GPU Computing Architecture for Irregular Parallelism

by

Wilson Wai Lun Fung

B. Applied Science, University of British Columbia, 2006

M. Applied Science, University of British Columbia, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

January 2015

Abstract

Many applications with regular parallelism have been shown to benefit from using Graphics Processing Units (GPUs). However, employing GPUs for applications with irregular parallelism tends to be a risky process, involving significant effort from the programmer and an uncertain amount of performance/efficiency benefit. One known challenge in developing GPU applications with irregular parallelism is the underutilization of SIMD hardware in GPUs due to the application's irregular control flow behavior, known as *branch divergence*. Another major development effort is to expose the available parallelism in the application as 1000s of concurrent threads without introducing data races or deadlocks. The GPU software developers may need to spend significant effort verifying the data synchronization mechanisms used in their applications. Despite various research studies indicating the potential benefits, the risks involved may discourage software developers from employing GPUs for this class of applications.

This dissertation aims to reduce the burden on GPU software developers with two major enhancements to GPU architectures. First, thread block compaction (TBC) is a microarchitecture innovation that reduces the performance penalty caused by branch divergence in GPU applications. Our evaluations show that TBC provides an average speedup of 22% over a baseline per-warp, stack-based reconvergence mechanism on a set of GPU applications that suffer significantly from branch divergence. Second, Kilo TM is a cost effective, energy efficient solution for supporting transactional memory (TM) on GPUs. With TM, programmers can uses transactions instead of fine-grained locks to create deadlock-free, maintainable, yet aggressively-parallelized code. In our evaluations, Kilo TM achieves 192$\times$ speedup over coarse-grained locking and captures 66% of the performance of fine-grained locking with 34% energy overhead.

Preface

This is a list of my publications at the University of British Columbia that have been incorporated into this dissertation, in chronological order:

[C1] Wilson W. L. Fung and Tor M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17), pp. 25-36, February 2011.

[C2] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, Tor M. Aamodt. Hardware Transactional Memory for GPU Architectures. In Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO-44), pp. 296-307, December 2011.

[T1] Wilson W. L. Fung, Inderpreet Singh, and Tor M. Aamodt. Kilo TM Correctness: ABA Tolerance and Validation-Commit Indivisibility. Technical Report, University of British Columbia, 24 May 2012.

[J1] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, Tor M. Aamodt. Kilo TM: Hardware Transactional Memory for GPU Architectures. IEEE Micro, Special Issue: Micro's Top Picks from 2011 Computer Architecture Conferences, Vol. 32, No. 3, pp. 7-16, May/June 2012.

[C3] Wilson W. L. Fung, Tor M. Aamodt. Energy Efficient GPU Transactional Memory via Space-Time Optimizations. In Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46), pp. 408-420, December 2013.

These publications are incorporated into this dissertation as follows.

Chapter 2. Portions of the text explaining the fundamental concepts of parallel computing are modified from previously written background material from my master's thesis "Dynamic Warp Formation: Exploiting Thread Scheduling for Efficient MIMD Control Flow on SIMD Graphics Hardware" (2008) completed at the University of British Columbia. The description of the baseline GPU architecture incorporated text from [C1] and [C3].

Chapter 3. A version of this material has been published as [C1]. In [C1], I conducted the research, analyzed the data and drafted the manuscript under the guidance of Dr. Tor M. Aamodt.

Chapter 4. A version of this material has been published as [C2] and later as [J1]. In [C2], I was the lead investigator, responsible for conceptualizing the major contributions, analyzing the data, and drafting the manuscript. Inderpreet Singh helped with data collection. He also analyzed the data for thread cycle distribution and drafted the text for this analysis in [C2]. Andrew Brownsword provided the original source code for the cloth simulation application, which Inderpreet Singh later modified to use transactions and fine-grained locks. Dr. Tor M. Aamodt was the supervisory author on this project and was involved in concept formation and manuscript edits. I wrote the manuscript for [J1] with help from Inderpreet Singh and Dr. Tor M. Aamodt.

Chapter 5. A version of this material has been published as [T1]. In [T1], I conceptualized the proof, drafted the manuscript under the guidance of Dr. Tor M. Aamodt. Inderpreet Singh helped validating the proof and editing of the manuscript.

Chapter 6. A version of this material has been published as [C3]. In [C3], I conducted the research, analyzed the data and drafted the manuscript under the guidance of Dr. Tor M. Aamodt.

Chapter 7. This chapter contains text from the related work sections from [C2] and [C3].

Table of Contents

List of Tables

List of Figures

List of Abbreviations

2PCR	Two-Phase Parallel Conflict Resolution
API	Application Programming Interface
BW	BandWidth
CID	Commit IDentifier
CMP	Chip-MultiProcessor
COHE	Coherent
CPU	Central Processing Unit
CU	Commit Unit
CUDA	Compute Unified Device Architecture
DIVG	Divergent
DLP	Data-Level Parallelism
DRAM	Dynamic Random-Access Memory
DWF	Dynamic Warp Formation
FG	Fine-Grained
FGL	Fine-Grained Locking
FGMT	Fine-Grained Multi-Threading
FLOPS	FLoating point Operations Per Second
FP	Floating Point
FR-FCFS	First-Ready First-Come First-Serve
GDDR	Graphics Double Data Rate memory
GPGPU	General-Purpose computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HM	Harmonic Mean

HTM	Hardware Transactional Memory
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
IPDOM	Immediate Post-Dominator
ISA	Instruction Set Architecture
Kilo TM	Kilo Transactional Memory
L1	Level 1
L2	Level 2
LCP	Likely-Convergence Point
LPC	Likely-Convergence Program Counter
LRU	Least Recently Used
LWH	Last Writer History
MC	Memory Controller
MIMD	Multiple-Instruction, Multiple-Data
MSHR	Miss-Status Holding Register
NoC	Network-on-Chip
OpenCL	Open Computing Language
PC	Program Counter
PDOM	Post-Dominator
PTX	Parallel Thread Execution
RF	Register File
RPC	Reconvergence Program Counter
RRB	Round-Robin
SCR	Serial Conflict Resolution
SDK	Software Development Kit
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SM	Streaming Multiprocessor
SMT	Simultaneous MultiThreading
SRAM	Static Random-Access Memory
SRR	Sticky Round-Robin
SSE	Streaming SIMD Extensions

STM	Software Transactional Memory
TBC	Thread Block Compaction
TCD	Temporal Conflict Detection
TDRF	Transactional Data-Race-Free
TLP	Thread-Level Parallelism
TM	Transactional Memory
TOS	Top Of Stack
VC	Virtual Channel
WarpTM	Warp-Level Transaction Management
YCID	Youngest Commit IDentifier
flit	flow control digit

Acknowledgments

The work presented in this dissertation would have only been possible with the help from many individuals and organizations. First and foremost, I would like to thank my supervisor, Dr. Tor Aamodt for his guidance and support throughout my M.A.Sc and Ph.D. programs. I was given the valuable opportunity to work along with Tor since he first joined University of British Columbia (UBC) as an assistant professor, building a fully functional research group from the ground up. Tor's dedication, passion, wisdom and his relentless pursue of high-impact, high-quality research has been truly inspirational to me both professionally and personally.

I would also like to thank my qualifying, departmental and final examination committee members: Dr. Matei Ripeanu, Dr. Sathish Gopalakrishnan, Dr. Mark Greenstreet, Dr. Wen-mei Hwu, Dr. Thomas Froese, Dr. Guy Lemieux, Dr. Steve Wilton, Dr. David Michelson, and Dr. Karthik Pattabiraman. Their valuable comments and feedback immensely improved the quality of this dissertation.

I would also like to thank Doug Carmean and Mark Hill for their enthusiastic support of my work on Kilo TM. I am also grateful to my research collaborators, Dr. Arrvindh Shriraman and Dr. Joseph Devietti, who both have expanded my knowledge horizon with their expertise.

I would also like to thank Inderpreet Singh for being a great colleague. The insightful midnight discussions between us as we worked through obstacles in our research have lead to many great ideas. I also thank the other members of UBC's computer architecture research group – Ali Bakhoda, Henry Wong, Xi Chen, Ivan Sham, George Yuan, Andrew Turner, Johnny Kuan, Tim Rogers, Rimon Tadros, Arun Ramamurthy, Jimmy Kwa, Hadi Jooybar, Ayub Gubran, Tayler Hetherington, Ahmed ElTantawy, Andrew Boktor, Myrice Li and Mahmoud Kazemi – for their valuable contributions to our research infrastructures, insightful discussions and timely supports.

My peers in the Department of Electrical and Computer Engineering in UBC have been helpful and entertaining to work with, and they helped making my experience in UBC an unforgettable one. In particular, I would like to thank Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, Emalayan Vairavanathan, Mohammad Afrasiabi, Bo Fang and Hao Yang. I also cherish the time I spent with many of my non-academic friends – David Mak, Derek Ho, Jennifer Li, Kenny Au, Fernando Har, Leo Wong, Amber Ting, Erica Ho, Sanford Lam, Tim Chan, Richard Leung and Karen Wong.

I am very grateful for the support and encouragement from my family. My parents has been fully supportive of my decisions to pursue my Ph.D. The encouragements from my mother has been heartwarming, and I am regretful for not being with her when she needed the support from me to fight her sickness (thankfully she has recovered since then). Despite being a physician himself, my father had taught me a great deal about computers and has instilled my interest in computers since my childhood.

Financial support for this research was provided through the Postgraduate Scholarships – Doctoral (PGS D) award from Natural Sciences and Engineering Research Council (NSERC) of Canada, as well as the NVIDIA Graduate Fellowship.

Chapter 1

Introduction

For the last few decades, advances in integrated-circuit (IC) fabrication technologies have followed the trend known as the “Moore’s Law”, which predicts an exponential growth of the number of transistors on a chip over time [112]. This growth continues with the area density of transistors on integrated-circuits doubling roughly every 24 months. Using these additional transistors to improve single-threaded performance on general purpose central processing units (CPUs) has been challenging for computer architects due to the “power wall” – the physical limit of power that an integrated-circuit may dissipate before cooling and power delivery become impractically expensive [9, 116]. This limits clock frequency scaling and has motivated the microprocessor industry to migrate towards parallel computing architectures in the form of chip-multiprocessors (CMPs).

A CMP system consists of multiple single-threaded CPU cores on a single chip sharing a common memory subsystem. While these CMP systems can use the extra transistors to provide more computation performance by scaling to more cores, this form of straight-forward scaling may also hit the power wall as well due to the significant slowdown of classical Dennard Scaling in the last few process generations [42, 50]. Dennard Scaling refers to the reduction of transistor operating voltage between successive IC fabrication process generations. In combination with the reduction of switching current from smaller transistor geometry, this voltage reduction has allowed the smaller transistors in the newer process to consume proportionally less dynamic power. As a result, a processor fabricated in a newer process can traditionally be extended to use more transistors without increasing either the power budget or the chip area. However, Dennard Scaling has significantly slowed recently, because lowering the transistor operating voltage any further can lead to transistors operating too close to their threshold voltages, introducing significant leakage current and noise margin reduction. Therefore, simply scaling the current CMP architectures with more CPU cores to consume the extra transistors made available by a newer process will result in a design that consumes more power. In response to this observation, much attention from the computer architecture research community has been focusing on more energy-efficient alternatives. One of the promising alternatives is off-loading computation to graphics processing units (GPUs), a computing paradigm known as *GPU Computing*.

1.1 GPU Computing Potential

GPUs started as fixed function accelerators for real-time 3D graphics rendering in computer games. Over time, GPU vendors have advanced GPU architectures to support various rendering effects as a way to differentiate from their competitors. This endeavor lead to the incorporation of programmable shading in GPUs. With programmable shading, an application can implement a new rendering effect via customizing the 3D rendering pipeline with "shaders", which are snippets of code that calculate the lighting and transformation of polygon vertices and pixels in a 3D digital scene. Since then, newer generations of GPUs have offered more programming flexibility in the shaders. This has eventually evolved modern GPUs into an emerging class of massively parallel compute accelerators that features theoretical throughput tens to hundreds of times higher than any general purpose CPUs available. Recent advances have also made it possible to program GPUs via simple extensions to C programming languages, such as CUDA [120, 122] and OpenCL [89]. These new application programming interfaces (APIs) allow non-graphics applications to harness the computing power of GPUs without using graphics-oriented APIs such as OpenGL and Direct3D.

A key feature that distinguishes GPU from traditional CMP architectures is the amount of exposed parallelism expected from the application. 3D graphics rendering contains plenty of data-level parallelism – the lighting and color of each pixel displayed on screen can usually be computed independently. Graphics applications use 3D graphics APIs such as OpenGL and Direct3D to specify the shader for each displayed pixel in the 3D scene, with the expectation that the GPU will execute the shaders for all pixels concurrently. Similarly, a GPU compute application typically partitions its workload into thousands, if not millions, of GPU threads, each working on a smallest indivisible task in the workload. This is in stark contrast to multithreaded applications for traditional CMP architectures, where the programmer would group the small tasks into a number of threads matching the number of CPU cores in the CMP system.

The abundance of application-exposed thread-level parallelism (TLP) in GPU applications has allowed GPU designers to focus hardware resource on achieving higher computation throughput, without having to maintain single thread performance. Specifically, GPU architectures can use the TLP to keep execution units busy without the expensive microarchitecture mechanisms used by traditional out-of-order cores to aggressively exploit instruction-level-parallelism (ILP) from a single thread. For example, instead of relying on branch prediction to speculatively fetch and execute instructions beyond a unresolved branch operation from a single thread, a GPU can schedule other threads for execution while a subset of its threads are resolving the outcome of their branch operations (hence, no wasted work from branch outcome misprediction). This technique, known as fine-grained multi-threading (FGMT), is discussed in more detail in Chapter 2. GPU architectures also use FGMT to tolerate the long latency from memory accesses. This reduces the need for low latency memory subsystems, and permits optimizations that trade latency for significantly higher memory bandwidth.

Furthermore, GPUs use wide single-instruction, multiple-data (SIMD) hardware to exploit the regularities in computation among different threads in GPU applications. Since scalar GPU threads tend to perform almost identical computation on different data, they are organized into SIMD execution groups called *warps* (or *wavefronts* in AMD terminology) in hardware. Threads in each warp share a common program counter, and they are executed in lockstep on wide SIMD hardware to amortize the control-logic overhead of each thread. Modern GPUs include special hardware to automatically serialize the execution of different subsets of threads that diverge to different control flow paths [56]. In combination with

Table 1.1: Computational efficiency of state-of-the-art CMPs and GPUs in 2014. This comparison uses single-precision floating point performance. The peak floating point performance for the various CMP systems assume the application uses SIMD instruction set extensions. TDP = Thermal Design Point, BW = Bandwidth

Processor	Type	IC Fab. Process	TDP	Peak FP Perf.	Memory BW	Computation Efficiency
NVIDIA Tesla K40	GPU	TSMC 28nm	235 W	4300 GFLOPS	288 GB/s	18.3 GFLOPS/W
AMD R290X	GPU	TSMC 28nm	290 W	5632 GFLOPS	320 GB/s	19.4 GFLOPS/W
Intel Xeon E7 v2	CPU	Intel 22nm	155 W	672 GFLOPS	85 GB/s	4.3 GFLOPS/W
Intel Core i7-4770K	CPU	Intel 22nm	84 W	499 GFLOPS	26 GB/s	5.9 GFLOPS/W
IBM Power 7+	CPU	IBM 32nm	200 W	795 GFLOPS	100 GB/s	4.0 GFLOPS/W

simple compiler analysis, this special hardware abstracts this SIMD organization from the GPU programmer – each scalar GPU thread is free to follow a unique execution path, but at a performance penalty. This abstraction is known as the single-instruction, multiple-thread (SIMT) execution model. Chapter 2 describes a baseline mechanism similar to the ones used in current GPUs to support the SIMT execution model. By combining FGMT with wide SIMD hardware, GPUs can deliver higher computation throughput while consuming less energy per operation than traditional CMP systems.

Table 1.1 illustrates this efficiency boost by comparing the peak floating point (FP) operation throughput and power consumption of different state-of-the-art GPUs and CMPs in 2014. Even with an inferior IC fabrication process (28nm vs. 22nm), the two GPUs manage to deliver $3-4\times$ more floating point operations per Watt (GFLOPS/W, a metric for computation efficiency) than the CMPs in this comparison. Table 1.1 also shows that GPU memory subsystems provide substantially higher bandwidth $(3-10\times)$ than traditional CMPs. Differences in peak performance aside, researchers have demonstrated that harnessing the computing power of GPUs in real world non-graphics application can lead to far more cost-effective computing solutions than solutions using traditional CMP systems comprised of only CPU cores [77, 95].

1.2 GPU Programming Challenges with Irregular Parallelism

Writing functionally correct software for GPUs is now relatively easy for applications with *regular parallelism*. These applications, such as dense matrix multiplication and simple image filtering, feature regular memory access patterns and control flow behavior among threads. While optimizing these applications to fully harness the computing power offered by GPUs is challenging, the regular parallelism in these applications maps naturally well to the SIMD hardware in GPUs. Consequently, the initial GPU implementations of these applications tend to offer a reasonable boost in performance/efficiency over the original CPU version. This gives the software developers incentives to refactor their software to incorporate GPU-accelerated modules.

In contrast, software developers have far more difficulties employing GPUs for applications with *irregular parallelism*. Software developers who attempt to use GPUs to exploit the irregular parallelism in these applications typically face the following risks:

Uncertain Performance/Efficiency Benefit Despite having plenty of data-level parallelism (DLP), these applications have threads with data-dependent control flow behavior and irregular memory access patterns. A group of threads with data-dependent control flow may diverge into different execution path. This behavior can cause underutilization of the SIMD hardware in GPUs, a performance issue in GPU computing known as *branch divergence* (see Chapter 2). Also, the GPU memory system expects adjacent threads to access data from the same memory block, so that it can service all the accesses in parallel by fetching the entire block at once via a wide data bus (this mechanism is known as *coalesced memory access* [122]). However, adjacent threads with irregular memory access patterns tend to access data in different memory blocks. Current GPUs fetch the entire memory block even through only a single word within the block is requested, so the irregular memory access patterns tend to waste a significant portion of the GPU memory bandwidth. The performance penalty from these issues may counteract the throughput advantage of the GPU, causing the GPU-accelerated version of the application to be slower than the initial CPU-only version.

Unpredictable Development Effort Besides introducing a performance penalty, a more important problem with these irregular memory accesses is the possibility of data-races. To guard against data-races without excessively serializing computation, the software developers have to use fine-grained locks, which are prone to deadlocks. GPUs are designed to execute thousands of threads concurrently. More threads, and correspondingly larger problem sizes, exacerbate the challenge of lock-based programming by increasing the amount of debug data analysis required to understand how a deadlock can manifest in the application [8]. Furthermore, implicit synchronization imposed by hardware that handles branch divergence in GPUs can cause some implementations of spin lock to deadlock [129] (see Chapter 4 for a more detailed discussion). The GPU software developer may need to implement *ad hoc*, error-prone data synchronization mechanisms to circumvent the use of locks [108]. Regardless of the solution employed, the developer will need to invest significant effort to verify the design and debug the implementation, just to eradicate every potential data-race and deadlock for every possible input. It is difficult for the software developers to estimate the amount of effort required for this process, making it difficult to fit into a time-driven development cycle.

Potentially Hard-to-Maintain Code Even if the software developer is able to overcome the mentioned challenges and has successfully produced a GPU-accelerated version of the application, the code would likely be hard to comprehend by programmers who do not understand the intricacies in the GPU microarchitecture. These intricacies may have driven many design decisions in the software. Maintaining this GPU-accelerated version of the application will require good understand of these design rationales to avoid introducing bugs/performance bottlenecks that the highly optimized code is designed to avoid in the first place. This adds significant burden onto the software maintainers who need to revisit these design decisions every time they update the code for new features and bug fixes.

In fact, these risks exist for each attempt to further optimize the application, because each optimization may introduce new bugs and does not necessarily produce the expected benefit.

Many research studies have demonstrated that with sufficient programming effort, using current GPUs to accelerate applications with irregular parallelism can provide substantial performance/energy boosts [4, 23, 66, 108, 118]. Nevertheless, the risks for accelerating these applications with GPUs can appear too high to software developers for production software development. The high risks consequently limit the range of applications that may harness the computing power of GPUs.

1.3 Thesis Statement

This dissertation enhances GPU architectures to boost their ability to accelerate applications with irregular parallelism. Each of the two major enhancements introduced by this dissertation addresses a specific risk in the software development of these applications. By reducing the risks in developing GPU applications with irregular parallelism, these enhancements aim to make GPU acceleration practical for these applications in the future.

Thread block compaction (TBC), the first enhancement, is a microarchitecture innovation that robustly boosts the GPU performance when the GPU executes control flow intensive applications that suffer from branch divergence. At each divergent branch, TBC temporarily rearranges scalar GPU threads from multiple warps (SIMD execution groups – see Section 2.3.2) into new warps. It compacts threads with the same branch outcome into the same set of new warps to boost their utility of the SIMD hardware. Threads from these temporary warps are restored into their original warps once they have reached the point in the program where both divergent execution paths converge. This restoration ensures full SIMD efficiency in code regions that do not suffer from branch divergence, and preserves the highly regular memory access patterns in these regions. This property significantly reduces the chance of TBC slowing down an application that does not suffer from branch divergence – a pathological behavior that plagues dynamic warp formation (DWF), the original work that introduced the concept of rearranging threads into new warps to boost SIMD efficiency [56]. The robustly boosted performance from

TBC increases the likelihood that the GPU acceleration can speed up the application. This in turn allows the software developers to tolerate more uncertainty in the benefit from GPU acceleration during the development.

Kilo TM, the second major enhancement, is a novel, scalable, and energy-efficient hardware proposal for supporting transactional memory (TM) on GPUs. With TM, programmers can replace lock-protected critical sections with atomic code regions, called *transactions* [75]. The underlying TM system optimistically executes transactions in parallel for performance and automatically resolves data-races (conflicts) between concurrently executed transactions in a deadlock-free manner. GPU software developers can use transactions instead of fine-grained locks to aggressively exploit the irregular parallelism in their applications. Freed from the concerns for data-races and deadlocks, the developers can verify the aggressively parallelized implementations with significantly less effort. This makes the development effort using transactions more predictable than using locks or other *ad hoc* data synchronization solutions. More importantly, transactions decouple the functional behavior of the code (operations performed by each transaction) from its performance (how the transactions are executed). This decoupling makes codes written with transactions easier to maintain than critical sections protected by fine-grained locks.

Kilo TM aims to support thousands of small, concurrent transactions. This design goal differs from the goal of most existing TM system proposals to support tens of large, unbounded transactions on traditional CMP systems. The unique properties of the GPU memory system and the need to scale to thousands of concurrent transactions have driven Kilo TM to deviate from the traditional cache-based TM system proposals. In particular, Kilo TM cannot rely on invalidation messages in cache coherence protocols to detect conflicts between transactions. Instead, it uses innovative alternatives to avoid exhaustive pair-wise conflict detection between all running transactions. The insights gained from exploring these alternatives in this work should transfer to other data synchronization mechanisms beyond transactional memory.

1.4 Contributions

This dissertation makes the following contributions.

1. It identifies challenges faced by dynamic warp formation running CUDA applications, and devises an improved scheduling policy to address these challenges (Section 3.2).

2. It presents thread block compaction (TBC) [54], a robust version of dynamic warp formation (DWF) [56]. TBC provides better performance than DWF with simpler design complexity (Section 3.3).

3. It extends the immediate post-dominator based reconvergence, a state-of-the-art mechanism for handling branch divergence, with *likely-convergence points* [54] (Section 3.4).

4. It evaluates the performance benefit of TBC. Together with likely-convergence points, TBC provides an average speedup of 22% over a baseline per-warp, stack-based reconvergence mechanism, and 17% versus DWF on a set of GPU applications that suffer significantly from branch divergence (Section 3.6).

5. It proposes the use of hardware transactional memory (HTM) for GPU computing (Chapter 4). In particular, it estimates the performance potential of transactional memory on a set of GPU computing workloads that employ transactions via a limit study with an ideal TM system (Section 4.1). This ideal TM assumes zero overhead for detecting conflicts among concurrent transactions and maintaining the atomicity of each transaction. On average, the GPU workloads running on this ideal TM achieve 279$\times$ speedup over serializing all transactions via a single global lock, and they perform comparably to fine-grained locking.

6. It highlights a set of challenges with employing existing HTM system proposals on GPUs (Section 4.1.1).

7. It proposes Kilo TM, a novel, scalable TM system that is designed specifically for GPUs [57] (Section 4.2). The design combines aspects of value-based conflict detection [37, 124], RingSTM [149], and Scalable TCC [27]

(Transactional Coherence and Consistency) to support 1000s of concurrent transactions without requiring a cache coherency protocol. Kilo TM detects conflicts at word-level granularity and employs various mechanisms to increase transaction commit parallelism.

8. It extends the SIMT [101] hardware to handle control flow divergence due to transaction aborts (Section 4.2.1).

9. It introduces the *recency bloom filter* which incorporates a notion of time and supports implicit, multi-item removal (Section 4.2.5). Kilo TM deploys one small (5kB) recency bloom filter in each GPU memory partition to boost transaction commit parallelism significantly. GPU architectures evaluated in this work have six to twelve memory partitions.

10. It shows that a simple extension of the GPU hardware thread scheduler to control transaction concurrency benefits high-contention workloads (Section 4.2.6).

11. It devises a theoretical framework to prove that Kilo TM satisfies *weak isolation* [16, 73] (Chapter 5).

12. It estimates the energy overhead of Kilo TM on a contemporary GPU design (Section 6.1 and Section 6.7).

13. It enhances the performance and energy efficiency of Kilo TM with *Warp-Level Transaction Management* (WarpTM) (Section 6.2). WarpTM enhances Kilo TM to leverage the thread hierarchy in GPU programming models to amortize the control overhead of Kilo TM and boosts the utility of the GPU memory system.

14. It proposes and evaluates two intra-warp conflict resolution schemes to resolve conflicts within a warp (Section 6.3). A low overhead intra-warp conflict resolution mechanism is crucial in maintaining the benefit from WarpTM.

15. It accelerates the execution of read-only transactions in Kilo TM with *Temporal Conflict Detection* (TCD) (Section 6.4). TCD is a low overhead mechanism that uses a set of globally synchronized on-chip timers to detect con-

flicts for read-only transactions. Once initialized, each of these on-chip timers runs locally in its microarchitecture module and does not communicate with other timers. TCD uses timestamps captured from these timers to infer the order of the memory reads of a transaction with respect to updates from other transactions.

16. It evaluates the combined benefits of WarpTM and TCD and shows that they complement each other (Section 6.7). The two enhancements together improve the overall performance of Kilo TM by 65% while reducing the energy consumption by 34%. Kilo TM with the two enhancements 192$\times$ speedup over coarse-grained locking, and captures 66% performance of fine-grained locking with 34% energy overhead. More importantly, the enhancements allow applications with small, rarely-conflicting transactions to perform equal or better than their fine-grained lock versions. This alludes to the possibility that GPU applications using transactions can be incrementally optimized to reduce memory footprint and transaction conflicts to take advantage of this. Meanwhile the transaction semantics can maintain correctness at every step, providing a low-risk environment for exploring optimizations.

1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 discusses background on contemporary GPU architectures and defines the baseline GPU architecture used throughout this work. It also provides background information on transactional memory (TM).
- Chapter 3 presents thread block compaction (TBC), a microarchitecture innovation that robustly boosts the GPU performance for applications that suffer from branch divergence.
- Chapter 4 highlights the difficulties in fine-grained locking and proposes Kilo TM, the first hardware proposal to support TM on GPU-like accelerators.

- Chapter 5 discusses the correctness of Kilo TM in providing the proper transaction semantic guarantees.
- Chapter 6 compares the performance and energy efficiency of Kilo TM with fine-grained locking and proposes two mechanisms, warp-level transaction management and temporal conflict detection, to boost the efficiency of Kilo TM.
- Chapter 7 discusses related work.
- Finally, Chapter 8 concludes this dissertation and discusses directions for future work for both TBC and Kilo TM.

Chapter 2

Background

This chapter provides the background for the rest of this dissertation. Section 2.1 briefly explains a set of fundamental concepts in parallel computing that are used throughout this dissertation, such as data-level parallelism, irregular parallelism, single-instruction, multiple-data, and data synchronization. Section 2.2 summarizes the background on transactional memory that is relevant to this work. Section 2.3 describes a modern GPU architecture model that serves as the baseline GPU throughout this work. Section 2.4 describes *dynamic warp formation*, a prior work aiming to reduce the GPU performance penalty due to branch divergence. Thread block compaction (introduced in Chapter 3) revises dynamic warp formation, fixing its pathological behaviors with a simple and robust mechanism.

2.1 Fundamental Concepts

This section explains a set of fundamental concepts in parallel computing that are used throughout this dissertation. While many of these concepts are common knowledge in computer architecture, the explanations in this section aim to clarify them in the context of GPU computing.

2.1.1 Data-Level Parallelism

An application contains *data-level parallelism* (DLP) if it contains computations that can operate on many data items at the same time [74]. One simple example of such an application is vector addition, in which each element from one of the source vectors is added to its corresponding element in the other source vector to produce an element in the destination vector. Since the sum for each element in the destination vector does not depend on the sum for another element in the same vector, the sums for multiple elements can be computed in parallel. DLP is commonly found in many scientific computations, multimedia applications and signal processing applications (usually in the form of matrix/vector operations).

Data-level parallelism is commonly discussed in contrast to *task-level parallelism*, which are multiple tasks in an application that can usually operate in parallel [74]. For example, a word processor application may render the user input onto the screen while simultaneously checking the spelling and grammar of the input text. Data-level and task-level parallelism in an application can be distinguished by how the parallelism scales. The amount of data-level parallelism in an application scales with the amount of data to be processed, whereas task-level parallelism increases as more functionality is added to the software.

This work further classifies data-level parallelism into *regular parallelism* and *irregular parallelism*.

2.1.2 Regular Parallelism vs. Irregular Parallelism

An application with *regular parallelism* processes a large pool of data in a regular fashion. In this type of application, the task for processing each data item performs almost identical computation and features a regular, predictable memory access pattern. Classic examples of such an application include dense matrix multiplication and simple image filtering. In both examples, the control flow behavior and the memory access pattern of the computation for each data item is largely independent of the value of the processed item. This decoupling between the processed data and the nature of the computation leads to regular, predictable behavior that can be exploited for optimizations. For example, the programmer can orchestrate the computations so that their memory accesses exhibit spatial and temporal locality, with high certainty that the optimization will yield a speedup for most common inputs. In particular, the programmer may arrange the dense matrix multiplication in tiles, so that computations that read from the same set of input data elements run concurrently to maximize cache hits. Also, the regularity of the tasks reduces the possibility of load imbalance at barriers. This permits an application with regular parallelism to use a single barrier, instead of fine-grained locks, to efficiently synchronize between sets of dependent tasks for communication.

Modern GPUs are designed to exploit regular parallelism with single-instruction, multiple-data (SIMD) hardware to amortize the instruction management overhead and to boost memory system utility. Consequently, the initial GPU implementation of applications with regular parallelism tend to offer a reasonable boost in performance/efficiency over the original CPU version. The stability of these applications' performance across different inputs encourages the software developers to further optimize the code to fully harness the computing power offered by GPUs.

Despite having plenty of data-level parallelism (DLP), an application with *irregular parallelism* processes a large pool of data concurrently in an irregular fashion. Depending on the data being processed, each task may need to perform a variable amount of computation and features irregular memory access pattern that is dictated by the input data. One example of such an application is ray tracing, a 3D graphics rendering algorithm commonly used in film production. In ray tracing, each task traces the light path from a pixel on the image plane (a ray) to one or more light sources in the rendered scene. The ray usually reaches a light source via a series of reflections and refractions with objects in the rendered scene, and each ray traverses through the scene differently, requiring a different amount of computation [4]. The irregularity among tasks makes it difficult to apply optimizations that work well with regular parallelism. For instance, the data-dependent control flow behavior in each task can cause load imbalance (leading to performance penalty for using barriers) and branch divergence (leading to underutilization of the SIMD hardware in GPUs). The irregular memory access pattern also makes it difficult for a programmer to group computations that exhibit temporal and/or spatial locality. Moreover, some applications require the programmer to use locks to protect shared data structures that can be modified by one or more concurrent tasks. The programmer has to verify that for every possible input to the application, the irregular memory access patterns among these concurrent tasks would never result in a deadlock. Overall, the irregularity among tasks poses significant challenges for software developers to parallelize and fine-tune these applications.

