1 GPU Memory Access Behavior

Figure 3.1(b) shows the memory access characteristics for GPU applications. In general, GPU applications are known to exhibit streaming memory access behavior as shown here for the benchmark Histogram (from the AMD APP benchmark suite [44]) in Figure 3.4. Streaming access behavior indicates a memory access pattern where the application sweeps through a large dataset with little or no reuse. Here, the application experiences the same

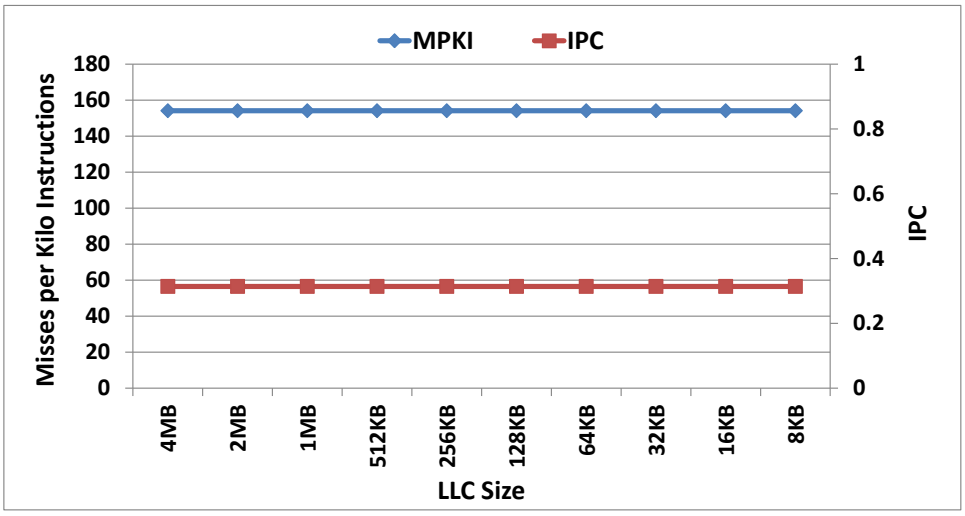


Figure 3.4: Cache sensitivity of GPU application Histogram. Histogram exhibits a streaming memory access behavior that is typical of many GPU applications.

(large) MPKI irrespective of the LLC size, and the change in LLC size has no impact on its performance (IPC).

However, with more general purpose applications being written for GPU cores, we are witnessing deviations from this behavior. Figure 3.5 shows the memory access characteristics for GPU application BoxFilter (from the AMD APP benchmark suite [44]). This GPU application displays cache performance (MPKI) variation with varying LLC sizes, however, this has little impact on the actual performance (IPC). For example, although the MPKI increases from 70 to nearly 150 while reducing the LLC size from 64KB to 8KB, this has no impact on its IPC.

This shows behavior where actual performance (IPC) is not directly related to cache performance (MPKI). This indicates that for GPU applications, cache performance is not necessarily a good indicator of actual performance.

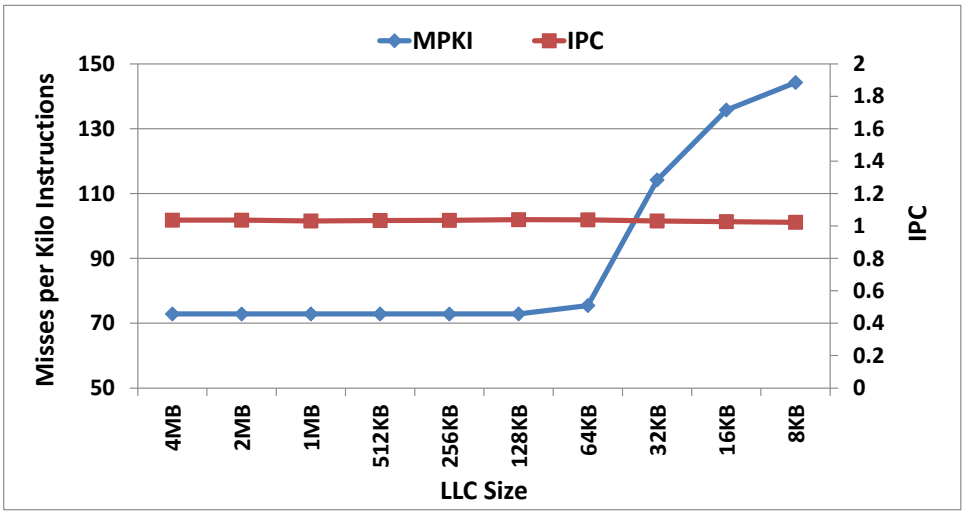


Figure 3.5: Cache sensitivity of GPU application BoxFilter. BoxFilter exhibits a memory access behavior that is divergent from the traditional understanding of the cache sensitivity of GPU applications.

3.5.2 TLP as a Runtime Metric

A GPU core is a highly parallel architecture supporting thousands of threads. In such cores, the high levels of TLP availability offers a large number of threads for scheduling when the pipeline is stalled waiting for memory accesses to complete. Figure 3.6 shows the performance characteristics of GPU application Floyd (from the AMD APP benchmark suite [44]) with the average amount of TLP available for varying LLC sizes. The available TLP at runtime is measured, using hardware performance monitors, as the number of *wavefronts*[1] ready to be scheduled at any given time. Higher number of ready wavefronts indicate higher TLP. We observe that even when MPKI increases while reducing the LLC size from 256KB to 64KB, the IPC is sustained because of the availability of threads that are ready to schedule. This shows that TLP can be utilized as a metric to identify the cache sensitivity of GPU applications.

[1]Work is allocated to the GPU cores as *kernels* that contain a large number of threads. A kernel is further partitioned and mapped to different GPU cores as *thread-blocks* or *workgroups*. Scalar threads within each

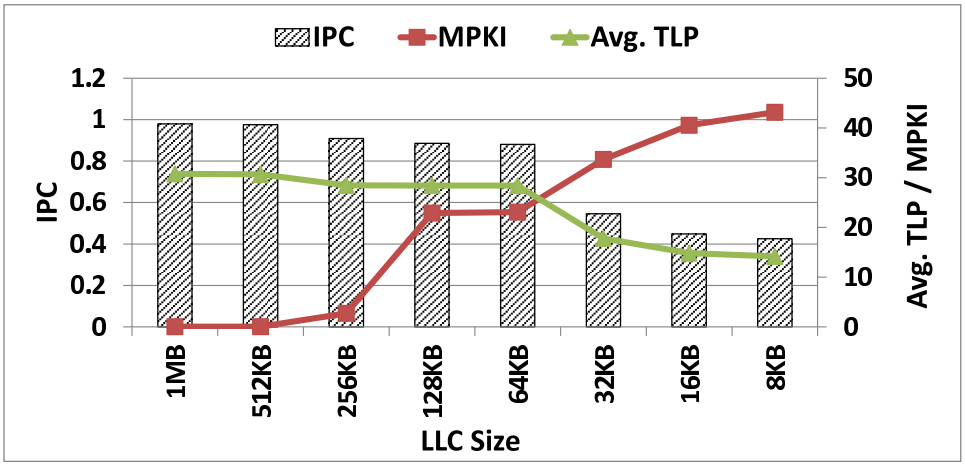


Figure 3.6: TLP availability and cache sensitivity characteristics of GPU application Floyd. Figure shows that available TLP helps Floyd sustain its performance in the face of increasing MPKI.

3.5.3 GPU LLC Bypassing

Cache management policies give low priority to the application identified as cache insensitive. TAP, for example, gives low priority to the memory accesses from the GPU core when the GPU application is identified as cache insensitive. However, GPU memory accesses still pass through the cache and occupy significant portion of the LLC owing to their order of magnitude higher memory access rate. Here, we explore avenues to let the GPU memory accesses bypass the LLC, if not found beneficial.

Figure 3.7 shows the impact of bypassing LLC for GPU applications in Table 4.4. Figure shows the data for randomly bypassing LLC for 25%, 50%, 75% and 100% of GPU memory accesses. Performance is relative to the IPC of the application executing under LRU policy without LLC bypassing. We can observe that, the impact of LLC bypassing is related to the cache sensitivity of the application and the amount of TLP available at runtime. On an average, GPU suffers minimal performance impact for 25% LLC bypassing.

GPU core are scheduled simultaneously as *warps* [34] or *wavefronts* [46] onto the SIMD computing engine.

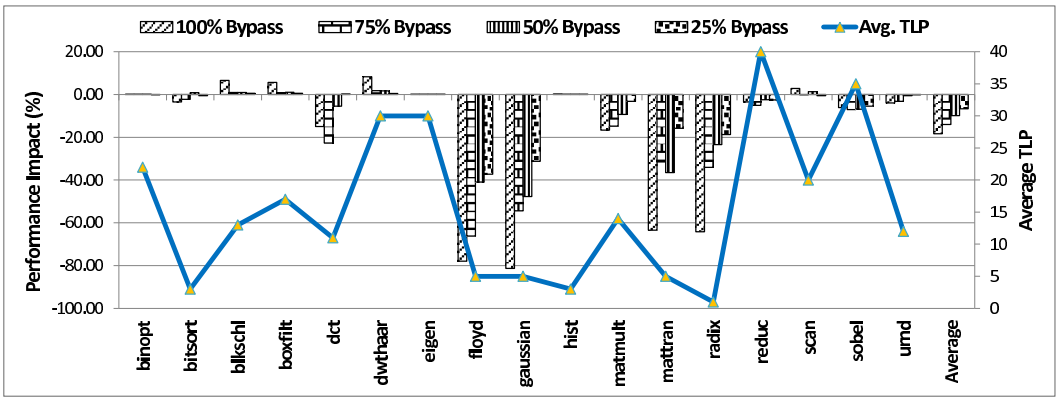


Figure 3.7: Performance impact of bypassing LLC for memory accesses of GPU applications in Table 4.4. Performance is relative to the performance of the application without LLC bypassing under LRU policy.

Furthermore, GPU applications can tolerate 50% and 75% bypassing without significant performance impact.

This insight offers a potentially effective way to manage LLC shared by CPU and GPU cores. GPU memory accesses could be made to bypass the shared LLC if those applications are found to be cache insensitive. The benefits of such a scheme could be two-fold:

- By bypassing LLC for GPU memory accesses, the LLC share of GPU could be reduced, leading to an increase in the LLC space for the cache sensitive CPU application.
- The cache insensitive GPU that does not benefit from the LLC could see performance improvement by skipping LLC lookup altogether and directly accessing data from memory.

3.6 Summary

In this chapter, we studied LLC management techniques in general. We analyzed the challenges of managing caches, particularly in the scenario of diverse cores sharing the

LLC. We studied the memory characteristics of GPU applications in order to develop techniques that better manage an LLC shared by heterogeneous cores. We describe this technique in Chapter 4.

Chapter 4

Managing the LLC in a Heterogeneous Processor

In this section, we describe a cache management mechanism that mitigates the performance impact of LLC sharing in heterogeneous multicore processors by throttling LLC accesses initiated by the GPU cores. We call our technique HeLM which stands for Heterogeneous LLC Management [47]. HeLM exploits the memory access latency tolerance capability of the GPU cores and allows the GPU cores to yield LLC space to the cache sensitive CPU cores without significantly degrading their own performance. In HeLM, we manage the LLC occupancy of the GPU cores by allowing the GPU memory traffic to selectively bypass the LLC, as shown in Figure 4.1, when: i) the GPU cores exhibit sufficient TLP to tolerate increased memory access latency; or ii) when the GPU application is not sensitive to LLC performance.

4.1 Heterogeneous LLC Management

For each GPU memory access, the decisions for bypassing the LLC is made at the shared LLC. On an L1 cache miss, the TLP information of the GPU core is attached to the LLC

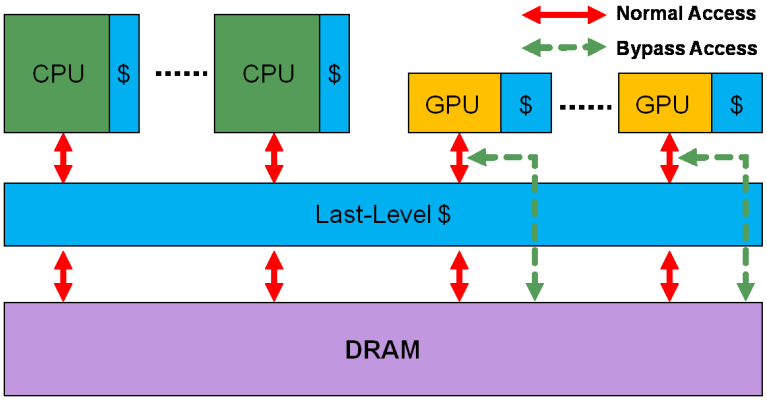


Figure 4.1: LLC bypassing employed by GPU cores in HeLM. Selected GPU memory accesses bypass the LLC to provide additional space to the cache sensitive CPU application.

access request. If the access misses at the LLC, the current TLP is compared with a selected threshold. If the current TLP is greater than the threshold, response to the cache miss bypasses LLC. The available TLP at runtime is measured using hardware performance monitors that measure the number of *wavefronts* ready to be scheduled at any given time. Higher number of ready wavefronts indicate higher TLP, which in turn suggests that GPU can tolerate higher memory access latency.

Figure 4.2 shows the high level view of HeLM. GPU LLC bypassing decisions are made on the basis of the cache sensitivities of both CPU and GPU applications. The CPU application is given higher priority in our algorithm as it is, in general, more cache sensitive. If the CPU application is found cache sensitive, GPU memory accesses are subject to aggressive LLC bypassing. If not, GPU LLC sensitivity is considered and a bypass aggressiveness is selected accordingly. When neither of the applications are cache sensitive, the bypassing aggressiveness selected does not impact the performance. However, it could have significant impact on the DRAM energy consumption and bandwidth utilization, both on-chip as well as off-chip.

Cache sensitivities of the CPU and GPU applications plays a critical role in making

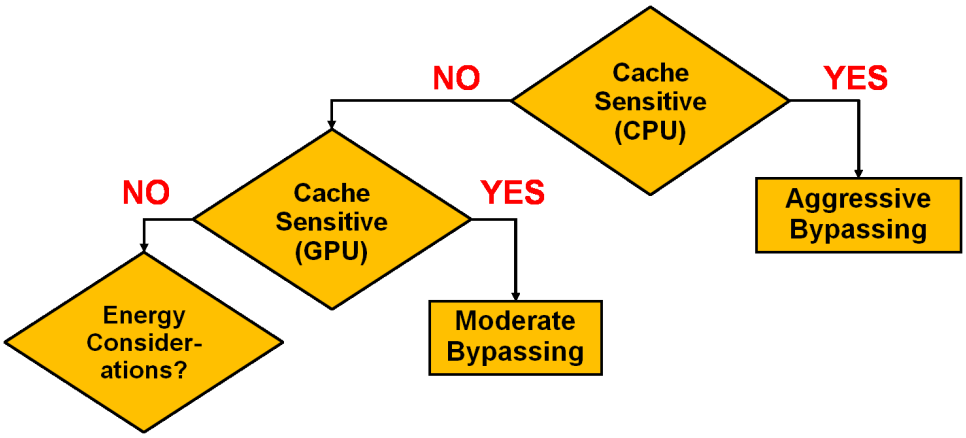


Figure 4.2: Flowchart for bypass algorithm in HeLM.

bypassing decisions. A cache insensitive CPU application does not benefit from increased LLC space available due to GPU LLC bypassing. Bypassing LLC for a cache sensitive GPU application executing along with such cache insensitive CPU applications could degrade GPU performance without improving the overall performance.

In the following subsections, we discuss in detail the techniques employed to identify: i) the cache sensitivities of the CPU and GPU applications; and ii) an effective TLP threshold to measure the memory access latency tolerance of the GPU application. We combine these metrics into a *Threshold Selection Algorithm* (TSA) that makes GPU LLC bypassing decisions.

4.1.1 Measuring Cache Sensitivity

We employ a mechanism based on the *set dueling* [21] technique to measure the cache sensitivity of the CPU and GPU applications. Set dueling applies two opposing techniques to two distinct sets, and identifies the characteristic of the application from the performance difference among the sets. Dynamic Set Sampling (DSS) [48] has shown that sampling a small number of sets in the LLC can indicate the cache access behavior with high accuracy. We use this technique by sampling 32 sets (out of 4096) to measure the cache sensitivity.

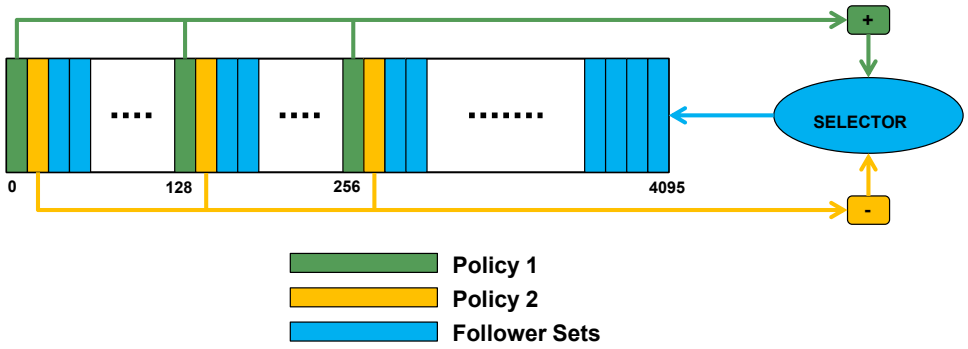


Figure 4.3: Overview of the set-dueling [21] technique.

Figure 4.3 shows a high-level view of the workings of the set-dueling technique. Here, for the 4096 set cache structure, every 128th set starting from Set$_0$ follows POLICY1, while every 128th set starting from Set$_1$ follows POLICY2. Performance events, such as cache hits, are monitored and these events increment a saturation counter when it occurs in POLICY1 sets and decrement the saturation counter when it occurs in POLICY2 sets. The value of the saturation counter is taken at the end of the sampling period to determine which policy is performing better. This policy is then applied on the *follower* sets.