In many applications, the irregularity of the concurrent tasks comes from the use of more work-efficient algorithms [72]. Parallel algorithms often perform more operations than their sequential counterparts, and algorithms that reduce this overhead often do so at a cost of increased irregularity. By removing wasted work, the complexities of these algorithms scale significantly better than their more regular, but less efficient counterparts for large working sets found in realistic workloads. One example is sparse matrix operations, which operate directly on sparse matrices stored in a compressed format recording only the location and value of the non-zero entries in matrices. Sparse matrix operations can leverage the sparsity of the matrices to remove wasted computations for entries with zeros, making these operations significantly more efficient than their dense matrix analogs [13, 164]. Even though algorithms with irregular parallelism seem ill-suited for GPU acceleration, the efficiency of these algorithms may compensate for their under-utility of the GPU hardware. Many research studies have demonstrated that with sufficient programming effort, using current GPUs to accelerate applications with irregular parallelism can provide a substantial performance/energy boost versus CPU only solutions [4, 13, 23, 66, 108, 118].

2.1.3 Thread-Level Parallelism

Thread-level parallelism (TLP) refers to the parallelism that is expressed explicitly by the software developers as threads in an application. These threads can run concurrently in the system, and each thread usually progresses independently through its own execution path through the program. Traditional chip-multiprocessors (CMPs) can harness TLP directly by executing each thread on a different processing core. This distinguishes TLP from instruction-level parallelism (ILP), which is the parallelism from independent operations within a single thread [74].

In this work, each thread has its own architectural state, which usually consists of a program counter (PC) and a set of registers. Threads in the same application share the same global memory space, such that when one thread updates the value at a memory location, the updated value will be visible to other threads. In some occasions, one thread may update a shared data structure in memory that is simultaneously used by other threads. These occasions are known as *data-races*.

Unexpected data-races in a multithreaded application can cause the application to behave erratically. Programmers may guard against unexpected data-races through the use of *data synchronization* mechanisms, such as locks, barriers and transactional memory. Section 2.1.4 gives a brief overview of the different data synchronization mechanisms used in existing CMPs and GPUs, and Section 2.2 explains the basics of transactional memory in more detail.

GPU compute applications use TLP to express its inherent task-level and data-level parallelism to the GPU hardware. Section 2.3 explains how GPU hardware uses a combination of SIMD hardware (Section 2.1.5) and fine-grained multithreading (Section 2.1.6) to harness the TLP in an efficient way.

2.1.4 Data Synchronization

In multithreaded programming, a data-race occurs when one thread updates a shared data structure in memory that is currently used by other thread. An unexpected data-race can cause a multithreaded application to behave erroneously in two ways: Two threads may clobber each other's update to the shared data structure, corrupting values in the structure; one thread may only read in part of the update from the other thread, corrupting its view of the shared data structure and the thread's behavior. The programmer can use data synchronization to avoid data-races in their application. This section gives an overview of the data synchronization mechanisms available on modern GPUs. These mechanisms are mentioned throughout this dissertation.

Atomic Operations

Atomic operations are mechanisms provided by modern processor architectures that are capable of updating a memory location atomically with a new value that is computed based on the old value [141]. They are also referred as read-modify-write operations, because each atomic operation involves reading the old value from the memory location, modifying it into a new value, and writing the new value back to memory. These operations differ from normal memory load/store operations – special microarchitecture mechanisms are added to ensure that all three steps are performed atomically with respect to every thread in the system. Most existing implementations only support atomic operations for a single 32-bit/64-bit word in the memory.

Atomic operations serve as the basic primitives for constructing more complex data synchronization mechanisms such as locks, barriers, nonblocking data structures and software transactional memory. Among different types of atomic operations, this work focuses on atomic *compare-and-swap* CAS [82], as it is used in the implementations of many synchronization mechanisms. There are three inputs to an atomic CAS: the memory location to be updated, the expected old value at the location, and a new value. Atomic CAS will only write the new value into the memory location if the current value at the location matches the expected old value. A successful swap returns the old value to the thread, whereas a failed swap returns the new value to indicate a failure. Software can use atomic CAS to emulate arbitrary single-word read-modify-write operation [141]:

1. Read the original value from memory.

2. Compute the new value from this original value.

3. Use atomic CAS to write the new value back to the memory location atomically if the value in memory has not been modified by another thread during the computation.

4. If atomic CAS detects a value different from the original value, repeat steps 1 to 3.

Notice that this emulation requires the computation of the new value from the original value (Step 2) to be idempotent [40], so that it can be repeated upon a CAS failure without accumulative effects.

Modern CPU processors usually implement atomic operations via a cache coherence protocol. A CPU can service an atomic operation by instructing its private cache to maintain exclusive access to the cache block containing the input memory location for the entire duration the atomic operation [34]. Modern GPUs do not have thread-private caches, nor are the per-core caches coherent. Instead, modern GPUs implement atomic operations with a set of execution units located in the access path to the shared last-level cache. Adopted from the raster operation units used for depth testing and pixel blending in 3D graphics [22], these units operate directly on the data stored in the last-level cache, and intercept any conflicting load-/store accesses to these locations during the computation. Similar implementations are also employed in Rigel [87] and IBM's Blue Gene/Q [71].

Barriers

A *barrier* provides global synchronization among a set of threads – none get past the barrier until every thread in the set has arrived [34]. The programmer can use barriers to arrange computations into multiple parallel phases, a programming paradigm known as *bulk-synchronous programming* [160]. In each phase, threads execute a set of independent tasks concurrently to avoid any potential data-race. The barrier between each phase ensures that all tasks in the previous phase are finished before proceeding to the next phase. It also waits until all memory stores from the previous phase have been committed to the global memory, so that tasks in the next phase can then use the update data from the prior phases.

Despite the simplicity of bulk-synchronous programming, load imbalance among threads can cause some threads to have long idle time at the barrier, waiting for other threads to catch up. This makes them more effective for applications with regular parallelism, in which threads are likely to complete their tasks at around the same time.

CMP systems usually implement barriers via the use of atomic operations. The simplest implementation involves each thread atomically incrementing a single counter, and then spinning until the counter equals to the number of threads participating in the barrier.

Modern GPUs provide hardware barrier instructions that synchronize threads within a single thread block (a unit in the GPU thread hierarchy explained in Section 2.3). Instead of using a shared counter in memory, the baseline GPU architecture in this work implements this barrier instruction by extending the per-core register scoreboard and the warp scheduler. Without going through the memory system, the hardware barrier can release its threads within a few cycles after the final thread has arrived. This per-block, low-overhead barrier is commonly used in GPU programming to synchronize accesses to a per-core on-chip scratchpad memory. GPU programmers can implement a global barrier by combining this hardware per-block barrier with atomic operations [166].

Locks

A *lock*, or mutex, is a software data object that allows multiple threads to negotiate mutual exclusion to one or more shared data objects in memory. A thread must *acquire* the lock associated with the shared data objects before accessing the shared data. Each lock may only be acquired by at most one thread at any time, so the thread holding the lock has exclusive access to the shared data. The thread should *release* the lock as soon as it has finished accessing the shared data to allow other threads to access the shared data. Since the shared data is only accessed by one thread at a time, no data-race can occur. If a thread tries to acquired a lock that has been acquired by another thread, the acquisition fails. A thread that has failed to acquire a lock may keep retrying, implementing a blocking lock, or it may fall back to an alternative routine to ensure the rest of the system can make forward progress.

If there are multiple locks in the system, a *deadlock* can occur when there is a cyclical dependency among a set of threads. For example, both thread A and thread B need to acquire both lock X and Y to progress, but each thread is holding one of the two locks and cannot release its own lock unless it has acquired the lock held by the other thread. In this case, both threads are blocked indefinitely, forming a deadlock. One ways to avoid deadlock is to ensure that every thread in the system acquires locks in a globally defined order. For example, a thread may only acquire lock X after acquiring lock Y, but not vice versa. This global lock order eliminates any potential cyclical dependency among threads, so that it is impossible for deadlock to form.

The simplest way to use locks in a software system is to associate all shared data in the entire system with one (or a few) lock(s). The approach, known as *coarse-grained locking*, trades multithreading performance for a simple design. With only a few locks in the system, the programmer can more easily verify that the system is free of deadlocks through exhaustively checking that the locks are always acquired in a globally defined order. However, coarse-grained locking can significantly reduce the amount of TLP in the system. By protecting a large amount of shared data with a single lock, coarse-grained locking can unnecessarily serialize threads that access disjoint parts of the protected shared data. Also, multiple threads that only read from the shared data may execute concurrently without causing any data-race, but are serialized at the lock acquisition.

One way to boost the TLP in the system is to employ more locks in the system, with each lock guarding a smaller portion of the shared data. We use this form of *fine-grained locking* to develop the lock version of the GPU-TM applications used in Chapter 4 and Chapter 6. It is also possible to use *readers-writer locks* to allow multiple threads to read the shared data concurrently while providing mutual exclusion to threads that modify the data [33, 141]. Despite their performance benefits, fine-grained locking and readers-writer locks makes a system significantly harder to verify. Fine-grained locking is more prone to deadlocks than coarse-grained locking and readers-writer locks are prone to starvation of the writer thread. The difficulty in using these more advanced locking mechanisms limits them to expert programmers.

GPU programmers may use a combination of atomic operations and memory fences to implement locks. However, implicit synchronization imposed by the SIMT execution model can introduce deadlocks to unsuspecting programmers. Chapter 4 summarizes a way to circumvent this issue that has been presented by Arun Ramamurthy [129].

2.1.5 Single-Instruction, Multiple-Data (SIMD)

Flynn's taxonomy first introduced SIMD as a class of parallel computing systems akin to vector processors specialized in scientific computing [53]. These computing systems are designed for applications that repeat sets of identical operations across a large array of data, such as vector multiplication. They harness the DLP in each of these operations efficiently via an instruction set architecture (ISA) that operates on vectors of data. Since each instruction performs the identical operation across every element in a vector, the hardware can perform the computation for multiple elements in parallel with multiple identical execution units, all sharing common control hardware. This sharing significantly reduces the hardware cost required to scale up the parallelism in a SIMD computing system. It saves the instruction bandwidth and control logic otherwise required for the extra processors to harness the same amount of DLP. This efficiency boost comes with the restriction that the execution units can only run in lockstep – they all execute the same instruction and advances to the next instruction simultaneously. This restriction distinguishes SIMD computing systems from *multiple-instruction, multiple-data* (MIMD) computing systems, another class of parallel computing system in Flynn's taxonomy that poses no lockstep restriction on the execution units [53]. Under this definition, most multiprocessor architectures are MIMD computing systems.

Nowadays, most contemporary CPU architectures contain SIMD execution units that are accessed via short-vector SIMD ISA extensions: Streaming SIMD Extension (SSE) and Advanced Vector eXtension (AVX) for x86 architectures [80, 154], AltiVec for POWER architectures [78], and NEON for ARM architectures [7]. These SIMD ISA extensions operate on short vector registers (up to 256-bit). Each register may hold a varying number of data elements depending on the size of each element. For example, a 256-bit short vector register can hold 32 8-bit elements or 8 32-bit elements. The length of the short vector register is fully exposed to the software, and the software is responsible for splitting operations on long vectors into multiple loop iterations. Until recently, these SIMD ISA extensions only featured vector load/store operations to contiguous memory blocks. This limits the use of these SIMD extensions to expert programmers. Nevertheless, with the proper software support, these SIMD execution units can boost the computation throughput of a CPU core by 3 to 4$\times$ [138].

Modern GPUs employ wide SIMD hardware (1024-bit to 2048-bit data paths) to exploit the DLP in 3D graphics rendering. Unlike SIMD ISA extensions in CPU architectures, the width of the data path is not directly exposed to the software. Instead, non-graphics GPU APIs, such as CUDA and OpenCL, feature a MIMD-like programming model that allows the programmer to launch a large array of scalar thread onto the GPU. At runtime, the GPU hardware groups scalar threads into SIMD execution groups and runs these SIMD execution groups on SIMD hardware. This execution model, called *single-instruction, multiple-thread* (SIMT), is explained in more detail in Section 2.3.2.

2.1.6 Fine-Grained Multithreading

Fine-grained multithreading (FGMT) is a microarchitecture mechanism that exploits thread-level parallelism (TLP) to increase hardware utility. In FGMT, multiple threads interleave their execution on a FGMT processor. Unlike the traditional multitasking provided by an operating system to time-share a uniprocessor CPU core, switching from one concurrent threads to another on the FGMT processor does not require flushing the pipeline nor offloading architectural states to the main memory. Instead, the FGMT processor has a shared register file that stores the architectural states (program counters and registers) of a pool of concurrent threads sharing the processor. Every cycle, a hardware scheduler selects and issues an instruction from the pool of threads that are not stalled by any data-dependency or resource hazards. Instructions from different threads may coexist in different pipeline stages and execution units inside the FGMT processor.

While FGMT can boost hardware utility in a processor, it also requires a significantly larger register file than the ones in a single-threaded processor. Having a larger register file can increase the cycle-time and can introduce extra operand access energy. This overhead penalizes single-threaded applications, so FGMT has been limited to architectures that optimize for applications with plenty of TLP: CDC 6600 [156], Heterogeneous Element Processor [83], the Horizon architecture [155] and Sun Microsystem's Niagara processor [93, 152].

Similar to FGMT, simultaneous multithreading (SMT) is a microarchitecture mechanism that exploits TLP to boost hardware utilization of an out-of-order, superscalar processor [158]. While FGMT features hardware thread schedulers near the front-end of the processor, SMT leverages the existing superscalar instruction scheduler in the out-of-order processor to execute multiple threads concurrently. The superscalar instruction scheduler is designed to harness instruction-level parallelism (ILP) within a single thread and has the capability to simultaneously issue multiple instructions to multiple functional units. SMT leverages this multi-issue capability to allow instructions from different threads to be issued to different execution units in a single cycle.

Finally, GPU applications are designed to expose an abundant amount of TLP to hardware. Modern GPUs uses the plentiful TLP (thousands of concurrent threads) via FGMT to tolerate pipeline latency within a core as well as memory access latency outside the core. By tolerating the latency of individual threads, GPU designs can focus its hardware resource on improving the overall throughput across all the threads. Section 2.3.3 describes how FGMT is implemented in our baseline GPU architecture.

2.2 Transactional Memory

Transactional memory (TM) is a parallel programming model proposed by Herlihy and Moss as a deadlock-free alternative to lock-based parallel programming [73, 75]. It has recently seen a large growth in interest, and as a result, various commercial CMP systems have incorporated hardware support for TM [71, 80, 81]. Inspired by transactions in database systems, TM simplifies software development for parallel architectures by providing the programmer with the illusion that a block of code, called a *transaction*, execute atomically. With TM, the programmer does not need to write code with locks to ensure mutual exclusion. The underlying TM system optimistically executes transactions in parallel for performance and automatically resolves data-races (conflicts) between concurrently executed transactions in a deadlock-free manner.

Despite having a different programming syntax, TM has been shown to be less error-prone than locks [133], and TM code is easier to understand than lock code [126]. With a well-implemented TM system, transactions can approach, or even supersede, the performance and parallelism only possible with fine-grained locking. Software developers can use transactions instead of fine-grained locks to aggressively parallelize their applications. Freed from the concerns for data-races and deadlocks, the developers can verify the aggressively parallelized software with significantly less effort. TM also allows the development of *composable* concurrent data structures [75], which further boosts programmer productivity. Overall, TM holds the promise of enabling programmers to deliver working, maintainable parallel software with reasonable development effort.

Example 1 shows two conflicting transactions. Transaction A reads from shared variable `H`, multiplies the value by 10, and accumulates the product into shared variable `K`. The set of memory locations read by transaction A, or its *read-set*, consists

Example 1 Example of two conflict transactions. `r` is a private variable in both transactions.

```
Transaction A                  Transaction B

atomic {                       atomic {
 int r = H * 10;                int r = K * 10;
 K = K + r;                     H = H + r;
}                              }
```

of both `H` and `K`, and the set of memory locations updated by transaction A, or its *write-set*, consists of only `K`. Transaction B performs the same operation, but with the roles of `H` and `K` reversed. When both transaction A and B execute concurrently, a conflict can occur if transaction A updates `K` after transaction B has read from it, but before transaction B can update `H` with product computed from the original value of `K`. A TM system detects this conflict dynamically by monitoring memory accesses from both transactions and tracking their read-sets and write-sets (assuming eager conflict detection and resolution, see Section 2.2.2). If transaction A updates `K`, the TM system detects that `K` has been read by transaction B earlier, making `K` part of transaction B's read-set. Since transaction B has not committed yet, this is a conflict. The TM system can then resolve this conflict by aborting transaction B, forcing it to re-execute from the beginning with the updated value of `K` from transaction A. This effectively serializes the two transactions to resolve the potential data-race between them.

Example 2 Example of a transaction (left) and its lock-based analog (right).

```
                               Lock(X[hash(tid)]);
atomic {                       Lock(Y);
 if(X[hash(tid)] > 100){       if(X[hash(tid)] > 100){
  Y += X[(hash(tid)];           Y += X[hash(tid)];
  X[(hash(tid)] = 0;            X[hash(tid)] = 0;
 }                             }
}                              Unlock(X[hash(tid)]);
                               Unlock(Y);
```

Example 2 compares a simple transaction to its lock-based analog. The transaction first reads an element from a shared array `X` according to a hash generated from its thread ID (`hash(tid)`). If the element is greater than 100, it adds the value to another shared variable `Y`, and resets the element to 0. The code within the transaction is enclosed in an atomic block. The lock-based analog of this transaction acquires two locks, one for the element in the shared array (`X[hash(tid)]`) and one for the shared variable `Y`.

The lock-based code proactively serializes all threads at the acquisition of the lock for `Y`. Assuming that in the common case, the element from `X` is smaller than 100, so the element and shared variable `Y` are rarely modified, and this serialization is usually unnecessary. The programmer can attempt to use this assumption to enable more TLP by only conditionally acquiring the lock for `Y` (i.e., moving line 2 to after line 3). However, the program still needs to acquire the lock for the element from `X` in advance, serializing all the threads that just want to read from the element. A well-implemented TM system can exploit this common case automatically to boost TLP. Also, in the absence of a writer, this TM system would permit all threads that are just reading from the same element in `X` to execute concurrently. This illustrates how TM can expose more TLP in an application with little effort from the software developer.

Many processor vendors have started incorporating hardware support for TM in their processors, implementing *hardware transactional memory* (HTM). To date, Intel Haswell [80], IBM's Blue Gene/Q [71] and System Z [81] have hardware support for transactional memory. Legacy CMP systems without hardware support for TM can use TM systems that are implemented as software runtime systems and compilers [73]. These *software transactional memory* (STM) systems rely on code transformation and/or binary translation to insert software routines for transaction management and conflict detection into the TM applications. STM can add some performance overhead to the application. In systems with only a few cores, applications parallelized with STM may run slower than their non-parallelized version.

The rest of this section is divided in two parts. The first part introduces several key correctness criteria for a TM system implementation. These correctness criteria impact the design of Kilo TM presented in Chapter 4, and the discussion in Chapter 5 uses the definition of these concepts to show that Kilo TM has correctly implemented a TM system. The second part discusses some of the design issues involved in implementing a TM system.

2.2.1 Correctness Criteria

This subsection discusses several key criteria that a TM system should meet to execute transactions correctly. The ACID properties define the key attributes that the programming model expects from a transaction [73, 163]. Serializability defines how a set of committed transactions may update the memory system [73, 75, 163]. Opacity ensures that *doomed transactions*, whose memory contents have been clobbered by conflicting transactions, will not corrupt the rest of the system [68].

ACID Properties

A transaction is a set of indivisible operations. When committed, these operations all appear to complete instantaneously in the system. Traditional databases systems guarantee transaction with four properties – atomicity, consistency, isolation, and durability [73, 163]. Together, they are known as the *ACID properties*.

Atomicity From the application's point of view, a transaction is either executed completely, or it has never been invoked at all. A transaction that has failed to commit, due to a conflict or other error, should be aborted. The TM system should ensure that the memory modifications from this transaction do not propagate to the rest of the system. It may either restore the modified locations to their original data, or confine the modification by buffering them. For a successfully committed transaction, the TM system should guarantee that all of its buffered modifications appear to propagate atomically to the rest of the system.

Consistency A transaction should preserve the consistency constraints that are defined by the application. Usually, this refers to a set of invariants among multiple data elements (e.g., different fields in a data structure entry). The application developers should ensure that each transaction is consistent on its own – leading from one consistent memory state to another. A transaction that fails to satisfy this requirement should be aborted to avoid clobbering data in memory.

Isolation Each transaction should execute as if it is executed alone in the system. Concurrently running transactions should not observe the presence of each other. During execution, a transaction should see a memory state with all transactions either committed completely, or not invoked at all. In particular, it should not observe any partial commit from another transaction. Isolation abstracts the concurrency of the system away from the application developer. One way to ensure isolation is to show that the concurrent executions permitted by the TM system are *serializable*.

Durability The effects of the operations performed by a committed transaction are persistent in the system and available to all subsequent transactions. This property is important for databases, which store data on persistent storage like hard disks, but not too relevant to transactional memory.

The ACID properties form a programming interface, a contract between the programmer and the underlying TM/database system. A correctly implemented TM system should satisfy all of the ACID properties.

Serializability

Even though a TM system may execute transactions concurrently, it must ensure that the current execution is *serializable* – equivalent to executing the same set of transactions in a serial order. In a serializable order, every transaction behaves as if it is executed serially one after another. By showing that all transactions committed by a TM system satisfy serializability, one can prove that the TM system enforces atomicity and isolation.

Notice that serializability only requires the concurrent execution of transactions to have at least one equivalent serial order. It does not restrict *which* serial order should be matched by the concurrent execution. The TM system is free to schedule the execution order of transactions for performance, as long as the policy does not lead to starvation [18].

There are many different variants of the definition of serializability. This work focuses on *conflict serializability* [163]. This definition of serializability is based on the *conflict relations* between all committed transactions. A conflict relation between two transactions (A $\leftarrow$ B) occurs when transaction A reads from or writes to a memory location that has been written by the other transaction B. One can construct a conflict graph for a set of committed transactions from their conflict relations, with each conflict relation forming a directed edge between the two transactions defining this relation. If this graph is acyclic, the committed transactions are conflict serializable. This is because one can construct a equivalent serial order by traversing the conflict graph in breath-first order. The correctness discussion in Chapter 5 shows that Kilo TM satisfies conflict serializability.

Opacity

While serializability is concerned about transactions that have been successfully committed, *opacity* defines the behavior of transactions whose memory contents have already been clobbered by conflicting transactions [68, 73]. These transactions, called *doomed transactions*, are destined to be aborted.

A doomed transaction can occur if a TM system, to reduce its conflict detection overhead, allows a conflicting transaction to commit without resolving all of its conflicts. Memory read by a doomed transaction may be inconsistent with the values it has read in before – i.e., the read values may violate invariants set up by the application for correct behavior. Since the doomed transaction will not be committed, values computed using the inconsistent memory inputs will only be visible to the transaction itself. However, the inconsistent inputs may clobber the address calculation in the transaction. This can cause it to access invalid memory regions, potentially triggering page faults and exceptions in a way that is not immediately comprehensible by the programmer. The inconsistent inputs can also cause the transaction to run in an infinite loop (e.g., using -1 instead of 2 for a loop bound).

Many proposed TM systems avoid doomed transactions by aborting a transaction immediately before committing a conflicting transaction. Doing so ensures that the transaction is always observing a consistent view of the memory throughout its execution, i.e., transactions can never proceed in a doomed state. However, implementing this behavior can add significant overheads. It either requires a transaction to check for a conflict whenever it reads from memory, or to resolve all of its conflicts prior to committing to memory.

Recently, several TM proposals opt for *sandboxing*, which use mechanisms to confine the corruption caused by doomed transactions away from the rest of the system [36, 124]. To do so, these TM systems buffer all memory updates made by a transaction as it executes, and handle any access to faulty pages caused by the transaction reading in an incorrect memory address from conflicting memory location. The sandboxing mechanism also needs to detect the infinite loops caused by inconsistent memory values and abort the transaction to avoid deadlock. Sandboxing transactions allows transactions to execute without having to detect conflicts at every memory access. This can reduce the complexity of these TM systems. Kilo TM, presented in Chapter 4, uses sandboxing to support opacity.

2.2.2 Design Space

This subsection briefly summarizes a common design space for a TM system. It discusses the trade-offs for each of the design choices. This set of design choices form a taxonomy to allow one to compare between different TM system designs. This allows us to compare and relate the design of Kilo TM (introduced in Chapter 4) to other existing TM system proposals.

Weak and Strong Isolation

While the ACID properties specifies that execution of a transaction is isolated from other concurrent transactions, it does not specify how it should interact with non-transactional operations. For example, it does not clearly define the interaction between a transaction and load/store operations that are not part of any transaction. A TM system may choose to support either *strong isolation* or *weak isolation* [16, 73, 106]. TM systems that support strong isolation guarantee transactional semantics between transactions and non-transactional operations, whereas those supporting weak isolation only guarantee transactional semantics between transactions.

TM systems supporting weak isolation permit non-transactional operations to inject data-races into transactions. If the TM application needs to share data with non-transactional code, such as legacy libraries, weak isolation would be insufficient to protect transactions against data-races. Also, updates from weakly isolated transactions may not appear atomic to non-transactional operations. These non-transactional operations may behave incorrectly due to the observation of partial updates from a transaction. This can occur in applications under development, when the programmers are still deciding what operations should be in a transaction. Proponents of strong isolation argue that weak isolation makes this process much harder, because the errors from weak isolation appear less intuitive to the programmer [16]. These two issues with weak isolation are some of the main motivations towards supporting strong isolation.

On the other hand, proponents of weak isolation argue for a transactional data-race-free (TDRF) programming model [35]. TDRF can be supported by weak isolation, and is analogous to data-race-free memory consistency models adopted by high-level programming languages such as C++ and Java [147].

Since strong isolation requires monitoring non-transactional memory accesses, implementing them on STM is difficult (but not impossible [2, 140]). On the other hand, HTM implementations that extend the cache coherence protocol for conflict detection are already monitoring non-transactional memory accesses for cache coherence. These HTM systems can support strong isolation with little overhead. For systems without cache coherence protocols, such as GPUs, it remains unclear whether supporting strong isolation is worth its overhead. Kilo TM, as presented in this dissertation, supports weak isolation. Section 8.2.2 discusses potential ways to extend Kilo TM to support strong isolation.

Conflict Detection and Resolution

Since the memory locations accessed by each transaction are unknown prior to its execution, *conflicts* can occur when multiple transactions execute concurrently. A conflict between two transactions occurs when both transactions have transactionally accessed the same data, and one of the accesses is a write. A TM system is responsible for *detecting* these conflicts, and *resolving* these conflicts before they generate errors in the rest of the system. Both conflict detection and conflict resolution of a TM system can be classified according to *when* the TM system detects a conflict and *when* it resolves the detected conflict [113].

With *eager conflict detection*, the TM system detects the conflict as soon as it has occurred. This is done in lock-based STMs and most HTMs that extend cache coherence protocols [75, 111, 113, 145, 157, 170]. In these HTMs, the detection mechanism piggybacks on the invalidation messages or exclusive access requests generated by a write to a cache line. These messages are relayed to the sharers (or the current exclusive writer) of the same cache line, informing the recipients of the existence of a conflicting transaction.

With *lazy conflict detection*, the TM system defers the detection, possibly until one of the transactions attempts to commit. By deferring the detection, the TM system may aggregate and compress the communication required for the detection (e.g., broadcasting the read-set and write-set of a transaction using bloom filters [26]), or it may reduce the frequency of conflict detection by only doing it once after the transaction has finished execution [67, 124, 149].

A TM system with eager conflict detection may choose to resolve the conflict immediately, doing *eager conflict resolution*. Alternatively, it may defer the resolution until one of the conflicting transactions attempts to commit, doing *lazy conflict resolution* [145, 157]. A TM system with lazy conflict detection can employ lazy conflict resolution. With either policy, the TM system can eliminate the conflict by aborting one of the conflicting transactions.

With eager conflict detection, the TM system may alternatively stall one of the conflicting transactions, and resume its execution after the other transaction has finished (committed or aborted). This avoids aborting a transaction and wasting away the work it has already done. Also, direct update (discussed in Section 2.2.2) requires eager conflict detection and resolution to avoid doomed transactions. However, resolving conflict eagerly may lead to dueling upgrades, where two transactions keep aborting each other so that neither can ever commit [18]. Lazy conflict resolution avoids this forward progress concern by having the TM system prioritizing the first transaction that attempts to commit. A TM system may dynamically switch between the two policies to capture the benefits from both policies [145, 157].

The TM system may also *validate* a transaction – checking whether the transaction has experienced conflicts. In this work, we distinguish validation from other forms of conflict detection in that a validation only detects the *existence* of conflicts, but not the exact transactions involved in the conflicts. A transaction may validate eagerly, at every transactional memory access, or lazily, only before it attempts to commit. With either policy, the transaction can only resolve the detected conflict by aborting itself. We call this *self-abort*. Kilo TM performs value-based validation lazily for each transaction and resolves conflict via self-abort (explained in Chapter 4). Chapter 6 augments Kilo TM with a novel eager conflict detection mechanism, called temporal conflict detection, to accelerate read-only transactions.

Version Management

When a transaction executes, it may make updates to memory. These updates should remain invisible to the rest of the system until the transaction commits. This creates two versions of data for each updated memory location: one visible to the transaction itself, and the other visible to the rest of the system. *Version management* of a TM system refers to how it manages the versions created by these updates.

TM systems with *direct update*, or eager version management, allows the transaction to update the memory location in global memory during its execution. The original version in global memory is stored in an *undo log* [113, 170]. If a transaction is aborted, the TM system rolls back memory updated by this transaction through restoring those memory locations with values from its undo log. Since the updated data are already in global memory, the TM system needs to ensure that they are not visible to other transactions via *eager conflict detection and resolution*. Another transaction that attempts to access locations updated by this transaction immediately invokes a transaction resolution manager, which either aborts or stalls one of the transactions. Committing a transaction with direct update simply involves discarding the undo log. Since the commit does not involve updating any global metadata, multiple conflict-free transactions may commit in parallel.

TM systems with *deferred update*, or lazy version management, provide a buffer to contain the memory updates from each running transaction. Each transactional read needs to check this *write buffer* to see if it should read from the buffer for a transactional version of the data. In many HTM implementations, the write buffer is implemented using a private L1 data cache, and therefore does not add much overhead to transaction execution [26, 27, 75, 111, 145]. The L1 data cache is extended to withhold coherence visibility of transactional data, making them invisible to the rest of the system. Since the content of the write buffer is invisible to the rest of the system, multiple conflicting transactions may execute concurrently. The TM system can use lazy conflict detection and resolution to resolve conflicts among these transactions at a later time to ensure forward progress. At commit, contents in the write buffer are made visible to the global memory atomically. A TM system with deferred update can enforce this atomic propagation by serializing the commit of each transaction [26], or it may employ a more complex protocol to allow multiple non-conflicting transactions to commit in parallel [27]. Moreover, since a transaction can exceed the capacity of the L1 cache, a TM system that uses the L1 cache as a write buffer needs to handle the write buffer overflow.

Kilo TM employs deferred update so that it can perform value-based validation lazily for each transaction. Chapter 4 explains how Kilo TM stores the write buffer of each transaction in a linear log.

Bound on Transaction Footprint and Irrevocable Operations

A TM system supports *bounded transactions* if it restricts the size of the memory footprint that can be accessed by a transaction. These restrictions usually originate from using the L1 cache as a buffer for transactional data in HTM systems. When the memory footprint no longer fits in the buffer, the TM system aborts the transaction. It then informs the application of the overflow, and lets the application handle it via alternative means. All existing HTMs implemented on real CMP systems have this restriction [71, 80, 81]. As a result, the runtime TM system usually complements the HTM with a STM system to handle the rare, large-footprint transactions, an approach known as hybrid TM [39, 45, 111].

A HTM system can employ specific mechanisms to support transactions that overflow the write buffer. The simplest mechanism is to serialize the execution of all overflowing transactions [17]. These overflowing transactions may execute concurrently with other non-overflowing transactions, but always have the top priority during conflict resolution.

A TM system can also use this implicit serialization to elegantly support *irrevocable operations*. These operations, such as I/O operations and page faults, cannot be rolled back when the transaction is aborted.

Kilo TM supports unbounded transactions by storing the read-set and write-set

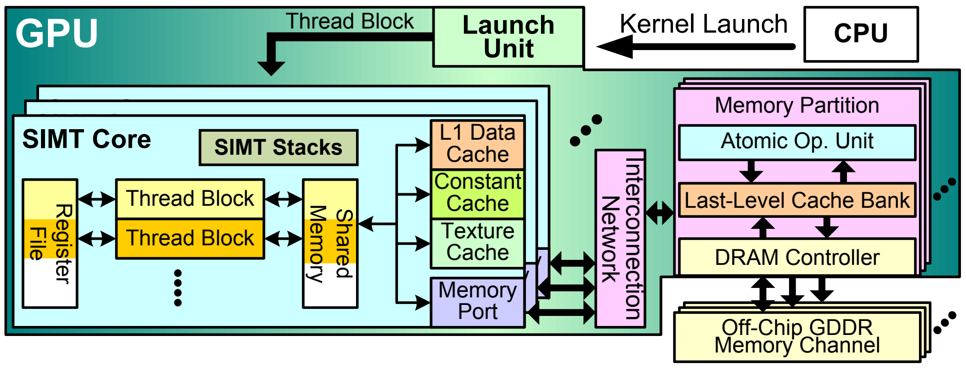


Figure 2.1: High-level GPU architecture as seen by the programmer.

of a transaction in linear logs that can be spilled to DRAM automatically. The GPU TM applications evaluated in this dissertation do not contain irrevocable operations. Nevertheless, Kilo TM may adopt the implicit serialization approach described above to support irrevocable operations.

Nesting

Nesting occurs when an outer transaction contains one or more inner transactions. This can happen when a transaction calls another transactional subroutine. The support for nesting with TM simplifies the development of composable concurrent software.

The exact semantics for nested transactions has yet to be settled [73]. In this work, Kilo TM supports nested transactions by *flattening* them – a conflict detected in an inner transaction aborts the outer transaction.

2.3 GPU Architectures

This section describes the baseline GPU architecture used throughout this dissertation. The description covers a generic architecture with a number of design parameters that can be configured to model different GPU hardware designs. The exact parameters used in the evaluations of the subsequent chapters are presented in the methodology section in each of the chapters.

2.3.1 Programming Model

Figure 2.1 shows the high-level overview of our baseline GPU architecture. A GPU application starts on the CPU and uses a compute acceleration API such as CUDA or OpenCL to launch work onto the GPU [89, 120, 122]. Each launch consists of a hierarchy of *scalar threads*, called a *grid* in CUDA. All scalar threads in a grid execute the same *compute kernel*. The thread hierarchy organizes threads as *thread blocks*. Each block is dispatched to one of the heavily multi-threaded SIMT cores as a single unit of work. It stays on the SIMT core until all of its threads have completed execution. A SIMT core is similar to a *Streaming Multiprocessor* (SM) in NVIDIA GPUs, a *Compute Unit* in AMD GPUs, or an *Execution Unit* (EU) in Intel GPUs. Threads within a block can communicate via an on-chip scratchpad memory called *shared memory* (*local memory* in OpenCL), and can synchronize quickly via hardware barriers. The SIMT cores access a distributed, shared, read-/writeable last-level (L2) cache and off-chip DRAM via an on-chip interconnection network.

The application may launch a thread hierarchy that far exceeds the GPU on-chip capacity. The GPU launch unit automatically dispatches as many thread blocks as the GPU on-chip resources can sustain, and dispatches the rest of the thread hierarchy as resources are released by completed thread blocks. This hardware-accelerated thread spawning mechanism distinguishes GPUs from more traditional vector processors and multi-core processors. It allows GPU applications to decompose their workloads into as many threads as possible without introducing significant overhead.

The memory model in CUDA and OpenCL features multiple memory spaces. These memory spaces share storage in the on-board DRAM on the GPU hardware, except for the shared and host memory space. Each memory space has its own functional semantics and performance characteristics that reflect its microarchitecture implementations in GPU hardware. In CUDA, each thread can access the following memory spaces [122]:

Global memory space contains data that can be accessed by all threads running on the GPU. Its capacity is limited by the amount of on-board DRAM on the GPU.

Table 2.1: Memory space mapping between CUDA and OpenCL

CUDA	OpenCL
Global memory	Global memory
Local memory	Private memory
Shared memory	Local memory
Constant memory	Constant memory
Texture memory	Image object
Host memory	Host pointer

Local memory space contains data that is private to a single GPU thread. The GPU application typically uses this memory space for register spilling and storage for thread-private memory structures/arrays.

Shared memory space contains data that can be shared among threads in the same thread block. It represents the scratchpad memory in each SIMT core, and has limited capacity.

Constant memory space contains read-only data that remains unchanged throughout a kernel launch. These data are cached in a special constant cache in the GPU memory system. In some GPU architectures, it is used for storing kernel parameters that are passed from the host CPU at the kernel launch.

Texture memory space contains texture data that are accessed via the texture unit with special texture fetch instructions. The texture data are cached in read-only texture caches. Before GPUs feature caches for data stored in global and local memory spaces, many early GPU compute applications store read-only data as textures to make use of these read-only texture caches.

Host memory space represents data in the system main memory that belongs to the host CPU. GPU threads can use this memory space to communicate directly with the host CPU thread.

Table 2.1 shows the mapping of memory spaces between CUDA and OpenCL [89].

2.3.2 Single-Instruction, Multiple-Thread (SIMT) Execution Model

Modern GPUs employ wide SIMD hardware to exploit the DLP in GPU applications. Instead of exposing this SIMD hardware directly to the programmer, GPU computing APIs, such as CUDA and OpenCL, feature a MIMD-like programming model that allows the programmer to launch a large array of scalar threads onto the GPU. Each of these scalar threads can follow its unique execution path and may access arbitrary memory locations. At runtime, the GPU hardware executes groups of scalar threads in lockstep on SIMD hardware to exploit their regularities and spatial localities. This execution model is called single-instruction, multiple-thread (SIMT) [101, 119].

With the SIMT execution model, scalar threads are managed in SIMD execution groups called *warps* (or *wavefronts* in AMD terminology). In the CUDA programming model, each warp contains a fixed group of scalar threads throughout a kernel launch. This arrangement of scalar threads into warps is exposed to the CUDA programmer/compiler for various control flow and memory access optimizations [122]. In Chapter 3, we refer to warps with this arrangement as *static warps* to distinguish them from the *dynamic warps* that are dynamically created via dynamic warp formation or thread block compaction.

Ideally, threads within the same warp execute through the same control flow path, so that the GPU can execute them in lockstep on SIMD hardware. Given the autonomy of the threads, a warp may encounter a *branch divergence* when its threads diverge to different targets at a data-dependent branch. Modern GPUs contains special hardware to handle branch divergence in a warp. Section 2.3.3 describes the SIMT stack, which handles branch divergence in our baseline GPU architecture by serializing the execution of the different targets. This serialization correctly handles the branch divergence, but with a performance penalty. Thread block compaction, introduced in Chapter 3, is a novel alternative that reduces this performance penalty.

Notice that thread blocks and warps are two orthogonal organizations of scalar threads. Conceptually it is possible for two scalar threads from different thread blocks to be grouped into the same warp. However, existing GPU architectures restrict each warp to only contain threads from the same thread block. With this restriction, one may treat warps as yet another level of thread hierarchy – each thread block is consisted of a set of warps running on the same SIMT core. This restriction is imposed to simplify the implementations of various features. For example, it allows the thread spawning mechanism in GPU architectures to manage

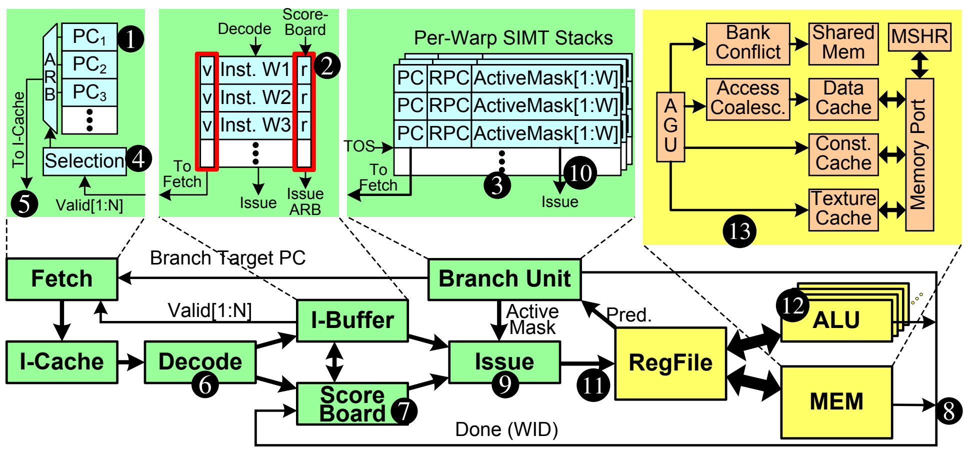


Figure 2.2: SIMT core microarchitecture of a contemporary GPU. N = #warps/core, W = #threads in a warp.

hardware resources allocated to a thread block, such as slots in the warp scheduler and register spaces, at warp-level granularity. It also simplifies the implementation of per-block barriers by ensuring that a warp can only update the barrier arrival counter for at most one thread block.

2.3.3 Microarchitecture

Figure 2.2 illustrates the multi-threaded microarchitecture within a SIMT core. We defined this microarchitecture, described below, by considering details found in recent NVIDIA patents [30, 31, 102]. In our evaluation of thread block compaction in Chapter 3, we approximate some details to simplify our simulation model: We model the fetch unit as always hitting in the instruction cache, and allow an instruction to be fetched, decoded and issued in the same cycle. We also employ a simplified scoreboard that forbids concurrent execution of subsequent instructions from the same warp and a unified pipeline that services both ALU and MEM instructions.

Fine-Grained Multithreading Hardware

Each SIMT core interleaves up to N warps on a cycle-by-cycle basis. The number of warps that run concurrently on the SIMT core (N) depends primarily on the register capacity of the SIMT core and the number of registers required by each thread, as well as the availability of several other hardware resources. Each warp has a program counter (PC) in the fetch unit (see ❶ in Figure 2.2), a dedicated slot (❷) in the instruction buffer, and its own stack (❸) to manage branch divergence *within* that warp.

Each slot (❷) in the instruction buffer contains a v-bit indicating when an instruction is present, and an r-bit indicating that it is ready for execution. Every cycle the fetch unit selects the PC for a warp with an empty instruction slot (❷), and fetches the corresponding instruction from the instruction cache (❺). The instruction is decoded (❻) and placed in an empty slot in the instruction buffer (❷). This instruction waits in the instruction buffer until its ready bit (❷) is set by the scoreboard (❼), indicating the completion of dependent prior instructions from this warp. Instructions within a warp execute in-order. Our evaluation of thread block compaction in Chapter 3 employs a simple scoreboard that only tracks prior instruction completion for each warp. On the other hand, our evaluation of Kilo TM in the rest of this dissertation uses a more advanced scoreboard that can track per-warp register dependencies [31], potentially allowing multiple instructions per warp in the pipeline.

The issue logic (❾) selects a warp with a ready instruction in the instruction buffer to issue for execution. As an instruction issues it acquires the active mask (❿) from the top entry on the corresponding warp's reconvergence stack in the branch unit. The active mask disables threads in the warp that should not execute due to branch divergence. Once issued (⓫), the slot in the instruction buffer (❷) that contained the instruction is marked *invalid*, signaling the fetch unit that it may fetch the next instruction for this warp.

After a branch from a warp is decoded, none of its instructions can be fetched, effectively stalling the warp, until the branch outcome is known. In the meantime, the fetch unit fetches instructions for other warps in the SIMT core. Stalling the warp until its branch target is known removes the need for branch predication hardware, and eliminates any unnecessary instruction fetch.

The issued instruction fetches its operands from the register file. It is then executed in the corresponding pipeline (ALU ⓬ or MEM ⓭). Upon completion, the instruction writes its results to the register file and notifies the scoreboard (❽).

When the scoreboard detects that the next fetched instruction for the corresponding warp no longer has any pending register dependency hazard, it updates the r-bit (❷) of this instruction in the instruction buffer. This allows the warp to be selected by the issue logic in subsequent cycles.

Handling Branch Divergence with SIMT Stack

Each warp has a SIMT stack (❸) to handle its branch divergence. The SIMT stack serializes the execution of different subsets of threads that diverge to different control flow paths. We summarize the SIMT stack mechanism in our baseline below [56, 59].

If some threads in a warp have different outcomes when a branch executes, i.e., the branch diverges, new entries are pushed onto the warp's SIMT stack. Each entry contains a reconvergence PC (RPC) which is set to the immediate post-dominator of the branch – the closest point in the program that all paths leaving the branch must go through before exiting the function [114]. The immediate post-dominator of each branch is obtained via an offline analysis during compilation, and it is encoded in the branch instruction (or encoded via a special instruction). Each bit in the active mask indicates whether the corresponding thread follows the control flow path corresponding to the stack entry. The PC of the top-of-stack (TOS) entry indicates the target path of the branch. Reaching the reconvergence point is detected when the next PC equals the RPC at the TOS entry. When this occurs, the top of the stack is popped (current GPUs use special instructions to manage the stack [5, 6, 30, 100]). This switches execution to the next branch target that is to be executed by the other subset of threads. After all threads have reached the reconvergence point, the TOS entry will reveal a full active mask with the reconvergence PC of the divergent branch, indicating that the threads have reconverged.

Figure 2.3 shows an example of how a SIMT stack handles two levels of branch divergence encountered by a warp with four threads (1, 2, 3, 4).

1. When the warp starts to execute basic block A, its SIMT stack consists of a single entry with a fully populated active mask (1111), indicating that every thread in the warp is active.

2. The warp has encountered a divergent branch at the end of basic block A,

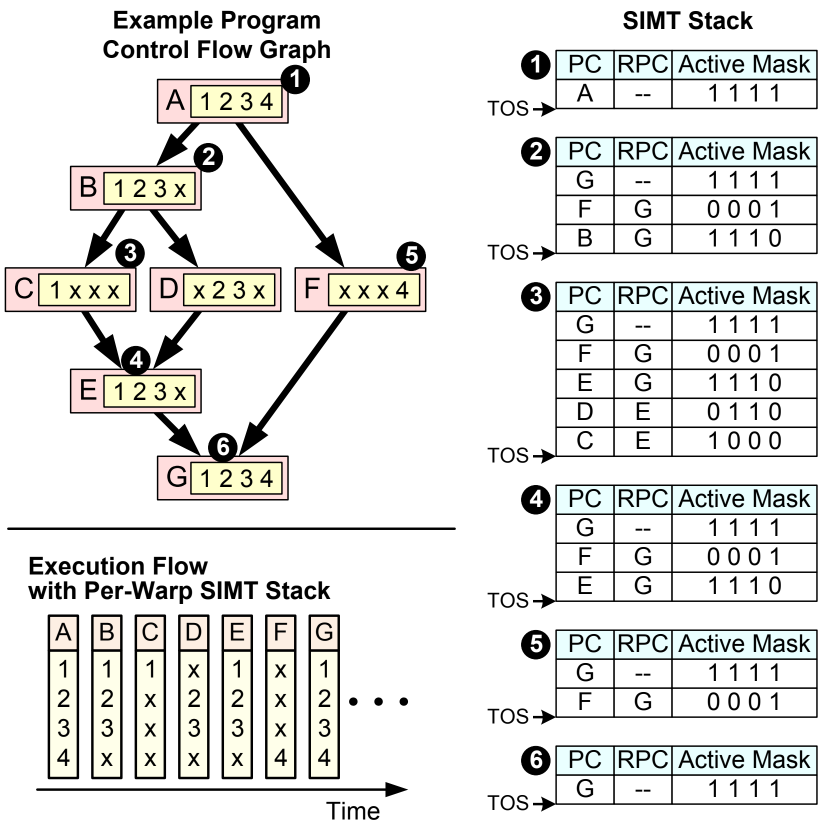


Figure 2.3: Example operation of a SIMT stack. The number label of each SIMT stack state on the right shows the warp's execution state at the same number label on the control flow graph.

with thread (1, 2, 3) diverging to block B, and thread (4) diverging to block F. The warp pushes two new entries onto the SIMT stack, with the RPC in both entries set to basic block G, the immediate post-dominator of block A. Each entry has a partially disabled active mask to indicate the subset of threads that are active for the target (1110 for block B and 0001 for block F). The warp also modifies the PC at the bottom of the stack to basic block G.

3. The warp has encountered another divergent branch at the end of the basic block B. The warp pushes two more new entries onto the SIMT stack, each with the RPC set to basic block E. One entry corresponds to the execution

of basic block C with only thread (1) active (active mask = 1000); another entry corresponds to the execution of basic block D with thread (2, 3) active (active mask = 0110).

4. When the warp has finished executing block C, it detects that the next basic block equals to the one stored in RPC of the TOS entry in its SIMT stack. It pops the stack to switch execution to block D. It also pops the stack again after executing block D, revealing the reconvergence entry at basic block E. This allows threads (1, 2, 3) to execute block E together.

5. After executing block E, the warp detects that the next PC equals to the RPC of the TOS entry. It pops the SIMT stack to switch execution to block F, with only thread (4) active.

6. After executing block F, the warp pops the stack again to reveal the top-level entry with a full active mask for block G. The diverged threads in the warp now reconverge back together to execute block G with full SIMD efficiency.

Through the above interactions with the SIMT stack, the warp executes the example program with the execution flow shown in Figure 2.3.

While the SIMT stack allows divergent threads to reconverge at the earliest *statically* known convergence point, it may result in low SIMD efficiency in applications with deeply nested data dependent control flow or loop bounds that vary across threads in a warp. In these situations different threads within a warp may follow different execution paths.

Memory Subsystem

When a memory instruction issues (⓭ in Figure 2.2), the address generation unit (AGU) generates addresses for each thread in the warp. For each memory instruction, each scalar thread in the warp can generate a scalar memory access. These accesses are served in parallel by the memory subsystem in the SIMT core. Shared memory accesses (accesses to on-chip per-core scratchpad memory) are served by 32 shared memory banks. Accesses contending for the same bank are serialized. For *global* and *local* memory spaces [122], accesses from different threads in the same warp to the same 128-Byte memory chunk are merged (coalesced) into a single wide access. The L1 data cache services one wide access per cycle. The constant cache operates similarly, except accesses from a warp to different addresses are serialized without coalescing. Texture accesses are serviced by the texture unit, which accesses a texture cache [69].

The L1 data caches in the SIMT cores are not coherent [121]. They can be used to cache read-only shared data as well as thread-private data. However, GPU applications (such as the TM applications evaluated in Chapter 4 and 6) can store shared data in the global memory space so that it can be updated by threads from different SIMT cores. To avoid access to stale (non-coherent) data, these applications can configure the GPU architecture (via a compiler flag [122]) so that all global memory accesses skip the L1 cache. In this configuration, they are serviced directly by the L2 cache bank at the corresponding memory partition. The accesses from each thread in the same warp are still coalesced into 128-Byte wide accesses. Each SIMT core can inject one wide memory access into the on-chip interconnection network per cycle.

Each thread can store thread-private data and spilled registers in a private *local memory space* [122]. The local memory is stored in off-chip DRAM and cached in the per-core L1 data cache and the shared L2 cache. It is organized such that consecutive 32-bit words are accessed by consecutive scalar threads in a warp. When all the threads in a warp are accessing the same address in their own local memory space, their accesses fall into the same cache line in the L1 cache and are serviced in parallel in a single cycle.

Current GPUs provide hardware atomic operations for simple single-word read-modify-write operations [89, 122]. The SIMT cores send atomic operation requests to a set of raster operation units in the memory partitions (Atomic Op. Unit in Figure 2.1) to perform these read-modify-write operations to individual locations atomically within the memory partitions [22]. Programmers can use these atomic operations to implement locks.

Memory Partition

GPU features a distributed shared memory architecture that is implemented with a set of memory partitions. The physical linear memory address space is interleaved among these partitions in chunks of 256 Bytes. This fine-grained division of the address space reduces the likelihood of load imbalance at a particular memory partition. A memory access that cannot be serviced within the SIMT core is sent, through the on-chip network, to the memory partition that corresponds to the accessed memory location. Each memory partition contains an off-chip DRAM channel, and one or more L2 cache banks that cache data from the off-chip DRAM. The L2 cache is coherent – each bank is responsible for part of the physically linear memory address space, so that no two banks store the same data.

Each memory partition also contains a set of atomic operation units in the memory partitions (Atomic Op. Unit in Figure 2.1). These atomic operation units work with the L2 cache logic to perform single-word read-modify-write operations atomically.

2.4 Dynamic Warp Formation

Dynamic warp formation (DWF) [56, 59] improves the performance of GPU applications that suffer from branch divergence by rearranging threads into new *dynamic warps* in hardware. The arrangement of scalar threads into *static warps* is an arbitrary grouping imposed by the GPU hardware that is largely invisible to the programming model. Also, with GPU implementing fine-grained multi-threading to tolerate long memory access latency, there are many warps in a core, hundreds of threads in total. Since these warps are all running the same compute kernel, they are likely to follow the same execution path, and encounter branch divergence at the same set of data-dependent branches. Consequently, each target of a divergent branch is probably executed by a large number of threads, but these threads are scattered among multiple static warps, with each warp handling the divergence individually. DWF exploit this observation by rearranging these scattered threads that execute the same instruction into new dynamic warps. At a divergent branch, DWF can maintain high SIMD efficiency by effectively compacting threads scattered among multiple diverged static warps into several non-divergent dynamic warps. In this way, DWF can capture a significant fraction of the benefits of MIMD hardware on SIMD hardware.

However, DWF requires warps to encounter the same divergent branch within a short time window. As a result, the warp scheduling policy can have a significant impact on DWF [56, 59]. Chapter 3 shows how the best performing scheduling policy for DWF from prior work suffers from a two major of performance pathologies: (1) A greedy scheduling policy can starve some threads, leading to a SIMD efficiency reduction; (2) Thread regrouping in DWF increases non-coalesced memory accesses and shared memory bank conflicts. These pathologies cause DWF to slowdown many existing GPU applications. Chapter 3 also shows how applications relying on implicit synchronization in a static warp execute incorrectly with DWF.

2.5 Summary

In this chapter, we have explained a set of fundamental concepts in parallel computing that are relevant to this work. We have also summarized important aspects of transactional memory that impacts the design of Kilo TM in Chapter 4, 5 and 6. We have also presented a modern GPU architecture that serves as our baseline in subsequent chapters. Finally, we have briefly discussed the insight behind dynamic warp formation, which forms the basis of our work on thread block compaction in Chapter 3.

Chapter 3

Thread Block Compaction for Efficient SIMT Control Flow

In this chapter, we propose *thread block compaction* (TBC), a novel hardware mechanism for improving the performance of applications that suffer from control flow divergence on GPUs. A version of this chapter has been published earlier [54].

The development of TBC is motivated by our investigation of performance pathologies that cause dynamic warp formation (DWF), our prior work tackling the same goal (See Chapter 2), to slowdown some GPU applications. These pathologies, presented in Section 3.2, can be partially addressed with an improved scheduling policy that effectively separates the compute kernel into two sets of regions, divergent and non-divergent (coherent) regions. The divergent regions benefit significantly from DWF, whereas the coherent regions are free of branch divergence but are prone to the DWF pathologies. We found that the impact of the DWF pathologies can be significantly reduced by forcing DWF to rearrange scalar threads back to their static warps in the coherent regions.

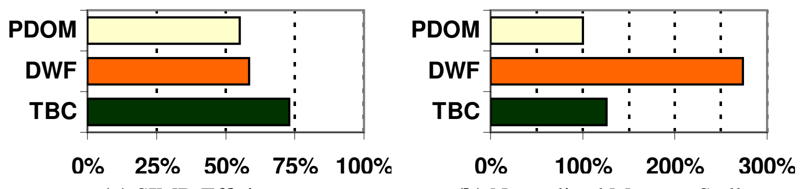
TBC builds upon this insight with the observation that rearrangement of threads into new dynamic warps continually does not yield additional benefit. Instead, the rearrangement, or *compaction*, only needs to happen right after a divergent branch, the start of a divergent region, and before its reconvergence point, the start to a coherent region. We note the existing per-warp SIMT stack (described in Chapter 2) implicitly synchronizes threads diverged to different execution paths at the reconvergence point of the divergent branch, merging these diverged threads back into a static warp before executing a coherent region. One can extend the SIMT stack to encompass all warps executing in the same core, forcing them to synchronize and compact at divergent branches and reconvergence points to achieve robust DWF performance benefits. However, synchronizing all the warps within a core at each divergent branch for compaction can greatly reduce the available thread-level parallelism (TLP). GPU architectures rely on the abundance of TLP to tolerate pipeline and memory latency.

TBC settles for a compromise between SIMD efficiency and TLP availability by restricting compaction to only occur within a *thread block*. GPU applications usually execute multiple thread blocks concurrently on a single core to overlap the synchronization and memory latency. TBC leverages this software optimization to overlap the compaction overhead at divergent branches – when warps in one thread block synchronize for compaction at a divergent branch, warps in other thread blocks can keep the hardware busy. Section 3.3 describes how the per-warp SIMT stack is extend to encompass warps in a thread block, and how the implicit synchronization can simplify DWF hardware.

This chapter also describes an extension to immediate post-dominator (IPDOM) based reconvergence called *likely-convergence points* (LCP). Section 3.4 describes how using IPDOM as the reconvergence point in applications with unstructured control flow can miss some opportunities to reconverge the warp before the immediate post-dominator. We show how a SIMT stack (per-warp and per-thread block) can be extended with these likely-convergence points to achieve higher SIMD efficiency.

Our simulation evaluation in Section 3.6 quantifies that TBC with LCP achieves an overall 22% speedup over a per-warp SIMT stack baseline (PDOM) for a set of divergent GPU applications, while introducing no performance penalty for a set of coherent GPU applications. Our analysis shows that TBC achieves a significantly higher SIMD efficiency versus PDOM or DWF (Figure 3.1(a)) and fewer memory pipeline stalls compared to DWF (Figure 3.1(b)).

The rest of this chapter is organized as follows: Section 3.1 classifies the workload used in this chapter according to their default SIMD efficiency. Section 3.2 discusses our findings on the various DWF pathologies. Section 3.3 describes TBC



(**a**) SIMD Efficiency (**b**) Normalized Memory Stalls

Figure 3.1: Overall performance of TBC (details in Section 3.5).

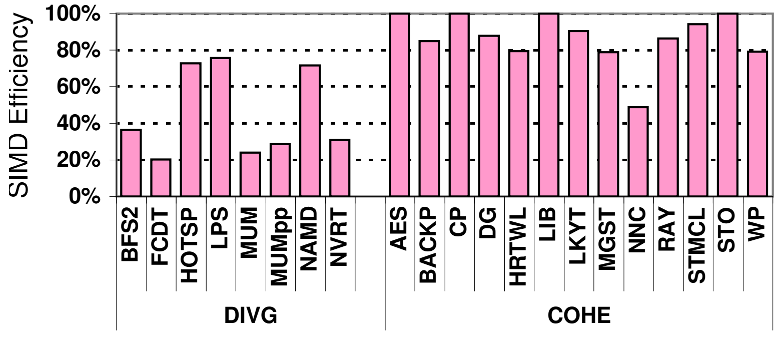


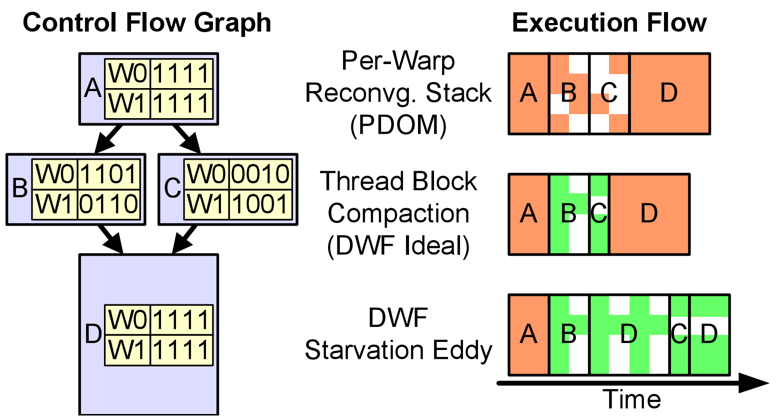
Figure 3.2: SIMD efficiency of GPU applications used in our evaluation. See Section 3.5 for methodology.

in detail. Section 3.4 describes likely-convergence points and how they may be applied to an existing SIMT stack. Section 3.5 describes our evaluation methodology. Section 3.6 presents results, Section 3.7 estimates the implementation complexities of TBC, and Section 3.8 summarizes this chapter.

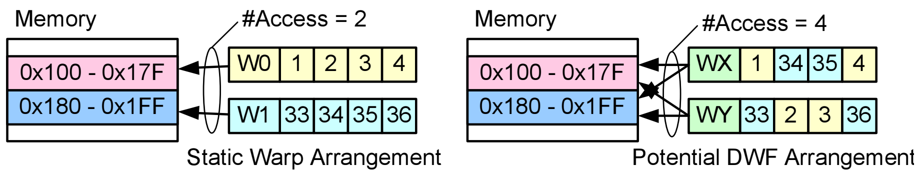
3.1 Workload Classification

In this work, we use SIMD efficiency to identify CUDA applications that diverge heavily (DIVG) but also study a representative set of *coherent* (COHE) applications[1] in which threads in a warp follow the same execution paths (Figure 3.2). Here SIMD efficiency is the average fraction of SIMD processing elements that perform useful work on cycles where the SIMD processing unit has a ready instruc-

[1]The graphics community has long used the term coherent to mean that different threads access adjacent memory locations which is important for DRAM performance. This usage is different from that in the computer architecture community, but equally well entrenched.



(a) Starvation eddy scheduling problem. While not-reconverging as soon as possible may benefit latency tolerance [107], it is a major cause of reduced SIMD efficiency in dynamic warp formation [59]. In the execution flow, shaded = utilized SIMD processing element, white = idle SIMD processing element.



(b) Extra memory accesses introduced by random thread grouping.

Figure 3.3: Dynamic warp formation pathologies.

tion to execute. We classify a benchmark as DIVG if it has SIMD efficiency below 76% and COHE otherwise (NNC contains 16 threads per block and no branch divergence). This classification simplifies our performance analysis throughout this chapter. TBC and DWF should aim to speedup the DIVG applications while preserving the performance of the COHE applications.

3.2 Dynamic Warp Formation Pathologies

Dynamic warp formation (DWF) [56] regroups threads executing the same instruction into new warps to improve SIMD efficiency. It promises to capture a significant fraction of the benefits of MIMD hardware on multithreaded SIMD hardware employing large multi-banked caches. However, the benefits of DWF can be affected by the warp scheduling policy [56, 59] and memory systems that limit bandwidth to first level memory structures. Figure 3.3(a) shows an example that helps to illustrate a form of pathological scheduling behaviour that can occur in DWF. This figure compares various SIMT control flow handling mechanisms for a branch hammock[2] diverging at block `A`. Each basic block contains the active mask for two warps where a “0” means the corresponding lane is masked off. The per-warp stack-based reconvergence mechanism (PDOM) executes block `B` and `C` with decreased SIMD efficiency, but reconverges at block `D`. The bottom right of Figure 3.3(a) illustrates a case where the threads at block `C` fall behind those at `B`. While this scheduling can increase latency tolerance [107], it can also lead to a reduction in performance [59] since ideally the warps at block `B` and `C` could form fewer warps at block `D`. We call this fall-behind behaviour a *starvation eddy*.