TLP Thresholds

For HeLM, the two opposing policies used in set-dueling are: bypassing with high aggressiveness and bypassing with low aggressiveness. In HeLM, the bypassing aggressiveness is adapted by choosing an appropriate threshold for bypassing. The threshold corresponds to the available TLP in the GPU application. When the available TLP is higher than a selected threshold, the

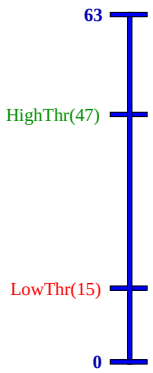


Figure 4.4: Range of threshold for TLP in a GPU architecture supporting up to 64 simultaneous wavefronts (warps).

GPU application has enough parallelism to sustain increased memory latency and is suitable for LLC bypassing.

Figure 4.4 shows the range for TLP threshold in a GPU that supports up to 64 simultaneous wavefronts (warps). It also shows two arbitrary thresholds: HighThr and LowThr. HighThr corresponds to less aggressive bypassing as current TLP has to be higher than HighThr to enable bypassing. Similarly, LowThr corresponds to more aggressive bypassing.

CPU LLC Sensitivity

We evaluate the cache sensitivity of the CPU application by monitoring the impact of GPU LLC bypassing on the performance of the CPU application. Since CPU applications are more cache sensitive than GPU applications, change in cache misses directly affects the performance of CPU applications. We measure two CPU LLC misses *MissLow* and *MissHigh* corresponding to GPU bypassing at LowThr and HighThr respectively. Two *set dueling monitors* (SDM) are used at the LLC to obtain the MissLow and MissHigh numbers, each bypassing GPU accesses at LowThr and HighThr respectively. For these SDMs, the bypassing decision is made using its unique TLP threshold irrespective of which GPU core initiated the access.

Since GPU takes more LLC space with HighThr than with LowThr, MissHigh is always greater than MissLow. If the difference between MissHigh and MissLow (ΔMISS_{CPU}) is greater than *mThreshold*[1], GPU bypassing is affecting the CPU LLC behavior, and hence its performance. This criterion can also identify compute intensive as well as streaming CPU workloads[2]. Dynamic Set Sampling (DSS) [48] has shown that sampling a small number of sets in the LLC can indicate the cache access behavior with high accuracy. We use this technique by sampling 32 sets (out of 4096) to measure the cache sensitivity.

[1]Based on empirical analysis, we set mThreshold to 10%.

[2]For compute intensive workloads, MissHigh $\approx$ MissLow $\approx$ 0, while for streaming workloads, MissHigh $\approx$ MissLow $\approx$ K (positive number).

GPU LLC Sensitivity

Figure 3.5 shows that cache miss rate is not a direct indicator of performance for GPU applications. Hence, we adapt the set-dueling technique to enable measuring GPU LLC sensitivity by directly measuring the performance of the GPU core. For this purpose, we utilize two GPU *sampling cores* and the two TLP thresholds: *LowThr* and *HighThr*. In every sampling period, one of the GPU cores (*LowGPU*) performs LLC bypassing at LowThr, while the other core (*HighGPU*) uses HighThr. LowThr is always smaller than HighThr and indicates a higher rate of bypassing. Hence, LowGPU bypasses more memory accesses than HighGPU. A significant performance difference (ΔIPC_{GPU}), greater than *pThreshold*[3], between these two cores indicates that LLC bypassing is having an adverse impact on the GPU performance and hence the GPU application is cache sensitive. If the performance difference is within the limit, the GPU application is considered cache insensitive.

4.1.2 Determining Effective TLP Threshold

Determining the effective TLP threshold to initiate GPU LLC bypass is critical. To adapt to the diversity among GPU applications, and the runtime variations within an application itself, we propose an algorithm to dynamically adapt LowThr and HighThr. Our heuristic is inspired by the *binary-chop* algorithm that is commonly used for searching an element in a sorted list bound by limits *MaxLimit* and *MinLimit*. Binary-chop algorithm starts with two parameters U and L such that U $\geq$ L, and calculates a decision element E as the average of U and L (Avg(U, L)). At the beginning of the algorithm, U and L are initialized to MaxLimit and MinLimit, respectively, and a prediction is made. If the decision element is lower than expected, the search window is moved up (GO UP) by updating U and L as shown in Table 4.1. If the prediction is higher than expected, the search window is moved down (GO DOWN). At each step, E is recalculated, and the process is continued until E

[3]Based on empirical analysis, we set pThreshold to 5%.

matches with the searched element.

Action	U	L
INIT	MaxLimit	MinLimit
GO UP	AVG(MaxLimit, U)	E
GO DOWN	E	AVG(L, MinLimit)

Table 4.1: Binary-chop algorithm for adapting sampling thresholds at runtime.

Our adaptation of the binary-chop algorithm recomputes the higher and lower bypass thresholds at runtime depending upon the application behavior. We start by initializing HighThr (U) = $\frac{3}{4}\times$MAX$_{wavefronts}$, and LowThr (L) = $\frac{1}{4}\times$MAX$_{wavefronts}$. Here, MaxLimit = MAX$_{wavefronts}$, MinLimit = MIN$_{wavefronts}$. After every sampling period, HighThr and LowThr values are updated with new values of U and L, respectively.

Figure 4.5 demonstrates the working of this algorithm with an example. Here, we start with HighThr and LowThr as shown in PHASE 0. In each phase, the selected threshold is underlined. If HighThr is selected and the program behavior indicates the need to reduce bypassing aggressiveness, binary-chop algorithm will perform the GO UP step, increasing the thresholds as shown in PHASE 1. If LowThr is selected and the program behavior indicates the need to increase bypassing aggressiveness, GO DOWN step will be taken. However, if there is a switch in direction from GO UP to GO DOWN, as shown in PHASE 2, we detect a toggle and retain the previously selected threshold. A toggle indicates reaching of a stable phase and, as observed in PHASE 3, the thresholds are maintained for a specified number of sampling periods. After this, the algorithm restarts as shown in PHASE 4.

4.1.3 Putting It All Together

We combine the cache sensitivity information and TLP thresholds into a *Threshold Selection Algorithm* (TSA). TSA monitors the workload characteristics continuously and re-evaluates the TLP threshold at the end of every sampling period (1M execution cycles in

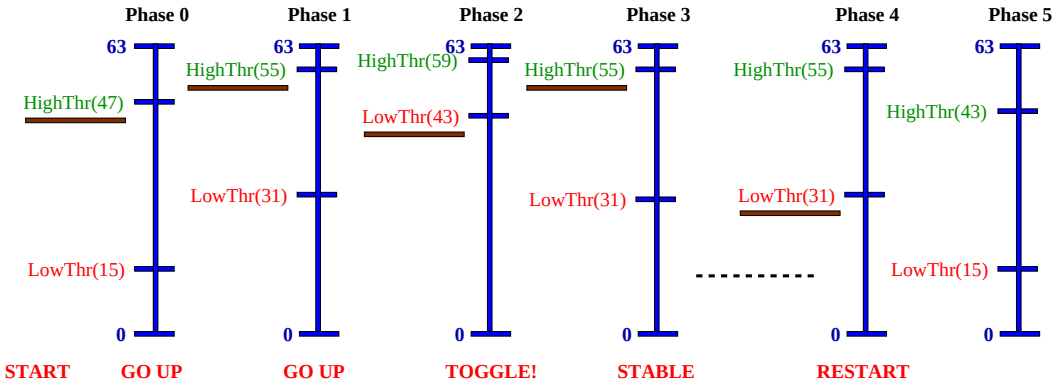


Figure 4.5: Binary-chop algorithm for varying bypass threshold.

our study). Once the threshold is chosen, it is enforced on all the *follower* GPU cores. The pseudocode for TSA is shown in Algorithm 1. If the CPU application is cache sensitive, LowThr is selected for bypassing LLC for GPU memory accesses. Otherwise, the choice of threshold depends on the characteristics of the GPU application. If GPU application is cache sensitive, HighThr is selected for bypassing LLC for GPU memory accesses.

Although the threshold selected when both CPU and GPU applications are identified as cache insensitive does not impact performance significantly, it could have a significant impact on the off-chip DRAM access rate. In such a case, we take a conservative approach by evaluating which metric is nearer to the limit and select the TLP threshold accordingly.

Based on the threshold selected by TSA, HighThr and LowThr are re-calculated using the binary-chop heuristic discussed in Section 4.1.2. For cache sensitive GPU applications, LLC bypassing aggressiveness is reduced by action GO UP; otherwise, the aggressiveness is increased by action GO DOWN. If the actions toggle between GO UP and GO DOWN for consecutive sampling periods, we maintain the existing HighThr and LowThr values for next five sampling periods.

CPU cache management policies have employed thread awareness to avoid the domination of one application on the sharing policy. Mechanisms such as thread-aware DIP [49] and thread-aware DRRIP [22] (referred to as DRRIP here) utilize separate set of SDMs to

Data: ΔIPC$_{GPU}$,ΔMISS$_{CPU}$
Result: Bypass TLP Threshold
if $\Delta MISS_{CPU} \geq mThreshold$ **then**
  Set LowThr as TLP threshold;
end
else if $\Delta IPC_{GPU} > pThreshold$ **then**
  Set HighThr as TLP threshold;
end
else
  if *delta*(ΔIPC_{GPU},*pThreshold*) $\geq$ *delta*($\Delta MISS_{CPU}$,*mThreshold*) **then**
    Set LowThr as TLP threshold;
  end
  else
    Energy considerations decide TLP threshold;
  end
end

Algorithm 1: Pseudocode for the Threshold Selection Algorithm (TSA).

isolate the influence of applications on each other. Similarly, HeLM is made thread aware by assigning individual MissLow and MissHigh counters to calculate ΔMISS$_{CPU}$ for each thread. For thread awareness, TSA selects LowThr as the TLP threshold if any of the ΔMISS$_{CPU}$ is $\geq$ mThreshold.

4.1.4 Other Design Considerations

Impact on On-Chip Energy and Off-Chip Access

Allowing memory accesses that are unlikely to be reused in the cache to bypass the LLC can potentially improve cache utilization and reduce dynamic energy of LLC accesses. However, LLC bypassing could also increase the off-chip DRAM accesses. Due to the streaming nature of GPU applications, the blocks in the LLC are mostly dead, and the accesses go off-chip to fetch data blocks from DRAM. Thus we observe that LLC bypassing does not increase off-chip DRAM accesses significantly. We evaluate the impact of bypassing on LLC energy consumption and off-chip accesses in Section 4.4.

Handling Coherence

The contemporary GPU does not support coherent memory hierarchy. However, if coherence is supported in future GPUs, bypassing can also be easily supported. The additional support for maintaining coherence with GPU bypassing may or may not be required depending upon the inclusion property of GPU cache hierarchy. Inclusion ensures that cache blocks present in high level caches are also present in the LLC, while non-inclusion or exclusion relaxes such a constraint. Bypassing essentially turns the inclusive LLC into a non-inclusive cache. Therefore, the support necessary for maintaining coherence in non-inclusive LLC can also be used to support bypassing in inclusive LLC. Coherence in non-inclusive caches is maintained by employing mechanisms such as snoop filter [50], which is essentially a replica of higher level cache tags at the LLC. Therefore, bypassing for non-inclusive LLC will not require any modifications for handling coherence, while support similar to snoop filter will be necessary for inclusive LLC.

While this work evaluates workloads where the CPU and GPU applications have disjoint address spaces, we expect the proposed technique to be equally effective when these applications share the same address space. When data is shared between the CPU and the GPU, the underlying cache coherence mechanism will ensure correctness of data accesses, and the proposed bypass mechanism can be deployed with a snoop filter as discussed above. In this work, we model a GPU cache hierarchy that is write-through; however, it will work equally well with a write-back cache.

4.2 Experimental Methodology

In this section, we describe the evaluation infrastructure and the experimental methodology used to evaluate HeLM.

4.2.1 Evaluation Infrastructure

In this section, we discuss the simulators used, architectural parameters simulated and the benchmarks and workloads considered.

Simulator

We evaluate HeLM on a cycle accurate simulator, Multi2Sim [51], that simulates both CPU and GPU cores as depicted in Figure 2.1. The CPU cores are modelled on a 4-wide out-of-order x86 processor, while the GPU cores are based on AMD Evergreen [7] architecture. We extended the memory subsystem in Multi2Sim to support shared LLC between CPU and GPU cores. Table 4.2 shows the parameters of the cores we simulate.

CPU	
Core	1-4 cores, 2.6GHz, 4-wide out-of-order, 64-entry RoB
L1 Cache	4-way, 32KB, 64B line, private I/D (2 cycles)
L2 Cache	8-way, 256KB, 64B line, unified (8 cycles)
GPU	
Core	4 cores, 1.3GHz, 8-wide SIMD, 16K register file, 64 wavefronts, round-robin scheduling
L1 Cache	4-way, 8KB, 64B line, private I/D (2 cycles)
Shared Memory	32KB, 256B block (2 cycles)
Shared Components	
LLC	32-way, 2-8MB, 64B line, 4-tiles (20 cycles)
DRAM	4GB, 4 controllers (200 cycles)
NoC	Mesh topology, 32B flit-size

Table 4.2: Configuration parameters for the heterogeneous evaluation infrastructure.

We evaluate 500 million instructions for each CPU benchmark and 150 million instructions[4] for each GPU benchmark. The 500 million representative interval for each CPU benchmark, with *ref* input, is obtained through SimPoint [52] analysis. As followed in previous works [17, 22, 45], early finishing benchmarks continue to execute until all the

[4]An instruction executed by all threads in a wavefront is counted as one instruction.

benchmarks execute the specified number of instructions. We utilize McPAT [53] for studying on-chip energy consumption, and DRAMSim2 [54] to simulate DRAM timing and to calculate off-chip DRAM energy consumption.

Baseline Processor Configuration

Selecting an appropriate baseline processor model is necessary so as not to skew the evaluations in favor of any policy. We follow a heterogeneous processor model as shown in Figure 2.1, and here, we discuss the number of CPU and GPU cores chosen for our evaluations.

We model our baseline design on AMD Fusion APU [8] (A4-5300) which contains two x86 CPU cores and 128 AMD Radeon GPU stream processors (grouped into 4 compute units). We term this, the 2C4G configuration where 'C' stands for CPU core and 'G' stands for GPU core (compute unit). We select the number of GPU cores as four so that the die area taken by these four cores match the die area taken by one CPU core. This method justifies comparison between performances of an application executing on a CPU core and an application executing on the GPU cores.

An inappropriately chosen configuration could bias the study of the impact of GPU cores on the performance of the CPU application shown in Figures 3.2 and 3.3. This bias could occur when the GPU cores are significantly larger than the CPU core, and as a result, overwhelm the LLC occupancy and performance. Hence, it is essential to avoid any such bias. To take into account any bias that could result from the 2C4G configuration, we consider two more configurations on the either side of it: 1C4G and 4C4G. In 1C4G, one CPU core shares the on-chip resources with four GPU cores, and in 4C4G, four CPU cores share the on-chip resources with four GPU cores.

Benchmarks

The CPU benchmarks evaluated belong to the SPEC CPU2006 benchmark suite [43]. The GPU benchmarks evaluated are OpenCL programs from AMD APP (Application Parallel Programming) v2.5 software development kit [44]. We classify these benchmarks into different categories, depending upon their cache performance, as shown in Tables 4.3 and 4.4.

Category	**Benchmarks**
Cache sensitive	bzip2, gcc, mcf, perlbench, dealII, omnetpp, astar, soplex, povray, h264
Cache insensitive	**Streaming:** libquantum, bwaves, milc, zeusmp, cactusadm, gemsfdtd, lbm, leslie3d **Compute intensive:** gobmk, hmmer, gromacs, sjeng, gamess, calculix, tonto, namd

Table 4.3: Classification of CPU benchmarks.

Category	**Benchmarks**
Cache sensitive	matrix-multiplication, matrix-transpose, gaussian, fast-walsh transform, floyd-warshal
Cache insensitive	**Streaming:** histogram, radix sort, blackscholes, reduction **Performance insensitive:** sobel, dwthaar1D, scanarray, dct, box filter **Compute intensive:** binomial option, eigen, bitonic sort

Table 4.4: Classification of GPU benchmarks.

We evaluate multiprogrammed workloads on heterogeneous processors with core configuration as shown in Table 4.5. For the three processor configurations, namely 1C4G, 2C4G and 4C4G, we consider three workload combinations by the same name. Each of the CPU core executes a CPU benchmark while all the four GPU cores execute the same

GPU benchmark. Thus, 1C4G contains one CPU and one GPU application, while 2C4G contains two CPU applications with one GPU application.

Equal number of CPU and GPU benchmarks are selected from cache sensitive and insensitive categories. Workloads in Table 4.5 are formed by randomly selecting benchmarks from each category. Processor with one CPU core shares a 2MB LLC with the four GPU cores, while processor with two and four CPU cores share 4MB LLC and 8MB LLC, respectively, with the four GPU cores.

Core Configuration	**Workloads**
1CPU + 4GPU (1C4G)	100
2CPU + 4GPU (2C4G)	40
4CPU + 4GPU (4C4G)	30

Table 4.5: Heterogeneous workloads evaluated.

4.2.2 Evaluation Metrics

We use instructions per cycle (IPC) as the main performance metric. Speedup of each application is calculated as the ratio of the IPC for a policy to the IPC for the baseline policy (Eq. 4.1). Geometric mean (GM) of individual speedup (Eq. 4.2) gives the overall speedup for any configuration.

$$speedup_i = \frac{IPC_i}{IPC_i^{baseline}} \tag{4.1}$$

$$speedup = GM(speedup_{(0 \text{ to n-1})}) \tag{4.2}$$

4.2.3 Comparable Cache Management Policies

HeLM is decoupled from the underlying cache management policy, which brings flexibility to the mechanism as it can be adapted to work with any cache management policy. We choose to implement HeLM over the DRRIP policy, however, it could well be implemented over other cache management policies such as Utility-based Cache Partitioning (UCP) [17]. TAP was proposed with two variants: TAP-UCP and TAP-RRIP, built on top of UCP and DRRIP respectively. Since TAP-RRIP outperforms TAP-UCP in all evaluations, we choose to compare HeLM against TAP-RRIP. In DRRIP policy, incoming cache blocks are inserted at non-MRU position and are promoted later on cache hits. Similar to TAP, we do not promote GPU blocks on LLC hits. Also, when both CPU and GPU blocks are available for replacement, we replace the GPU block first.