This behavior originates from a greedy scheduling policy, majority, that has been shown to work best with DWF. It incurs poor performance when a small number of threads “falls behind” the “majority” of threads [59]. The starvation eddy phenomenon reduces opportunities for such threads to regroup with the “majority” leading over time to lower SIMD efficiency. We observed this issue lowering the SIMD efficiency of many CUDA applications run with DWF as shown in Figure 3.1(a) (applications and configuration described in Section 3.5).

Furthermore, CUDA applications tend to be written assuming threads in a warp will execute together and should therefore access nearby memory locations. DWF tries to optimize control flow behaviour at the potential expense of increasing memory accesses (demonstrated in Figure 3.3(b)). Across the workloads we study, this leads to 2.7× extra stalls at the memory pipeline (Figure 3.1(b)).

Figure 3.4 compares DWF with majority scheduling against the baseline per-warp SIMT stack (PDOM) on the DIVG applications. While DWF improves performance significantly on BFS2, FCDT, and MUMpp, other applications suffer a slowdown[3]. One application, NVRT executes incorrectly with DWF, because it

[2]Note that the mechanisms studied in this work support CUDA and OpenCL programs with arbitrary control flow within a kernel.

[3]In our previous evaluation [56, 59], each SIMT core had large multi-bank L1 caches to buffer the memory system impact of DWF, whereas each SIMT core in this work only has a much smaller, single banked L1 cache, which may be desirable in practice to reduce the complexity and area of the memory system. The applications evaluated in our prior study [56, 59] also lacked the memory coalescing optimizations found in most CUDA applications (including those evaluated here) masking the impact of thread regrouping on the memory system.

uses a single manager thread in each static warp to continually acquire tasks from a global queue (atomically acquire a range of task IDs) for other worker threads in the warp. The per-warp SIMT stack enforces an implicit synchronization as it forces the worker threads in the warp to wait while this manager thread is acquiring tasks. DWF executes NVRT incorrectly because it does not enforce this behaviour. With DWF the worker threads incorrectly execute ahead with obsolete task IDs when the manager thread is acquiring new tasks. In general, we observed three problems when running CUDA applications on DWF enabled execution model: (1) Applications relying on implicit synchronization in a static warp (e.g. NVRT) execute incorrectly; (2) Starvation eddies may reduce SIMD efficiency; (3) Thread regrouping in DWF increases non-coalesced memory accesses and shared memory bank conflicts.

3.2.1 Warp Barrier

In this section we propose an extension to DWF, called a *warp barrier*. This mechanism keeps threads in their original static warps until they encounter a divergent branch. After a top-level divergence, threads can freely regroup between diverged warps but a "warp barrier" is created per static warp at the immediate post-dominator of the top-level divergent branch. A *top-level divergence* is a divergent branch that is not control dependent upon an earlier divergent branch. A dynamic warp may contain threads with different warp barriers. When a dynamic warp reaches a warp barrier those threads associated with the barrier are restored to their original static warp and wait until all threads from this static warp have arrived at the barrier. The remaining threads in the dynamic warp continue execution using the original DWF mechanisms [56]. The warp barrier mechanism confines starvation eddies between top-level divergence and reconvergence points while preserving the static warp arrangements reduces memory divergence [107] and shared memory bank conflicts. Warp barriers are distinct from `__syncthreads()` in CUDA – they are created dynamically only at divergent branches and there is one per static warp rather than one per thread block. Note that warp barrier only partially addresses the incompatibility of DWF with applications that rely on synchronous behavior in a static warp: DWF with warp barrier is compatible with

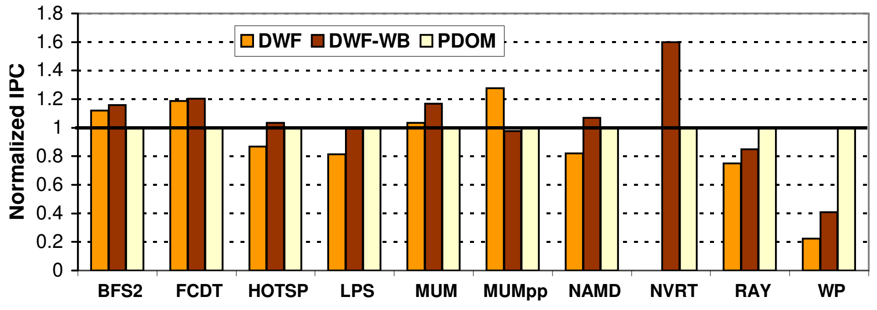


Figure 3.4: DWF with and without warp barrier compared against baseline per-warp SIMT stack reconvergence (PDOM).

applications that rely on warp synchronous behavior only in the coherent code region without any branch divergence (e.g., NVRT), and remains incompatible with applications that expect warp synchronous behavior in the presence of branch divergence (e.g., parallel reduction).

Figure 3.4 compares the performance of DWF with this warp barrier mechanism (DWF-WB) against the original DWF and the baseline per-warp SIMT stack (PDOM). With the warp barrier, NVRT executes properly and achieves a 60% speedup over PDOM. Three other applications that suffer slowdowns with the original DWF (HOTSP, LPS, NAMD) now achieve speedup. However, MUMpp loses performance with the warp barrier and shows a slight slowdown versus PDOM. In addition, RAY and WP continue to suffer from starvation eddies while using warp barriers. A deeper investigation suggests these applications require additional barriers between the top-level divergence and reconvergence of a static warp. Such barriers are a natural property of SIMT stack-based reconvergence and this led us to propose *thread block compaction*.

3.3 Thread Block Compaction

In CUDA (OpenCL) threads (work items) are issued to the SIMT cores in a unit of work called a “thread block” (work group). Warps within a thread block can communicate through shared memory and quickly synchronize via barriers. Thread block compaction extends this sharing to exploit control flow locality among threads within a thread block. Warps *within a thread block* share a block-wide SIMT stack for divergence handling instead of having separate *per-warp* stacks. At a divergent branch, the warps synchronize and their threads are “compacted” into new warps according to the branch outcome of each thread. The compacted warps then execute until the next branch or reconvergence point, where they synchronize again for further compaction. Compaction of all the divergent threads after they have reached the reconvergence point will restore their *original* warp grouping before the divergent branch was encountered.

As threads with a different program counter (PC) value cannot be merged in the same warp, DWF is sensitive to scheduling. When branch divergence occurs, a sufficient number of threads need to be present at the divergent branch to be merged into full warps. Ideally, threads should be encouraged to be clustered at a local region in the kernel program, but variable memory access latency and complex control flow make this hard to achieve. Moreover, even if this scheduling could be achieved, it will cluster memory accesses, discouraging overlap between memory access and computation and may increase memory latency via increased contention in the memory system.

Thread block compaction simplifies this scheduling problem with block-wide synchronization at divergent branches. This ensures that the maximum number of threads wait at a branch or reconvergence point, while other threads can be scheduled focusing on improving pipeline resource utilization. With the use of a SIMT stack for reconvergence, we can keep track of the warps that will eventually arrive at the reconvergence point and eliminate the starvation eddy problem described in Section 3.2. The synchronization overhead at branches can be covered by switching the execution to a different thread block running on the same SIMT core. In Section 3.6.2, we explore the performance impact of several thread block prioritization policies.

While sharing a single SIMT stack among all warps in a block can in theory reduce performance when threads in *different* warps follow diverging control flow paths, we find this type of code to be rare. It tends to occur where CUDA programmers work around the limitation of one concurrent kernel launch at a time on pre-Fermi NVIDIA GPUs by having different warps in a block execute different code following a top-level “`if`” or “`switch`” statement. There are also emerging GPU programming models that use warp specialization, using each warp to perform a

Example 3 Code that exhibits branch divergence.

```
t = threadIdx.x; // block A
flag = (t==1)||(t==6)||(t==7);
if( flag )
    result = Y;  // block B
else
    result = Z;  // block C
return result;   // block D
```

different subtask in a thread block, to manage memory transfer [11] or to support domain specific languages [12]. If better support were desired for such code, we could employ prior proposals for enabling stack-based reconvergence mechanisms to overlap execution of portions of the same warp that follow different control flow paths [107]. We leave evaluation of such extensions to future work and instead focus on what we observe to be the common case for existing applications.

3.3.1 High-Level Operation

Figure 3.5 illustrates the high-level operation of thread block compaction. The code in Example 3 translates into the control flow graph in Figure 3.5. In this example, each warp contains four threads and each thread block contains eight threads. The numbers in each basic block of the control flow graph (left portion of figure) denote the threads that execute that block. All threads execute block A and D, while only threads (1,6,7) execute block C and only threads (2,3,4,5,8) execute block B.

Two warps composed of threads 1-4, and 5-8 begin executing at the start of block A. Since there is no divergence, there is only a single entry in the block-wide SIMT stack (❶ in Figure 3.5). Two warps (❷) are created from the active threads (❶). The two warps are scheduled on the pipeline independently until they reach the end of block A (❸), where they "synchronize" at the potentially divergent branch. This synchronization allows the hardware to determine if any of the warps has diverged at the branch. As an optimization, the programmer/compiler can statically annotate non-divergent branches (such as `bra.uni` in the PTX-ISA [123]) in the kernel, allowing the warps to skip synchronization at these branches. After both warps have executed the branch, two new entries (❹) will have been pushed

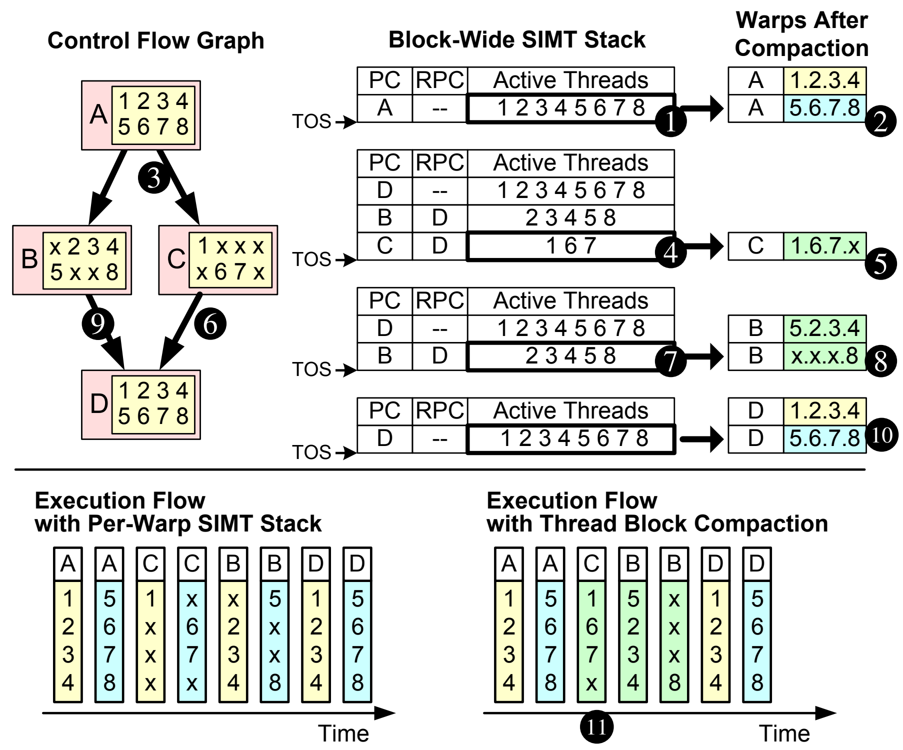


Figure 3.5: High-level operation of thread block compaction.

onto the stack, each containing the active threads that will execute the “taken” or “not taken” side of the branch (block C or B, respectively). The active threads on the top entry are compacted into a single warp that executes basic block C (❺). As this warp reaches the reconvergence point D (❻), its entry is popped from the SIMT stack (❼), and the active threads that execute basic block B are compacted into two warps (❽). After these two warps have reached the reconvergence point D (❾), their corresponding entry is popped from the stack, and threads resume execution in two full warps with their original arrangements before the divergent branch (❿).

The lower part of Figure 3.5 compares the execution flow of thread block compaction with the baseline per-warp reconvergence mechanism. In this example, thread block compaction compacts threads (1,6,7) into a single warp at basic block C (⓫). This reduces the overall execution time by 12.5% over the baseline in this

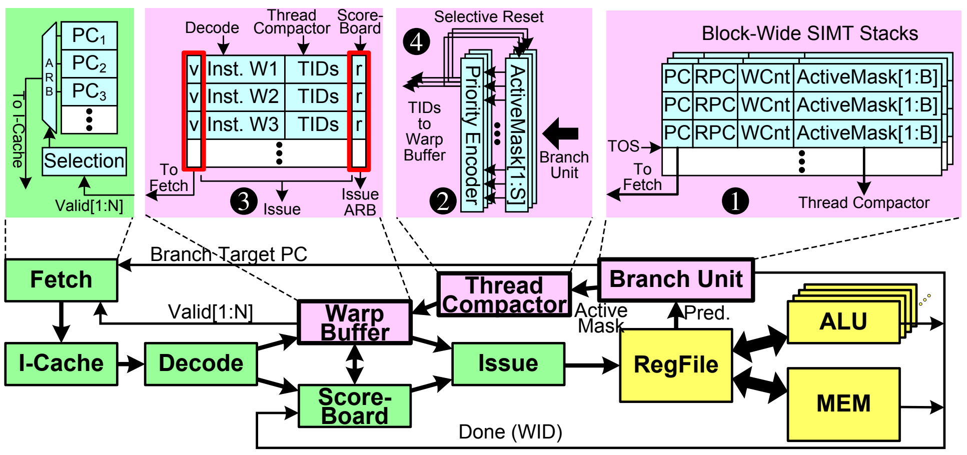


Figure 3.6: Modifications to the SIMT core microarchitecture to implement thread block compaction. N = #warps, B = maximum #threads in a block. S = B ÷ W where W = #threads in a warp.

example.

The pushing and popping of the entries on and off the block-wide SIMT stack, as branches and reconvergence points are encountered, uses the same reconvergence points as the per-warp SIMT stack in the baseline SIMT core.

3.3.2 Implementation

Figure 3.6 illustrates the modifications to the SIMT core microarchitecture to implement thread block compaction. The modifications consist of three major parts: a modified branch unit (❶), a new hardware unit called the thread compactor (❷), and a modified instruction buffer called the warp buffer (❸). The branch unit (❶) has a block-wide SIMT stack for each block. Each entry in the stack consists of the starting PC (PC) of the basic block that corresponds to the entry, the reconvergence PC (RPC) that indicates when this entry will be popped from the stack, a warp counter (WCnt) that stores the number of compacted warps this entry contains, and a block-wide active mask that records which thread is executing the current basic block. The thread compactor (❷) consists of a set of priority encoders that compact the block-wide active mask into compacted warps with thread IDs. The warp buffer (❸) is an instruction buffer that augments each entry with the thread

IDs associated with compacted warps.

To retain compatibility with applications that rely on static warp synchronous behaviour (e.g., `reduction` in the CUDA SDK), the thread compactor can optionally be disabled to allow warps to retain their static/compile-time arrangement (e.g., when launching a kernel via an extension to the programming API).

In comparison to DWF [56], thread block compaction accomplishes the lookup-and-merge operation of the "warp LUT" and the "warp pool" [56] with simpler hardware. In DWF, an incoming warp is broken down every cycle and the warp LUT has to locate an entry in the warp pool that can merge with the individual threads. In thread block compaction, warps are only broken down at potentially divergent branches and partial warps are accumulated into block-wide active masks. The compaction only occurs once after the active masks have been fully populated and the compacted warps are stored at the warp buffer until the next branch or reconvergence point.

3.3.3 Example Operation

Figure 3.7 presents an example of how the hardware in Figure 3.6 implements thread block compaction. The active mask in the block-wide SIMT stack is divided into groups, each corresponding to one vector lane in all static warps in the thread block[4]. Threads are constrained to stay in their vector lane during warp compaction to avoid the need to migrate register state and to simplify the thread compactor. Each thread can locate its corresponding bit inside its vector lane group via its associated static warp. For example, thread 5 in the first vector lane of static warp W2 corresponds to the second bit of the first group (❹).

The active mask in the block-wide SIMT stack is incrementally populated as warps arrive at a branch. Since warps are allowed to execute independently as long as they do not encounter branches or reconvergence points, it is possible for warps to arrive at the branch at the end of block `A` (❶) in an arbitrary order. In the example, warp (W2) arrives at the branch first and creates two target entries on the stack (❷). Each thread in the warp updates one of these entries based upon its branch outcome. For instance, since thread 5 goes to block `C` (❸), it updates the

[4]Example shows 3 warps, but each thread block can have up to 32 warps/1024 threads [122].

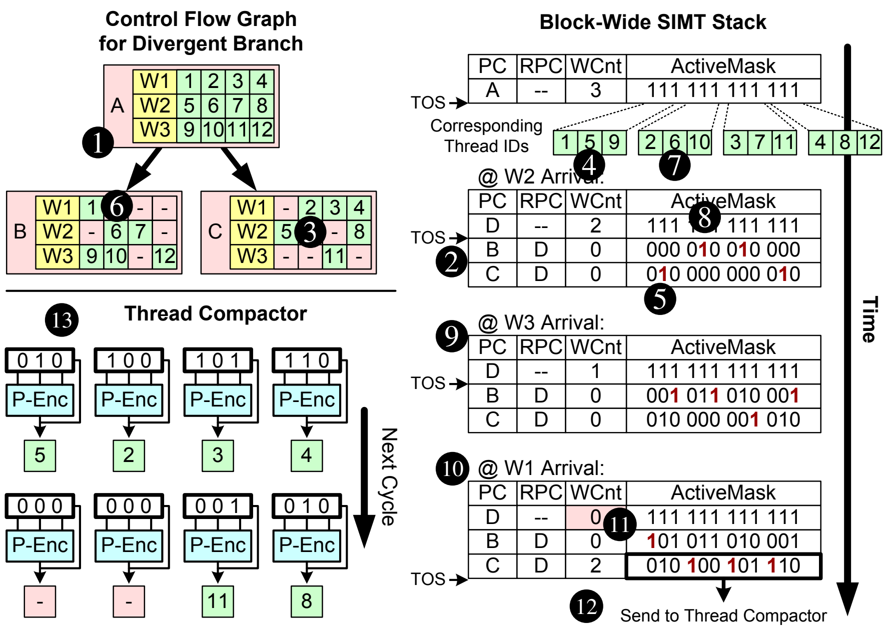


Figure 3.7: Thread block compaction example operation showing how the active mask in the block-wide SIMT stack can be incrementally populated for a divergent branch.

first new entry on the stack (❺). On the other hand, since thread 6 goes to block `B` (❻), it updates the second new entry on the stack (❽). In subsequent cycles, the other warps (W3 and W1) arrive at the branch and update the active masks of the two new stack entries (❾ and ❿, bits updated at each warp's arrival are shown in bold).

As the warp arrives at the potentially divergent branch, WCnt of the original TOS entry is decremented to keep track of pending warps. When WCnt reaches zero (⓫), the TOS pointer increments to point at the new top target entry. The active mask of this new TOS entry is sent to the thread compactor for warp generation (⓬). WCnt of this entry is also updated to record the number of compacted warps to be generated (calculated by counting the maximum number of set bits in all lanes' active mask). A single block-wide active mask can generate at most as many compacted warps as the static warps in the executing thread block.

For branches that NVIDIA's CUDA compiler marks as potentially divergent, we always create two entries for taken and not-taken outcomes when the first warp reaches a branch even if that warp is not divergent. If all threads in a block branch to the same target one of the entries will have all bits set to zero and will be immediately popped when it becomes the top of stack.

Figure 3.7 also shows the operation of the thread compactor (⓭). The block-wide active mask sent to the thread compactor is stored in multiple buffers, each responsible for threads in a single home vector lane [56]. Each cycle, each priority encoder selects at most one thread from its corresponding inputs and sends the ID of this thread to the warp buffer (❹ in Figure 3.6). The bits corresponding to the selected threads will be reset, allowing the encoders to select from the remaining threads in subsequent cycles.

When the compacted warps encounter another divergent branch, the process described above repeats, pushing new entries onto the block-wide SIMT stack. Eventually, as each of these compacted warps reaches the reconvergence point (block `D` in this example), WCnt of block `C` entry decrements. When this WCnt reaches zero, the entry is popped (TOS pointer decrements). The active mask of the new TOS entry is sent to the thread compactor to generate warps that execute block `B`, and WCnt of this entry is updated accordingly. After these compacted warps have reached the reconvergence point as well, the block-wide stack is popped again, shifting execution to the top-level entry. Similarly, the full activemask of this entry is sent to the thread compactor, with its WCnt updated to the corresponding warp count in preparation for the next divergent branch.

3.4 Likely-Convergence Points (LCP)

The post-dominator (PDOM) stack-based reconvergence mechanism [56, 59] uses reconvergence points identified using a unified algorithm rather than by translating control flow idioms in the source code into instructions [5, 30, 100]. The immediate post-dominator of a divergent branch selected as the reconvergence point is the earliest point in a program where the divergent threads are *guaranteed* to reconverge. In certain situations, threads can reconverge at an *earlier point*, and if hardware can exploit this, it would improve SIMD efficiency. We believe this observation motivates the inclusion of the `break` instruction in recent NVIDIA GPUs [30].

Example 4 Example for likely-convergence.

```
while (i < K) {
  X = data[i];       // block A
  if( X == 0 )
    result[i] = Y;   // block B
  else if ( X == 1 ) // block C
    break;           // block D
  i++;               // block E
}
return result[i];    // block F
```

The code in Example 4 exhibits this earlier reconvergence. It results in the control flow graph in Figure 3.8 where edges are marked with the probability with which individual scalar threads follows that path. Block `F` is the immediate post-dominator of `A` and `C` since `F` is the first location where *all* paths starting at `A` (or `C`) coincide. In the baseline mechanism, when a warp diverges at `A`, the reconvergence point is set to `F`. However, the path from `C` to `D` is rarely followed and hence in *most* cases threads can reconverge at `E`.

We extend the PDOM SIMT stack with *likely-convergence points* to capture the potential performance benefits from such "probabilistic" reconvergence. We add two new fields to each stack entry: one for the PC of the likely-convergence point (LPC) and the other (LPos), a pointer that records the stack position of a special likely-convergence entry created when a branch has a likely-convergence point that differs from the immediate post-dominator. The likely-convergence point of each branch can be identified with either control flow analysis or profile information (potentially collected at runtime). Figure 3.8 shows a warp that diverges at `A` (❶). When the divergence is detected three entries are pushed onto the stack. The first entry (❷) is created for the likely-convergence point `E`[5]. Two other entries for the taken and fall through of the branch are created as in the baseline mechanism. The warp diverges again at `C` (❸), and two[6] new entries are created (❹). Execution continues with the top entry until it reaches `E`, and the likely-convergence point is detected since PC == LPC. When this occurs, the top entry is popped and merged

[5]In our experimental evaluation, we restricted likely-convergence points to the closest enclosing backwards taken branch to capture the impact of "`break`" statements within loops [30].

[6]Since likely-convergence and immediate post-dominator are the same.

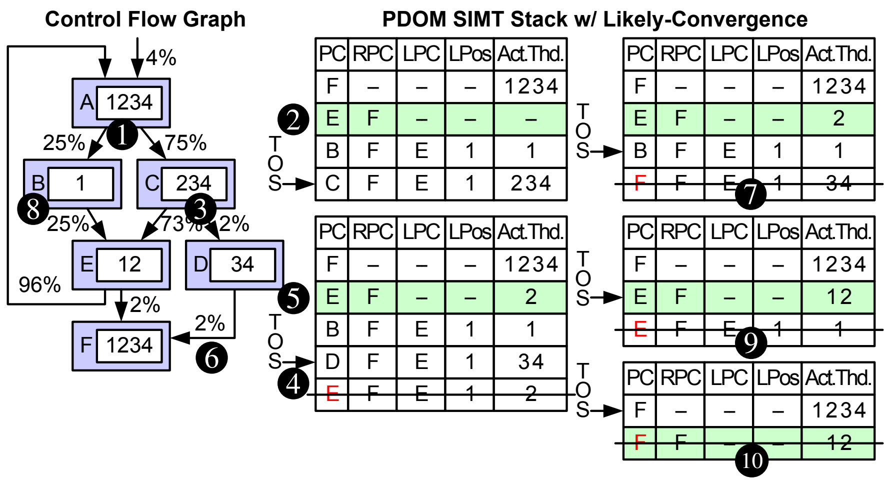


Figure 3.8: Likely-convergence points improve individual warp SIMD efficiency by reconverging *before* the immediate post-dominator.

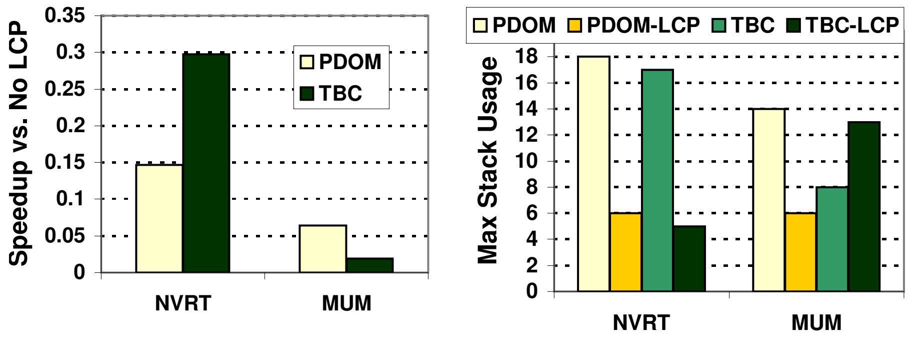


Figure 3.9: Performance and resource impact of likely-convergence points on both baseline per-warp SIMT stack (PDOM) and thread block compaction (TBC).

with the likely-convergence entry (❺) as the LPos field indicates. When thread 3 and 4 reach `F` (❻), since PC == RPC, the stack is popped (❼). Thread 1 then executes `B` (❽) and its entry is popped at `E` (❾), when PC == LPC. Finally, the likely-convergence entry executes until it reaches the immediate post-dominator, where it is popped (❿).

Both the baseline per-warp SIMT stack (PDOM) and the block-wide stack used in thread block compaction (TBC) can be extended with LCP. Figure 3.9 shows the performance impact of extending PDOM and TBC with likely-convergence points

(-LCP). Only data for MUM and NVRT are shown because we have only identified likely-convergence points that differ from the immediate post-dominators in these two applications. The impact of LCP for MUM is minimal: 2% speedup for TBC and 5% for PDOM. In contrast, it greatly benefits NVRT: 30% speedup for TBC, 14% for PDOM. Although LCP pushes an extra entry onto the stack for each divergent branch applicable, it can reduce the stack capacity requirement if multiple divergent entries are merged into fewer likely-convergence entries. This happens in most cases, except for TBC running MUM, where the unused likely-convergence entries increases the maximum stack usage.

3.5 Methodology

We model our proposed hardware changes using a modified version of GPGPU-Sim (version 2.1.1b) [10]. The version of GPGPU-Sim modeling our proposed hardware changes is available online [63]. We evaluate the performance of various hardware configurations on the CUDA benchmarks listed in Table 3.1. Most of these benchmarks are selected from Rodinia [28] and the benchmarks used by Bakhoda et al. [10]. We did not exclude any benchmarks due to poor performance on thread block compaction but excluded some Rodina benchmarks that do not run on our infrastructure due to their reliance on undocumented behaviour of barrier synchronization in CUDA. We also use some benchmarks from other sources:

Face Detection is a part of Visbench [104]. We applied the optimizations [103] recommended by Aqeel Mahesri to improve its SIMD efficiency.

MUMMER-GPU++ improved MUMMER-GPU [139] reducing data transfers with a novel data structure [65].

NAMD is a popular molecular dynamics simulator [127].

Ray Tracing (Persistent Threads) dynamically distributes workload in software to mitigate hardware inefficiencies [4]. We render the “Conference” scene.

Our modified GPGPU-Sim is configured to model a GPU similar to NVIDIA’s Quadro FX5800, with the addition of L1 data caches and a L2 unified cache similar to NVIDIA’s Fermi GPU architecture [121]. We configure the L2 unified cache to

Table 3.1: Benchmarks in thread block compaction evaluation.

Name	Abbr.	BlockDim	#Instr.	Blocks/core
Divergent Set				
BFS Graph Traversal [28]	BFS2	(512x1x1), (256x1x1)	28M	2
Face Detection [104]	FCDT	2x(32x6x1)	1.7B	5,4
HotSpot [28]	HSP	(16x16x1)	157M	2
3D Laplace Solver [10]	LPS	(32x4x1)	81M	6
MUMMER-GPU [10]	MUM	(256x1x1)	69M	4
MUMMER-GPU++ [65]	MUMpp	(192x1x1)	140M	3
NAMD [127]	NAMD	2x(64x1x1)	3.8B	7
Ray Tracing (Persistent Threads) [4]	NVRT	(32x6x1)	700M	3
Coherent Set				
AES Cryptography [10]	AES	(256x1x1)	30M	2
Back Propagation [28]	BACKP	2x(16x16x1)	193M	4,4
Coulumb Potential [10]	CP	(16x8x1)	126M	8
gpuDG [10]	DG	(84x1x1), (112x1x1), (256x1x1)	569M	4,5,6
Heart Wall Detection [28]	HRTWL	(512x1x1)	8.9B	1
LIBOR [10]	LIB	2x(64x1x1)	162M	8,8
Leukocyte [28]	LKYT	(175x1x1)	6.8B	5,5,1
Merge Sort [28]	MGST	(96x1x1) 2x(32x1x1), 2x(128x1x1), 2x(256x1x1), (208x1x1)	2.3B	1,3,8,3,4,2,4
NN_cuda [28]	NNC	(16x1x1)	6M	5,8,8,8
Ray Tracing [10]	RAY	(16x8x1)	65M	3
Stream Cluster [28]	STMCL	(512x1x1)	941M	2
StoreGPU [10]	STO	(128x1x1)	131M	1
Weather Prediction [10]	WP	(8x8x1)	216M	4

be significantly larger than that on Fermi (1MB vs. 128 kB per memory channel) to make dynamic warp formation more competitive against thread block compaction. In Section 3.6.4, we explore the sensitivity of both techniques to changes in memory system by reducing the L2 cache to 128kB per memory channel. Table 3.2 shows the major configuration parameters.

3.6 Experimental Results

Figure 3.10 shows the performance of TBC and DWF relative to that of PDOM. TBC uses likely-convergence points whereas PDOM does not and DWF and DWF-

Table 3.2: GPGPU-Sim configuration for thread block compaction evaluation

# Streaming Multiprocessors	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 64B line, 8-way assoc.
L2 Unified Cache	1MB/Memory Channel, 64B line, 64-way assoc.
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [56]
Warp Scheduling Policy	Loose Round Robin
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

WB use majority scheduling [56]. For the divergent (DIVG) benchmark set, TBC has an overall 22% speedup over PDOM. Much of its performance benefits are attributed to speedups on the applications that have very low baseline SIMD efficiency (BFS2, FCDT, MUM, MUMpp, NVRT). While DWF can achieve speedups on these benchmarks as well (except NVRT, which fails to execute), it also exhibits slowdowns for the other benchmarks that have higher baseline SIMD efficiency (HOTSP, LPS, NAMD), lowering the overall speedup to 4%. DWF with warp barrier (DWF-WB) recovers from most of this slowdown and executes NVRT properly, but loses much of the speedup on MUMpp. Overall, TBC is 17% faster than the original DWF and 6% faster than DWF-WB.

Applications in the coherent (COHE) benchmark set are not significantly effected by branch divergence, hence we do not anticipate significant benefits from DWF or TBC. DWF suffers significant slowdowns on some applications in this benchmark set (HRTWL, LKYT, RAY, STO and WP), due to starvation eddy and extra memory stalls from thread regrouping (see Section 3.6.1). DWF-WB recovers much of this slowdown, however, RAY and WP still suffer from the starvation eddy problem.