Reuse-based Cache Management

Reuse-based mechanisms [55, 56, 57, 58, 59] have been studied extensively in CPU domain to improve cache utilization. Also known as dead-block predictors, they predict whether a cache block is dead or live and improve cache performance by replacing dead blocks first, by bypassing dead blocks, or by prefetching data into dead blocks. Lai et al. [56] propose a dead-block predictor to prefetch data into L1 data cache. While Kharbutli et al. [58] propose counting-based dead-block predictors that consider the number of accesses to a cache block, Cache Bursts [57] observes references to a cache block at MRU position to make dead block prediction.

Since HeLM is based on bypassing LLC for GPU memory accesses, we compare the performance of HeLM with two reuse-based bypass mechanisms, MAT [55] and Sampling Dead-Block Predictor (SDBP) [59]. MAT is an address-based reuse mechanism in which cache block reuse is determined using a *Memory Address Table* (MAT) at a *macroblock* granularity. MAT bypasses a cache block if the victim block has higher reuse than the one being inserted. SDBP, on the other hand, is a PC-based reuse mechanism in which

the reuse pattern is learned from accesses to the cache blocks in a few sampled sets in the LLC. SDBP updates its prediction table using the PC of the last instruction that accesses a cache block in the sampled sets. On an access to the LLC, SDBP predicts the cache block as dead or live by referring to the prediction table, indexed by the PC of the instruction initiating the access. If the block being accessed is dead, it can be bypassed from the LLC.

4.3 Evaluation of HeLM

In this section, we evaluate the impact of HeLM on the effectiveness of shared LLC in a heterogeneous multicore processor. We compare the performance of HeLM with DRRIP, MAT, SDBP, and TAP-RRIP, all normalized to LRU. MAT and SDBP were originally proposed for bypassing CPU memory accesses. However, to compare with HeLM, we employ these techniques to bypass only the GPU memory accesses.

4.3.1 Performance

We start our evaluation with the impact of these policies on the cache performance for CPU and GPU cores. Figure 4.6 shows the reduction in CPU and GPU LLC misses for these policies (normalized to LRU) for 1C4G, 2C4G, and 4C4G workloads. In case of CPU, although all of them perform better than LRU, the improvement for DRRIP and the reuse-based mechanisms (MAT and SDBP) are lower than TAP and HeLM. Additionally, HeLM outperforms TAP. Overall, HeLM reduces CPU LLC misses by 29.5%, 39.1%, and 33% over LRU for 1C4G, 2C4G, and 4C4G workloads, while the corresponding reduction for TAP are 13.9%, 18.1%, and 18.8%. On the other hand, TAP and HeLM increase the GPU LLC misses as they give lower priority to GPU over CPU. While DRRIP does better than TAP and HeLM, MAT and SDBP are the most favorable towards GPU.

For the CPU, cache performance directly translates to speedup as shown in Figure 4.7. Here, HeLM outperforms all other policies convincingly. The lower priority given to the GPU is evident in the speedup of TAP and HeLM. However, the impact of increased

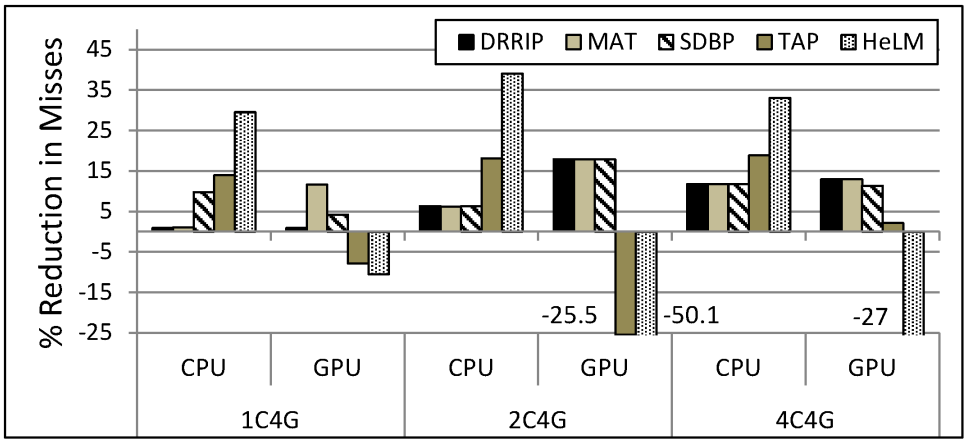


Figure 4.6: Impact of different cache management policies on the cache performance of CPU and GPU benchmarks. Graphs show results for workloads: 1C4G, 2C4G, and 4C4G. Results are relative to the LRU policy.

cache misses do not translate linearly into performance degradation for the GPU. This is due to the difference in cache sensitivity among the CPU and GPU cores. This shows that since the CPU benchmarks are more cache sensitive than the GPU benchmarks, it is preferable to prioritize CPU benchmarks while managing the shared LLC space. In our experiments, the GPU benchmarks in 2C4G workloads show slightly higher performance degradation overall when compared to 1C4G and 4C4G workloads. This is potentially due to the random selection of the workload mix which contains more cache sensitive GPU benchmarks than 1C4G and 4C4G.

Combined speedup for CPU and GPU for all workloads is shown in Figure 4.8. Speedup is calculated as the geometric mean of individual benchmark speedups as in Eq. 4.2. The figure shows that HeLM outperforms all other replacement policies consistently in overall system performance. As the number of CPU benchmarks increases, speedup from HeLM also increases, indicating that HeLM is able to scale with the number of CPU cores in a heterogeneous multicore. Overall, HeLM performs 7.7%, 10.4%, and 12.5% better than LRU for 1C4G, 2C4G, and 4C4G workloads, respectively. The corresponding improvements

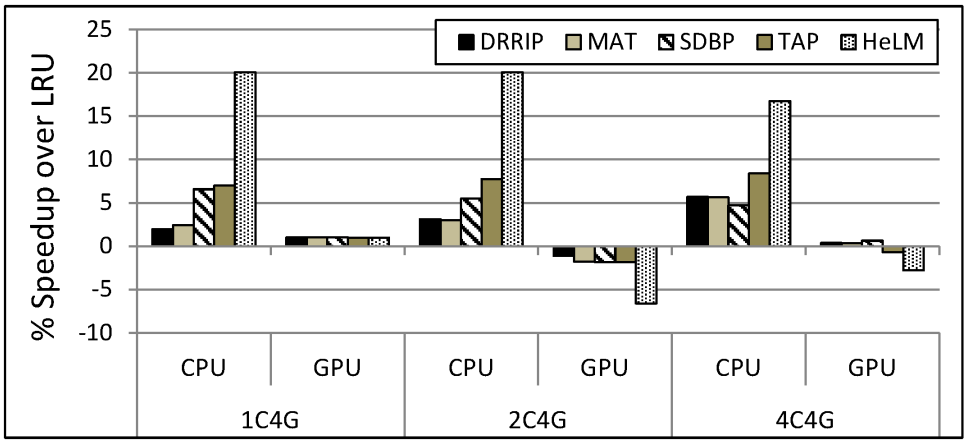


Figure 4.7: Impact of different cache management policies on the speedup of CPU and GPU benchmarks. Graphs show results for workloads: 1C4G, 2C4G, and 4C4G. Results are relative to the LRU policy.

in TAP over LRU are 2.6%, 4.4%, and 6.5%, respectively.

4.3.2 Comparison with Other Policies

Here we discuss the reasons for the performance improvement of HeLM over various cache management policies we evaluate.

DRRIP

The effectiveness of DRRIP in multicore environment is visible in Figures 4.6 and 4.7 as DRRIP outperforms LRU. However, DRRIP faces difficulty in adapting to the heterogeneous characteristics of the cores. DRRIP policy, similar to LRU, does not consider the diversity among the on-chip cores and gives equal priority to both. Therefore, the higher LLC access rate from the GPU cores tends to skew the cache management policy in their favor. Thus, the performance improvement for the CPU core is limited, while GPU cores do not benefit much from the additional LLC space. Hence, the overall speedup for DRRIP in Figure 4.8 is low.

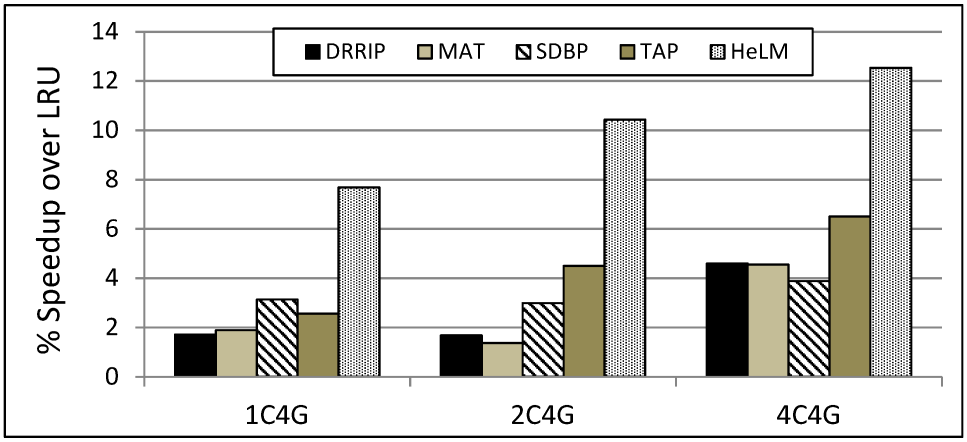


Figure 4.8: Combined speedup for various workloads under different policies. Combined speedup is calculated as the geometric mean of individual benchmark speedups. Results are relative to the LRU policy.

MAT/SDBP

The performance of reuse analysis based policies, MAT and SDBP, is very similar to DRRIP, however for different reasons. These mechanisms, although capable of LLC bypassing of memory accesses, are overly conservative in their approach towards the GPU applications. They detect reuse pattern in GPU memory access behavior and preserve the GPU blocks in LLC. This improves the cache performance of GPU as in Figure 4.6. However, GPU does not benefit significantly from the increased LLC space due to their TLP and the ability to tolerate higher memory access latency. This additional LLC space would have been better utilized had it been provided to the CPU application. Hence, these mechanisms also observe low overall speedup as in Figure 4.8.

TAP

TAP considers the diversity of on-chip cores in optimizing the cache management policy, and improves performance over existing policies by prioritizing CPU over GPU. However, HeLM still outperforms TAP for two reasons:

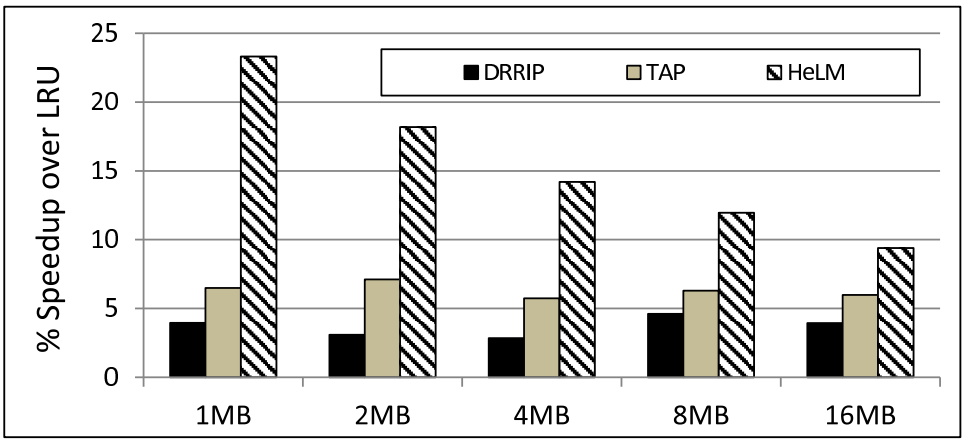


Figure 4.9: Performance of 4C4G workloads with varying cache sizes. Result are relative to the LRU policy.

(i) the *core sampling* technique used by TAP leaves a significant portion of the shared LLC to be occupied by the GPU cores. Majority of these blocks originate from the GPU core that inserts at the MRU position in the core sampling technique, and end up being dead blocks. In our experiments with 1C4G workloads, we observe that nearly 40% of the shared LLC space is occupied by the GPU dead blocks that were inserted at the MRU position. This leads to eviction of useful CPU blocks, leaving significant room for improvement. Since HeLM does not suffer from this side effect, it performs better than TAP.

(ii) TAP takes a binary decision on whether the GPU application is cache sensitive or not. This decision is then used to override the underlying policy for all the accesses in the sampling period. However, such a binary decision is at a coarse granularity, while a more fine-grained ability to control the LLC share between CPU and GPU could potentially improve the performance of both the cores. HeLM is able to control the cache occupancy of the GPU core at a finer granularity, by taking bypass decision for each GPU access, which also helps in outperforming TAP.

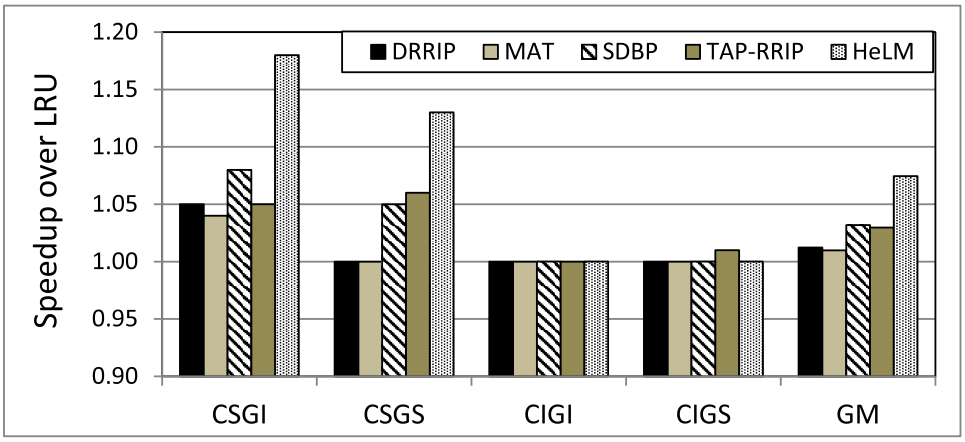


Figure 4.10: Speedup for 1C4G workloads category-wise. Result are relative to the LRU policy.

4.3.3 Sensitivity to Cache Size

Figure 4.9 presents the sensitivity of HeLM to varying LLC sizes for 4C4G workloads. To configure different LLC sizes we vary the LLC associativity. As shown in the figure, HeLM outperforms other policies for all cache configurations. Although the performance benefits of HeLM is more evident with smaller LLC size, it is able to preserve its benefits with increasing LLC sizes. This shows that HeLM can adapt well to variations in cache configurations.

4.3.4 Workload Types

Mixing of CPU and GPU applications in a heterogeneous multicore processor creates workloads with unique characteristics. To evaluate the potential opportunities in these workloads, we broadly classify them based on their cache sensitivity, resulting in four different categories:

- *CPU cache Sensitive, GPU cache Insensitive* (CSGI): CSGI is perhaps the most common category in which GPU occupies majority of the LLC space leading to poor

performance of the CPU application in existing cache policies.

- *CPU cache Sensitive, GPU cache Sensitive* (CSGS): Although GPU is cache sensitive in this combination, it has the advantage of high TLP. Hence, any additional LLC space given to CPU could bring a larger overall performance improvement.

- *CPU cache Insensitive, GPU cache Insensitive* (CIGI):

- *CPU cache Insensitive, GPU cache Sensitive* (CIGS): These categories do not leave significant room for performance improvement as the CPU applications are cache insensitive.

Figure 4.10 presents the speedup for the evaluated policies, over LRU, for the four categories. As expected, all the policies show performance improvement over LRU in the first two categories (CSGI, CSGS) where CPU is cache sensitive. Also, we can observe that HeLM outperforms all the other policies in these categories, particularly by a significant margin for CSGI which is the most common category. In the last two categories (CIGI, CIGS), there is hardly any performance improvement over LRU for any of the policies.

4.4 Energy Consumption

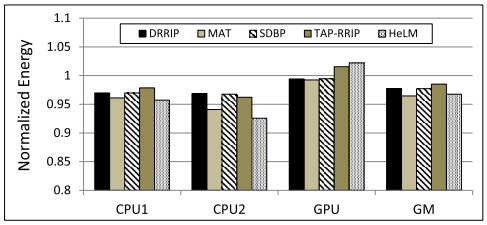
HeLM's LLC bypassing technique could alter the energy consumption profile of the system, both on-chip as well as off-chip. On-chip, there are two scenarios contributing to LLC dynamic energy: (i) on an LLC hit, tag array and cache block data are accessed; and (ii) on an LLC miss, tag array is accessed, a cache block is written back to memory if it is dirty, and data for the missed access is written to the cache block. When an LLC access is bypassed on a miss, only the tag array is accessed, eliminating the energy consumption of data block accesses. Additionally, dynamic access energy of the memory controller could also be altered by LLC bypassing because of the changes in off-chip access requests. Static

energy, on the other hand, is dependent on the total execution time, and is hence related to the performance of the policy.