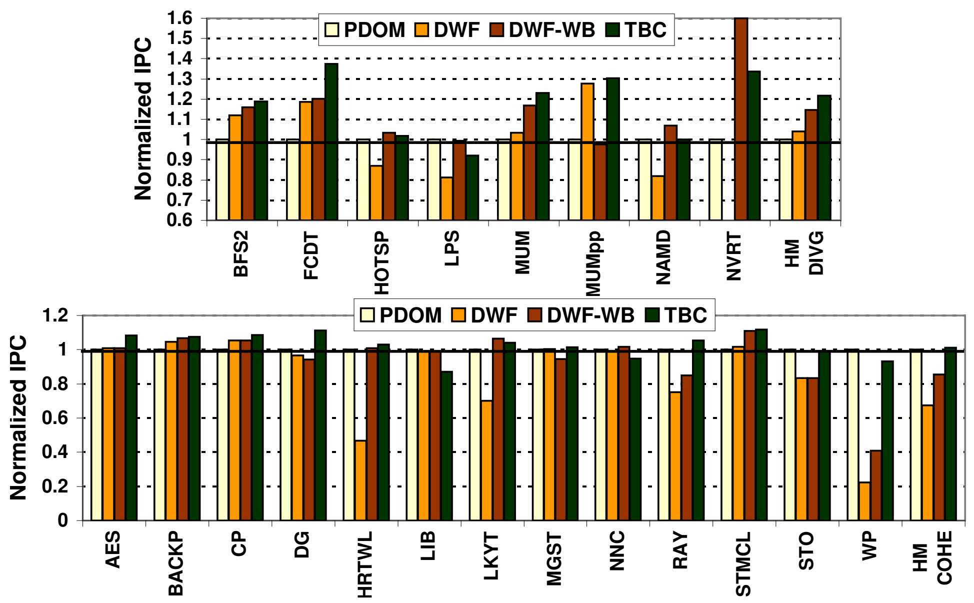
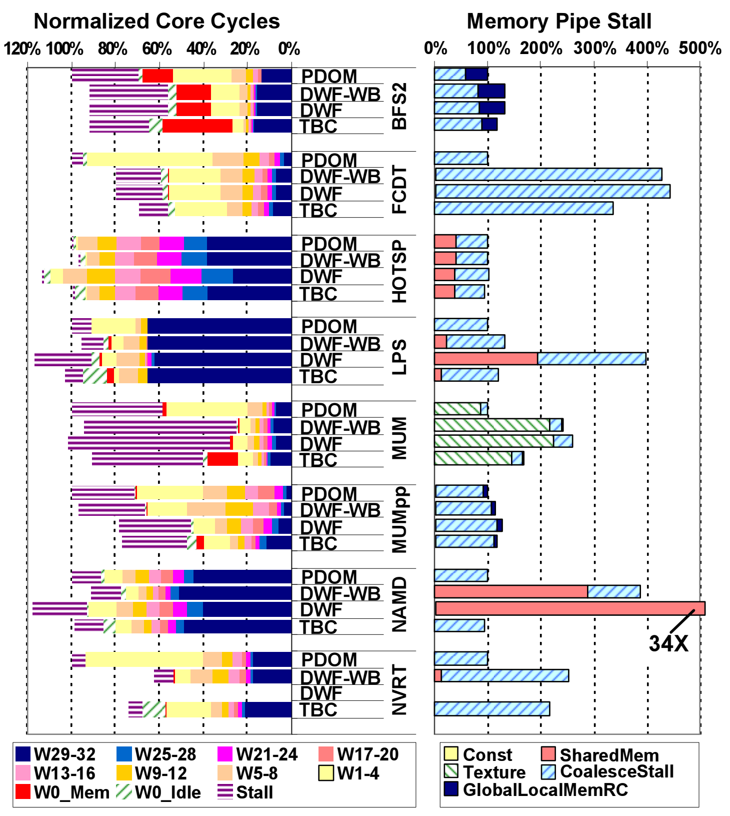


Figure 3.10: Performance of thread block compaction (TBC) and dynamic warp formation (DWF) relative to baseline per-warp post-dominator SIMT stack (PDOM) for the DIVG and COHE benchmark sets.

Across all benchmarks, TBC obtains an overall 10% speedup over PDOM. The performance benefits of DWF-WB are mostly offset by slowdowns in other applications, making it perform evenly with PDOM.

3.6.1 In-Depth Analysis

Figure 3.11(a) and 3.12(a) show the breakdown of the SIMT core cycle for both DIVG and COHE benchmark sets. At each cycle, the SIMT core can either issue a warp containing a number of active threads (W*n*-*m* means between *n* and *m* threads are enabled when a warp issues), be stalled by the downstream pipeline stages (Stall), or not issue any warp because none are ready in the I-Buffer/Warp-Buffer (W0_Mem if the warps are held back by pending memory accesses, and W0_Idle otherwise). This data shows that in some applications (HOTSP, NAMD, HRTWL, MGST, RAY and WP), starvation eddies cause DWF to introduce extra divergence, turning some of the warps with more active threads (W29-32) into warps with fewer active threads. In other applications (e.g. LPS, MUM and NAMD), the extra

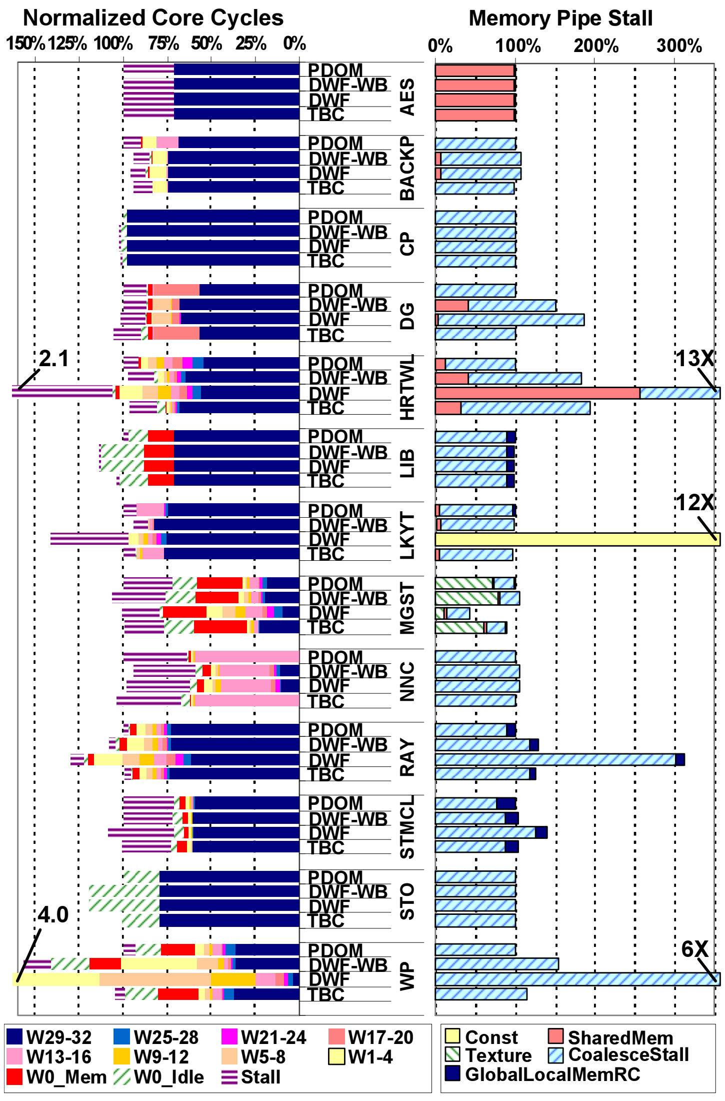


(a) Core Cycle Breakdown (b) Memory Pipe Stall Breakdown

Figure 3.11: Detail performance data of TBC and DWF relative to baseline (PDOM) for the DIVG benchmark set.

stalls from DWF undermine the benefit of merging divergent threads into warps. DWF-WB reduces stalls and divergence versus DWF in these applications, but the starvation eddy problem persists with DWF-WB for RAY and WP.

TBC can usually improve SIMD efficiency as well as DWF-WB, and it does not introduce significant extra stalls or divergence. However, the synchronization overhead at branches can introduce extra W0_idle and W0_Mem cycles. This is the main reason why DWF-WB performs better than TBC for HOTSP, LPS, NAMD and NVRT. For NAMD and NVRT, TBC achieves a lower SIMD efficiency than DWF-WB because DWF can form warps from any threads within a SIMT core, while TBC can only do so from threads within a thread block. BFS2, MUM, and



(a) Core Cycle Breakdown (b) Memory Pipe Stall Breakdown

Figure 3.12: Detail performance data of TBC and DWF relative to baseline (PDOM) for the COHE benchmark set.

MGST transition from compute-bound to memory-bound with TBC (indicated by the extra W0_Mem for them), limiting its benefit. TBC is more prone to being memory-bound than DWF because it requires warps to synchronize at every divergent branch for compaction. These extra block-wide synchronizations reduce the number of warps available to cover the long memory access latency in GPUs. Rhu and Erez [130] tackle this issue by extending TBC with a compaction-adequacy predictor (CAPRI) to avoid unnecessary synchronizations for branches that do not benefit from compaction (see Chapter 7.1).

In Figure 3.11(a), TBC has the total number of core cycles as DWF for BFS2 while Figure 3.10 shows that TBC has a higher IPC than DWF. This is because IPC in Figure 3.10 is calculated using the total execution time, whereas core cycles only account for the time when a SIMT core is running at least one thread. BFS2 features a series of relatively short kernel launches, and the load imbalance between different SIMT cores causes the total execution time of DWF to be longer than TBC.

Figure 3.11(b) and Figure 3.12(b) show the breakdown of memory pipeline stalls modeled in our simulator normalized to the baseline and highlights that DWF introduces extra memory stalls. TBC introduces far fewer extra memory stalls. These extra stalls do not out-weight the benefits of TBC to control flow efficiency on several applications (e.g. FCDT, MUMpp, NVRT and HRTWL). Section 3.6.3 shows how the L1 data cache in each SIMT core absorbs the extra memory accesses generated by TBC, leaving the memory subsystem undisturbed.

3.6.2 Thread Block Prioritization

Figure 3.13 compares the performance of TBC among different thread block prioritization policies. A thread block prioritization sets the scheduling priority among warps from different thread blocks, while warps within a thread block are always scheduled with loose round-robin policy [56]:

Age-based (AGE) The warps from the oldest thread block (in the order that thread blocks are dispatched to a SIMT core) have the highest priority. This tries to stagger different thread blocks, encouraging them to overlap each other's synchronization overhead at branches.

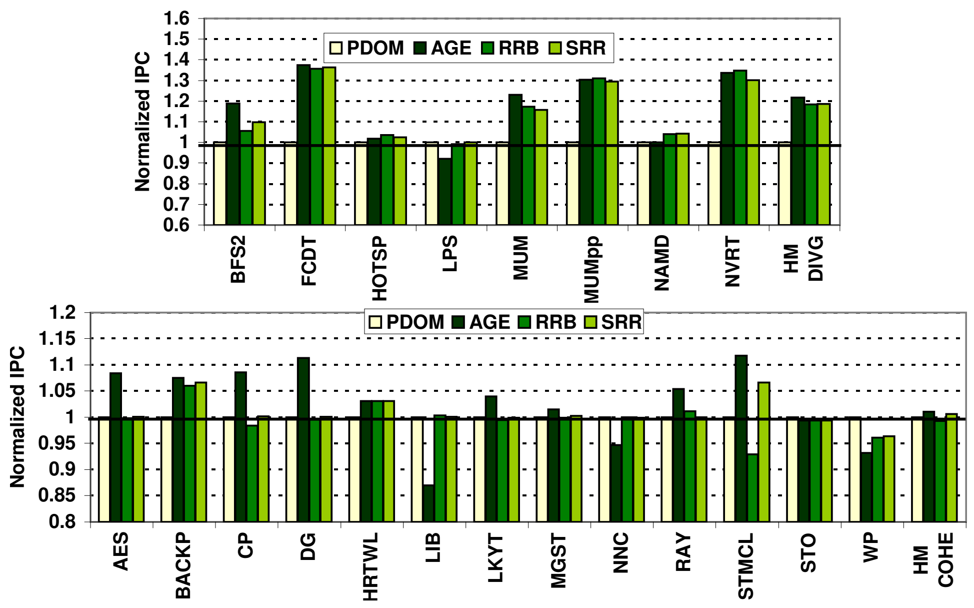


Figure 3.13: Performance of TBC with various thread block prioritization policies relative to PDOM for the DIVG and COHE benchmark sets.

Round-robin (RRB) Thread block priority rotates every cycle, encouraging warps from different thread blocks to interleave execution.

Sticky round-robin (SRR) Warps in a thread block that is currently issuing warps retain highest priority until none of the thread block's warps are ready for issue. Then, the next thread block gets highest priority.

Overall, AGE (default policy for TBC in this work) achieves the highest performance, but it can leave the SIMT cores with a lone-running thread block near the end of a kernel launch. The lack of interleaving reduces overall performance for LPS, NAMD and LIB. RRB encourages even progress among thread blocks, but increases memory system contention in STMCL. SRR provides the most robust performance among the policies, with its performance consistently staying above 95% of the PDOM performance for all evaluated benchmarks.

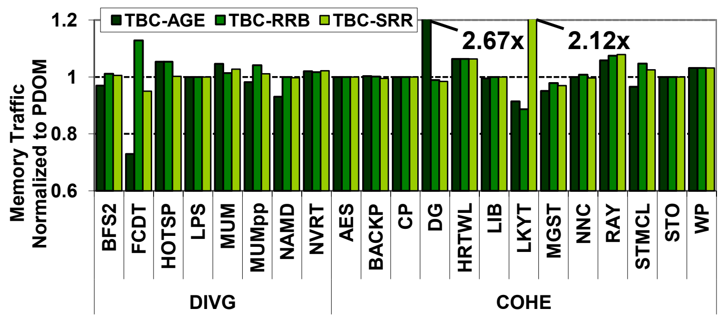


Figure 3.14: Average memory traffic of a SIMT core for TBC relative to baseline (PDOM).

3.6.3 Impact on Memory Subsystem

Figure 3.14 shows the average memory traffic (in bytes for both reads and writes) between a SIMT core and the memory subsystem outside the core with TBC normalized to PDOM. Traffic does not increase significantly with TBC (within 7% of PDOM), indicating that the L1 data cache has absorbed the extra memory pressure due to TBC. If memory accesses from a static warp would have coalesced into a 128-Byte chunk, and this static warp is compacted into multiple dynamic warps by TBC upon a branch divergence, then they will access the same blocks in the L1 data cache. The L1 data cache can generally capture this data locality, and no extra memory accesses will be sent out from the SIMT core.

Memory traffic of DG increases by 2.67× due to increased texture cache misses when using AGE based prioritization (which tends to reduce interleaving from different thread blocks). TBC combined with the RRB policy reduces traffic of DG by increasing the texture cache hit rate. Similarly, memory traffic of LKYT increases by 2.12× with SRR due to increased texture cache misses. In both cases, the extra memory traffics hit at the L2 cache and have little impact on the overall performance.

3.6.4 Sensitivity to Memory Subsystem

Figure 3.15 shows the speedup of TBC and DWF over PDOM, but with smaller L2 caches (128 kB, 8-way per memory channel). The relative performance between the different mechanisms remains unchanged for the COHE benchmarks. Two of

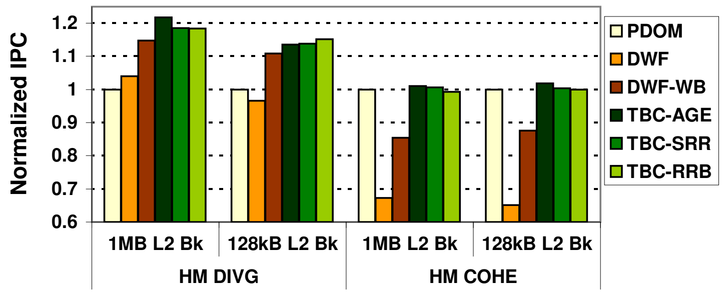


Figure 3.15: Average speedup of TBC and DWF over baseline (PDOM) with a smaller (128 kB/Memory Channel) L2 cache. The speedups are normalized to PDOM with the same L2 cache capacity.

the DIVG benchmarks (MUM and MUMpp) become more memory-bound with a less powerful memory system, lowering the speedup of TBC (15% with RRB and 13.5% with AGE for the DIVG benchmarks). Smaller L2 caches reduce DWF's performance on MUMpp from 28% speedup to 4% slowdown. In comparison, the speedups with DWF-WB and TBC remain robust to the change in the memory system.

3.7 Implementation Complexity

Most of the implementation complexity for thread block compaction is the extra storage for thread IDs in the warp buffer, and the area overhead in relaying these IDs down the pipeline for register accesses. The register file in each SIMT core also needs to be banked per lane as in dynamic warp formation [56] to support simultaneous accesses from different vector lanes to different parts of the register file.

The scheduler complexity that DWF imposes is mostly eliminated via the block-wide SIMT stacks in thread block compaction. The bookkeeping for thread grouping is done via active masks in the stack entries. The active masks can be stored in a common memory array. Each entry in this memory array has T bits (T = max #threads supported on a SIMT core). The T bits in each entry are divided among the multiple thread blocks running on a SIMT core. In this way, the total #bits for active mask payload does not increase over the baseline per-warp stacks.

Table 3.3: Maximum stack usage for TBC-LCP

DIVG		COHE			
	#Entries		#Entries		#Entries
BFS2	3	AES	1	NNC	3
FCDT	5	BACKP	2	RAY	9
HOTSP	2	CP	1	STMCL	3
LPS	5	DG	2	STO	1
MUM	13	HRTWL	5	WP	8
MUMpp	14	LIB	1		
NAMD	5	LKYT	3		
NVRT	5	MGST	38		

One potential challenge to TBC is that the block-wide stack can in the theoretical worst case be deeper than with per-warp stacks. Table 3.3 shows the stack usage for TBC with likely-convergence points across all the applications we study. Most applications use fewer than 16 entries throughout their runtime. For exceptions such as MGST the bottom of the stack could potentially be spilled to memory. We synthesized the 32-bit priority encoders used in thread compactors (sufficient for the maximum thread block size of 1024 threads [122]) in 65 nm technology and found their aggregate area to be negligible ($<< 1mm^2$).

3.8 Summary

In this chapter, we proposed thread block compaction, a novel mechanism that uses a block-wide SIMT stack shared by all threads in a thread block to exploit their control flow locality. Warps run freely until they encounter a divergent branch, where the warps synchronize, and their threads are compacted into new warps. At the reconvergence point the compacted warps synchronize again to resume in their original arrangements before the divergence. We found that our proposal addresses some key challenges of dynamic warp formation [56]. Our simulation evaluation quantifies that it achieves an overall 22% speedup over a per-warp SIMT stack baseline for a set of divergent applications, while introducing no performance penalty for a set of control flow coherent applications.

Chapter 4

Kilo TM: Hardware Transactional Memory for GPU Architectures

In this chapter, we explore how to support a transactional memory (TM) [73, 75] programming model on GPU architectures. This exploration was motivated by the challenge of managing the irregular, fine-grained communications between threads in GPU applications with ample irregular parallelism. We believe TM can ease this challenge, and perform a limit study in Section 4.1, showing that a set of GPU-TM applications can perform nearly as well as their fine-grained locking analogs with an idealized TM system. The rest of this chapter (and Chapter 6) focus on enabling TM on GPU efficiently. A version of this chapter has been published earlier [57].

Our proposed solution, Kilo TM, is the first hardware TM proposal for GPU architectures. The heavily multithreaded nature of GPU introduces a new set of challenges to TM system designs. Instead of running tens of concurrent transactions with relatively large footprint – the focus of much recent research on TM for multicore processors – Kilo TM aims to scale to tens of thousands of small concurrent transactions. This reflects the heavily multithreaded nature of GPU, with tens of thousands of threads working in collaboration, each performing a small task towards a common goal. These small transactions are tracked at word-level granularity, enabling finer resolution of conflict detection than cache blocks. Moreover, each per-core private cache in a GPU is shared by hundreds of GPU threads. This drastically reduces the benefit of leveraging a cache coherence protocol to detect conflicts, a technique employed on most HTMs designed for traditional CMPs with large CPU cores.

Kilo TM employs value-based conflict detection [37, 124] to eliminate the need for global metadata for conflict detection. Each transaction simply reads the existing data in global memory for validation – to determine if it has a conflict with another committed transaction. This validation leverages the highly parallel nature of the GPU memory subsystem, avoids any direct interaction between conflicting transactions, and detects conflicts at the finest granularity. However, a native implementation of value-based conflict detection requires transactions to commit serially. To boost commit parallelism, we have taken inspiration from existing TM systems [27, 149] and extended them with innovative solutions. In particular, we have introduced the *recency bloom filter*, a novel data structure that uses the notion of time and order to compress a large number of small item sets. We use this structure to compress the write-sets of all committing transactions in Section 4.2.5. Each transaction queries the recency bloom filter for an approximate set of conflicting transactions. Kilo TM uses this approximate information to schedule hundreds of non-conflicting transactions for validation and commit in parallel. Using the recency bloom filter to boost transaction commit parallelism is an integral part of Kilo TM.

This chapter also proposes a simple extension to the SIMT hardware to handle conflict flow divergence due to transaction aborts. This extension, described in Section 4.2.1, is independent of other design aspects of Kilo TM. Despite its simplicity, it is a necessary piece for supporting TM on GPUs.

Our evaluation with a set of GPU-TM applications shows that Kilo TM captures 59% of the performance of fine-grained locking. We find that Kilo TM outperforms fine-grained locking for low contention applications that require acquiring multiple locks to enter a critical section. On the other hand, we find that TM applications ported from CPU-optimized versions can perform poorly on GPUs regardless of the data synchronization mechanism used (fine-grained locking or TM). Optimizing these applications for GPUs would involve redesigning the algorithm and data structures to expose more thread-level-parallelism. Our estimation with CACTI [144] indicates that implementing Kilo TM on an NVIDIA Fermi GPU [121] would increase area by only 0.5%, a small overhead for the large increase in programmability.

The rest of this chapter is organized as follows: Section 4.1 motivates TM on GPUs, and outlines the challenges in adopting prior HTMs on GPUs. Section 4.2 describes Kilo TM, our TM design for a GPU that supports 1000s of concurrent transactions. Section 4.3 describes our methodology and benchmark applications. Section 4.4 presents our evaluation results. Section 4.5 summarizes this chapter.

4.1 Transactional Memory on GPU: Opportunities and Challenges

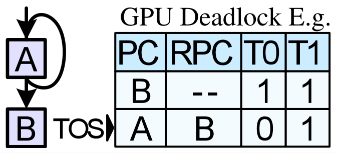
Atomic operations on current GPUs enable implementation of locks, allowing complex irregular algorithms [23]. Fine-grained locking enables higher concurrency in applications, but requires the application developer to consider all possible interactions between locks to ensure deadlock-free code – a challenging task [94]. With tens of thousands of threads running concurrently on a GPU, the number of possible interactions among these fine-grained locks can be overwhelming in practical applications. This problem is well known to the supercomputing community and has inspired special debugging tools to summarize thread behaviours for deadlock/data-race analysis [8].

In this chapter, we propose to increase support for irregular algorithms on GPUs by extending GPU architecture to support TM [75]. While originally proposed for CPUs, we find TM to be a natural extension to the existing GPU/CUDA programming model. From a programmer's perspective, a transaction is executed as an atomic block of code in isolation. A thread in a transaction is never blocked waiting to synchronize with another thread. This is important because a CUDA/OpenCL application can launch many more threads than the GPU hardware can concurrently execute. Like transactions, thread execution sequencing is abstracted away in the CUDA programming model. The hardware thread schedulers on current GPUs can execute transactions with simple extensions.

In addition to the traditional deadlock problem, GPU application developers have to deal with interactions between the SIMT stack and atomic operations. Ex-

Example 5 CPU spin-lock code. CAS = compare-and-swap.

```
A: while(CAS(lock,0,1)==1);
B: // Critical Section ...
C: lock = 0;
```



PC	RPC	T0	T1
B	--	1	1
A	B	0	1

Example 6 Spin-lock implementation on GPU to avoid deadlock due to implicit synchronization in warps [1].

```
A: done = 0;
B: while(!done){
C:   if(CAS(lock,0,1)==0){
D:     // Critical Section ...
E:     lock = 0;
F:     done = 1;
G:   }
H: }
```

ample 5 shows how a critical section may be guarded by the acquisition and release of a fine-grained lock (lines A and C) on the CPU. On a GPU, this code may deadlock [1]. This can happen if the threads in the same warp attempt to acquire the same lock at line A. For example, consider a warp with two threads, T0 and T1, both trying to acquire the same lock. T0 succeeds and exits the loop, but waits at the start of the critical section (line B) for reconvergence, while T1 still spins in the loop (see inset at right in Example 5). T1 will continue spinning and waiting for the lock held by T0 and never exit, forming a deadlock. To remove the deadlock, the program needs to be modified; Example 6 shows a typical solution. This issue is known among GPU application developers [1] and explored in more detail by Ramamurthy [129]. With TM, the GPU hardware can be designed to handle such interactions between transactions and the SIMT stack (see Section 4.2.1).

Figure 4.1 compares the performance of a set of GPU TM applications (described in Section 4.3) running on an ideal GPU TM system against fine-grained lock versions of the applications. In this ideal TM system, TM overheads related to detecting conflicting transactions are removed. Each committing transaction can instantaneously abort all of its conflicting transactions to resolve its conflicts, and it can update the memory with its write-set in a single cycle. This ideal TM retains the overheads related to re-executing aborted transactions. The performance shown is normalized to that obtained by serializing all transaction executions via a single global lock. On average, the applications running on ideal TM achieve 279× speedup over serializing all transactions, which is 24% faster than fine-grained locking. In BH, ideal TM is slightly slower than fine-grained locking because the thread that failed to acquire a lock can just wait until the lock is freed, whereas an aborted transaction requires full re-execution from the start, producing wasted work. In CC, ideal TM is slower because each pixel in the fine-grained lock version uses atomic add operations to update its neighbouring pixels. These atomic operations are implemented with special hardware that performs computation directly with data in the L2 cache (as explained in Section 2.3.3). The TM version, on the other hand, has to fetch the data into the core for the same computation.

Table 4.1 shows the IPC of these applications with the ideal TM system and fine-grained locking. For many of our applications, Ideal TM has lower IPC than fine-grained locking even though it has the lower execution time. This is because the fine-grained lock versions of these applications execute extra instructions to acquire/release locks and to spin when the acquisition fails. For other applications, re-execution of aborted transactions causes Ideal TM to have an higher IPC than fine-grained locking even though it has the higher execution time. Some of our applications (CL, BH and CC) achieve reasonable performance, while others suffer from GPU performance bottlenecks such as control flow and memory divergence (see Section 4.4.2 for further discussion). Notice that even though HT-L has lower IPC than HT-H for both Ideal TM and fine-grained locking, HT-L has lower execution time (i.e., runs faster) than HT-H. With fewer conflicts among transactions, HT-L has fewer aborted transactions than HT-H. This allows HT-L to complete with fewer instructions, causing it to have a lower IPC. We believe these applications can be optimized via performance tuning – identifying bottlenecks and redesigning the applications incrementally to address these bottlenecks one by one. Application performance tuning is beyond the scope of this work. Transactional memory arguably provides an easier programming model for performance tuning because it allows GPU application developers to rework algorithms and data structures without concern for deadlock. This work focuses on enabling TM on GPUs efficiently,

Table 4.1: Raw performance (instructions per cycle, or IPC) of GPU-TM applications described in Section 4.3 (Peak IPC = 240).

Applications →	HT-H	HT-L	ATM	CL	BH	CC	AP
Ideal TM	6.6	5.9	4.2	9.4	10.5	33.4	0.5
FG Lock	8.1	6.5	4.2	8.8	9.5	51.0	0.5

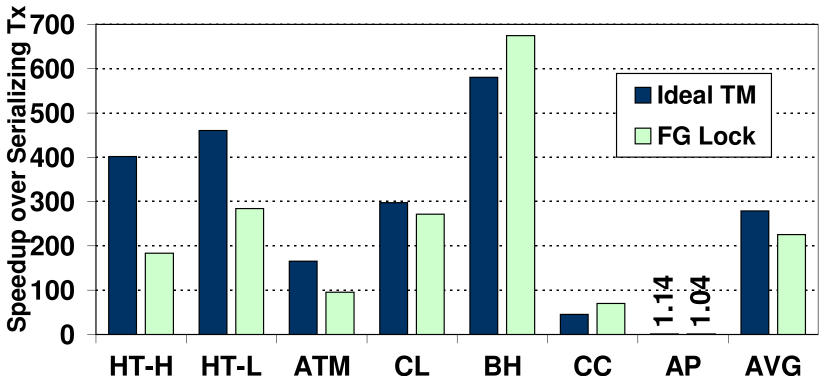


Figure 4.1: Performance comparison between applications running on an ideal TM system and their respective fine-grained (FG) locking version (applications described in Section 4.3).

and with minimum overhead.

4.1.1 Challenges with Prior HTMs on GPUs

Hardware transactional memory (HTM) has been researched extensively (see Section 7.2). Many proposed HTMs leverage cache coherence for conflict detection among concurrent transactions, while assuming that each transaction owns a private L1 cache. Even though recent GPUs have caches [121], the caches local to a SIMT core are not coherent and they are shared among 100s of threads that execute concurrently on the core. GPUs are designed to exploit fine-grained data parallelism; adjacent memory words are often accessed by different threads. These differences raise many challenges in adopting existing HTMs for GPUs.

An emerging class of manycore accelerators, such as Intel's Larrabee [142], feature fewer concurrent threads per core and coherent caches that can be partitioned per thread. The following challenges may be less severe for this class of manycore accelerators.

Access Granularity and Write Buffering

Each line in the L1 data caches could be extended to identify and isolate speculative data written by individual transactions. However, each transaction might obtain only a few cache lines before the cache overflows because there are fewer L1 cache lines than scalar threads on a SIMT core. Transactions typically lack the spatial locality required to fully use a cache line and make poor use of the few lines they can access. Furthermore, the fine-grained, interleaved accesses among different threads can introduce significant false-sharing and reduce the accuracy of conflict detection.

Transaction Rollback

Many proposed HTMs checkpoint the architectural state of the hardware thread at the start of a transaction for restoration upon rollback. Maintaining copies for 10s of registers at transaction boundaries in a CPU core is relatively cheap. GPUs, however, are designed to execute 1000s of concurrent threads, and spend significant hardware resources on register file storage. NVIDIA Fermi has 2MB of register file storage, which exceeds its aggregate cache capacity [121]. Naively checkpointing this many registers would introduce significant overheads.

Scaling Conflict Detection

A key challenge for scaling TM beyond 1000s of concurrent transactions is designing a conflict detection mechanism that works effectively at this scale. Naive broadcast-based conflict detection scales poorly; T concurrent transactions will broadcast to T-1 other transactions, generating $O(T^2)$ traffic.

Many proposed TMs use global metadata, such as a cache coherence directory, to eliminate unnecessary traffic. Recently, directory based cache coherence protocols supporting up to 1000 cores have been proposed [51, 86, 173]. However, GPUs such as Fermi [121] do not have a private cache for each thread.

Using bloom filters to represent read- and write-sets of a transaction [26, 111, 170] allows each thread to quickly react to incoming requests and enables fine-grained conflict detection. We experimented with an ideal version of a signature-based HTM (lazy conflict detection and lazy version management) with each transaction maintaining both its read- and write-sets in a bloom filter. We used the parallel bloom filters described by Sanchez et al. [137]. Each filter contained 4 separate sub-signatures and each sub-signature was indexed by a unique H_3 hash function. We had to use a 1024-bit filter per thread (3.8MB of total storage for 30720 threads) to keep the false conflict rate below 20% for the benchmarks CL, BH and AP. Using 512-bit filters increased the rate to 60%.

Commit Bottleneck

Even if we reduce the bloom filter storage by limiting the number of concurrent transactions (Section 4.2.6), the bloom filters of all transactions and the directory cannot be modified when one of the transactions is committing. Otherwise, a conflicting access may go undetected when a transaction that has resolved all of its conflicts is updating memory [27]. Scalable TCC [27] solves this issue by locking entries in the directory, but its commit protocol serializes transaction commit at each directory bank. LogTM-SE [170] uses eager version management, writing speculative data directly to global memory, to allow transactions to commit in parallel. However, this requires eager conflict detection *and* resolution to isolate the speculative data between concurrent transactions. It is not clear how eager conflict detection can be implemented on current GPUs, which do not feature invalidation-based cache coherence protocols. The potential commit bottleneck and the signature storage explosion (Section 4.1.1) issue persuaded us to explore alternatives.