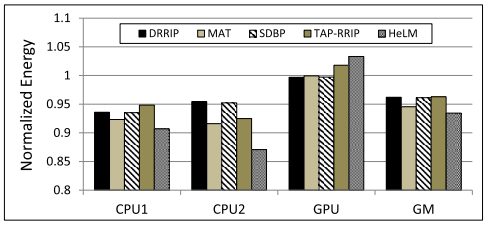
Here, we discuss the energy consumption of our baseline 2C4G configuration. The energy consumption shown here for each of the policy and configuration is normalized to the LRU policy. Figure 4.11(a) shows the on-chip dynamic energy consumption (cores, LLC, memory controller) for the various policies. HeLM improves the on-chip dynamic energy consumption by about 2% compared to TAP. Figure 4.11(b), on the other hand, shows the on-chip static energy consumption (cores, LLC, memory controller). Here also, HeLM improves the overall on-chip static energy consumption due to the performance improvement. Overall, Figure 4.11(c) shows the combined (static + dynamic) on-chip energy for the 2C4G configuration. HeLM consumes about 2% less on-chip energy than TAP.

LLC bypassing, on the other hand, could lead to an increase in off-chip main memory (DRAM) accesses, resulting in a potential increase in DRAM dynamic energy consumption. Figure 4.12 shows the DRAM energy consumption for 2C4G workloads. HeLM increases the DRAM energy consumption when compared to DRRIP, MAT and SDBP. However, its DRAM energy consumption is comparable with TAP, as TAP also increases DRAM energy consumption due to the increase in DRAM access rate as a result of the low priority given to GPU memory accesses.

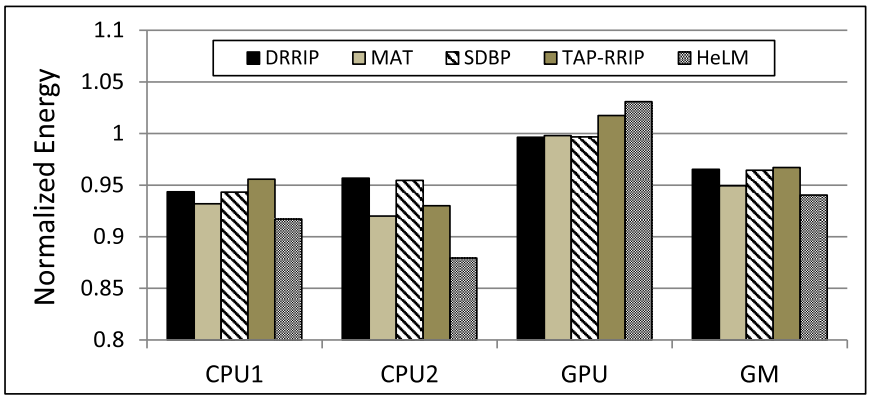
Figure 4.13 shows the overall system (on-chip + DRAM) energy consumption for the 2C4G configuration. Here, we see that HeLM's energy consumption is similar to DRRIP and MAT, and infact lower than TAP and SDBP. HeLM outperforms TAP in energy consumption by about 3%. This shows that energy consumption is not a major concern for the bypassing based HeLM.



(a) On-chip dynamic energy consumption



(b) On-chip static energy consumption



(c) On-chip total energy consumption

Figure 4.11: On-Chip energy consumption for the 2C4G configuration. Figures show the energy consumption for the two CPU applications, the GPU application, and the system average (geometric mean). The energy consumption for each policy is normalized to the energy consumption for LRU policy.

4.4.1 Energy Delay-Squared Product

Energy efficiency has emerged as an important metric in architectural evaluations. Several works [5, 60, 61] have turned to energy delay-squared product (ED^2) [62] as the metric of choice to study energy efficiency as it considers both energy consumption and performance in determining the efficiency of a processor. In this section, we evaluate HeLM on the basis of the energy delay-squared product metric. For a multiprogrammed workload such as 2C4G, however, calculating ED^2 gets tricky. Below we describe how we calculate the ED^2 metric.

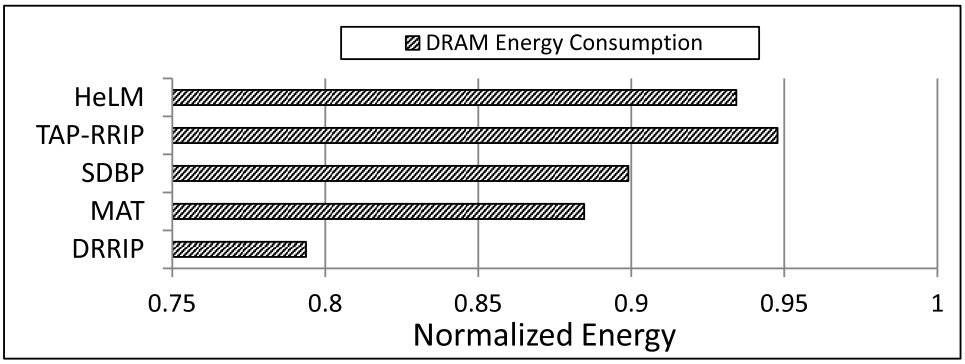


Figure 4.12: DRAM energy consumption for the different policies for the 2C4G configuration. The energy consumption for each policy is normalized to the energy consumption for LRU policy.

For our baseline 2C4G workload, ED^2 for each CPU application and the GPU application is calculated separately as they experience different execution times and performance improvements. Energy consumption for each core is calculated separately, while energy consumed by shared resources, such as LLC and memory controllers, is pro-rated for each application on the basis of execution time (static energy) or access numbers (dynamic energy).

Once the ED^2 value is calculated for each application and policy, for each core, normalized ED^2 is calculated for each policy as shown in Eq. 4.3. Geometric Mean (GM) is then applied on this value for all applications to calculate normalized ED^2 for the core and the policy as shown in Eq. 4.4. GM is applied once more, as shown in Eq. 4.5, to calculate normalized ED^2 for the policy for the entire system.

$$normalized\ ED^2{}_{(core,application,policy)} = \frac{ED^2{}_{(core,application,policy)}}{ED^2{}_{(core,application,LRU)}} \tag{4.3}$$

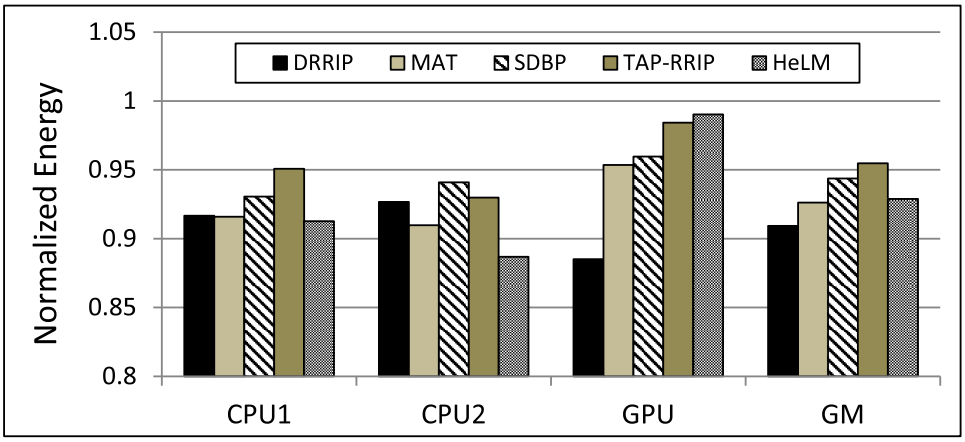


Figure 4.13: Total system energy consumption for the different policies for the 2C4G configuration. The energy consumption for each policy is normalized to the energy consumption for LRU policy.

$$normalized\ ED^2{}_{(core,policy)} = GM(normalized\ ED^2{}_{(\text{core,application=(0 to n-1),policy})}) \tag{4.4}$$

$$normalized\ ED^2{}_{(policy)} = GM(normalized\ ED^2{}_{(\text{core=(0 to m-1),policy})}) \tag{4.5}$$

Figure 4.14 shows the normalized ED^2 for all the policies for 2C4G configuration. We see that for the CPU applications, HeLM's ED^2 value improves significantly compared to other policies, due to the performance improvement. For the GPU application, ED^2 worsens as we sacrifice GPU performance for overall performance improvement. However, we can note that HeLM's ED^2 is only slightly worse than that for TAP. Overall, HeLM

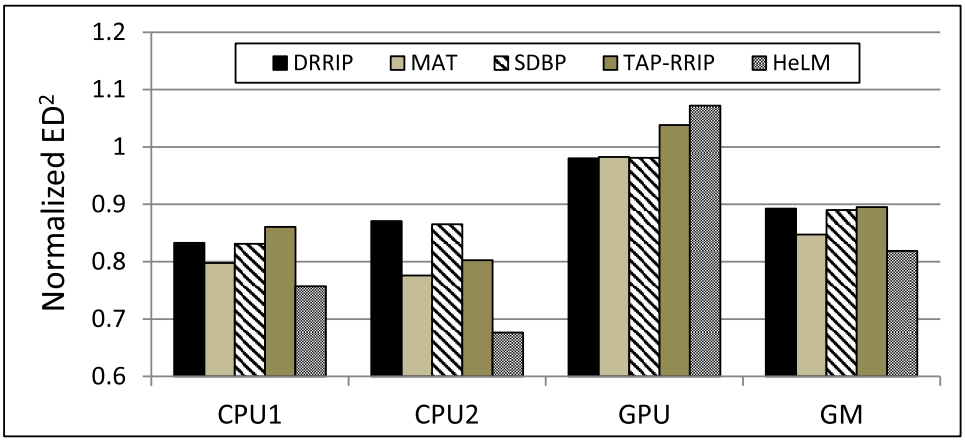


Figure 4.14: Total system energy delay-squared (ED^2) product for the different policies for the 2C4G configuration. The ED^2 for each policy is normalized to the energy consumption for LRU policy.

performs better than other policies in terms of ED^2. It does nearly 8% better than TAP. These evaluations have shown that HeLM fares better than other policies from the energy efficiency perspective.

4.5 Hardware Overhead

Table 4.6 presents the hardware overhead for various cache management policies, including HeLM. While MAT and SDBP require significant amount of storage to track the reuse of GPU LLC blocks, TAP and HeLM can be implemented using simple hardware counters. HeLM utilizes MissHigh, and MissLow counters to track GPU and CPU LLC access behavior. Also, instruction counters and core IDs are required to identify the cache sensitivity of the GPU application every sampling period. The TLP threshold register holds the TLP threshold selected by TSA. In summary, hardware overhead for HeLM is comparable to that for TAP, and both these mechanisms use significantly less additional hardware than MAT or SDBP.

Policy	Hardware	Overhead
MAT [55]	4K-entry memory address table (each entry: 20-bit tag, 8-bit counter, 1 valid bit)	14.5 KB
SDBP [59]	4K-entry x 3 prediction tables (each entry: 2-bit counter), sampler sets	13.7 KB
TAP [45]	Instruction counters (20-bit x 4 GPU cores), core IDs (10-bit x 4 LLC tiles)	120 bits
HeLM	MissHigh and MissLow counters (20-bit each), TLP threshold register (6-bit), instruction counters, core IDs	166 bits

Table 4.6: Hardware overhead for various cache management mechanisms.

4.6 Summary

The growing importance of data-parallel accelerator cores, such as GPU, has lead to their integration with general purpose CPU cores on the same die. Such architectures with heterogeneous processing cores present a significant challenge to optimal sharing of on-chip resources such as the LLC. Our heterogeneous LLC management mechanism, HeLM, monitors the TLP available in the GPU application, and uses this information to throttle the GPU LLC accesses when the application has enough TLP to sustain longer memory access latency. This in turn provides an increased share of the LLC to the CPU application, thus improving its performance. HeLM monitors the cache sensitivity of both CPU and GPU applications in the heterogeneous workload, and achieves LLC sharing that improves overall system performance and energy efficiency.

We evaluate HeLM against: (i) existing shared LLC management techniques (LRU, DRRIP); (ii) reuse-based bypassing mechanisms (MAT, SDBP); and (iii) the only current technique proposed for heterogeneous multicore processor (TAP). HeLM outperforms all these mechanisms in overall system performance. HeLM improves over LRU policy by 10.2% and outperforms TAP by 5.7% for the baseline processor configuration with two

CPU and four GPU cores. HeLM also outperforms these policies in energy consumption and energy efficiency. HeLM consumes nearly 3% less total energy compared to TAP, while reducing ED^2 value by 8%, for the baseline configuration. HeLM scales well with varying processor count and performs consistenty in terms of both performance and energy efficiency.

Chapter 5

Parallel Execution Correctness

With the increased availability of on-chip parallelism in modern multicore processors, programmers are actively parallelizing applications from diverse domains to take advantage of the abundant computing power at their disposal. However, ensuring the correct execution of parallel applications is challenging due to the difficulty in tracking concurrency bugs [36, 37]. Data race is one of the main classes of concurrency bugs. When two threads access the same memory location without a separating synchronization, with at least one of the accesses being a write, there is a data race. A data race could lead to incorrect or unexpected program behavior, and is a potential security risk.

An example of a data race is shown in Figure 5.1, where *thread0* and *thread1* access the same memory location *addr2*, without an intervening synchronization operation, and one of the instructions is a write. This is a potential concurrency bug as thread0 could modify the value at address addr2 before its next intended use by thread1. Data races can be divided into three categories: (i) read-after-write (RAW); (ii) write-after-read (WAR); and (iii) write-after-write (WAW). A WAR data race condition is shown in Figure 5.1. Not all data races are hazardous or potential security risks; some of them could be benign. However, it is essential for data race detectors to identify and evaluate all potential data race conditions.

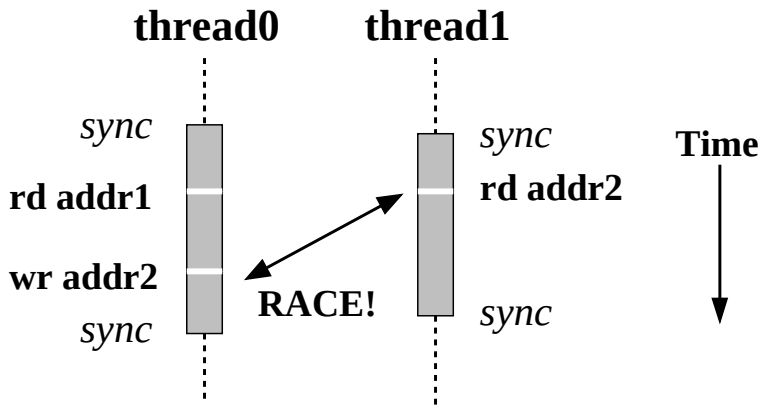


Figure 5.1: An example of a data race, on address *addr2*, between *thread0* and *thread1*. Figure shows the synchronization and memory instructions in the two threads. Synchronization instructions define the beginning and end of *epochs*. Shaded boxes mark epochs in the instruction stream.

5.1 Data Race Detection

There are two primary classes of data race detection algorithms: *lockset* [29, 27] and *happened-before* [28, 26]. Lockset-based schemes track the set of locks held by threads while accessing a shared variable, and report a data race when the accesses are not protected by common locks. Happened-before (H-B) algorithm is based on Lamport's happened-before relation [63]. In this scheme, memory accesses between synchronizations are grouped into *epochs*. Epochs belonging to different threads are *concurrent* only if their execution times overlap.

Figure 5.2 shows how epochs and concurrency is defined in H-B algorithm. H-B schemes compare memory accesses in concurrent epochs and report a data race condition if they contain accesses to the same memory location with at least one of the access being a write. Lockset-based schemes do not track synchronizations other than locks, whereas, H-B scheme covers all types of synchronizations and hence can potentially detect more data races. For this reason, H-B algorithm has found wider applicability in data race detection mechanisms.

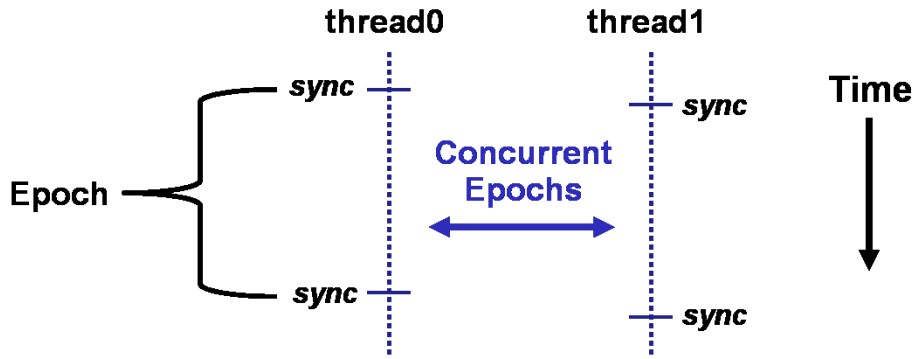


Figure 5.2: Epochs and concurrency in happened-before algorithm. Epoch is the execution stream between two synchronization operations. Two epochs are concurrent if their execution times overlap.

5.1.1 Challenges in Data Race Detection

Being able to detect data races efficiently at runtime is essential to assure correctness and reliability of parallel programs.

Runtime Support for Data Race Detection

There exists a large body of work for identifying data races offline, either through static analysis [64] or by post-mortem analysis [65]. Static analysis based approaches analyze the source code to detect data race conditions and experience high false positive rates due to their conservative nature. Post-mortem methods that collect and analyze the execution trace of applications have significant storage overhead and cannot identify potential security risks in real-time. Due to these drawbacks, current data race detection techniques emphasize on runtime support.

Implementation of Runtime Techniques

Several runtime techniques have been proposed for data race detection utilizing the algorithms discussed. They can be divided into software-based and hardware-based techniques.

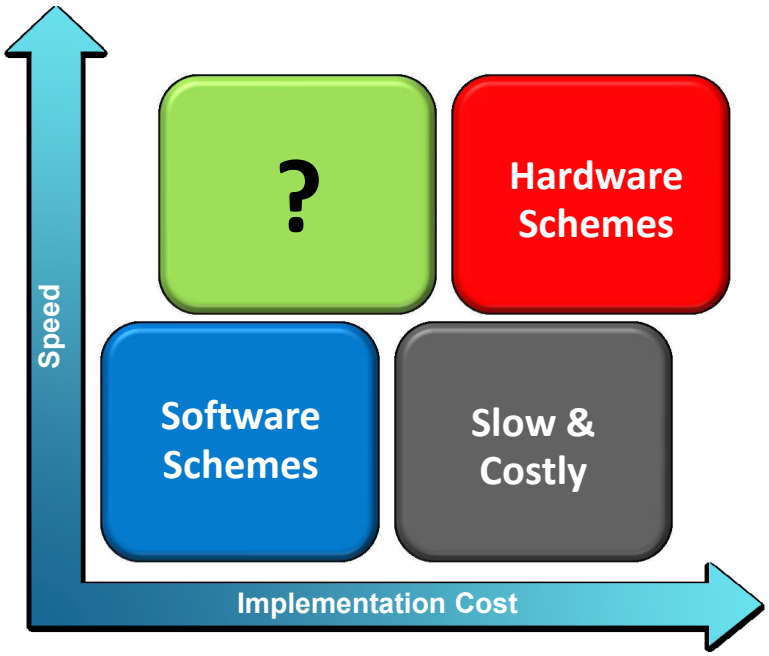


Figure 5.3: Classification of data race detection techniques based on performance and cost of implementation.

Software-based data race detection tools [28, 29], although cost-effective to implement often slow down the monitored application by orders of magnitude, thus, limiting their applicability. Although hardware-based data race detection tools [26, 27] inflict a near-zero performance impact on the monitored application, they often have a significant implementation overhead that limits their scalability.

Figure 5.3 shows the classification of various data race detection techniques based on their cost and performance. In this work, we aspire to build a data race detection tool that achieves the performance of hardware-based mechanisms with minimal implementation overhead. To achieve this goal, we explore the resources available on-chip a heterogeneous multicore processor.

5.2 GPU Accelerated Data Race Detection

Integration of data-parallel accelerator cores with CPU cores on the same chip has emerged as a new trend to facilitate energy-efficient computing with diverse cores. The AMD Fusion APU [8] and Intel Sandy Bridge [9] have become part of mainstream computing. The data-parallel cores in these designs can support a significant number of parallel threads providing computing power needed for executing data race detection algorithms efficiently.

In this chapter, we design and evaluate an efficient and scalable CPU data race detection mechanism utilizing data-parallel accelerator cores available on-chip current heterogeneous multicore processors. When these cores are not being employed for performance acceleration, we propose to utilize them for detecting data races in the application executing on the CPU cores. Without loss of generality, we consider the Graphics Processing Unit (GPU) as the data-parallel accelerator in our proposal. In the following sections, we refer to our design as GUARD which stands for GPU Accelerated Data Race Detector [66].

5.2.1 GUARD Overview

A snapshot of the basic GUARD mechanism is shown in Figure 5.4. The heterogeneous architecture we model consists of CPU and GPU cores sharing on-chip resources through a common on-chip interconnection network (ICNT). Solid lines with double arrows indicate data communication paths between the cores and the caches, through the interconnection network. Dotted lines indicate the flow of data race detection related information in GUARD. Features of the heterogeneous multicore processor modeled here are discussed in Section 5.3.

GUARD utilizes minimal hardware support to improve the performance of data race detection. One of the primary tasks of a data race detection mechanism is to extract the memory access trace of the application being monitored. In GUARD, the memory access trace generation is orchestrated by a dedicated hardware component we refer to as the *Memory Trace Extractor* (MTE).

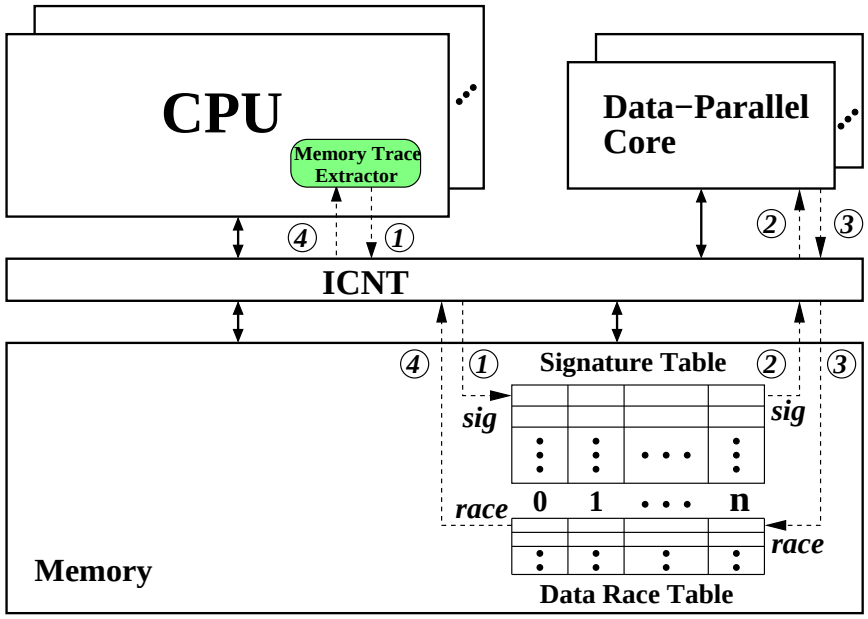


Figure 5.4: GUARD mechanism is based on a heterogeneous multicore processor with CPU cores and Data-parallel accelerator (GPU) cores. *Memory Trace Extractor*, the only hardware modification to the baseline processor, is highlighted.

When GUARD is enabled for data race detection, a library function is invoked. It creates two data structures, the *signature table* and the *data race table*, in the GPU memory space. The MTE is configured with the starting addresses of these tables. Henceforth, the MTE is able to write generated signatures to the signature table and read flagged data race conditions from the data race table. It then launches the GPU kernel that performs the happened-before algorithm. In GUARD, the GPU cores work in tandem with the CPU cores to detect data races. We describe the CPU-side actions (① and ④ in Figure 5.4) and GPU-side actions (② and ③ in Figure 5.4) in detail in the next two sections.

5.2.2 CPU-Side Actions

The primary task at the CPU side is the extraction of memory access information from the application being monitored. We call this step: *memory trace extraction*. However, the

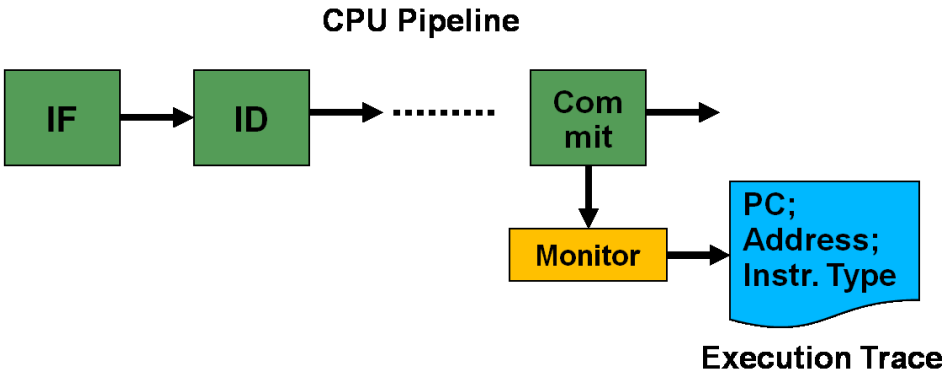


Figure 5.5: High level view of an instruction-grain program monitor.

volume of the trace generated at runtime makes it intractable to be processed in real-time, even with GPU. Hence, extracted memory needs to be compressed. We call this step: *memory trace compaction*. In the next two sections, we describe these two tasks.

Memory Trace Extraction

Instruction-grain program monitors have been proposed to efficiently extract runtime information from the CPU. These tools monitor programs at an instruction-level granularity and collect information such as program counter, instruction type, input/output operands, and access addresses. Such monitors have been used for specialized purposes such as memory checking, security tracking, and taint analysis [67, 29, 27]. Runtime data race detection requires extraction of memory access information from the CPU cores while the parallel applications are executing. General purpose instruction-grain program monitors such as Log-based Architecture [68] can efficiently extract runtime information from the CPU without significant hardware modifications.

Figure 5.5 shows the high-level view of an instruction-grain program monitor. Here, a small hardware module snoops the commit stage of the pipeline to extract instruction-level information. Previously, we have proposed utilizing hardware support for extracting

runtime information for dynamic program execution monitoring [67]. GUARD utilizes a similar hardware *extraction logic* that tracks the program execution and extracts the execution trace of the CPU application being monitored. GUARD's *Memory Trace Extractor* is build on top of such previously proposed instruction-grain program monitors.

Memory Trace Compaction

Memory trace generated by each CPU core is partitioned into chunks called *epochs*. Synchronization instructions, such as lock/unlock, barriers, etc. define epochs. For efficient communication and computation, all the addresses belonging to an epoch are encapsulated into representative signatures using Bloom Filters [69] and H3 hash functions [70]. For each epoch, the MTE generates two signatures: a read (RD) and a write (WR).

Once the signatures are generated, they are written to the signature table (action ① in Figure 5.4) stored in the GPU memory space. The signature table contains signatures from all CPUs, and forms the input to the H-B algorithm running on the GPU. It is a circular queue structure where the oldest *processed* entry for each processor is over-written by the latest entry. A flag is maintained for each signature entry indicating whether the entry has been processed by the GPU or not. The MTE refers to this flag before the entry is over-written with a new signature, and resets the flag when a new entry is written.

The potential speed difference between the CPU application and the GUARD kernel means that the CPU could retire instructions faster than GUARD's ability to process them. This could lead to GUARD missing some instructions and consequently missing data race conditions. To avoid this, we design the MTE on a feedback-based architecture where the CPU retire stage and the MTE communicate through special registers. When GUARD is enabled, the CPU retire stage checks the MTE state through the special registers and if MTE is stalled, CPU pipeline is stalled to avoid missing any races. We evaluate the impact of this design on the performance of the CPU application in Section 5.4.

Signature Selection: Signatures are long bit vector structures used to encapsulate

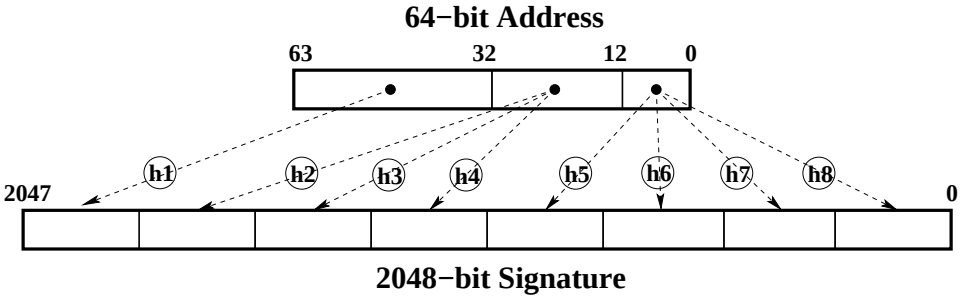


Figure 5.6: Signature formation: A 2048-bit signature is divided into 8 bins and 8 different hash functions are applied on the 64-bit address to set the signature bits.

addresses in the memory access trace in a compressed form. Figure 5.6 shows the signature creation process used for a signature of size 2048-bits, divided into eight bins. The 64-bit address in the extracted memory access instruction is divided into three sections and these sections are passed through eight different H3 hash functions, *h1* through *h8*, and a particular bit is set in each of the signature bins. Two signatures indicate a potential data race only when all the eight bins have at least one common bit set. Here, we analyze the effect of various signature parameters on its false positive rate. A *false positive* is defined as an incorrectly flagged data race condition due to two unique addresses mapping to the same signature bits.

We observe that the false positive rates are 18.78%, 37.88%, and 89.86% for 2048-bit, 1024-bit, and 512-bit signatures respectively. Use of hardware signatures in data race detection has been explored by previous works [26, 27] and the false positive rates we observe are similar to the rates observed by these works. This false positive data is based on epochs that could contain up to 2000 individual instructions. Higher instruction count inside an epoch will lead to higher false positive rate for signatures of the same length. Ideally, an epoch is closed by a synchronization instruction. However, if there are no synchronization instructions within 2000 instructions, we forcibly close the epoch and write the signature to the signature table. This is a practical design choice as data race

conditions between memory accesses that execute close to each other in time are the most critical, while those which occur far apart in time are potentially benign data races.

Signature Table Size: The difference in frequency of instructions in different cores could mean that an epoch in one core needs to be H-B compared with a significantly large number of concurrent epochs from another core. This, in turn, means that the H-B scheme would need a significantly large number of entry slots in the signature table to perform ideal data race detection without missing any concurrent epochs. However, in practical systems such as GUARD, we cannot afford such a large signature table. In addition to the larger signature table size, this will also lead to a greater performance penalty, as the data race detector will have to perform a significantly higher number of H-B signature comparisons. Limiting the number of entries in the signature table, on the other hand, inevitably leads to missing the comparison of some concurrent epochs; a parameter we refer to as *Missed Epoch Comparisons*.

In our experiments, we evaluate the missed epoch comparisons for a 16-entry and a 64-entry signature table compared with an ideal signature table with an infinite number of entries. We observe that the 16-entry signature table misses 3.16% of epoch comparisons, while the 64-entry signature table misses 0.12% of epoch comparisons, versus the ideal signature table. Since the 64-entry signature table incurs a significantly higher performance overhead compared to the 16-entry signature table, for a small improvement in missed epoch comparisons, we chose to evaluate GUARD with a 16-entry signature table.

Size of the signature table grows linearly with the number of CPU cores monitored and the number of signature entries. Even for a small core count and number of signature entries, this is high overhead to be constructed as a dedicated on-chip hardware structure. For example, a four-core CPU with 2048-bit signatures and 16 signature entries has a signature table size of 32 kilo bytes [26]. GUARD stores the signature table in the GPU last-level cache (LLC), without any additional hardware overhead. GUARD shares the LLC space with other GPU applications and hence the space is reusable. Designs that

store the data race detection related information as an extension of the cache line, such as HARD [27], suffer from lost detection opportunities when the lines are evicted. GUARD does not suffer from this limitation as the signatures are not based on information in cache line extension.

Post Detection Actions

Once a data race is detected, the related information is written to the data race table by the GPU kernel and a notification is sent to the CPU in the form of an exception. An appropriate response such as rollback or replay is then initiated (action ④ in Figure 5.4). GUARD can utilize existing record/replay mechanisms [71] to perform this step. Efficient checkpointing systems such as Revive [72] can create checkpoints with low overhead. An appropriate checkpoint for rollback or replay is selected using information from the data race table. Further analysis could include detailed debugging to find out the exact memory location and instructions responsible for the data race. Information from the data race table and checkpoints could also be used to modify the thread scheduling to avoid the occurrence of data race conditions in re-execution.

5.2.3 GPU-Side Actions

Due to its wider applicability, compared to lockset technique, we use happened-before (H-B) technique as the baseline algorithm in GUARD.

Data Race Detection Algorithm

The happened-before comparisons of the signatures generated by the MTE is performed by a GPU kernel. In a nutshell, each signature from a particular CPU thread is compared with each concurrent signature from all other CPU threads. If the intersection of the concurrent signatures is not *NULL*, we have a potential data race condition. Figure 5.7 shows the signature table with entries for an n-core CPU.

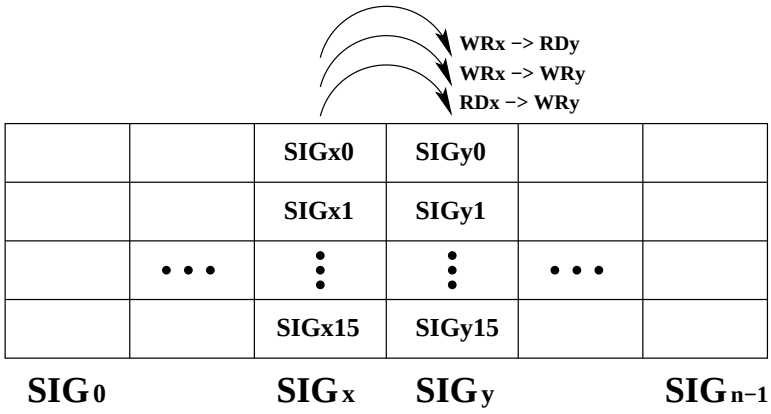


Figure 5.7: Structure of the signature table for an n-core CPU. Curved arrows indicate the three required comparisons in H-B algorithm between every concurrent signatures of the CPUs X and Y.

Entries for processors CPUX and CPUY are marked. Each entry in the signature table, say SIGX0, consists of a read (RDX) signature and a write (WRX) signature along with the epoch start (TS1) and end (TS2) timestamps. Since a read-after-read access is not potentially harmful, we have to compare only three signature combinations for CPUs X and Y: RDX-WRY, WRX-RDY, and WRX-WRY. These combinations are indicated in the figure. This signature comparison is extremely parallel and we utilize the data-parallel architecture of GPU to perform these comparisons.

Algorithm 2 shows the GPU kernel algorithm for GUARD. The GPU threads monitor the signature table for new incoming signatures. Once a new signature entry (SIGX) is identified, the GPU thread iterates through each of the current SIGY entries present in the signature table. Potential data race is identified if the *intersection* of concurrent signatures is not *NULL*. Timestamp information embedded in the signature is used to test the concurrency of these signatures using the function *concurrent*(). Bitwise AND operation is used to efficiently calculate the intersection of the signatures. When a data race condition is identified, information related to the race condition such as thread and epoch numbers is written to the data race table. Once the GPU thread has iterated through all the current

```
Data: Memory access signatures (SIG).
Result: Flagged data race conditions.
while (1) do
    if isNew(SIGX) then
        for each current SIGY do
            if concurrent(SIGX,SIGY) && (SIGX ∩ SIGY ≠ NULL) then
                Flag data race condition;
            end
        end
        mark SIGX for graduation;
    end
    __gpu_sync();
end
```

Algorithm 2: The GPU kernel for the H-B comparison between CPU cores X and Y. SIGX and SIGY corresponds to signature from CPUX and CPUY. Custom kernel synchronization function __gpu_sync() and function concurrent() are discussed in Section 5.2.3.

SIGY signatures, it marks SIGX for graduation and moves on to the next SIGX. The signature marked for graduation can now be overwritten by CPUX with new signature.