4.2 Kilo Transactional Memory

In this section, we present Kilo Transactional Memory (Kilo TM), a TM system scalable to 1000s of concurrent transactions. Kilo TM does not leverage a cache coherence protocol for conflict detection among running transactions. Instead, each transaction performs *word-level, value-based conflict detection* against committed transactions by comparing the saved value of its read-set against the value in memory upon its completion [37, 124]. A changed value indicates a conflict. This mechanism offers weak isolation [73]. Each transaction buffers its saved read-set

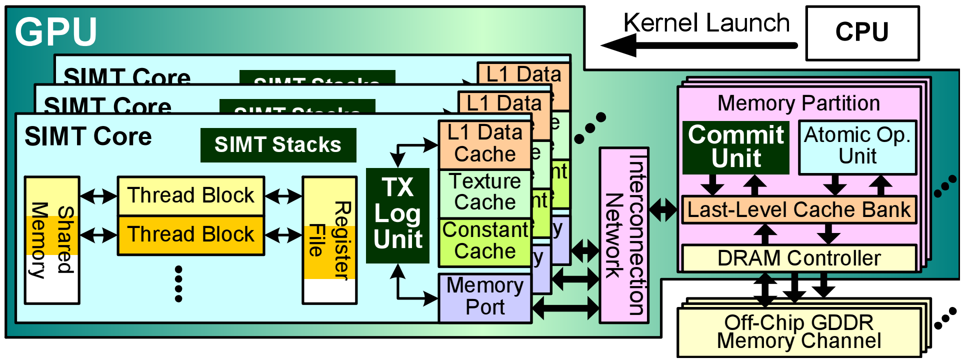
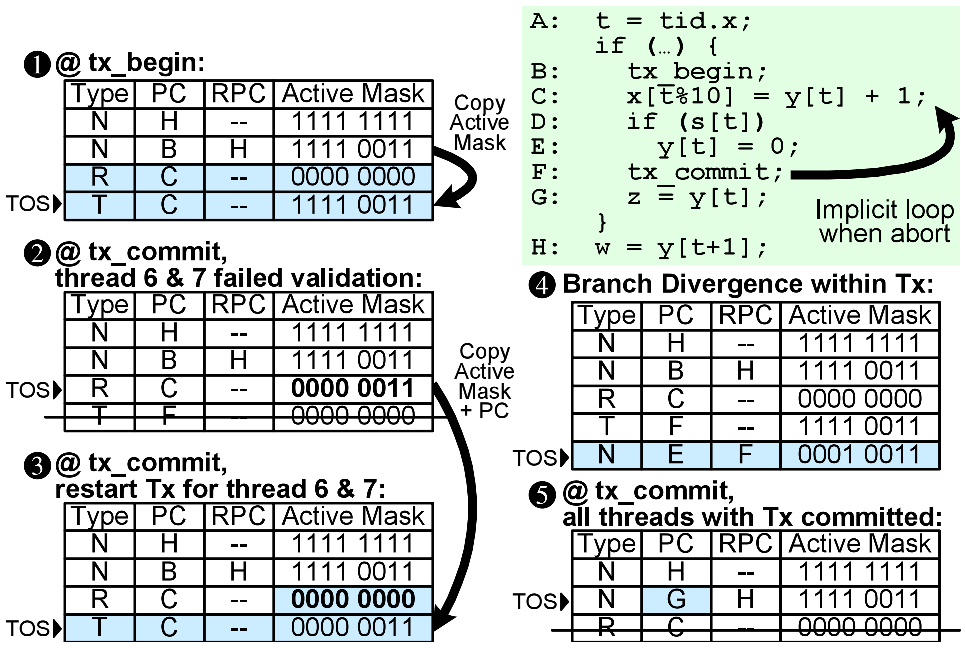


Figure 4.2: Kilo TM Implementation Overview. Kilo TM adds a transaction (TX) log unit to each SIMT core and a commit unit to each memory partition. The SIMT stacks are extended to support transactions.

values and memory writes in a read-log and a write-log (in address-value pair) in local memory (lazy version management). When a transaction finishes executing, it sends its read- and write-log to a set of commit units for conflict detection (validation), each of which replies with the outcome (pass/fail) back to the transaction at the core. Each commit unit validates a subset of the transaction's read-set. If all commit units report no conflict detected, the transaction permits the commit units to publish the write-log to memory. To improve commit parallelism for non-conflicting transactions, transactions speculatively validate against committed transactions in parallel, leveraging the deeply pipelined memory subsystem of the GPU. The commit units use an address-based conflict detection mechanism to detect conflicts among these transactions (we call these *hazards* to distinguish them from the conflicts detected via value comparison). A hazard is resolved by revalidating one of the conflicting transactions at a later time. Section 4.2.5 describes the protocol in detail.

In Kilo TM, transaction-specific communication (conflict detection and memory updates) occur only between the commit units and the committing thread (shown in Figure 4.4a). This restriction permits the communication packets from different threads to be pipelined and interleaved, as long as the end-to-end message order between the SIMT cores and the commit units is maintained. Kilo TM restricts each transaction to have a single entry and a single exit, matching



```
A:  t = tid.x;
    if (…) {
B:    tx_begin;
C:    x[t%10] = y[t] + 1;
D:    if (s[t])
E:      y[t] = 0;
F:    tx_commit;
G:    z = y[t];
    }
H:  w = y[t+1];
```

❶ **@ tx_begin:**

	Type	PC	RPC	Active Mask
	N	H	--	1111 1111
	N	B	H	1111 0011
	R	C	--	0000 0000
TOS	T	C	--	1111 0011

❷ **@ tx_commit, thread 6 & 7 failed validation:**

	Type	PC	RPC	Active Mask
	N	H	--	1111 1111
	N	B	H	1111 0011
TOS	R	C	--	**0000 0011**
	~~T~~	~~F~~	~~--~~	~~0000 0000~~

❸ **@ tx_commit, restart Tx for thread 6 & 7:**

	Type	PC	RPC	Active Mask
	N	H	--	1111 1111
	N	B	H	1111 0011
	R	C	--	**0000 0000**
TOS	T	C	--	0000 0011

❹ **Branch Divergence within Tx:**

	Type	PC	RPC	Active Mask
	N	H	--	1111 1111
	N	B	H	1111 0011
	R	C	--	0000 0000
	T	F	--	1111 0011
TOS	N	E	F	0001 0011

❺ **@ tx_commit, all threads with Tx committed:**

	Type	PC	RPC	Active Mask
	N	H	--	1111 1111
TOS	N	G	H	1111 0011
	~~R~~	~~C~~	~~--~~	~~0000 0000~~

Figure 4.3: SIMT stack extension to handle divergence due to transaction aborts (validation fail). Thread 6 and 7 have failed validation and are restarted. Stack entry type: Normal (N), Transaction Retry (R), Transaction Top (T). For each scenario, added entries or modified fields are shaded.

'atomic{}' semantics in common TM language extensions [73]. Transaction boundaries are conveyed to hardware with tx_begin and tx_commit instructions in the compute kernel. Nested transactions are flattened [73] into a single transaction.

Figure 4.2 highlights the changes required to implement Kilo TM on our baseline GPU architecture. These include an extension to the SIMT stack, a Transaction Log Unit, and a Commit Unit.

4.2.1 SIMT Stack Extension

When a warp finishes a transaction, each of its active threads will try to commit. Some of the threads may abort and need to reexecute their transactions, while other threads may pass the validation and commit their transactions. Since this outcome may not be unanimous across the entire warp, a warp may diverge after validation.

Figure 4.3 shows how the SIMT stack can be extended to handle control flow divergence due to transaction aborts. When a warp enters the transaction (at line B, `tx_begin`), it pushes two special entries onto the SIMT stack (❶). The first entry of type R stores information to restart the transaction. Its active mask is initially empty, and its PC field points to the instruction after `tx_begin`. The second entry of type T tracks the current transaction attempt. At `tx_commit` (line F), any thread that fails validation sets its mask bit in the R entry. The T entry is popped when the warp finishes the commit process (i.e., its active threads have either committed or aborted) (❷). A new T entry will then be pushed onto the stack using the active mask and PC from the R entry to restart the threads that have been aborted. Then, the active mask in the R entry is cleared (❸). If the active mask in the R entry is empty, both T and R entries are popped, revealing the original N entry (❺). Its PC is then modified to point to the instruction right after `tx_commit`, and the warp resumes normal execution. Branch divergence of a warp within a transaction is handled in the same way as non-transactional divergence (❹).

4.2.2 Scalable Conflict Detection

Section 4.1.1 discussed how signature-based conflict detection is prone to the commit bottleneck and storage explosion when scaled to 1000s of threads. Typical conflict detection used in HTMs checks the existence of conflicts *and* identifies the specific conflicting transactions. One insight Spear et al. [149] present with RingSTM is that a committing transaction only needs to detect the existence of conflicts with transactions that have committed. Transactions with detected conflicts can self-abort without interfering with the execution of other running transactions. This reduces storage and traffic requirements because the TM system does not need to maintain a set of in-flight sharers/modifiers for each memory location, and each transaction only performs the detection once before it commits. However, in our experiment with RingSTM, we had to use 512-bit write-signatures in the commit record ring to keep the false conflict rate below 40% (1.9MB of total storage for a ring with 30720 records to support 30720 concurrent transactions). Value-based conflict detection [37, 124] exhibits similar traffic requirements as RingSTM.

Transactions detect conflicts with other committed transactions, but without using any global metadata – only values from global memory are used. Kilo TM combines aspects of RingSTM and value-based conflict detection in hardware, and extends them to permit concurrent validations (Section 4.2.5).

A transaction is *doomed* if it has observed an inconsistent view of memory (e.g., in between memory reads to two different locations, another transaction has committed and updated both locations). These doomed transactions may enter an infinite loop. To ensure that doomed transactions are eventually aborted, we use a watchdog timer to trigger a validation pass. This satisfies *opacity* [68] with minimum overhead for GPUs.

4.2.3 Version Management

Kilo TM manages global memory accesses in hardware and uses software for version management of registers and local memory space. Section 4.1.1 discussed how blindly checkpointing each transaction is too expensive on GPUs. We observed that the original values in many registers are rarely used when a transaction restarts, and do not need to be restored. A compiler could determine which registers are both read and written within a transaction and insert code to checkpoint and restore them before/after a transaction – similar to existing code generation technique to create idempotent code regions [40, 41]. We observed that transactions in the BH benchmark require restoring *two* registers on average. Other benchmarks do not require any register restoration upon transaction aborts. Hence, we do not model the register checkpoint overhead in our evaluation as we believe it to be minor compared to validation and commit overheads in our workloads.

Accesses to global memory are buffered in the read/write-log in local memory. A small bloom filter can be used to detect whether a transaction is reading a value in its write-set. A hit in the filter will trigger the transaction log unit to walk the write-log. Since the member set of the filter is constrained to only the memory accesses of a single transaction, a small filter should produce reasonably few false positives. In our evaluations, this detection is perfect.

4.2.4 Transaction Log Storage and Transfer

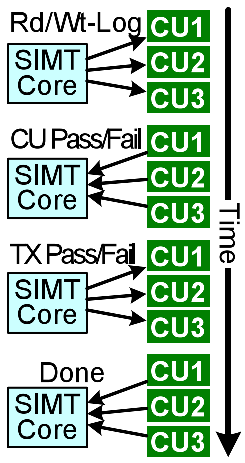
The read- and write-logs of transactions in Kilo TM are stored as linear buffers in local memory located in off-chip DRAM, cached in the per-core L1 data cache, and mapped to physical addresses such that consecutive 32-bit words are accessed by consecutive scalar threads in a warp. GPU applications can specify the maximum size of local memory to avoid overflow.

When a warp accesses global memory in a transaction, a new entry is appended to the read/write-log for all threads in the warp. Entries for the inactive threads are marked with a special address to void the entry. This organization allows the log accesses to be coalesced. If only part of a warp needs to walk the write-log for data, the entire warp will wait for the walk to finish before proceeding to the next instruction. When threads in a warp are ready to validate their transactions (before commit), the transaction log unit walks the read- and write-logs of each thread and sends the address-value pairs to the commit unit in the corresponding memory partition. Entries from different threads at the same log offset are accessed in parallel with coalesced memory reads. The individual entries sent to the same memory partition are grouped into a single larger packet to reduce interconnection traffic.

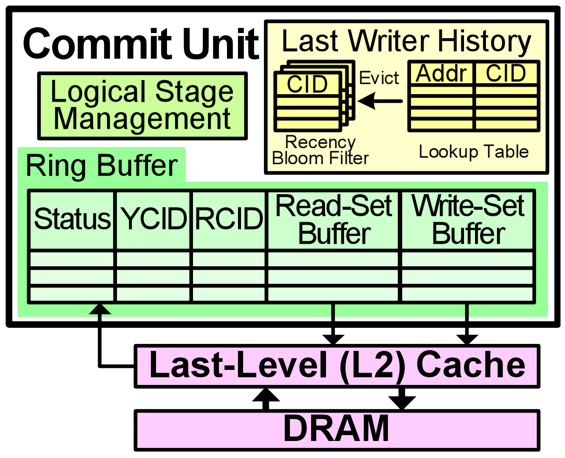
This transaction log design addresses the fact that per-core caches in contemporary GPUs are shared by 100s of threads. GPUs employ a flexible register allocation scheme that balances the number of registers per warp against the number of concurrent warps to avoid spilling registers. Hence, memory reads rarely access data written by the same transaction, reducing the penalty of storing the write-log as a linear buffer.

4.2.5 Distributed Validation/Commit Pipeline

A naive implementation of value-based conflict detection serializes transaction commits. Memory updates from a transaction (its write-set) are invisible to others until the transaction commits. Two conflicting transactions validating concurrently will observe no changes to their read-sets, and will subsequently update global memory with their contradicting write-sets. While serializing all transaction commits prevents this potential data race, it also prevents non-conflicting transactions



(a)



(b)

Figure 4.4: Commit unit communication and design overview. (a) Communication flow with a SIMT core. (b) Overview of a commit unit.

from committing in parallel [18].

To enable parallel commits, prior STMs with value-based conflict detection [37, 124] use a set of versioned locks, each serializing commit to a memory region. Each transaction checks/acquires the locks of all the memory regions that require protection during validation and commit. Acquiring locks imposes significant overhead.

Kilo TM increases commit parallelism by using a set of *commit units* that quickly detect conflicts among a limited set of transactions. GPU memory subsystems are deeply pipelined to support a large number of in-flight accesses to maximize throughput. The commit units leverage this capability. In each commit unit, a subset of transactions are speculatively validated in parallel. This validation only detects conflicts with the already committed transactions. Later, a *hazard detection* mechanism is applied to conservatively detect all potential conflicts. Any hazard is resolved by deferring one of the conflicting transactions and *revalidating* its read-set after the other transaction has updated global memory. Revalidation serializes the validation/commit process among transactions when necessary. This mechanism guarantees forward progress by giving the deferred transaction a sec-

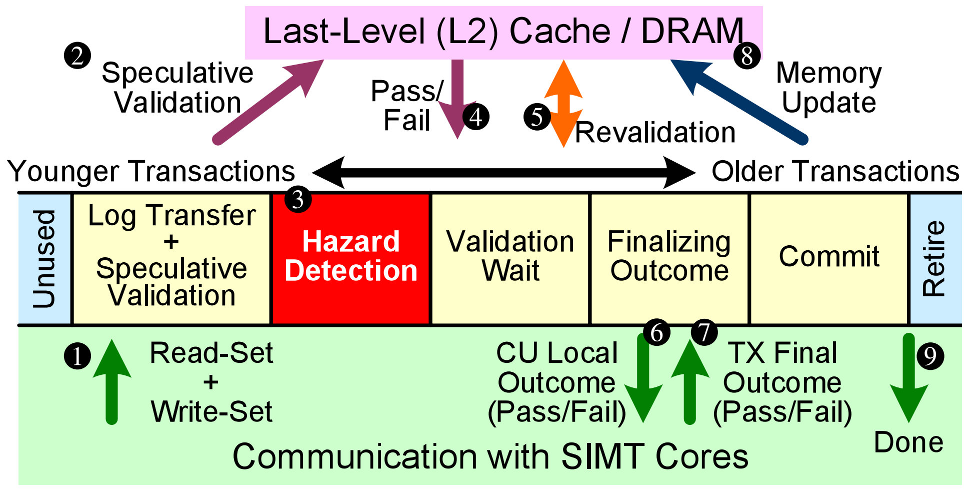


Figure 4.5: Logical stage organization and communication traffic of the bounded ring buffer stored inside commit unit.

ond chance to validate and commit in case the earlier transaction failed.

Each memory partition has a commit unit (shown in Figure 4.4b) that handles validations and commits of TM accesses to that memory partition. Before a transaction starts the validation/commit process, it acquires a commit ID (CID) from a centralized ID vendor (similar to Scalable TCC [27]). This commit ID is associated with a logical entry in the commit unit at every memory partition, and dictates the relative commit order of this transaction (so that the conflict/hazard resolution is unanimous among all commit units). Each commit unit has a ring buffer of commit entries [149] organized in the logical stages shown in Figure 4.5. Each entry tracks the state of a committing transaction in this memory partition. The Status field in each commit unit ring buffer entry in Figure 4.4b indicates the current status of the transaction. The YCID (youngest commit ID) and RCID (retired commit ID) fields are used for hazard detection. The RCID of a transaction is a pointer to the oldest committing transaction when this transaction started speculative validation. Transactions that committed before this transaction started speculative validation do not trigger a hazard with this transaction. The YCID of a transaction points to the youngest conflicting transaction detected to have a hazard with this transaction. The transaction needs to wait for the conflicting transaction to retire before it can start revalidation. The Read-Set and Write-Set Buffers consist of bounded linear buffers that store, for value comparison, the exact address-value pairs of each transactional access to this memory partition.

The following is an overview of the validation/commit process of a transaction at different logical stages (Figure 4.5):

Log Transfer + Speculative Validation. The transaction transfers its read- and write-logs to an allocated entry in the commit unit (➊). The incoming read-set is speculatively validated against the current values in global memory (accessing L2 cache/DRAM ➋).

Hazard Detection. Once the read- and write-logs have been transferred to the commit unit, the read-set of the transaction is checked against the *Last Writer History* unit (LWH) for *hazard detection* (➌), detecting conflicts between the transaction and all committing transactions in the later stages. Existence of a hazard indicates that the speculative validation may have accessed stale data in global memory that will be updated before the transaction commits. The hazard is resolved in the Validation Wait stage.

Validation Wait. Each transaction waits for the speculative validation to complete before advancing to later stages (➍). Transactions with hazards will wait until all conflicting transactions have retired to revalidate their read-set with the updated global memory (➎).

Finalizing Outcome. This stage finalizes the outcome of each transaction by replying with the local outcome (pass/fail) of the transaction to the core (➏). After a transaction has received replies from all commit units, it will broadcast the final outcome (pass/fail) to all commit units (➐). Each commit unit entry waits for its final outcome before proceeding to the next stage.

Commit. Each passed transaction updates the global memory at this stage (➑). Failed transactions are skipped.

Retire. The commit unit entry associated with each transaction is deallocated, releasing storage for future transactions. The core is informed so that the thread running the transaction can proceed to the next instruction, or restart the failed transaction (❾).

Commit Unit Resource Allocation

When a warp executes `tx_commit`, the transaction log unit acquires credits from a per-core credit pool of commit unit entries before acquiring contiguous commit IDs and proceeding with the commit. Insufficient credits prevent the warp from acquiring the commit IDs until the credits are returned when the validation/commit operation of another warp completes. In this work, we assume that the commit units always have enough entries to support all in-flight transactions. Section 4.4.4 measures the resources required.

Hazard Detection, Last Writer History Update

At the Hazard Detection stage, each transaction checks the integrity of its speculative validation via the *Last Writer History* unit (LWH) in Figure 4.4b. This unit has an approximate but conservative representation of the write-sets of all older transactions at the later stages that have not yet retired. The LWH unit identifies the youngest conflicting transaction (returns its commit ID) in the later stages that may modify the read-set of the transaction at the hazard detection stage. If this conflicting transaction retired before the current transaction started validating (its CID<RCID of the current transaction), no hazard remains. Otherwise, a hazard is detected. The hazard is resolved in the Validation Wait stage by waiting for this conflicting transaction (now tracked by YCID) to retire, and then revalidating the transaction with the updated memory. After detection, the current transaction updates the LWH unit with its write-set. This mechanism leverages the same intuition described in Section 4.2.2. The LWH unit can approximately maintain the latest pending writer to each memory location, as a slightly younger false writer only slightly lengthens the wait at Validation Wait stage.

The LWH unit has an address-indexed set-associative lookup table and a *recency bloom filter*. The two structures, in combination, conservatively track the CID of the *youngest* transaction in later stages that may write to a given memory location. The lookup table stores the exact write-sets from recent transactions, whereas the bloom filter stores the approximate write-sets from distant transactions. As write-sets from newer transactions are deposited into the lookup table, entries are updated (replacing the CID if addresses match), or evicted into the recency bloom filter to free up storage for different addresses. A large transaction with its write-set exceeding the capacity of the lookup table automatically overflows part of the write-set into the recency bloom filter. The recency bloom filter has multiple sub-arrays of buckets (each bucket storing a CID) indexed by a hash of the given memory address. Each evicted entry updates a CID bucket in each sub-array according to the hashed written memory address. Due to address aliasing in each sub-array, an older CID writing to an address may be replaced by a younger CID writing to a different address. When the bloom filter is queried with an address, one CID is retrieved from each sub-array. The *oldest* retrieved CID is returned as it is least likely to have been aliased by a younger writer. This oldest CID can also be aliased, causing the LHW unit to report a false writer. The write-set of a retiring transaction is implicitly removed from the LWH unit as its CID can no longer trigger a hazard.

Unbounded Transactions

If the commit entry's read-set buffer overflows, the commit unit will continue to speculatively validate the address-value pair of the incoming read-set, but will stop populating the read-set buffer. The commit unit will ask the transaction to resend its read-set from the SIMT core during hazard detection and revalidation. Similarly, if the commit entry's write-set buffer overflows, the commit unit will ask the transaction to resend the write-set during LWH update after hazard detection and memory update at the Commit stage.

4.2.6 Concurrency Control

While Kilo TM can support thousands of concurrent transactions, limiting the number of concurrent transactions can improve the performance of high-contention applications (with transactions that are likely to abort), and lowers the resource requirement for Kilo TM. To limit the number of concurrent transactions within a SIMT core, we use a counter to track the number of warps currently in transactions.

Table 4.2: General characteristics of evaluated GPU-TM applications. Instruction count (#Inst) obtained from Ideal TM version.

Name	Abbr.	#Inst	Blk Size	Grid Size	#Blk/ SIMT Core
Hash Table (CUDA)	HT-H	632k	192	120	4
	HT-L	501k	192	120	4
Bank Account (CUDA)	ATM	4.1M	192	120	3
Cloth Physics [20] (OpenCL)	CL	6.8M	512	118	1
Barnes Hut [23] (CUDA)	BH	15M	288	60	1
CudaCuts [162] (CUDA)	CC	104M	256	133	1
Data Mining [3, 88] (CUDA)	AP	39M	64	112	4

We leave to future work exploration of adaptive mechanisms (e.g., [15, 171]) that react to the dynamic contention in applications.

4.3 Methodology

We model our proposed hardware changes by extending GPGPU-Sim 3.0 [10]. We evaluate performance of various hardware configurations on a set of GPU-TM applications listed in Table 4.2 and Table 4.3. Since we do not have access to the CUDA compiler front end source code, we need an alternative mechanism to communicate transaction boundaries to the simulated hardware. Our approach was to add transactions with empty functions `tx_begin()` and `tx_commit()`. Calls to these functions are recognized by the simulator as transaction boundaries and reinterpreted by the simulator as hardware instructions. Since these functions are empty, we need to ensure that CUDA and OpenCL compilers do not optimize them out. We do this by using the `__noinline__` keyword in CUDA and by making the functions self-recursive in OpenCL (as the OpenCL driver we used does not provide a no-inline option). Note these dummy functions are never actually called during simulation. The following CUDA/OpenCL applications are used in our evaluations.

Hash Table (HT) is a microbenchmark in which each thread inserts an element into a chained hash table. Each slot in the hash table is a linked list of key-value pairs. We use two table sizes to create high contention (HT-H with 8k entries) and

Table 4.3: TM-specific characteristics of evaluated GPU-TM applications. Metrics collected with to Kilo TM with unlimited concurrency.

Name	#Committed TX	#Inst per TX (Avg)	#Aborts per TX Avg	#Aborts per TX Max	Read-Set (#Words) Avg	Read-Set (#Words) Max	Write-Set (#Words) Avg	Write-Set (#Words) Max	Max # Concurrent TX (Kilo TM)
HT-H	23040	26	1.39	2	1.0	1	4.0	4	23040
HT-L	23040	26	0.14	2	1.0	1	4.0	4	23040
ATM	122880	8	0.03	3	3.0	3	2.0	2	16131
CL	60200	53	1.06	8	11.2	12	4.8	8	22816
BH	264106	48	0.15	14	4.3	40	0.82	14	8640
CC	114677	21	0.004	3	1.4	4	1.4	4	735
AP	4550	89	0.32	6	15.7	174	6.2	109	192

low contention (HT-L with 80k entries) workloads.

Bank Account (ATM) is a microbenchmark with ∼16k concurrent threads accessing an array of structs that represents 1M bank accounts. Each transaction transfers money between two accounts.

Cloth Physics (CL) is based on “RopaDemo”, which simulates the cloth physics for a T-shirt [20]. Performance is limited by the Distance Solver kernel, which implements a spring-mass system using a set of constraints between cloth particles. To forbid two constraints concurrently modifying the same particle, the original demo processes these constraints sequentially in octets (i.e., 8 at a time). We modified this kernel to process all ∼4k distance constraints of each T-shirt in parallel transactions.

Barnes Hut (BH) is based on the tree-based n-Body algorithm implemented by Burtscher et al. [23] with 30000 bodies. We focus on the iterative tree-building kernel using lightweight locks, which we modified to use transactions. Each thread in this kernel appends a body into the octree, and inserts any branch node required to isolate its body in a unique leaf node. Each level of traversal down the tree and the node insertions are protected by separate transactions.

CudaCuts (CC) applies a maxflow/mincut algorithm to segmentation of a 200×150 pixel image [162]. It consists of Push kernels that use atomic operations to push excessive flow from a node to its neighbours, and Relabel kernels that change the height of a node when excessive flow cannot be pushed. We grouped consecutive atomic operations in the Push kernels into transactions.

Table 4.4: GPGPU-Sim configuration for Kilo TM evaluation

# SIMT Cores	30 (10 clusters of 3)
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Branch Divergence Method	PDOM [56]
Warp Scheduling Policy	Loose Round Robin
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	5KB, 32B line, 20-way assoc.
L1 Data Cache / Core	48KB, 128B line, 6-way assoc. (transactional and local memory access only)
L2 Unified Cache	64KB/Memory Partition, 128B line, 8-way assoc.
Interconnect Topology	1 Crossbar/Direction (SIMT Core Concentration=3)
Interconnect BW	32 (Bytes/Cycle) (160GB/dir.)
Interconnect Latency	5 Cycle (Interconnect Clock)
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
# Memory Partitions	8
DRAM Req. Queue Capacity	32
Memory Controller	Out-of-Order (FR-FCFS)
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8 t_{CDLR}=6 t_{WR}=11
Memory Channel BW	8 (Bytes/Cycle)
Min. L2/DRAM Latency	460 Cycle (Compute Core Clock)
Kilo TM	
Commit Unit Clock	650 MHz
Validation/Commit BW	1 Word/Cycle/Memory Partition
#Concurrent TX	2 Warps/Core (1920 threads)
Last Writer History Unit Size	5kB (See Section 4.4.3)

Data Mining (AP) is based on Apriori in the RMS-TM benchmark suite [3, 88]. We evaluate the `apriori_gen()` function, which was modified [129] to use CUDA, with each thread processing a unique record. As in the CPU TM version, transactions are used to protect candidate insertion and support value counting.

Our modified GPGPU-Sim is configured to model a GPU similar to NVIDIA Quadro FX5800, extended with L1 data caches and a L2 unified cache similar to NVIDIA Fermi [121]. We validated GPGPU-Sim 3.0 with the NVIDIA Quadro FX5800 configuration (no cache extensions and using PTX instead of SASS) against the hardware GPU and observed an IPC correlation of ~0.93 for a subset of the CUDA SDK benchmarks. GPGPU-Sim incorporates a configurable interconnection network simulator [38]. Traffic in each direction between the SIMT cores and the memory partitions are serviced by two separate crossbars. The crossbars can transfer a 32-byte flit per interconnect cycle to/from each memory partition ($\sim$160GB/s per direction). Each flit takes 5 cycles to traverse the crossbar. The 30 SIMT cores are grouped in 10 clusters. Cores in a cluster share a common port to each crossbar (concentration of three). The memory partition has an out-of-order memory access scheduler. We model detailed GDDR3 timing. Every memory access sent to L2 cache/DRAM has a minimum pipeline latency of 460 cycles (in compute core clocks) to match that observed by microbenchmarks of NVIDIA Quadro FX5800 [165]. The actual latencies of individual accesses can be higher due to delays from memory access scheduling and queuing as DRAM bandwidth saturates. We have an optimistic performance model for atomic operations (used in fine-grained locking). Atomic compare-and-swap operations on GPGPU-Sim have $\sim 4\times$ higher throughput than on NVIDIA Fermi GPU, while other types of atomic operations on GPGPU-Sim perform roughly the same as Fermi. Table 4.4 lists the other major configuration parameters.

We model all interconnection network traffic between the SIMT cores and the commit units. Packets from the transaction log unit are sized according to the payload within the packet, and they contend for the same interconnection port with packets for normal memory accesses. Each short commit protocol message occupies a single flit. Packets containing multiple read/write-log entries (see Section 4.2.4) may occupy multiple flits, taking multiple cycles to transfer. In our evaluations, Kilo TM validates and commits each transaction as directed by the timing simulation. In our simulation, timing of committing transactions affects functional behaviour of the application, and hence any undetected data-race would likely lead to an application error, which we verify does not occur.

This modified version of GPGPU-Sim and the evaluated GPU-TM applications are available online [61, 62].

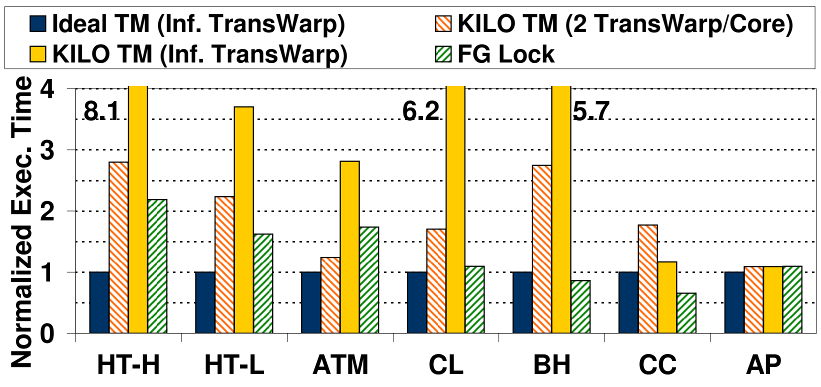


Figure 4.6: Execution time of GPU-TM applications with Kilo TM. Lower is better.

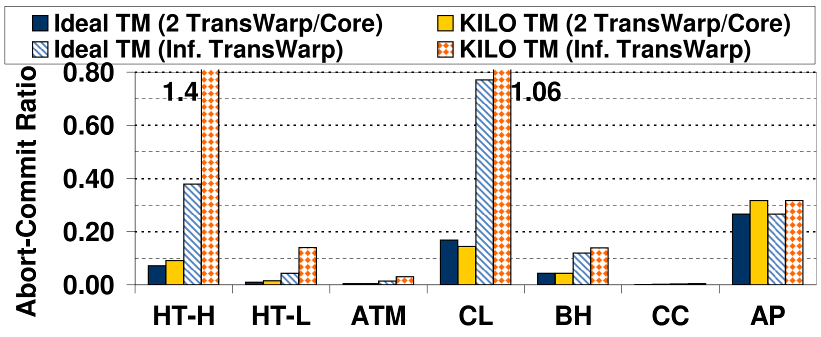


Figure 4.7: Abort/commit ratio of GPU-TM applications with Kilo TM. Lower is better. Ratio = 1 if on average transactions abort once.