GPU Kernel Synchronization

The GPU kernel synchronizes all the threads after the comparison of the current SIGX with all the present SIGY entries, using a custom synchronization function *__gpu_sync*(). The current SIGX is then graduated before each thread moves to a new SIGX. This *lock-step* behavior ensures correctness of signature data accessed by GPU threads by avoiding untimely overwriting of SIGX by CPUX. Since GUARD's GPU Kernel could utilize several thread blocks, spread across multiple SMs, it is essential for __gpu_sync() to be able to synchronize across SMs. While the CUDA library function __syncthreads() [34] can only synchronize among threads in a block, __gpu_sync() utilizes a global mutex variable and atomic operations to synchronize among multiple SMs. __gpu_sync() is inspired by the GPU Lock-based Synchronization discussed by Xiao and Feng [73].

GPU Kernel Parallelization

We map the H-B algorithm to the GPU in the following way: each GPU thread is assigned to perform signature comparison between two CPU threads X and Y. Each thread is also assigned a particular signature combination, among RDx-WRy, WRx-RDy, or WRx-WRy. The H-B algorithm is highly parallel. To improve the performance of GUARD, we parallelize the GPU kernel at different levels:

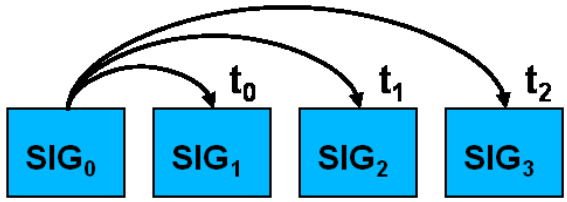
- For comparison of CPU_0 with, say, CPU_1, CPU_2, CPU_3, three different set of GPU threads are utilized as shown in Figure 5.8(a). We refer to this as *core-level* parallelization.

- The current signature of CPUx could be compared with all 16 signatures of CPUy in parallel as shown in Figure 5.8(b).We refer to this as *signature-level* parallelization. We evaluate three signature-level parallelization (*throttling*) schemes: *full, half,* and *quart*. In *full* throttle, 16 different GPU threads are used to H-B compare the current signature of CPUx with the 16 signatures of CPUy in parallel. *Half* and *quart* throttle, on the other hand, use 8 and 4 GPU threads, respectively.

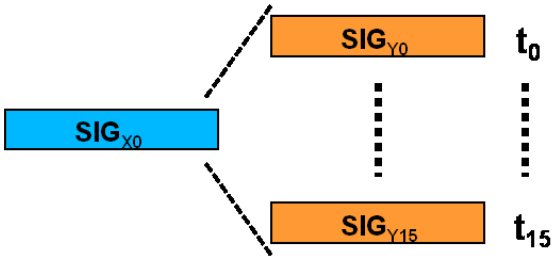
- We read the 2048-bit signatures in chunks of 64-bit UNSIGNED INTEGER data type for bitwise AND calculations to perform the intersection operation of the H-B algorithm. To further parallelize GUARD's GPU kernel, we utilize different threads to perform the bitwise AND calculations on the different chunks of the same signature. This is shown in Figure 5.8(c) and is referred to as *chunk-level* parallelization.

5.3 Evaluation Infrastructure

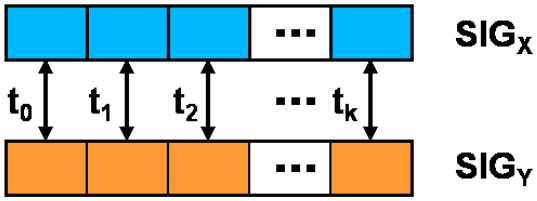
In spite of the recent heterogeneous designs [8, 9] that are already in the market, the optimal design of a multicore CPU with on-chip data-parallel cores is still unclear. The memory hierarchy design and shared memory consistency models are ambiguous and the



(a) Core-level parallelization.



(b) Signature-level parallelization.



(c) Chunk-level parallelization.

Figure 5.8: Parallelizations for the GPU kernel in GUARD.

programming model is still in its nascent state. Nevertheless, such designs provide a suitable infrastructure to off-load the task of CPU data race detection to on-chip accelerator cores. In this work, we utilize a generic execution model and propose a data race detector inspired by these designs.

Heterogeneous Execution Environment

We utilize a heterogeneous multicore processor, consisting of CPU and GPU cores on the same die, as shown in Figure 5.4. The cores share on-chip resources and are connected through a common on-chip interconnection network. Communicating through the shared on-chip interconnection network improves the efficiency of GUARD. These cores work on

CPU		GPU	
Cores	4 / 8 / 16 / 32	Warp Size	32
Frequency	2600MHz	Frequency	1300MHz
Pipeline Width	4	SIMD Pipeline Width	8
L1 Cache (Size/Assoc/Line)	32KB / 2 / 64B	L1 Cache (Size/Assoc/Line)	32KB / 2 / 64B
L2 Cache (Size/Assoc/Line)	2MB / 4 / 64B	L2 Cache (Size/Assoc/Line)	512KB / 4 / 64B
RoB / IW Size	64 / 32	Shared Memory per Core	16KB
MSHR / TLB Entries	256 / 64	Threads / Registers per core	1024 / 16384
L2 / DRAM Access Latency	6 / 200 Cycles	Memory Channels	8

Table 5.1: System configuration parameters for the heterogeneous CPU-GPU evaluation infrastructure for GUARD.

different address spaces and hence we do not consider the complexities of coherence between CPU and GPU cores in our design. We base our evaluation on a GPU SM, with 8 SPs, that can each support up to 1024 threads. This is modeled on Nvidia Geforce® 8600 GTS. Various parameters of the CPU and GPU cores simulated are given in Table 5.1.

To simulate multicore CPU in detail, we use Simics [74] combined with GEMS [75]. The GPU cores are simulated using GPGPU-sim [76]. The on-chip interconnection network is simulated using Garnet [77]. GUARD GPU Kernel is compiled using CUDA 2.3 [34]. We evaluate GUARD with applications from two widely used benchmark suites: PARSEC [78] and SPLASH-2 [79]. Our evaluation reports data from 15 programs in total: seven PARSEC and eight SPLASH-2 programs as indicated in Table 5.2. The evaluated PARSEC benchmarks are: BLACKSCHOLES, BODYTRACK, CANNEAL, FLUIDANIMATE, FREQMINE, STREAMCLUSTER, and SWAPTIONS. The evaluated SPLASH-2 benchmarks are: BARNES, CHOLESKY, FFT, LU, OCEAN, RADIOSITY, RAYTRACE, and WATERNS.

Using Simics and GEMS, we simulate a many-core system with Sun Microsystem's UltraSPARC® III processor running Solaris® 8 operating system. All the benchmark programs are written in C/C++ and parallelized using either OPENMP or PTHREADS. They are compiled using GCC 4.5.2 at -O3 optimization level. The reported results are based on running the selected benchmarks for 1 billion instructions in total from the start of their respective parallel sections, also known as the *region of interest*. Full system

simulation is extremely time consuming, and therefore it is practical to simulate 1 billion instructions in the region of interest. We observe that GUARD's ability to detect data race conditions and its performance characteristics are comprehensively evaluated by simulating 1 billion instructions in the region of interest.

Cycle accurate simulators are utilized to evaluate the performance impact of GUARD on the CPU application being monitored. GUARD GPU kernel invocations, data transfer operations and signature comparison operations are simulated in a cycle accurate manner. We enable L1 data cache in the GPU to improve the performance of GUARD kernel. Potentially, we could also make use of the GPU shared memory to store the signature table. However, we utilize the L1 data caches as the access times are similar. Shared memory in the GPU is explicitly managed by the programmer and when the signature table is updated at regular interval by the CPU, copy of the signature table in shared memory will also need to be manually updated. This will prove to be an additional overhead when using GPU shared memory.

5.4 Evaluation

This section performs a detailed evaluation of the effectiveness of GUARD. First of all, we look at the effectiveness of our scheme in detecting data races. Then, we move on to the performance characteristics of GUARD. We also discuss the performance-accuracy trade-off achievable, given the limited on-chip resources available.

Table 5.2 shows the number of data races GUARD detects. GUARD is based on the happened-before principle that has been used by prior work such as SigRace [26], and thus is expected to capture the same set of data races. It is worth pointing out that similar to SigRace, GUARD does not capture all potential data races. The set of data races captured are only those that lead to violation of happen-before principal at runtime. GUARD works at address level granularity and hence each data race reported corresponds to a unique address. The ability to detect actual data races proves the effectiveness of GUARD. Some

Parsec	Races	Splash – 2	Races
blackscholes	1	barnes	2
bodytrack	0	cholesky	2
canneal	1	fft	4
fluidanimate	4	lu	2
freqmine	0	ocean	0
streamcluster	7	radiosity	2
swaptions	1	raytrace	1
		waterNS	0

Table 5.2: Number of data race conditions detected by GUARD.

of the data race conditions reported here are benign, harmless, or intended race conditions. However, it is essential for a concurrency bug detection tool to report all potential bugs and let the programmer make a decision on its severity.

5.4.1 Performance-Accuracy Trade-offs

Although massively parallel, signature comparison based data race detection involves significant amount of computational work. If not properly managed, it could slow down the data race detection process and, in turn, stall the CPU application. Here, we analyze the performance cost of GUARD and the performance-accuracy trade-offs we could make. In particular, we look at two main parameters of GUARD, *signature size* and *throttling*:

- We consider three signature sizes in our experiments: 2048-bits, 1024-bits, and 512-bits. The maximum size of an epoch is limited to 2000 instructions. The false positive rate increases with decreasing signature size as discussed in Section 5.2.2.

- We consider three levels of parallelization (*throttling*) as discussed in Section 5.2.3: *full, half,* and *quart*. For an n-core CPU, the number of GPU threads required for GUARD throttling at T grows at the rate of $\mathcal{O}(n^2 * \mathrm{T})$.

Figure 5.9 presents the performance-accuracy trade-off characteristics of GUARD for a 4-core CPU. The performance overhead (in bars) is evaluated as the slowdown (% increase

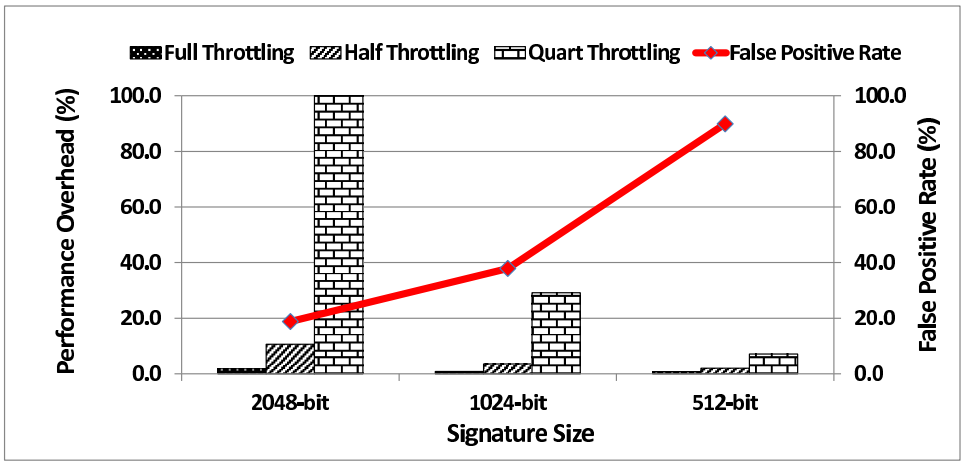


Figure 5.9: Performance and accuracy characteristics of GUARD for a 4-core CPU. The graph shows (in bars) the slowdown (%) of application being monitored for different throttling levels and signature sizes. The graph also shows (in line) the false positive rate (%) for different signature sizes used.

in cycles per instruction) of the CPU application being monitored with GUARD, over its native execution. The values shown are the geometric mean of all the 15 benchmarks we evaluated. The accuracy (in lines) is evaluated as the false positive rate (% of data races reported that are false) for the signature size used in GUARD.

We observe that the difference in throttle level is well pronounced in the results. For any particular signature size, *full* throttle performs better than *half* throttle which in turn performs better than *quart* throttle. This is expected as the data race detection algorithm is extremely parallel and with more GPU threads assigned, better performance is obtained.

Similarly, for any particular throttling, the performance of GUARD improves with decreasing signature size as GPU kernel has less signature comparisons to perform. However, this performance improvement is accompanied by increase in the false positive rate. We observe that at *full* throttle, we are able to achieve near-zero performance overhead for data race detection on a 4-core CPU. Furthermore, by scaling the number of GPU cores

employed for data race detection, GUARD is able to perform data race detection for 8-core, 16-core, and 32-core CPUs with near-zero (less than 2%) performance overhead at *full* throttling.

On detailed analysis of the performance of the GPU kernel, we observe that the performance overhead of GUARD is mainly due to two reasons: (i) data accesses related to the long signatures; and (ii) synchronization of the hundreds of threads used for H-B comparisons. GUARD's GPU kernel stalls only for about 1.54% of its execution cycles due to unavailability of data in any threads (memory related stalls). We see that the signature table size is small enough to fit inside the GPU L2 cache. For a reasonable GPU L1 data cache size, as in Table 5.1, the L1 data cache hit rate is more than 99%. We also observe that GPU does a good job of *coalescing* memory accesses and limiting the impact of data access latency on the performance of GUARD. Thread synchronizations, on the other hand, are necessary for the correctness of H-B algorithm when mapped to a highly parallel architecture like GPU.

Customizable Design

The high performance of *full* throttle mode is obtained at the cost of utilizing large amount of on-chip GPU resources. If on-chip resources are constrained, we could also select a smaller signature size and still achieve better performance for the same level of throttling as shown in Figure 5.9. However, this will be achieved at the cost of higher false positive rate. GUARD allows customizing either of these parameters, signature size or throttling, to achieve the performance goal we set for a particular accuracy constraint. This level of performance-accuracy customizability is hard to achieve in hardware-based data race detection mechanisms.

Kernel Parallelizations

We observe that not all parallelization opportunities discussed in Section 5.2.3 work equally well. In addition to throttling, we also discussed utilizing multiple threads to compare the chunks inside each signature. While we observe that throttling has a significant impact on the performance of GUARD, the signature chunk-level parallelism does not improve the performance significantly. When utilizing chunk-level parallelism, each GPU thread performs a very short computation (comparing two 64-bit unsigned integers) which does not yield significant benefits. Additionally, the overhead of managing a high number of GPU threads is not recovered by the short 64-bit comparison. This indicates that the H-B algorithm used in GUARD benefits more from coarse-grained parallelism than from fine-grained parallelism.

Bandwidth Utilization

Signature transfer between CPU and GPU consumes on-chip bandwidth. For a 2048-bit signature, we observe that GUARD utilizes less than 15% of the on-chip bandwidth provided by current designs [8] to transfer signatures. This bandwidth utilization can further be reduced by using additional hardware to compress the signatures [26] before transferring through the on-chip interconnection network.

Effect on GPU Applications

GUARD shares the GPU computational power with other GPU applications. Hence, while GUARD is enabled, other applications will have less GPU resources available and their performance could suffer. GUARD, however, is envisioned as a runtime tool that is exclusively used for debugging purposes and not for continuous usage while other applications are utilizing GPU resources. Hence, the impact of GUARD on the performance of other GPU applications is minimal.

Supporting Thread Migration & Simultaneous Multithreading

Thread migration in a multicore processor enables application threads to migrate from one core to another. GUARD can support thread migration as the signature table entries correspond to a thread, and are not tied to any particular core. When a thread migrates from a core, the current signature is forcibly closed and transferred to the signature table for data race detection. Additionally, GUARD can handle parallel applications utilizing more number of threads than the number of cores present in the processor. Since the signature table is stored in memory, instead of dedicated hardware, GUARD is able to adapt to the number of threads utilized by the application. This capability also lets GUARD support simultaneous multithreading.

Hardware Support

In baseline GUARD, the only additional hardware support required is the MTE. We build MTE on top of well studied generic instruction-grain program execution monitors [68, 67] that is used for efficient extraction of execution trace. Bloom filter hardware is used to compress the extracted traces into signatures. Hardware buffers are used to temporarily store the signature while an epoch is being created. For a 2048-bit signature, combined RD/WR signature size will be 512 bytes per core.

5.5 Related Work

Several data race detection techniques have been proposed by both academia and industry. They can broadly be divided into hardware [25, 26, 27] or software [28, 29, 30, 31] schemes. While hardware schemes offer high performance data race detection at the cost of implementation overhead, software schemes offer cheaper data race detection capability, albeit at the cost of performance. Techniques [32, 33] have been proposed to achieve low

performance overhead at the cost of minimal hardware modifications. However, these techniques usually trade-off on accuracy or coverage to achieve the performance goal. Although these techniques are primarily based on happened-before [63] or lockset [29] algorithms, techniques that utilize a hybrid of these algorithms have also been proposed [80]. GUARD is based on happened-before algorithm and utilizes on-chip data-parallel cores to perform data race detection without compromising on accuracy or coverage.

Previous work has proposed utilizing hardware transactional memory (HTM) mechanism for data race detection. RaceTM [81] utilizes lightweight *debug transactions* to detect data races with the help of the conflict detection mechanism of the HTM. However, GUARD differs from such an HTM-based data race detection. RaceTM requires the underlying HTM support for operation, whereas, GUARD requires minimal hardware support for the extraction of memory access trace. The crux of GUARD's data race detection, the signature comparison, is performed by the general purpose GPU cores available on-chip. However, the memory access trace extraction mechanism can potentially be shared by several functionalities, including GUARD and HTM.

Concurrency bug detection tools for applications executing on GPU architectures have been proposed [82, 83, 84]. In general, these tools instrument the GPU application to log memory accesses, and then utilize these logs to perform data race detection. GUARD differs from these software-based mechanisms as it targets data race detection for CPU application, by utilizing on-chip GPU cores. Holey et al. [85] have proposed a hardware accelerated technique to efficiently detect data races in GPU applications.

A recently proposed work, KUDA [86], proposes to utilize GPU threads to improve the performance of data race detection on CPU threads. GUARD, however, differs from KUDA in several aspects. KUDA needs binary instrumentation and the help of additional CPU threads (worker threads) for the extraction of memory access trace. Additionally, the memory trace compression technique employed by GUARD helps in outperforming KUDA.

5.6 Summary

As the integration of data-parallel accelerator cores onto the modern multicore processor becomes common, it is desirable to be able to utilize this computing power for enhancing non-performance aspects of parallel execution. Concurrency bug detection, particularly data race detection, assumes increased importance in the current landscape of parallel computing. In this chapter, we design, implement, and evaluate a GPU accelerated data race detector (GUARD). GUARD utilizes GPU cores available on-chip to perform data race detection for the multithreaded applications running on the CPU cores. The GPU cores are employed for data race detection when they are not being utilized for performance acceleration of applications.

GUARD proposes different optimizations each allowing a different trade-off between performance and accuracy of data race detection: (i) accelerating CPU data race detection utilizing available on-chip data-parallel cores; and (ii) compressing generated memory traces into signatures. Using a single GPU core that can support up to 1024 threads (SM architecture described in Section 5.3), GUARD performs data race detection on a 4-core CPU with 1.8% performance overhead and 18.8% false positive rate. Furthermore, by scaling the number of GPU cores employed for data race detection, GUARD is able to perform data race detection for 8-core, 16-core, and 32-core CPUs with near-zero performance overhead at *full* throttling.

With minimal hardware support, GUARD can be invoked for data race detection with negligible performance impact. Overall, GUARD proves to be a powerful tool in the parallel programming environment, necessitated by the emergence of many-core processors and facilitated by the development of heterogeneous architectures with on-chip data-parallel cores.

Chapter 6

Addressing Accuracy & Scalability of GUARD

Accuracy and scalability are two critical issues for any data race detection mechanism. In this chapter, we analyze GUARD for these characteristics and propose techniques to improve them.

6.1 Accuracy of Data Race Detection

GUARD uses signature for efficient computation and communication. However, signatures are lossy compression techniques. Two distinct addresses could map to the same bit pattern in a signature and this could lead to incorrect data race detection, also known as *false positives*.

Figure 6.1 shows the impact of using signatures on the accuracy of GUARD. To improve the effectiveness of GUARD, it is essential to improve the accuracy of the signature-based compression technique. In this section, we discuss a novel coherence-based filtering mechanism that improves the accuracy of data race detection in GUARD.

Figure 6.1: Inaccuracy in GUARD due to signatures. Figure shows the false positive rate for different signature sizes.

6.1.1 Coherence-Based Filtering

GUARD compresses load (LD) and store (ST) addresses into separate read (RD) and write (WR) signatures of same size for comparison purposes. However, we observe that LD instructions generally outnumber ST instructions by nearly five to one. This means that LD instructions are the major source of false positive rate in GUARD. The false positive rate can be reduced by increasing signature sizes. However, this increases the signature table size and the signature comparison effort leading to significant performance penalty.

We propose a filtering mechanism that utilizes coherence state information to identify the LD instructions that access private and shared read-only addresses, and filters them out. This way, only LD instructions that access shared addresses modified by other threads are compressed into the RD signature. These are the potential data race accesses and we call such addresses *shared-modified*. Since the impact of ST instructions on accuracy is very low, and since each write access is a potential data race candidate, we do not apply any filtering on them. By filtering out innocuous LD instructions, we aim to bring down the false positive rate for GUARD without any negative impact on performance or data

race detection capability.

We consider a memory hierarchy design with private L1 caches and a shared LLC which is common in current multicore processors. When the filtering mechanism is enabled, the MTE monitors the data response message from the LLC to check the *shared-modified* state. If the data was written to by another thread and is in modified state in the LLC, the shared-modified state is set by the LLC controller. When the state is set, MTE concludes that this is a potential data race candidate and adds the address to the RD signature. Otherwise, the address is filtered out. The filtering mechanism considers the following three scenarios:

- L1 Hit: When a LD instruction hits the L1 data cache, the data is either private or shared read-only. Such an access will not cause a data race, and hence it is considered safe and the address is filtered out.

- L1 Miss & LLC Hit: When a LD instruction misses L1 data cache and hits the shared LLC, the LLC controller uses the coherence information to identify the state of the address. If the address was in a modified state prior to the load request, it was written to by another thread recently. Hence, this address is considered shared-modified and the corresponding bit is set in the response message.

- LLC Miss: If the access misses the shared LLC, it is potentially a cold miss or an access to the address after a long interval. Such accesses are considered safe as they will not cause a data race. Hence, the LLC controller resets the shared-modified bit and the address is filtered out.

These scenarios, however, could still experience a situation where the access could lead to a data race condition. In a potential write-after-read (WAR) race condition scenario, as shown in Figure 6.2, when the read instruction occurs at first to ADDR2, there is not enough information to make a decision on filtering. However, a future write to the same memory location by another thread in a concurrent epoch results in a potential data race. Hence, if this LD instruction was filtered out due to insufficient information, a potential

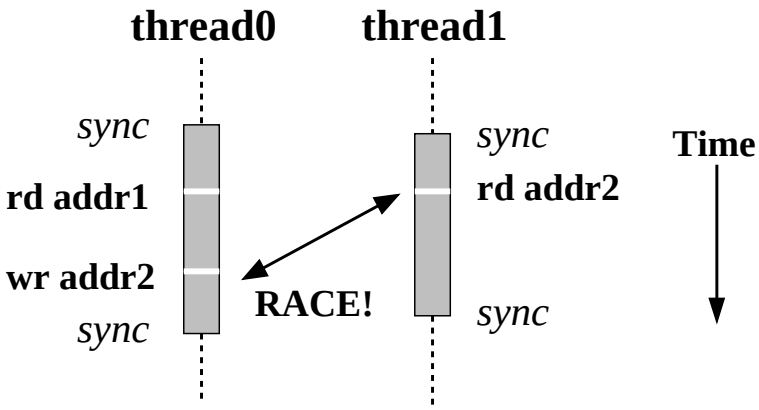


Figure 6.2: A WAR data race condition that coherence filtering could potentially miss.

WAR race condition could be missed.

This issue can be addressed by using temporary hardware signatures. For every thread, the filtered LD addresses from the current epoch are compressed and stored in temporary signatures. When a ST occurs (rather infrequent) in a thread, LLC controller sends invalidation messages to sharers and the cache line is set to *modified* state. The MTE in these sharers compare the address in the invalidation message with the addresses in their temporary RD signature, and if there is a match, the address is added back to the thread's RD signature.

However, only the addresses from the current epoch could be saved as the previous epochs would have already been dispatched to the GPU for data race detection. Also, limited capacity of the LLC or time gap between the two instructions could lead to the related cache line being evicted from the shared LLC. The scheme will then filter out the LD instruction, due to lack of information in the LLC. However, it should be emphasized here that the most crucial data race accesses are the ones that occur in close proximity, and those are unlikely to be filtered out due to this limitation.

MTE picks up the shared-modified state of the address from the cache response message from the LLC controller. Since the MTE is private to a core, it need only monitor coherence

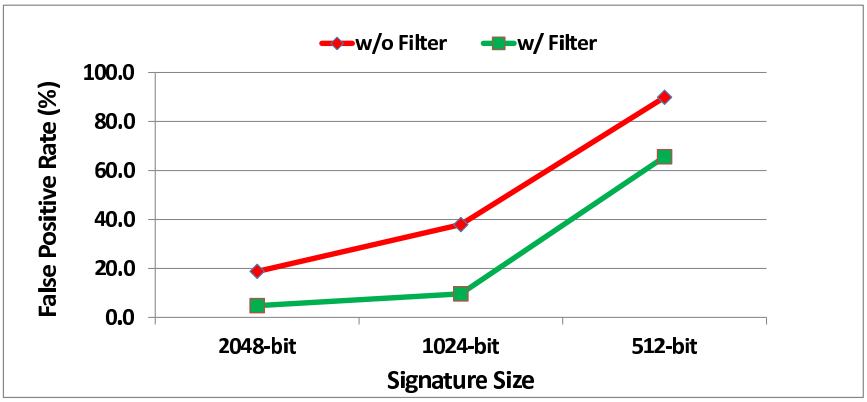


Figure 6.3: Impact of coherence-based filtering on the inaccuracy of GUARD. Figure shows the false positive rates with and without filtering for different signature sizes.

messages destined to the local first-level cache. Only a single additional *shared-modified* bit is required to pass this information. The filtering mechanism does not alter the cache coherence scheme in any way, which is desirable as they are highly optimized designs. The temporary signatures will be the size of a RD signature per core, which is 256 bytes for a 2048-bit signature.

Prior work [33] has proposed an algorithm that uses coherence state information to detect data races. Also, software-based data race detection mechanisms [87] have employed techniques to filter stack and duplicate addresses to improve performance. However, to the best of our knowledge, this is the first work to utilize a coherence-based filtering technique to improve the accuracy of a data race detection tool that already works at near-hardware speed.

6.1.2 Evaluation

The coherence-based mechanism filters 93.6% of all LD instructions, which results in filtering out accesses to 96.56% of unique addresses. With filtering, the false positive rate drops significantly as shown in Figure 6.3. Additionally, the filtering mechanism achieves this improvement without missing any data race conditions in our experiments. Thus,

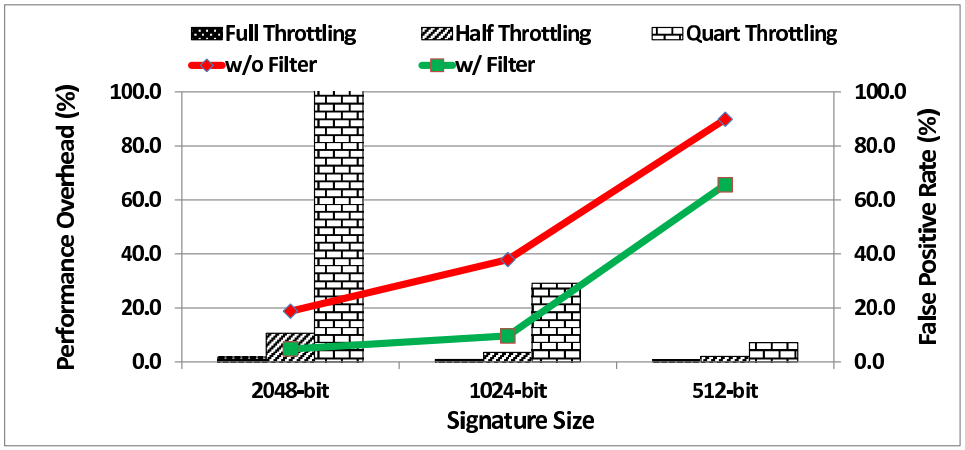


Figure 6.4: Performance-Accuracy trade-offs in GUARD. Figure shows the performance overhead and the false positive rates for different throttling levels and signature sizes.

coherence-based filtering proves to be very efficient in improving the accuracy of GUARD. Our evaluations are based on MOSI coherence protocol. However, the filtering mechanism can easily be adapted to other coherence protocols.

In addition to improving accuracy of GUARD, filtering enables reduction in GPU resource utilization. With filtering, false positive rate for 1024-bit signature is now under 10% as shown in Figure 6.4. Hence, *half* throttling with 1024-bit signatures can be utilized to run GUARD with negligible performance overhead, reasonable accuracy, and low GPU utilization. This is particularly attractive for CPUs with higher number of cores as the GPU resources required to perform data race detection at *full* throttling can become quite large as shown in Figure 6.5 and discussed in the next section.

6.2 Scalability of Data Race Detectors

Although, GUARD is able to achieve near-zero performance overhead for data race detection in CPUs with up to 32 cores, this performance comes at a resource penalty. Figure 6.5 shows the amount of GPU resources required to perform data race detection, for different

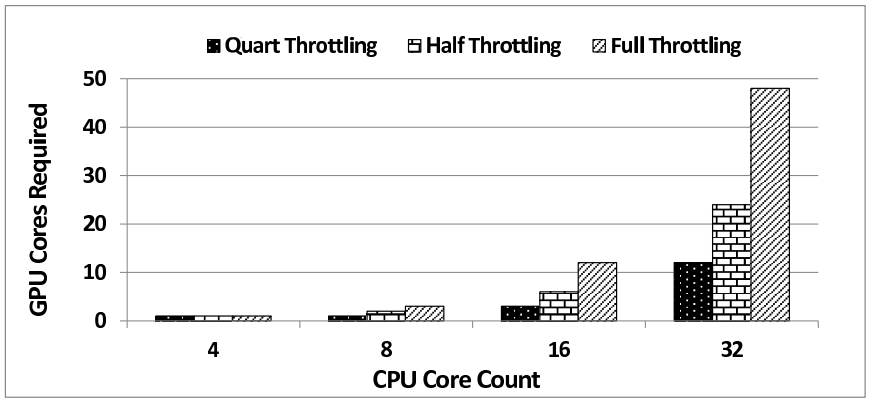


Figure 6.5: Number of GPU cores required for data race detection, for different CPU core count, at different throttling. The GPU SM architecture is described in Section 5.3.

CPU configurations, at different throttling. Although coherence-based filtering enables GUARD to perform efficient data race detection with *half* throttling, even that faces scalability issues with increasing CPU core count. To address scalability issues, we propose a clustered architecture for GUARD.

6.2.1 Clustered GUARD

Figure 6.6 shows the clustered implementation of GUARD. Let us consider a 16 core CPU divided into 4 clusters of 4 cores each. Here, instead of each processor performing signature comparisons with each of the other 15 cores, we perform a two-level signature comparison. At the inner level, in *intra-cluster* comparisons, each of the cores perform signature comparison with the other three cores. At the outer level, in *inter-cluster* comparisons, each of the clusters perform signature comparisons with the other three clusters. These inter-cluster comparisons are performed on separate *SuperSignatures*, unique to each cluster. SuperSignatures are created from the memory accesses by all the cores inside a particular cluster.

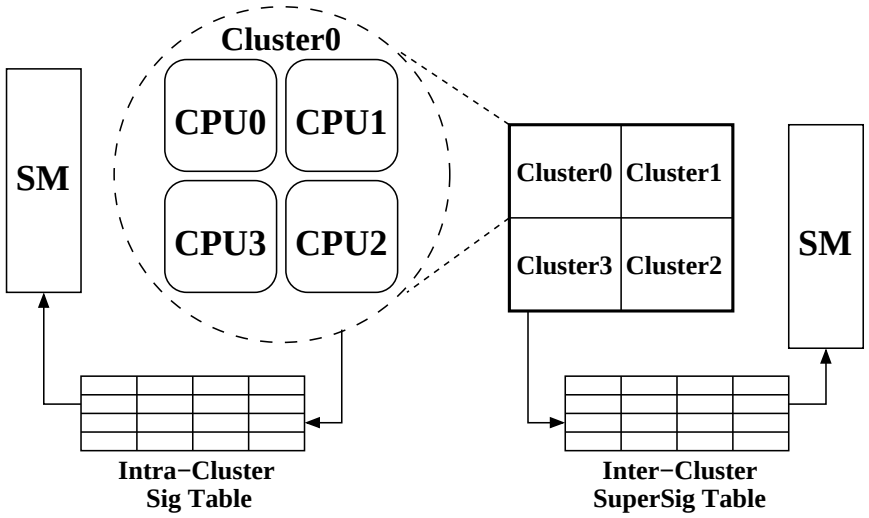


Figure 6.6: Clustered race detection mechanism. A 16-core CPU is divided into 4 clusters of 4 cores each. Inter-cluster SuperSig table is shown in addition to intra-cluster signature table.

SuperSignature Formation

SuperSignature formation is as shown in Figure 6.7. Here, we consider two clusters: cluster0 and cluster1. In vertical lines, we express the execution flow of each thread t0 to t7. Horizontal dashed lines indicate the cluster epochs formed with concurrent memory accesses among all the threads in the cluster. Let us consider epoch2 in cluster0, as shown in Figure 6.7. This cluster epoch encapsulates all the accesses between *sync* operations: t01 and t02, t11 and t12, t21 and t22, and t31 and t32. Here, the epoch starts with the first *sync* operation among the eight: t21. All the corresponding addresses are combined into a SuperSignature that represents the cluster epoch. To protect the concurrency characteristics of individual threads inside a cluster, the first and last *sync* timestamp among the four threads are taken as the *start-time* and *end-time* for the cluster epoch. Hence, the *start-time* of this cluster epoch is t21 and *end-time* is t32. H-B algorithm is then applied on these SuperSignatures to perform the inter-cluster comparisons.

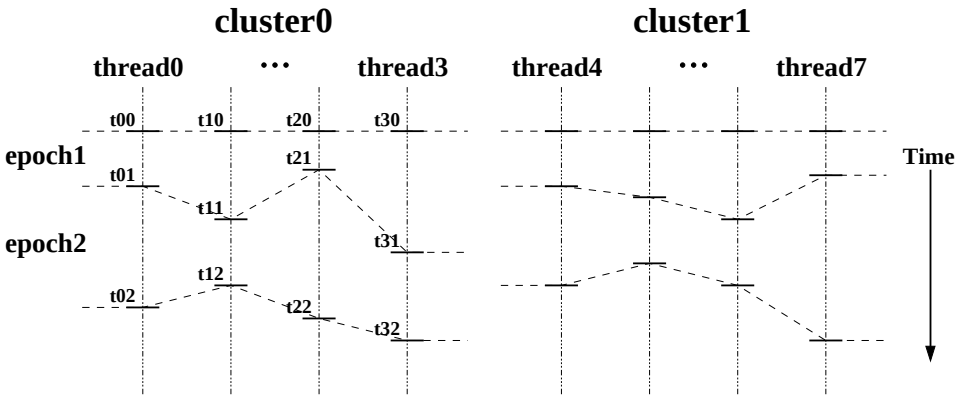


Figure 6.7: SuperSignature formation for clusters with 4 cores each. Vertical lines represent each thread's execution stream, and horizontal dashed lines represent cluster epoch boundaries.