4.4 Experimental Results

4.4.1 Performance

In this section, we compare the performance of Kilo TM against the ideal TM system (Ideal TM) and fine-grained locking (FG Lock) described in Section 4.1. Figure 4.6 shows the execution time of each application with Kilo TM and fine-gained locking normalized to the execution time of Ideal TM. In our evaluations, Kilo TM uses commit units with unlimited capacity.

With unlimited transaction concurrency (Inf. TransWarp), Kilo TM is on average 4.1× slower than Ideal TM. HT, CL, and BH are affected the most. These

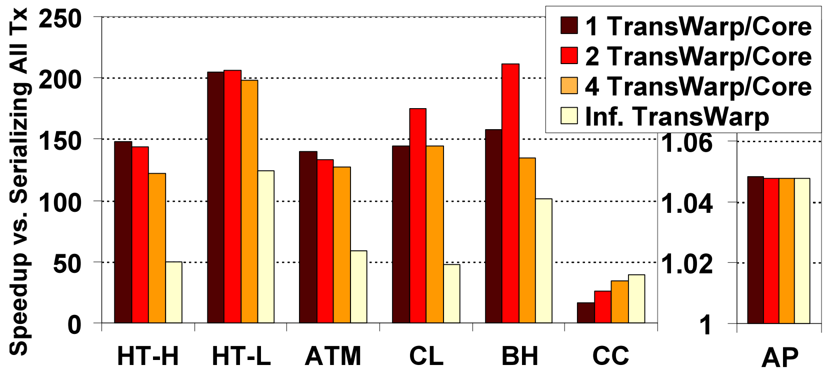


Figure 4.8: Performance scaling with increasing number of concurrent transactions with Kilo TM. Higher is better.

applications have many concurrent transactions with high contention (Figure 4.7). Although BH's overall abort-commit ratio is relatively low, it starts with a high-contention period when all transactions are competing to insert nodes near the root of the octree. When conflicting transactions attempt to commit concurrently, the commit unit defers revalidating one transaction. This reduces overall performance. Notice that AP has relatively few concurrent transactions, so its high abort-commit ratio has little impact on performance (Table 4.3).

Limiting each core to two transaction warps (2 TransWarp/Core in Figure 4.6, 1920 threads globally) reduces contention in HT-H, CL, and BH and improves their performance with Kilo TM by 2-3×. ATM speeds up by 2.3× from improved hazard detection accuracy. The performance of HT-L improves by 66%, while AP is unaffected. CC's performance drops by 34% because of this limit. In CC, warps are typically diverged before entering transactions. CC would not be penalized with thread-level concurrency control. Overall, Kilo TM performs significantly better with this concurrency limit, capturing 52% of Ideal TM and 59% of fine-grained locking performance.

Concurrency Control

Figure 4.8 compares the performance of Kilo TM under different concurrency limits versus serializing execution of all transactions. HT-L, ATM, CL and BH achieve the best performance with transaction execution limited to two warps per core (2

TransWarp/Core), while HT-H performs best with transaction execution limited to one warp per core (1 TransWarp/Core). CC prefers unlimited transaction concurrency. AP is insensitive to the limit. Overall, Kilo TM performs best with transaction concurrency limited to two warps per core, achieving on average 128× speedup over serially executing each transaction.

Effects on Abort-Commit Ratio

Figure 4.7 compares the abort-commit ratios between Kilo TM and the ideal TM system. Kilo TM and Ideal TM show similar abort-commit ratios with transaction concurrency limited to two warps per core. With unlimited transaction concurrency, contention at the commit unit defers memory updates from older transactions that would have been made visible much earlier with Ideal TM. Younger transactions that were originally reading the updated values in Ideal TM now conflict with the older uncommitted transactions.

4.4.2 Execution Time Breakdown

To provide further insight, Figure 4.9 shows a breakdown of the cumulative per-hardware thread cycles, scaled by the overall execution time of each application. At each cycle, a thread can be in a warp stalled by *Concurrency control* (TC), be in a warp committing its transactions (TO), have passed commit and be *Waiting* for other threads in its warp to pass (TW), be executing an eventually *Aborted* (TA) or committed/*Useful* (TU) transaction, be acquiring a lock or performing an *Atomic operation* (AT), be waiting at a *Barrier* (BA), or be performing non-atomic non-transactional work (NL). We compare the thread-state distributions between the fine-grained locking versions of the benchmarks (FGL), and the transactional versions running on Ideal TM (IDEAL), Kilo TM with transaction concurrency limited to two warps per core (KL), and Kilo TM with unlimited transaction concurrency (KL-UC).

We observe the overheads of lock acquisition (AT) in the lock-based versions to be proportional to the inherent contention in their transactional versions. Transactional HT-H and CL have the largest abort-commit ratios in Figure 4.7 and their lock-based counterparts have the greatest locking overheads in Figure 4.9. HT-

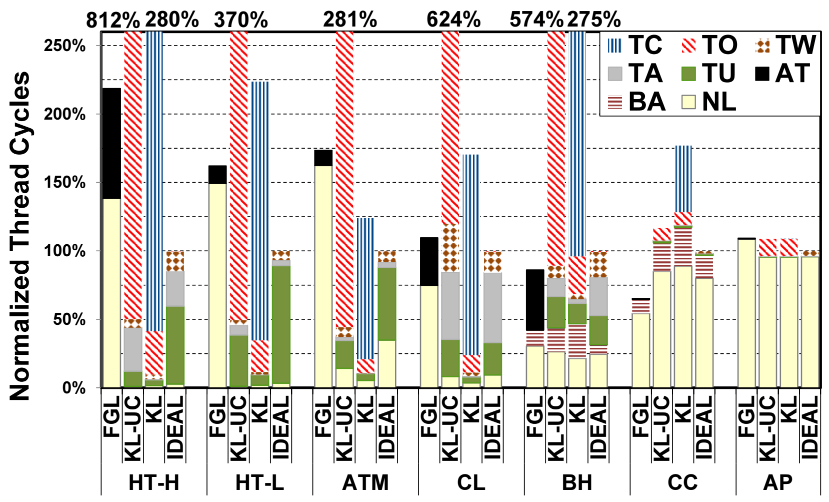


Figure 4.9: Breakdown of thread execution cycles for Kilo TM. Scaled by the overall execution time, normalized to IDEAL TM.

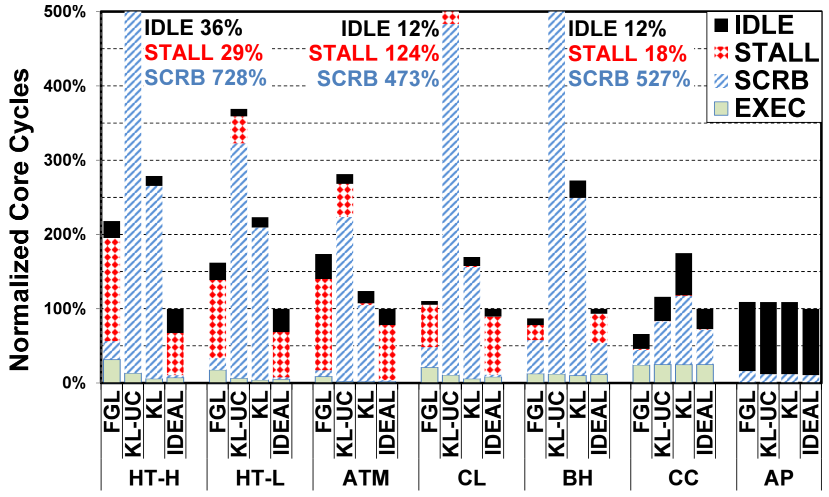


Figure 4.10: Breakdown of core execution cycles for Kilo TM.

L, ATM and CC have lowest abort-commit ratios and the smallest locking overheads. Lock-based BH has a significant locking overhead because of the initial high-contention period, as explained in Section 4.4.1. Lock-based AP shows insignificant locking overhead, despite a high abort-commit ratio in its transactional version, due to limited parallelism in its implementation. For the lock-based benchmarks, the NL cycles include the execution of the critical sections and are therefore greater than in the transactional versions. Detailed analysis (not shown) indicates that lock-based benchmarks suffer from increased branch divergence, further increasing their NL cycles.

Threads running on Kilo TM with unlimited transaction concurrency spend much of their time waiting to be committed (TO). This overhead is significantly reduced by limiting transaction execution to two warps per core in exchange for long waits in concurrency control (TC). For most benchmarks this provides an overall gain in performance. CC's performance on Kilo TM, however, degrades with concurrency control. This is because CC's originally low commit overhead remains unchanged with reduced concurrency, and because CC benefits from increased transaction concurrency as indicated by its scaling performance in Figure 4.8. Figure 4.10 shows the cumulative execution cycle breakdown of each core. At each cycle, a SIMT core may issue a warp (EXEC), be stalled by downstream pipeline stages (STALL), have all warps blocked by the scoreboard due to data hazards, concurrency control, pending commits or any combination thereof (SCRB), or not have any warps ready to issue in the instruction buffer (IDLE). This figure shows that limiting concurrency in Kilo TM reduces stalling and waiting at the scoreboard. Stalling is reduced as a result of fewer concurrent transactional memory accesses, while shorter and fewer commits reduce the amount of time spent waiting at the scoreboard.

The amount of time spent on transactional work, indicated by TU and TA in Figure 4.9, is lower on KL than on KL-UC and IDEAL. This is also due to the reduction in STALL cycles in Figure 4.10 for KL. Reduced stalling leads to faster transaction completion. BH saw only a small decrease in transaction time when concurrency was reduced. This is because BH contains inherent and limiting memory dependencies that are visible in Figure 4.10 as SCRB cycles on IDEAL TM. Similar to TU and TA, TW also decreases with reduced concurrency as passed transactions spend less time waiting for failed transactions in their warp to re-execute and commit. The amount of time spent on non-transactional work (NL) varies among KL, KL-UC and IDEAL. This is because threads doing transactional work and non-transactional work may execute in parallel, allowing the differences in transaction behavior in the different TM systems to have an impact on the performance of the non-transactional work.

In Figure 4.9, HT-H, HT-L and ATM spend less time doing useful transactional work (TU) on KL-UC than on IDEAL, even though both have unlimited concurrency. This is because Kilo TM caches global memory writes in write-logs stored in the L1 data cache. HT-H benefits most from this buffering during transaction execution as its transactions are dominated by writes (See Table 4.3). HT-L and ATM's lower data locality negates some of the benefit of write buffering. CL and BH are dominated by reads and gain little benefit from write buffering. The memory write overhead of write buffering is eventually incurred during transaction commit (TO).

CC and AP both suffer from load imbalance as indicated in Figure 4.10 by the significant portion of IDLE cycles - the portion of the time when the cores run out of warps to execute. The inter-thread load imbalance suffered by CC is exacerbated by transactional overheads. AP suffers from inter-core load imbalance. AP spends most its execution in non-transactional work, but the overhead of Kilo TM still impacts performance because of the time involved in transferring logs for the large transactions. AP spends 90% of its core cycles in IDLE. This behaviour contributes to the low absolute performance of AP. We created the CUDA version of AP from its CPU TM version without changing much of the algorithm and data structures. An improved version may redesign the algorithm to spread the workload across more threads.

Overall, even with a significant portion of time spent on executing aborted transactions, the Ideal TM system performs comparably to fine-grained locking. This indicates that the performance penalties of Kilo TM may be reduced with future refinements.

4.4.3 Sensitivity Analysis

L2 Cache Miss from Validation Access

We observe that $>$90% of validation accesses for Kilo TM hit in the L2 cache for all benchmarks with transactional execution limited to two warps per core. This also applies to most benchmarks with unlimited concurrent transactions, but for

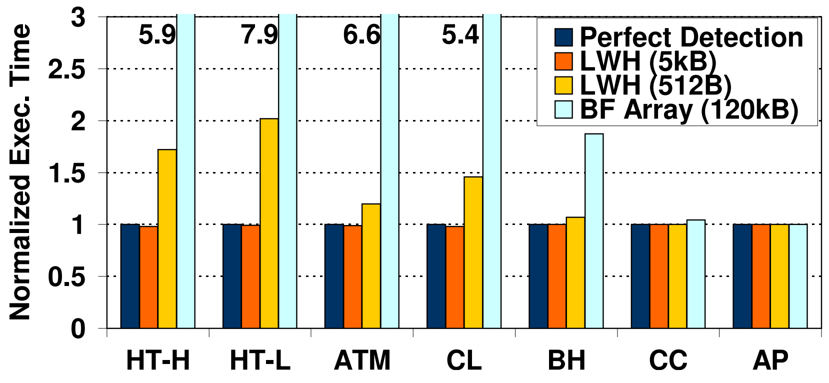


Figure 4.11: Sensitivity to hazard detection mechanism. LWH = Last Writer History (Section 4.2.5)

HT-L, ATM and CL, the cache hit rate for validation access is lower (70% for HT-L, 46% for ATM and 62% for CL). These extra accesses are easily handled by the GPU memory subsystem. In a sensitivity study with idealized validation accesses that always hit in the L2 cache, performance of ATM and CL improves only by 11% and 17%, respectively. Other benchmarks (including HT-L) are insensitive to this change. In this study, Kilo TM employs LWH units that detect hazards perfectly. About 50% of the validation-induced L2 cache accesses in CL are pending hits. In ATM, the extra L2 cache misses improve the row-hit rate in the open-row, out-of-order DRAM controller, increasing the bandwidth efficiency by 5%. The improved efficiency partly compensates the penalty from validation-induced DRAM accesses. In HT-L, these L2 cache misses increase the DRAM bandwidth utilization by 5% and do not impact performance. This ability to handle extra memory accesses in GPUs shows why value-based conflict detection is a viable solution for supporting TM on GPUs.

Hazard Detection Sensitivity

We explored the performance of Kilo TM with different hazard detection mechanisms. In Figure 4.11, we compared two versions of the LWH mechanism described in Section 4.2.5 and an additional mechanism based on a bloom filter array. The first 5kB LWH consists of a 512-entry, 4-way set-associative lookup table (3kB) and a 1024-bucket bloom filter (2kB) split into 4 separate sub-arrays, each array indexed by a unique H_3 hash function (similar to the parallel bloom signature described by Sanchez et al.[137] and Ceze et al. [26, 137]). The second 512B LWH configuration consists of a 64-entry lookup table and a 64-bucket bloom filter. A second detection mechanism, bloom filter array (BF Array), encodes the read-set and write-set of each transaction into two 512-bit signatures in the commit unit. Each signature consists of 4 sub-signatures with each indexed by a unique H_3 hash function. Incoming read/write accesses check for conflicts against all signatures in the commit unit in parallel.

Kilo TM with 5kB LWH unit performs almost identically to perfect hazard detection. The 512B LWH reduces the storage by 10$\times$ but increases execution time by 36% on average. Despite taking 24$\times$ more storage than the LWH unit (120kB vs. 5kB per commit unit), BF Array slows down Kilo TM by up to 7.9$\times$ (4.3$\times$ on average). The performance gap between the LWH unit and BF Array demonstrates how leveraging a pre-defined commit order to prioritize storage can lead to significantly more effective design than a design that dedicates the same amount of resource to represent the write-set of each transaction. The lookup table plus recency bloom filter design in a LWH unit dedicates extra resources to ensure that an unnecessarily revalidating transaction and its false writers are far apart in the commit unit pipeline, minimizing the stalling at the Validation Wait stage.

4.4.4 Implementation Complexity of Kilo TM

In each SIMT core, Kilo TM implementation involves extending the SIMT stack to support transactions, employing concurrency control, and adding a transaction log unit. Even though each transaction log unit manages 1000s of transactions, most of the bookkeeping is amortized across the warp. For example, threads in the same warp have the same read-log and write-log sizes, and they always have consecutive commit IDs. The L1 data cache stores the read-/write-logs. Evicted entries are written back to L2/DRAM. We believe the area overhead of a transaction log unit is negligible.

The area overhead of a commit unit consists of the storage required for the LWH unit, the entries in the ring buffer, and the read- and write-set storage buffers for each entry. Section 4.4.3 showed that a 5kB LWH unit is sufficient. Each

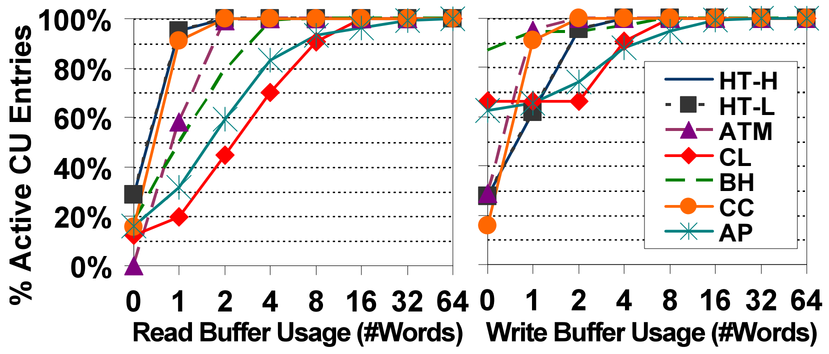


Figure 4.12: Buffer usage in active commit unit (CU) entries.

commit unit ring buffer entry occupies 10 bytes for the status, RCID and YCID fields, and pointers to a shared pool of the read- and write-set buffers. The area required for the read- and write-set buffers is a product of the size of each buffer and the number of buffers present. Section 4.4.4 examines how large each fixed-size buffer should be to limit buffer overflow. Section 4.4.4 examines how many of these fixed-size buffers are required concurrently.

Read-Set/Write-Set Buffer Capacity

Figure 4.12 shows the cumulative distribution of the read- and write-set buffer usage for the active ring buffer entries in the commit units. The distributions show that an 8-word (64 Bytes) read-set buffer and an 8-word write-set buffer can serve >90% of the commit unit ring buffer entries. If the read-set or the write-set buffer overflows (a rare event), the penalty involves resending the read- and write-log. We leave performance evaluation with finite-sized read- and write-set buffers as future work.

Commit Unit Capacity

Conservatively assigning each commit unit ring buffer entry with a dedicated 8-word read-set buffer and a dedicated 8-word write-set buffer can introduce significant storage overhead for the commit units (as explained later in this section). We observed that not all commit unit ring buffer entries will need read- and write-set buffers; in many case, the buffers are not used at all, or they are only needed at cer-

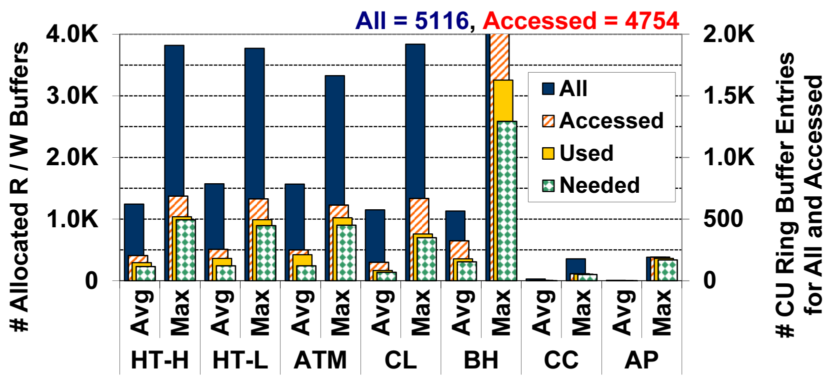


Figure 4.13: Number of in-flight, allocated read and write buffers for different buffer allocation schemes. All ⊇ Accessed ⊇ Used ⊇ Needed. Y-axis on right shows number of commit unit ring buffer entries with buffers allocated for All and Accessed.

tain phase of the validation and commit of a transaction. Dynamically allocating these buffers from a shared pool reduces the area overhead of buffers. Figure 4.13 shows the average and maximum number of read- and write-set buffers required in-flight throughout the execution of each TM application for 4 different buffer allocation schemes. The *All* and *Accessed* allocation schemes allocate fixed-size read- and write- buffers for a commit unit ring buffer entry when it is created, and deallocate the buffers when the entry retires. *All* allocates the two buffers for all active entries in the ring buffer, while *Accessed* only allocates buffers for the entries whose transaction has accessed this memory partition. The number of ring buffer entries in the *All* and *Accessed* schemes is given by the right Y-axis in Figure 4.13. The *Used* allocation scheme improves upon *Accessed* by allowing a single fixed-size read- or write-set buffer to be allocated if the transaction has an empty write-set or read-set buffer for this memory partition, respectively. *Needed* further improves upon *Used* by allowing buffers to be deallocated before the ring buffer entry retires. Instead of deallocating the buffers after all earlier transactions has retired, the buffers are deallocated as soon as a transaction fails. Also, the read buffers of a transaction are deallocated as soon as it has passed validation. In Figure 4.13, the number of concurrent transactions is limited to two warps per core (1920 in total) via concurrency control, and there is no capacity limit on the number of ring buffer entries.

HT-H, HT-L, ATM, CL and BH use $\sim$700 ring buffer entries on average (*All*). BH's maximum number of entries exceeds the concurrent transaction limit because it has many read-only transactions. The SIMT core considers a read-only transaction to be done when it receives the local outcome reply from commit units, allowing the next waiting transaction to proceed before the corresponding commit unit entries are retired. We observed that the number of in-flight entries exceeded the concurrent transaction limit of 1920 for only $<1\%$ of the execution time for BH. CC and AP have significantly fewer concurrent transactions and therefore require fewer entries. A commit unit with 1920 entries with two 64B buffers for all entries (*All*) would require 240kB for read- and write-set storage, and 19kB for the ring buffer storage (10B per ring buffer entry). The *Accessed*, *Used* and *Needed* optimizations reduce the storage requirement for buffers. To serve most of the *Needed* buffer usage, $\sim$500 buffers per commit unit (32kB per unit, 256kB for the whole GPU) are enough. The rare worst case can be handled by deferring validation/commit via the credit-based allocation mechanism described in Section 4.2.5. We leave the performance evaluation of this allocation mechanism as future work. Deferring validation/commit of a transaction when its allocation fails reduces the commit parallelism in Kilo TM; however, we expect the performance impact from these deferrals to be minimal, given how rarely they occur.

Area Estimation

We assume that both the 32kB read- and write-set buffer pool and the 19kB ring buffer have 4 banks of SRAM arrays. The 5kB last writer history unit (Section 4.4.3) consists of a 3kB lookup table and a 2kB bloom filter, each an SRAM array. Using CACTI 5.3 [144], we estimate the area of each commit unit (the aggregate area of these arrays) to be $0.40mm^2$ in a 40nm technology. NVIDIA Fermi GPU features 6 memory partitions [121], so implementing the commit units on Fermi architecture requires an area of $2.41mm^2$. This is just 0.5% of Fermi's $520mm^2$ die area.

4.5 Summary

In this chapter, we proposed the use of transactional memory for GPU computing. Transactions can simplify parallel programming by making it easier to reason about parallelism. This becomes more important as the number of threads increases and as more software is ported to take advantage of GPUs' better peak performance and power efficiency. Compared to lock-based programming, TM simplifies the porting/creation of applications that require data synchronization on GPUs. Specifically, TM is a better fit to the current GPU programming models. The isolation property of TM is similar to how GPU threads are exposed in the programming model. The application specifies as many transactions as it can, the TM system attempts to execute them in parallel, but transactions can run in isolation. The GPU hardware can be designed to automatically handle interactions between data synchronization and the SIMT stack, solving an obstacle that prevented fine-grained data synchronization from being widely used in GPU applications. Furthermore, TM frees the programmer from deadlock concerns as they rework the algorithms and data structures to optimize the performance of their application.

Kilo TM is a novel HTM system scalable to 1000s of concurrent transactions. It uses value-based conflict detection to offer weak isolation, avoid the need for coherence, reduce metadata overheads, support unbounded transactions, and detect conflicts at the granularity of individual words. We describe a scalable parallel commit protocol and the changes to a SIMT hardware organization required to support transactions. Kilo TM uses a novel speculative validation mechanism to improve the validation and commit parallelism for non-conflicting transactions. By design, it favors applications with low contention transactions. We have evaluated Kilo TM with a set of TM-enhanced GPU applications with various degrees of exposed parallelism (the granularity of the decomposition of work into threads) and contention. We find that applications with low exposed parallelism (e.g., AP) perform poorly on the GPU regardless of the data synchronization mechanism used. We argue that these applications can be further parallelized more easily with TM. Our evaluation suggests that Kilo TM performs well (relative to fine-grained locking) on applications with low contention and high exposed parallelism (HT-L, ATM, CC). Kilo TM performs poorly (relative to fine-grained locking) on applications with high contention and high exposed parallelism (HT-H, CL, BH). For these applications, limiting transaction concurrency lowers contention, and improves their performance with Kilo TM. The programmer can lower contention in their application via performance tuning, identifying transactions with high contention and reworking the code to reduce contention [174]. Applications with contention varying during execution (e.g., BH) may benefit from more dynamic mechanisms that control the transaction concurrency according to the current level of contention [15, 171].

Overall, our evaluation shows Kilo TM captures 59% of fine-grained locking performance and is $128\times$ faster than executing transactions serially on the GPU. Our evaluation with an idealized TM system indicates that TM on GPU can perform as well as fine-grained locking. These results motivate the need for TM on GPUs and the need for novel TM systems, like Kilo TM, that better address the challenges in this new domain.

The next two chapters in this dissertation address two pending issues with Kilo TM: correctness and energy-efficiency. The novel design of Kilo TM and its use of value-based conflict detection have lead to concerns regarding its correctness. In Chapter 5, we will present a semi-formal proof showing that value-based conflict detection can tolerate the ABA-problem and more generally that the implementation of Kilo TM presented in this chapter satisfies conflict serializability. The use of value-based conflict detection in Kilo TM has also lead to concern regarding its energy efficiency. Chapter 6 measures and characterizes the energy overhead of Kilo TM over fine-grained locking, and proposes two optimizations to significantly reduce the overhead.

Chapter 5

Kilo TM Correctness Discussion

In chapter 4, we have proposed Kilo TM, a hardware transactional memory (TM) system designed for GPU architectures. In Kilo TM, each transaction detects the existence of conflicts with other transactions via *value-based conflict detection* [37, 124]. With value-based conflict detection, each transaction buffers its writes to memory in a write-log and saves the values of its reads from memory in a read-log during execution. Upon its completion, the transaction compares the saved values of its read-set with the latest values in memory before it commits. We refer this comparison as *validation*. Any difference between the saved value and the latest value in memory indicates the existence of a conflict.

A general concern for the correctness of value-based conflict detection is the possibility of subtle bugs due to the ABA problem [109]. Examples of ABA problems have been found for published non-blocking algorithms [110, 151, 161]. These generally result from an implicit assumption that atomicity of a high-level operation on a concurrent data structure can be inferred as long as the value of a guard variable is the same after a sequence of low-level instructions (that implements the operation). This fallacy results in subtle bugs [109].

In this chapter, we address this concern with a semi-formal proof, showing that value-based conflict detection can tolerate the ABA problem. A version of this proof is available online [58]. Like NOrec [37], we can create a logical order for Kilo TM in which validation and commit of each transaction are indivisible – successfully committed transactions in Kilo TM appear atomic to each other. We summarize the insights behind the proof as described below.

Consider a TM system with address-based conflict detection (full knowledge of which locations have been modified by other transactions). If any of the locations read by a transaction have been changed and subsequently restored to their original value since the transaction originally read them, aborting the transaction and rerunning it instantaneously with the updated memory would yield the same result (same addresses and values in its write-set). This situation summarizes the behavior of a transaction in Kilo TM that has passed value-based conflict detection, where other conflicting transactions have changed and then restored values at memory locations that belong to the transaction's read-set during its execution. Committing the transaction directly without rerunning it effectively serializes it behind the last committed transaction. This effective serialization allows Kilo TM to tolerate the ABA problem. This requires the transaction to validate its read-set against a consistent view of memory (i.e. the conflict detection is not comparing values with a partially committed transaction). The transaction should also commit its write-set immediately after validation, before other transactions update locations in the transaction's write-set.

Kilo TM achieves this *logical indivisibility* between validation and commit by ordering transactions with their commit IDs. In Kilo TM, each transaction T_X is given a unique commit ID prior to validation and commit, and this ID defines the *commit order* of T_X. T_X will obtain a new commit ID for each execution attempt (i.e. a new ID is assigned every time T_X was aborted). Given two transactions T_X and T_Y with commit ID X and Y, where $X < Y$, Kilo TM's implementation guarantees the following partial ordering:

- Validation of each word w by T_Y always happens after any write to w by T_X.
- Any write to w by T_Y always happens after any write to w by T_X.
- Validation of w by T_X always happens before any write to w by T_Y.

These guarantees order validations and writes at each memory location (word) in ascending commit order.

With this per-word order, it can be shown that transactions committed by Kilo TM satisfy *conflict serializability* [163]: All conflict relations produced from the accesses at each word will obey the commit order. Hence, a conflict graph created from these conflict relations is always acyclic.

Conflict serializability implies a logical timeline in which validation and commit of each transaction are indivisible (performed without being interleaved by other transactions). The logical timeline also implies that validation of each transaction always observes a consistent view of memory. Therefore, by proving conflict serializability for Kilo TM, we can show that it can tolerate the ABA problem.

The remainder of this chapter contains a proof of Kilo TM correctness. The proof begins in Section 5.1, by defining a memory value-location framework that is used to represent the values observed and produced by each transaction. The framework is then used to model a general ABA problem scenario specific to TM systems that employ value-based conflict detection. The proof for Theorem 1 uses this framework to show that such TM systems can tolerate the ABA problem, under the assumptions that validation and commit of each transaction are indivisible, and validation is done on a consistent view of memory. To show that Kilo TM satisfies these assumptions, Section 5.2 shows how the implementation of Kilo TM provides a set of partial orderings, Claim 1 to 6. In Section 5.4, these claims are used to prove Lemma 1 and Lemma 2, which together show that accesses from transactions to each word are ordered by ascending commit IDs. In Section 5.5, from this per-word ordering, Lemma 3 shows that validation performed by each transaction accesses a consistent view of memory. In Section 5.6, Lemma 4 then shows that accesses from transactions satisfy *conflict serializability* [163], which means there exists a logical timeline in which validation and commit of each transaction is indivisible. Finally, in Section 5.7, Theorem 2 combines Lemma 3 and Lemma 4 to show that Kilo TM satisfies the assumptions for Theorem 1. Therefore, Kilo TM can tolerate the ABA problem.

5.1 Memory Value-Location Framework for the ABA Problem

Transactional memory (TM) provides the programmer with the abstraction that transactions are executed in some serialization order. In this serialization order, transactions are executed serially one after another. Each transaction T advances the memory state from the original state observed by T, M^O, to a new state M^N. We denote this transition from M^O to M^N with $M^O \rightarrow M^N$.

Each memory state is defined by the value at every location in the entire memory space M (i.e. a mapping from each address in M to its value). A memory state M^X is equivalent to another memory state M^Y if the value at each location in M^X is equal to the value at the corresponding location in M^Y. For the rest of this discussion, we use subscripts to denote subsets of locations within a memory space and superscripts to denote different memory states (i.e. values at a set of memory locations). For example, M_R is a subset of memory locations (not values) of the memory space M, M^L denotes the values in every location in the entire memory space M, and M_R^L is the set of address-value pairs for the memory locations (addresses) in M_R with values from the memory state M^L. We call M_R^L a partial memory state.

For each transaction T_X, the entire memory space M is divided into the following subsets:

$M_{R,X}$ =	The memory locations read by T_X as it executes (Read-Set).
$M_{W,X}$ =	The memory locations written by T_X when it commits (Write-Set).
$M_{A,X}$ =	$M_{R,X} \cup M_{W,X}$ = The memory locations that are accessed (either read or written) by T_X.
$M_{I,X}$ =	$M - M_{A,X}$ = Memory locations that are ignored (neither read nor written) by T_X.
$M_{RW,X}$ =	$M_{R,X} \cap M_{W,X}$ = Memory locations in T_X's Read-Set that are also part of its Write-Set.
$M_{RO,X}$ =	$M_{R,X} - M_{RW,X}$ = Memory locations that are only read by T_X.
$M_{WO,X}$ =	$M_{W,X} - M_{RW,X}$ = Memory locations that are only written by T_X.