Kernel Modifications

Clustered architecture for GUARD is designed as a software feature, without any hardware modifications. The SuperSignatures are created by GUARD's GPU kernel from individual signatures in the intra-cluster signature table. This SuperSignature is then stored inside the inter-cluster SuperSig table and used by the GUARD GPU kernel for inter-cluster data race detection. Three different tasks are implemented in clustered GUARD: (i) intra-cluster data race detection; (ii) SuperSignature generation; and (iii) inter-cluster data race detection; using thread partitioning inside the same GPU kernel. Newer GPU architectures allow multiple GPU kernels to run simultaneously, and this feature can be used to further speedup cluster architecture by assigning each of these three GUARD tasks to parallel kernels.

6.2.2 Evaluation

For the 16-core CPU design in Figure 6.6, the signature and SuperSignature sizes are fixed at 2048-bits and *full* throttling is applied for inter-cluster SuperSignature comparisons as well as for intra-cluster signature comparisons. As every cluster epoch comprises of multiple

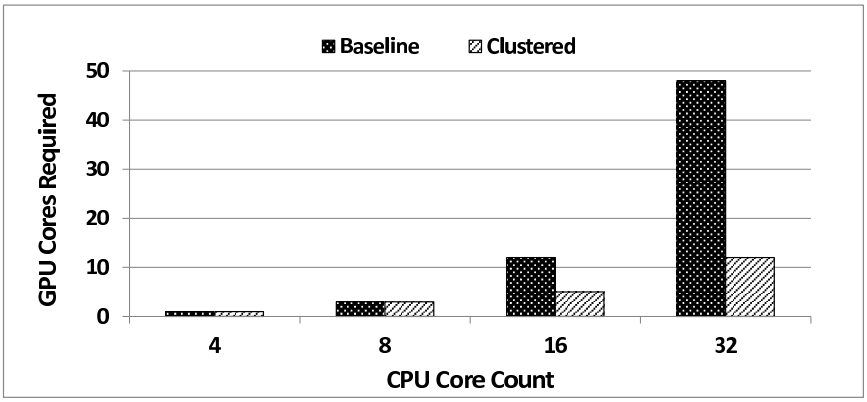


Figure 6.8: GPU resource utilization in clustered GUARD at *full* throttling.

individual epochs, SuperSignatures are produced slower than individual signatures. Our experiments show that the total number of SuperSignatures is about 16.51% of the total number of individual signatures. Thus, the rate of writing SuperSignatures to the SuperSig Table is slower than that of signatures to individual signature table. Hence, the intra-cluster data race detector remains the major bottleneck for GUARD and the SuperSignature comparisons do not add any additional performance overheads.

Clustered GUARD for a 16-core CPU reports a performance overhead of 0.88% which is similar to that for 16-core baseline GUARD. However, clustered GUARD performs significantly less computation than the baseline GUARD scheme. Figure 6.8 shows the number of GPU cores (architecture described in Section 5.3) required for perform data race detection at *full* throttling for baseline and clustered GUARD. We observe that, clustered GUARD scheme uses less resources than that utilized by baseline GUARD scheme. Thus, with the clustered architecture, we present a data race detection tool that can scale well in the many-core era.

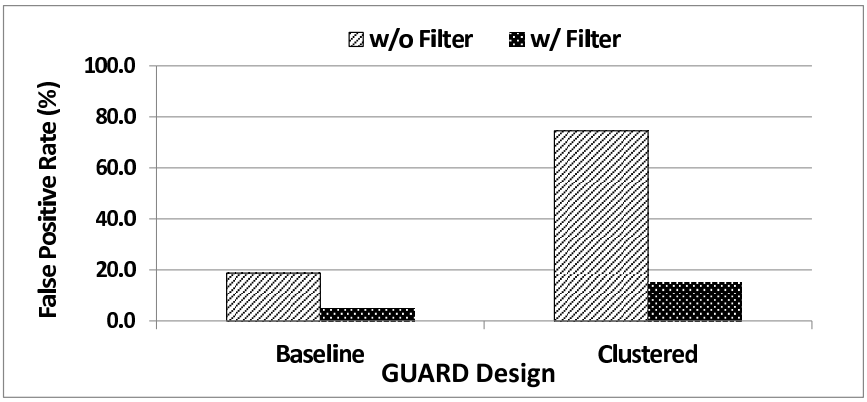


Figure 6.9: Accuracy characteristics of clustered GUARD at *full* throttling for a 16-core CPU.

Filtering in Clustered Architecture

In SuperSignature formation, we combine addresses from concurrent epochs of multiple threads into a cluster epoch. Thus, SuperSignature ends up encapsulating a much larger number of addresses than individual signatures. As a result, the false positive rate of SuperSignature becomes larger than that of individual signatures. Ideally, the SuperSignatures should be larger in size than the base-case of 2048-bit signature. However, an increased size of SuperSignature would demand increased GPU resources to perform the SuperSignature comparison at the same speed as the individual signature comparisons. On the other hand, limiting SuperSignature size to 2048-bits will be trading-off on precision.

In clustered architecture, we trade-off on precision at the inter-cluster SuperSignature comparison without compromising on performance. Consequently, for 2048-bit signatures, false positive rate increases from 18.78% for non-clustered architecture to 74.47% for clustered architecture as shown in Figure 6.9. Although good performance characteristics for GUARD is observed in the clustered design, this loss of precision is unacceptable. By applying the coherence-based filtering mechanism described in Section 6.1.1, the false positive rate improves from 74.47% to 14.98%. Thus, the coherence-based filtering mechanism

proves to be very efficient in improving the accuracy of GUARD in clustered architecture.

6.3 Summary

Accuracy and scalability are two key challenges in designing an efficient data race detection scheme. Use of signatures for improving performance leads to false positives in GUARD. We tackle this side-effect by utilizing a novel coherence-based filtering mechanism. All the currently proposed data race detection mechanisms will face scalability issues as we move towards many-core CPU designs. For GUARD, the GPU resource requirement to perform data race detection with low performance overhead will increase quadratically with the number of CPU cores. To improve this aspect, we propose a clustered architecture for GUARD that is able to scale with the CPU core count. Overall, these two optimizations improve GUARD in terms of two critical factors: accuracy and scalability.

Chapter 7

Conclusions and Future Directions

With the advancements in process technology and the need to improve performance of diverse applications, heterogeneous multicore processors that contain diverse cores on the same die appears to be the future of microprocessor design. However, designing heterogeneous multicore processors where on-chip resources are shared by cores with diverse characteristics is challenging. In this thesis, we study these challenges from the perspective of two key aspects: performance and correctness.

First, we study the impact of sharing the on-chip last-level cache (LLC) among diverse cores on their performance. We observe that introduction of GPU cores impact the performance of cache sensitive CPU applications under currently proposed cache management policies. Furthermore, the high rate of memory accesses from GPU increases the energy consumption leading to a reduction in energy efficiency. To develop a better cache management policy, we study the cache sensitivity characteristics of CPU and GPU applications. We observe that GPU applications are, in general, less cache sensitive than CPU applications. Also, the order of magnitude higher number of threads in GPU cores enable GPU applications to sustain performance in the presence of increased memory access latency.

With this knowledge, we propose a new cache management policy, HeLM, that improves the sharing of the on-chip LLC. By bypassing LLC for memory accesses from the GPU cores,

when the available TLP indicates so, HeLM is able to increase the cache occupancy of CPU applications and hence improve its performance. In our evaluations, HeLM outperforms all currently proposed policies. Performance improves over LRU policy by 10.2% and over TAP by 5.7% for the baseline processor configuration with two CPU and four GPU cores. HeLM also outperforms these policies in energy consumption and energy efficiency. It consumes nearly 3% less total energy compared to TAP, while reducing ED^2 value by 8%, for the baseline configuration. HeLM scales well with varying processor count and performs consistenty in terms of both performance and energy efficiency.

Second, we address the challenge of improving execution correctness of parallel programs in the context of the emerging heterogeneous multicore processors. We propose a data race detection tool, GUARD, that achieves the performance of hardware-based data race detectors without the implementation cost of currently proposed mechanisms. GUARD achieves this by utilizing available on-chip GPU cores to perform data race detection for the applications executing on the CPU cores. Using a single GPU core that can support up to 1024 threads, GUARD performs data race detection on a 4-core CPU with only 1.8% performance overhead. Furthermore, by scaling the number of GPU cores employed for data race detection, GUARD is able to perform data race detection for 8-core, 16-core, and 32-core CPUs with near-zero performance overhead.

Accuracy and scalability are two critical characteristics of an efficient data race detector. To improve the accuracy of GUARD, we propose a novel coherence-based filtering mechanism that filters innocuous load addresses from reaching the signatures used. On an average, the coherence-based filtering improves the accuracy of signature-based GUARD by nearly 75%. Although baseline GUARD is able to performance data race detection with near-zero overhead, the GPU resource required to do so becomes extremely high with increasing CPU core count. To address this scalability issue, we propose a clustered architecture for GUARD that performs data race detection at two levels. This leads to a significant reduction in the amount of GPU resources utilized for high performance data

race detection.

To summarize, the key insights of this dissertation are the following:

- The diversity in memory access characteristics among on-chip cores leads to the performance degradation for cache sensitive applications when on-chip cache is shared. Under the extremely high memory access rate from GPU cores, the energy efficiency of the heterogeneous core also suffers. This emphasizes the need for better cache management policies.

- GPU architecture provides extremely high number of concurrent threads that applications can utilize to sustain performance in the event of increased memory access latency. Thus, TLP availability could be utilized as a metric to guide the cache management policy in heterogeneous multicore processors.

- Releasing cache space to more cache sensitive CPU applications by bypassing LLC for memory accesses from GPU applications, when TLP metric indicates so, improves the overall performance, as well as the energy efficiency, of the heterogeneous system.

- Execution correctness is critical for parallel CPU applications, and can be improved by utilizing on-chip GPU cores to perform data race detection efficiently.

- Accuracy and scalability are critical characteristics that need to be addressed to improve the usefulness of a data race detection mechanism.

7.1 Future Directions

We see several avenues in which the research presented in this dissertation could be extended:

7.1.1 Improving Cache Management with Reuse Analysis

HeLM, the cache management policy proposed in this dissertation, is agnostic of the reuse characteristics of the GPU application. This could lead to suboptimal performance in certain situations:

- *Low TLP, Low Reuse*: In applications where available TLP is low, HeLM will bypass LLC conservatively. However, if the applications show low cache reuse, storing cache blocks in LLC will not yield performance benefits.

- *High TLP, High Reuse*: In applications where available TLP is high, HeLM will bypass LLC aggressively. However, if the applications show high cache reuse, bypassing LLC will miss opportunities to improve performance of the GPU application.

Reuse-based analysis could help in these situations. Reuse analysis could improve HeLM's understanding of the benefit of storing GPU blocks in LLC and can improve its cache sharing decisions.

Several reuse-based cache management policies have been proposed [55, 56, 57, 58, 59]. However, they have been developed targeting CPU cores and applications. GPU cores exhibit very different architectural characteristics and GPU applications display very different memory access characteristics when compared to CPU cores and applications. For example:

- PC-based reuse analysis has been studied to be beneficial for CPU applications. In GPU applications, however, the range of PC in a kernel is very small for PC-based reuse analysis to be potentially effective.

- Currently proposed reuse-based techniques do not consider the architectural characteristics of GPU such as the high number of concurrent threads. Hence, they experience difficulty in extracting performance from GPU applications.

Hence, adapting reuse analysis to suit GPU architecture and applications, and utilizing them to enhance HeLM could further improve the sharing of on-chip LLC among heterogeneous cores.

7.1.2 Application Phase Behavior in Data Race Detectors

CPU applications have been shown to have runtime phase behaviors [11, 88, 89, 90]. Knowledge of this phase behavior can be utilized to optimize both hardware and software support. There are phases where all the threads are involved in computation and the program is executing at high instructions per cycle (IPC). There are also phases where many are waiting for response from memory access instructions and hence operating at low IPC. This speed difference in operation during different phases could be utilized to adapt the data race detection scheme for lower resource utilization.

In particular, we could adapt the *throttling* in GUARD to the performance of the CPU application being monitored. When the CPU application is operating at a high IPC, higher throttling level (*full*) could be used. However, when the CPU application is executing at lower IPC, running GUARD on lower throttling levels (*half, quart*) can reduce the GPU resource requirement for effective data race detection. Thus, considering the phase behavior of the CPU application being monitored could improve the resource utilization of data race detection schemes such as GUARD.

7.1.3 Broader Challenges in Heterogeneous Architecture

Beyond the specific issues discussed in previous sections, there are broader challenges that need to be addressed in the designing the emerging heterogeneous multicore processor architecture.

(i) Shared Address Space: The heterogeneous multicore architecture we consider in this dissertation assigns disjoint address spaces for the diverse CPU and GPU cores. However, in future heterogeneous multicore processors, these diverse cores might share the address

space. Designing such processors will face significant challenges including the design of an efficient coherence protocol that handles diverse cores.

(ii) Unified Programming Model: Currently, CPU and GPU cores support separate programming models. A tightly integrated heterogeneous multicore processor might be better utilized by a more unified programming model. Designing such a program model that supports diverse processor architecture will also face significant challenges.

These challenges make the design of heterogeneous multicore processors an exciting area for future research work.

References

[1] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[2] J.M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 2002.

[3] Tilera Corporation. Tilera Tile Processor Family. http://www.tilera.com.

[4] Intel Corporation. Intel Many Integrated Core Architecture. http://www.intel.com.

[5] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[6] NVIDIA Corporation. Nvidia Fermi Architecture. http://goo.gl/GBSq7J.

[7] Advanced Micro Devices Incorporated. Evergreen Family Instruction Set Architecture . http://goo.gl/WQ5lE.

[8] Nathan Brookwood. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. *Advanced Micro Devices(AMD) White Paper*, 2010.

[9] Intel Corporation. Intel Sandy Bridge Microarchitecture. http://www.intel.com.

[10] Pradeep Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.

[11] Vineeth Mekkat, Ragavendra Natarajan, Wei-Chung Hsu, and Antonia Zhai. Performance characterization of data mining benchmarks. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, 2010.

[12] Ragavendra Natarajan, Vineeth Mekkat, Wei-Chung Hsu, and Antonia Zhai. Effectiveness of compiler-directed prefetching on data mining benchmarks. *Journal of Circuits, Systems and Computers*, 2012.

[13] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[14] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 1992.

[15] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[16] G.E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 2004.

[17] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[18] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[19] Miquel Moreto, Francisco. J. Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*. 2008.

[20] Yuejian Xie and Gabriel H. Loh. Scalable shared-cache management by containing thrashing workloads. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[21] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[22] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[23] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[24] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[25] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[26] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[27] Pin Zhou, R. Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[28] Intel Corporation. Intel Thread Checker. http://www.intel.com.

[29] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 1997.

[30] Sun Microsystems. Sun Studio Thread Analyzer. http://www.sun.com.

[31] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[32] M. Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2006.

[33] Rodrigo Gonzalez-Alberquilla, Karin Strauss, Luis Ceze, and Piuel Luis. Accelerating data race detection with minimal hardware support. In *Proceedings of the 17th International Conference Euro-Par Parallel Processing*, 2011.

[34] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. http://www.nvidia.com.

[35] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[36] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[37] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[38] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[39] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, 2007.

[40] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[41] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (PriSM). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[42] Mainak Chaudhuri. Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[43] Cloyce D. Spradling. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News*, 2007.

[44] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) . http://developer.amd.com/sdks/amdappsdk/.

[45] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, 2012.

[46] Advanced Micro Devices Incorporated. ATI Stream Computing Programming Guide. http://www.amd.com.

[47] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[48] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[49] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[50] Valentina Salapura, Matthias Blumrich, and Alan Gara. Design and implementation of the Blue Gene/P snoop filter. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008.

[51] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[52] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[53] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[54] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 2011.

[55] T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 1999.

[56] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[57] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[58] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 2008.

[59] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[60] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta,

and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 2000.

[61] Valentina Salapura, Randy Bickford, Matthias Blumrich, Arthur A. Bright, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Michael Gschwind, Manish Gupta, Shawn Hall, Ruud A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Martin Ohmacht, Rick A. Rand, Todd Takken, and Pavlos Vranas. Power and performance optimization at the system level. In *Proceedings of the 2nd Conference on Computing Frontiers*, 2005.

[62] Paul I Pénzes and Alain J. Martin. Energy-delay efficiency of vlsi computations. In *Proceedings of the 12th ACM Great Lakes Symposium on VLSI*, 2002.

[63] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 1978.

[64] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[65] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[66] Vineeth Mekkat, Anup Holey, and Antonia Zhai. Accelerating data race detection utilizing on-chip data-parallel cores. In *Fourth International Conference on Runtime Verification*. 2013.

[67] Guojin He and A. Zhai. Improving the performance of program monitors with compiler support in multi-core environment. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[68] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006.

[69] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 1970.

[70] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions. In *ACM Symposium on Theory of computing*, 1977.

[71] M. Xu, R. Bodik, and M.D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[72] M. Prvulovic, Zheng Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[73] Shucai Xiao and Wu-Chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

[74] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 2002.

[75] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 2005.

[76] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[77] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[78] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[79] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[80] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[81] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. Using hardware transactional memory for data race detection. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2009.

[82] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[83] Qiming Hou, Kun Zhou, and Baining Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. In *ACM Transactions on Graphics (TOG)*, 2009.

[84] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. Grace: a low-overhead mechanism for detecting data races in gpu programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011.

[85] Anup Holey, Vineeth Mekkat, and Antonia Zhai. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of the 42nd Annual International Conference on Parallel Processing (ICPP)*, 2013.

[86] U Can Bekar, Tayfun Elmas, Semih Okur, and Serdar Tasiran. KUDA: GPU accelerated split race checker. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2012.

[87] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[88] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[89] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[90] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 2004.