Notice $M_{I,X}$, $M_{RW,X}$, $M_{RO,X}$ and $M_{WO,X}$ are all disjoint sets, and $M = M_{I,X} \cup M_{RW,X} \cup M_{RO,X} \cup M_{WO,X}$.

During execution, each transaction T_X observes the partial memory state $M_{R,X}^O = M_{RO,X}^O \cup M_{RW,X}^O$[7], and writes to addresses in $M_{W,X}$, producing the new partial memory state $M_{W,X}^N = M_{WO,X}^N \cup M_{RW,X}^N$. In the mean time, other transactions may have

[7]T_X observes $M_{R,X}^O$ as long as no other transaction modifies the locations in $M_{R,X}$ while T_X executes. This holds for all transactions that successfully commit. See Section 5.1.4 for the discussion on how Kilo TM detects and handles transactions with inconsistent view of memory. We assume that transactions are weakly isolated [16] and ignore non-transactional writes in this work.

committed, advancing the latest global memory states to $M^L = M^L_{R,X} \cup M^L_{W,X} \cup M^L_{I,X}$. With value-based conflict detection, the transaction checks to see if $M^L_{R,X} = M^O_{R,X}$ before it commits. If the two partial memory states are indeed equivalent, T_X commits by advancing the partial memory state in its write-set from $M^L_{W,X}$ to $M^N_{W,X}$. This appends the serialization order with a new transition ($M^L \rightarrow M^N$).

5.1.1 ABA Problem

The ABA problem manifests in non-blocking algorithms, where multiple threads may operate on a data structure simultaneously, and the atomicity of each operation is presumed to be guaranteed via success of one or more atomicCAS operations. Many non-blocking algorithms rely on the following assumption: If the value of a guarding variable has not been modified since it was last read, then no other threads have modified the data structure, and thus this thread has performed the current operation in isolation. This assumption ignores the possibility that several other operations may have occurred in between, first modifying the guard variable to other values, then restoring the original value before the current thread uses atomicCAS to check the variable's value. The fallacy in this assumption is how the ABA problem manifests in various non-blocking algorithms, resulting in subtle bugs that are hard to detect [46, 109].

5.1.2 Potential ABA Problem in Transactional Memory

In the context of value-based conflict detection employed in Kilo TM, we consider the potential for ABA problems in the following form. A set of transactions T_{ABA} = $\{T_1, \ldots T_L\}$ have committed in between time t_1, when transaction T_X first started to read its read-set $M_{R,X}$ (observing state $M^O_{R,X}$ from M^O, the memory state before any transaction in T_{ABA} commits), and the time t_2, when T_X is validating its read-set against the latest global memory state. We assume that transactions are weakly isolated [16] and ignore non-transactional writes in this discussion. Transactions in T_{ABA} advance the global memory state from M^O through a series of memory states and eventually to M^L. M^L is not necessarily equivalent to M^O, but the part that belongs to T_X's read-set is equivalent: $M^O_{R,X} = M^L_{R,X}$. Value-based conflict detection performed by T_X will observe that its read-set has not been changed, and

T_X "assumes" no conflicting transaction has committed between t_1 and t_2 (i.e. the values in $M_{R,X}$ appear to have never been modified in this window). Subsequently, T_X commits by advancing $M^L_{W,X}$ to $M^N_{W,X}$, whereas the intended transition (one that would have occurred if T_X has executed in isolation without the presence of transactions in T_{ABA}) is from $M^O_{W,X}$ to $M^N_{W,X}$. This intended transition ($M^O \rightarrow M^N$) violates the existing serialization order because the latest memory state is M^L. A TM system with address-based conflict detection will regard this as a conflict. In such a system, T_X will be restarted to resolve this conflict. However, we will show that this restart is not needed.

5.1.3 Tolerance to the ABA Problem

The following proof shows that committing T_X directly in the situation described in Section 5.1.2 will result in the same serialization order in which T_X detects the conflict and reruns itself starting at time t_2. In other words, TM systems employing value-based conflict detection can tolerate the ABA problem by yielding the same memory state transition as TM systems employing address-based conflict detection.

Theorem 1. *Directly committing T_X in the situation described in Section 5.1.2 will result in the same serialization order in which T_X detects the conflict and reruns itself instantly starting at time t_2.*

Proof. Assume that the TM system employs a separate mechanism other than value-based conflict detection that is not prone to the ABA problem. T_X, upon detecting the conflict at time t_2, aborts itself and restarts immediately. Let T^1_X be this new instance of T_X. T^1_X will observe its read-set from M^L. Since $M^L_{R,X} = M^O_{R,X}$, T^1_X will produce the identical write-set partial memory state $M^N_{W,X}$ as T_X. If T^1_X finishes executing instantaneously with no other transactions committing in between, its commit will advance the partial memory state in $M_{W,X}$ from $M^L_{W,X}$ to $M^N_{W,X}$ and T^1_X will transition the global memory state from M^L to $M^N = (M^N_{I,X} \cup M^N_{RO,X} \cup M^N_{W,X})$. Values in $(M_{I,X} \cup M_{RO,X})$ remain unchanged between M^L and M^N (i.e. $M^N_{I,X} = M^L_{I,X}$ and $M^N_{RO,X} = M^L_{RO,X}$). Committing T_X at time t_2 would have produced the same transition ($M^L \rightarrow M^N$): $M^L_{W,X}$ is advanced to $M^N_{W,X}$, while values in $(M_{I,X} \cup M_{RO,X})$ remain unchanged. Therefore, as committing either T_X or T^1_X results in the same transition, the programmer cannot discern between the two instances of execution. □

The above proof made two assumptions:

Assumption 1. *T_X commits immediately after value-based conflict detection, such that no other transactions can commit in between to advance the memory state away from M^L.*

Assumption 2. *The value-based conflict detection performed by T_X is comparing the original read-set state $M^O_{R,X}$ against a consistent view of the global memory state $M^L_{R,X}$. Here consistent view means that during conflict detection, $M^L_{R,X}$ is not advanced to another memory state by the commit of another transaction (i.e. $M^L_{R,X}$ is the part of the memory state that exists in between the commits of two transactions).*

We proceed to demonstrate that Kilo TM, despite its distributed design, satisfies both assumptions.

5.1.4 Inconsistent Read-Set

While T_X can possibly have observed an inconsistent view of memory (e.g. partially committed states from transactions in T_{ABA}) during its execution, Theorem 1 holds as long as T_X's observed read-set equals to $M^L_{R,X}$. T_X may also observe inconsistent values from a single memory location. In Kilo TM, each transactional load appends the value read from global memory into a linear read-log (if it is not accessing the transaction's write-set) [57]. The inconsistent values observed from a single memory location by T_X will create multiple read-log entries that contain different values for a single location. During value-based conflict detection, only one of the values will match with the one in $M^L_{R,X}$. The mismatched entry will cause T_X's validation to fail (subsequently aborting T_X). T_X may enter an infinite loop due to the inconsistent view of memory. To ensure that T_X is eventually aborted, Kilo TM employs a watchdog timer to trigger a validation for T_X [57].

5.2 Transaction Components in Kilo TM

In Kilo TM, each transaction, T_X, is comprised of the following sequence of operations:

$$T_X = R(r_1)\dots R(r_m)\ Rv(r_1)\dots Rv(r_m)\ W(w_1)\dots W(w_n)$$

- $R(r_1)$ is a read operation from word r_1, and $M_{R,X} = \{r_1 \dots r_m\}$ is the read-set of T_X.

- $Rv(r_1)$ is a validation operation on word r_1, ensuring that the value obtained by R(r1) equals the value in global memory. (This is the operation that performs value-based conflict detection.)

- $W(w_1)$ is a write operation to word w_1, and $M_{W,X} = \{w_1 \dots w_n\}$ is the write-set of T_X. The write operation is performed only while T_X commits, and it updates the value of w_1 in global memory.

When there are multiple transactions involved, the notation $Rv(T_X, w)$ denotes the validation operation on word w for transaction T_X; whereas notation $Rv(T_X)$ denotes all of the validation operations required by T_X.

T_X is executed on a single SIMT core. The core interacts with a set of commit units (one in each memory partition) to validate and commit T_X. The following messages are sent between the commit units and the core that executes T_X:

- $V_k(T_X)$ = (pass/fail) is the validation outcome for T_X at commit unit k. $V_k(T_X)$ = pass if validation operations performed on each word w contained in commit unit k for T_X, where $w \in M_{R,X}$, all succeed; otherwise, $V_k(T_X)$ = fail. This message is sent from each commit unit k to the core running T_X.

- $F(T_X)$ = (pass/fail) is the final outcome for T_X. $F(T_X)$ = pass if all validation outcomes received by the core $V_k(T_X)$ = pass; otherwise, $F(T_X)$ = fail. This message is sent from the core to each commit unit. Write operations $W(T_X)$ are only performed if $F(T_X)$ = pass.

5.2.1 Commit ID and Commit Order

Prior to validation and commit, a transaction T is given a unique commit ID. This ID defines the *commit order* of T. A transaction with lower commit ID has an earlier commit order than those with a higher commit ID. Namely, given transactions T_X and T_Y with commit ID X and Y respectively, $X < Y \iff T_X <_t T_Y$. Here $<_t$ denotes the commit order. Each transaction will obtain a new commit ID for each execution attempt (i.e. a new ID is assigned every time the transaction was aborted).

The commit order limits how a transaction may appear in the serialization order. Let M^L be the latest memory state, and T_X and T_Y be two transactions, with $T_X <_t T_Y$, that are ready to commit. Only one of the following can happen:

- Both T_X and T_Y commit, resulting in transitions $M^L \to M^{X1} \to M^{Y2}$, where M^{X1} is the memory state from M^L after T_X has committed and M^{Y2} is the memory state from M^{X1} after T_Y has committed.
- Only T_X commits, resulting in transition $M^L \to M^{X1}$, where M^{X1} is the memory state from M^L after T_X has committed.
- Only T_Y commits, resulting in transition $M^L \to M^{Y1}$, where M^{Y1} is the memory state from M^L after T_Y has committed.
- Neither T_X nor T_Y commits, resulting in no transition.

Notice that transitions $M^L \to M^{Y1} \to M^{X2}$ (M^{X2} is the memory state from M^{Y1} after T_X has committed) is not allowed. This restriction is enforced by Claim 2 below. This allows Kilo TM to handle transactions with write-after-write conflicts by ordering their commits without synchronizing among different commit units (instead of aborting one of them). RingSTM [149] also uses this policy to handle write-after-write conflicts.

5.3 Partial Orderings Provided by Kilo TM

Kilo TM's implementation provides a set of partial orderings that we present in the following claims. The following discussions on the validity of these claims assume that the reader is familiar with the implementation of Kilo TM. An in-depth description of Kilo TM's implementation can be found in Section 4.2.5 of Chapter 4. These claims will be used to show that Kilo TM satisfies both Assumption 1 and Assumption 2 required for ABA problem tolerance (Theorem 1). We denote these partial orderings with $<_P$. Given two events/operations A and B, $A <_P B$ means A *happens before* B in real time.

Claim 1. *At each commit unit, given transactions T_X and T_Y, where $T_X <_t T_Y$, a write to a memory location w performed by a transaction T_X always happens before validation of the same location w performed by a transaction T_Y . In our notation, $W(T_X, w) <_P Rv(T_Y, w)$.*

Proof. In Kilo TM's implementation, transactions always perform hazard detection in commit order. Each commit unit can speculatively validate each memory location in the read-set of T_Y ($M_{R,Y}$) as the corresponding read-log entry arrives at the unit. Later in hazard detection, if the unit detects (via address-based conflict detection) that T_X is writing to any part of $M_{R,Y}$, the unit will revalidate the entire read-set of T_Y after T_X has finished committing. □

Claim 2. *Let T_X and T_Y be transactions with $T_X <_t T_Y$. If a memory location w is in the write-sets of both T_X and T_Y , writes to w by transaction T_X always happen before writes to w by transaction T_Y. Namely, $W(T_X, w) <_P W(T_Y, w)$.*

Proof. This is enforced at the commit stage in each commit unit. This ordering is guaranteed by issuing the write operations of each passed transaction in ascending commit order. The GPU memory subsystem in our architecture can reorder accesses to different locations to optimize for bandwidth, but it maintains the ordering of accesses to the same location. □

Claim 3. *At each commit unit, the write operations for a transaction T_X are only commenced after the commit unit has received the final outcome $F(T_X)$ of T_X from the core. Namely, $F(T_X) <_P W(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. □

Claim 4. *The transaction will not send out the final outcome $F(T_X)$ to commit units until it has received validation outcomes $V_k(T_X)$ from all commit units for the* *transaction. For each commit unit k that contains any location in the read-set of T_X, $V_k(T_X) <_P F(T_X)$.*

Proof. This is enforced by the Kilo TM implementation at each SIMT core. □

Claim 5. *At each commit unit k that contains any location in the read-set of T_X, the validation outcome $V_k(T_X)$ is sent after all validation operations are done. Namely, $Rv(T_X) <_P V_k(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. □

Claim 6. *At each commit unit, the write operations for a transaction T_X are only commenced after the commit unit has received the final outcomes $F(T_Y)$ from all transactions T_Y with earlier commit order if the commit unit contains any location in either read-set or write-set of T_Y ($M_{A,Y}$). I.e. For all transaction T_Y with ($T_Y <_t T_X$) and the commit unit that contains any location in $M_{A,Y}$, $F(T_Y) <_P W(T_X)$.*

Proof. This is enforced at the finalizing outcome stage in each commit unit. At each commit unit, the commit unit entry that corresponds to a transaction T_Y waits for the final outcome $F(T_Y)$ before proceeding to the commit stage. T_Y stalling at the finalizing outcome stage will forbid any transaction with a younger commit ID (e.g. T_X) to proceed to the next stage, even after $F(T_X)$ has been received by the unit. Since write operations are only issued in commit stage, this stalling behavior enforces Claim 6. □

5.4 Per-Word Access Ordering

Lemma 1 and 2 illustrate how Kilo TM orders accesses (validations and writes) to each memory location (word) in ascending commit order. This ordering is used in Lemma 3 to prove that all validation operations for a given transaction T_X are comparing against a consistent view of $M^L_{R,X}$ (Assumption 2).

Lemma 1. *Let T_Y be a transaction. For each memory location w in $M_{RW,Y}$, validation operation(s) to w by T_Y always happen before write operation(s) to w by T_Y.*

Proof. Let t_C be the time when T_Y starts sending the final outcome $F(T_Y)$ to each commit unit from the core. By Claim 5, all the validation operations of w by T_Y ($Rv(T_Y,w)$) at each commit unit k have to happen before the unit replies with the validation outcome $V_k(T_Y)$ back to the core. By Claim 4, all of these outcomes have to arrive at the core before T_Y sends out $F(T_Y)$, i.e. before t_C. The same commit unit k will receive $F(T_Y)$ at a time $t_K > t_C$. By Claim 3, any write operation to w in $M_{RW,Y}$ by T_Y ($W(T_Y,w)$) has to happen after t_K.

Putting it all together, all validation operations to w in $M_{RW,Y}$ by T_Y have to occur before t_C, which is before all write operations to w by T_Y. I.e. $Rv(T_Y,w) <_P t_C <_P t_K <_P W(T_Y,w) \Rightarrow Rv(T_Y,w) <_P W(T_Y,w)$. □

Lemma 2. *Let T_X and T_Y be transactions with $T_X <_t T_Y$. For each memory location w, operations (validation/write) to w by T_X always happen before operations to w by T_Y, except when both operations are validation.*

Proof. This can be broken down into three separate orderings:

O1. $W(T_X,w) <_P Rv(T_Y,w)$

O2. $W(T_X,w) <_P W(T_Y,w)$

O3. $Rv(T_X,w) <_P W(T_Y,w)$

The first two orderings follow directly from Claim 1 and Claim 2 respectively. The final ordering follows from Claim 4-6: At each commit unit k, the validation outcome of T_X, $V_k(T_X)$, is only sent after all its validation operations are done (Claim 5), and the final outcome $F(T_X)$ will only arrive after all validation outcomes have been received by the core (Claim 4). By Claim 6, the write operations of T_Y will not be issued until the commit unit has received $F(T_X)$. Putting it all together, let commit unit k be the unit containing location w, $Rv(T_X,w) <_P V_k(T_X) <_P F(T_X) <_P W(T_Y,w) \Rightarrow Rv(T_X,w) <_P W(T_Y,w)$. □

5.5 Validation Against a Consistent View of Memory

Lemma 3. *The value-based conflict detections (all validation operations) performed by T_Y compare the original read-set state $M^O_{R,Y}$ against a consistent view of a global memory state $M^L_{R,Y}$.*

Proof. Claim 2 (O2 in Lemma 2) specifies that each location in memory is written in ascending commit order. By O1 and O3 in Lemma 2, every validation by T_Y to a location w in $M_{R,Y}$ is performed after all transactions with earlier commit orders have written to w and before transactions with later commit orders write to w. The validation is also done before T_Y writes to w itself (by Lemma 1). Hence, each validation $Rv(T_Y, w)$ by T_Y will observe the value of w that is written by the transaction with the latest commit order before T_Y, which is the same value of w in the memory state right after T_{Y-1} commits ($M^L_{R,Y}$). Since this applies to the validation of every w in $M_{R,Y}$, the validation operations $Rv(T_Y, w)$ for all w in $M_{R,Y}$ are comparing against the same consistent view of global memory. □

Comment: This can be explained in a simpler way via the commit IDs. Each commit unit ensures that all validation operations of a transaction T_X with CID = X are reading from a memory state equivalent to the one right after T_{X-1} commits. Hence, T_X is validating against a consistent view of memory.

5.6 Logical Indivisibility of Validation and Commit

Kilo TM is designed to permit non-conflicting transactions to commit in parallel. This means that memory state M^L would likely be advanced to another memory state $M^{L'}$ by the commits of other transactions during the validation of T_X and before T_X can commit. This seems to violate Assumption 1. However, with the per-word operation ordering illustrated by Lemma 1 and 2, we can construct a logical timeline in which validation and commit of each transaction are indivisible.

To construct such a logical timeline, we first define the validation and write operations of a committed transaction T_X as a mini-transaction:

$C_X = Rv(r_1) \ldots Rv(r_m)\ W(w_1) \ldots W(w_n)$

Each transaction corresponds to a single mini-transaction. The commit order for each mini-transaction is the same as its transaction counterpart. We show that operations performed by these mini-transactions satisfy *conflict serializability* [163].

Lemma 4. *The sequence of operations performed by any arbitrary mini-transactions with Kilo TM satisfies conflict serializability.*

Proof. Let be C_X and C_Y be two mini-transactions with $C_X <_t C_Y$.

C_X has read-set $M_{R,X}$ and write-set $M_{W,X}$, and $M_{A,X} = M_{R,X} \cup M_{W,X}$.

C_Y has read-set $M_{R,Y}$ and write-set $M_{W,Y}$, and $M_{A,Y} = M_{R,Y} \cup M_{W,Y}$.

Each memory location w that is accessed by both transactions (i.e. w is in $(M_{A,X} \cap M_{A,Y})$) will have the operations ordered according to the ordering defined by Lemma 1 and Lemma 2. In all cases, the operations will produce the conflict relation (denoted by ordered pair $(Op(C_A,w), Op(C_B,w))$ below, see definition 3.12 in Weikum and Vossen [163]) that aligns with the commit order. Each conflict relation in turn creates directed edges for their corresponding mini-transactions in a conflict graph (denoted by ordered pair (C_A,C_B) below, see definition 3.15 in Weikum and Vossen [163]):

- $Rv(C_X,w)$ and $W(C_Y,w)$ are always ordered in $Rv(C_X,w) <_P W(C_Y,w)$, producing conflict relation $(Rv(C_X,w), W(C_Y,w))$ and conflict graph directed edge (C_X,C_Y).
- $W(C_X,w)$ and $W(C_Y,w)$ are always ordered in $W(C_X,w) <_P W(C_Y,w)$, producing conflict relation $(W(C_X,w), W(C_Y,w))$ and conflict graph directed edge (C_X,C_Y).
- $W(C_X,w)$ and $Rv(C_Y,w)$ are always ordered in $W(C_X,w) <_P Rv(C_Y,w)$, producing conflict relation $(W(C_X,w), Rv(C_Y,w))$ and conflict graph directed edge (C_X,C_Y).
- $Rv(C_X,w)$ and $Rv(C_Y,w)$ are freely ordered, but they produce no conflict relation.

Since every pair of mini-transactions has either no conflict, or produces conflict relation that aligns with the commit order, the conflict graph created from the conflict relations among all mini-transactions will not contain any cycle. Specifically, two transactions C_A and C_B with $C_A <_t C_B$ can never have a directed path from C_B to C_A in the conflict graph. Therefore, by Theorem 3.10 in Weikum and Vossen [163], any sequence of operations performed by the mini-transactions satisfies conflict serializability. $\square$

By Lemma 4 and the definition of conflict serializability (definition 3.14 in Weikum and Vossen [163]), we can imply that any sequence of operations performed by the mini-transactions with Kilo TM has a logically equivalent serial sequence. In this serial sequence, operations performed by each mini-transaction are not interleaved by those from other mini-transactions. This serial sequence forms a logical timeline in which validation and commit of each transaction are indivisible.

5.7 Tolerance to ABA Problem (Kilo TM)

With Lemma 3 and Lemma 4 proving how Kilo TM satisfies Assumption 2 and Assumption 1 for Theorem 1, Theorem 2 follows:

Theorem 2. *Kilo TM can tolerate the ABA problem.*

Proof. Lemma 3 and Lemma 4 show that Kilo TM satisfies both assumptions for Theorem 1. Hence, Kilo TM can tolerate the ABA problem. □

5.8 Summary

In this chapter, we presented a semi-formal proof to show that Kilo TM can tolerate the ABA problem. We first formalized the ABA problem in the context of a TM system with a memory value-location framework. We used this framework to show that in the presence of an ABA problem, committing the transaction directly produces the same memory state transition as aborting the transaction and rerunning it to completion instantaneously. This behavior holds as long as each transaction can commit immediately after validating its read-set. Hence, the rest of the proof shows that all transactions committed by Kilo TM satisfies conflict serializability – these transactions form a logical order in which the validation and commit of each transaction is logically indivisible. In other words, Kilo TM properly enforces atomicity and isolation among transactions.

Chapter 6

Energy Efficiency Optimizations for Kilo TM

In this chapter, we address concerns over the energy-efficiency of Kilo TM, our GPU TM system proposed in Chapter 4. In particular, we evaluate and analyze the performance and energy overhead of Kilo TM. The insights from this analysis lead to two distinct enhancements: *warp-level transaction management* (WarpTM) and *temporal conflict detection* (TCD). A version of this chapter has been published earlier [55].

Warp-level transaction management leverages the thread hierarchy in GPU programming models – the spatial locality among threads within a warp – to improve the efficiency of Kilo TM. In particular, WarpTM amortizes the control overhead of Kilo TM and boosts the utility of the GPU memory subsystem. These optimizations are only possible if conflicts within a warp can be resolved efficiency, and thus a low overhead intra-warp conflict resolution mechanism is crucial in maintaining the benefit from WarpTM. To this end, we propose a two-phase parallel intra-warp conflict resolution that resolves conflicts within a warp efficiently in parallel.

Temporal conflict detection is a low overhead mechanism that uses a set of globally synchronized on-chip timers to detect conflicts for read-only transactions. Once initialized, each of these on-chip timers runs locally in its microarchitecture module and does not communicate with other timers. This implicit synchronization without communication distinguishes TCD from existing timestamp-based con-

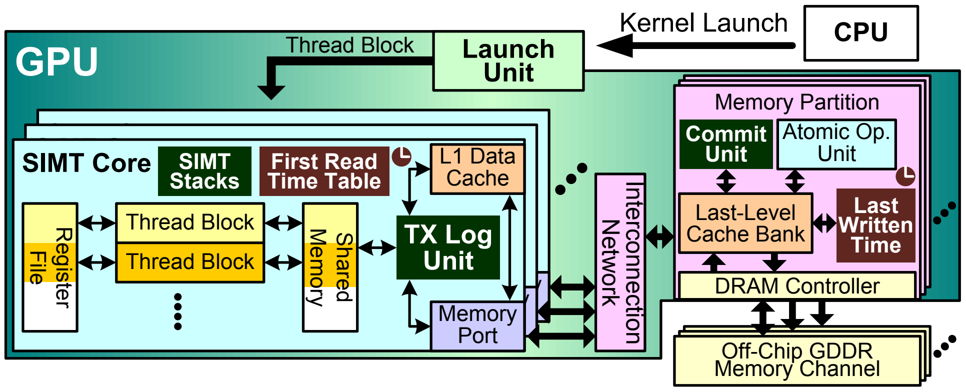


Figure 6.1: Enhanced Kilo TM implementation overview. TX Log Unit, Commit Unit added for Kilo TM. SIMT stack modified to support transactions (see Chapter 4). TX Log Unit extended to use shared memory for intra-warp conflict resolution. First Read Time Table, Last Written Time, and timers in SIMT Cores and Memory Partitions added for temporal conflict detection.

flict detections used in various software TM systems [37, 148, 167]. TCD uses timestamps captured from these timers to infer the order of the memory reads of a transaction with respect to updates from other transactions. Kilo TM incorporates TCD to detect conflict-free read-only transactions that can commit directly without value-based conflict detection. In doing so, it significantly reduces the memory bandwidth overhead for these transactions, which can account for 40% and 85% of the transactions in two of our GPU-TM applications.

Figure 6.1 shows the overall implementation of an enhanced Kilo TM that incorporates both WarpTM and TCD. The two enhancements complement each other to improve the overall performance of Kilo TM by 65% while reducing the energy consumption by 34%. This enhanced Kilo TM outperforms coarse-grained locking by 192× and achieves 66% of the performance of fine-grained locking with 34% energy overhead. More importantly, the enhancements allow applications with small, rarely-conflicting transactions to perform equal or better than their fine-grained lock versions. We believe that GPU applications using transactions can be incrementally optimized to reduce memory footprint and transaction conflicts to take advantage of this. Meanwhile the transaction semantics can maintain correctness at every step, providing a low-risk environment for exploring optimizations.

6.1 Performance and Energy Overhead of Kilo TM

Our evaluation shows that GPU TM applications running on a simulated NVIDIA Fermi GPU extended with our baseline Kilo TM only capture 40% of the performance of fine-grained locking (i.e., Kilo TM execution time is 1/0.4 = 2.5× slower), and consumes 2× the energy. We note this is lower than the 59% relative performance between Kilo TM and fine-grained locking in our previous evaluation in Chapter 4. The discrepancy is mainly contributed by the different core to memory ratio between the NVIDIA Fermi architecture and the cache-extended Quadro FX5800 architecture modeled in Chapter 4.

Our analysis has identified multiple sources of inefficiency in Kilo TM:

- While concurrency control can reduce the number of aborted transactions, the GPU TM application may contain phases of extremely high contention among transactions. The performance overhead of value-based conflict detection for these aborted transaction creates a bottleneck in Kilo TM.

- While transactions in Kilo TM are executed in the SIMT execution model, they validate and commit to global memory at scalar granularity. We opted for the scalar granularity based on our observation that threads in in GPU applications usually access adjacent words in memory. A significant overhead stems from the inherent mismatch between this scalar transaction management and the wide memory subsystem in GPUs designed to capture spatial locality. Although Kilo TM may validate and commit transactions in a warp as a group to exploit the spatial locality among threads – just like how current GPUs already coalesce non-transactional memory accesses into wider accesses, it may only do so for warps without *intra-warp conflicts*, or conflicts among transactions in the same warp. Extending the commit units in Kilo TM to detect and explicitly handle intra-warp conflicts in a distributed way can add complexity to the design.

- The protocol used to maintain the consistency between commit units in different memory partitions can introduce significant extra traffic in the on-chip interconnection network.

- Kilo TM has adopted value-based conflict detection for its lack of essential global metadata requirement. While it eliminates direct communication between transactions (a trait for scalability), we find that it incurs a significant energy overhead, even if most of the validation lookups hit at the L2 (last-level) cache.

In the following sections, we will present how warp-level transaction management and temporal conflict detection attempt to eliminate these sources of inefficiencies.

6.2 Warp-Level Transaction Management (WarpTM)

The GPU memory subsystem is designed to handle accesses with high spatial locality. The L2 cache bank in each memory partition can access a quarter of the cache block (32 Bytes) in a single cycle, and the accessed data is delivered through an interconnection network that can inject 32 bytes per cycle at each port. The use of wide cache ports and wide flit size matches well with the off-chip DRAM architecture, and delivers high bandwidth with relatively low control hardware overhead. GPU uses special coalescing logic to capture the spatial locality among scalar memory accesses from threads in the same warp.

Although Kilo TM executes transactions in the same warp in parallel, each transaction validates and commits individually. This scalar management simplifies the design of the commit units – conflicts between transactions within the same warp are handled just as conflicts between any two transactions in the system. However, this design simplification results in an inefficient utilization of the memory subsystem. Every cycle, each commit unit can only send one scalar, 4-Byte request (validation or memory writeback of a single 4-Byte word for a single transaction) to the L2 cache bank. This wastes at least 7/8 of the L2 cache bandwidth, creating a major energy overhead for Kilo TM.

The scalar management also introduces many extra protocol messages between the SIMT cores and the commit unit. Each transaction has to generate at least three messages: (1) a done-fill message to indicate that the entire read-set and write-set have arrived at the commit unit, (2) a response from the commit unit to relay the outcome of value-based conflict detection local to the unit, (3) a message broadcasting the overall transaction outcome to each commit unit. These extra protocol messages can significantly increase interconnection traffic.

Even though it is possible to improve the L2 cache bandwidth utility by extending each commit unit with an access combine buffer that opportunistically accumulates multiple scalar accesses and coalesces them into wider accesses, we believe a simpler alternative is to exploit the spatial locality that already exists in a warp. We call this Kilo TM extension *warp-level transaction management* (WarpTM). WarpTM uses a low-overhead *intra-warp conflict resolution* mechanism to detect and resolve all conflicts within a warp before validating and committing the warp via the commit units. The warp that is free of intra-warp conflicts can then be managed as a single entity, allowing various optimizations that boost the performance and efficiency of Kilo TM without introducing complex control logic.

The rest of this section describes the optimizations enabled by warp-level transaction management and the hardware modifications required to realize these optimizations. The implementation of intra-warp conflict resolution will be discussed in Section 6.3.

6.2.1 Optimizations Enabled by WarpTM

Without any potential conflicts within the warp, the commit unit can coalesce the validation and commit requests from all transactions within the warp into wider accesses to the L2 cache. It can also aggregate the protocol messages so that it is relaying the validation outcomes of the entire warp. The benefits from WarpTM can be categorized as follows.

Eliminate Futile Validation. By resolving conflicts within a warp prior to global commit, transactions that would have failed abort before generating any commit related traffic out of the SIMT core. This can reduce congestions at the commit units for workloads with high contention, improving their performance and energy usage. Figure 6.2 shows that conflicts between transactions within the same warp, *intra-warp conflicts*, rarely occur in most of our workloads. The exceptions, BH and AP, both feature a high contention period when many transactions are

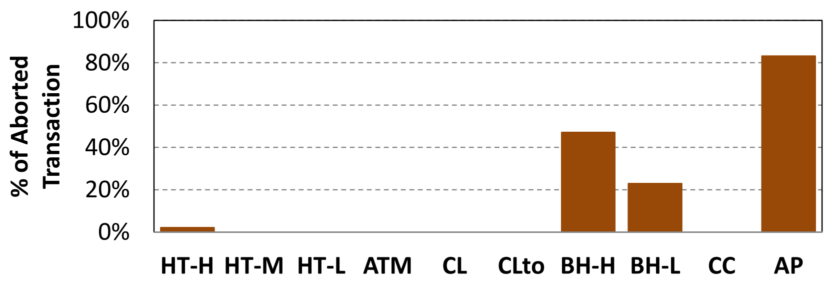


Figure 6.2: Transaction conflicts within a warp.

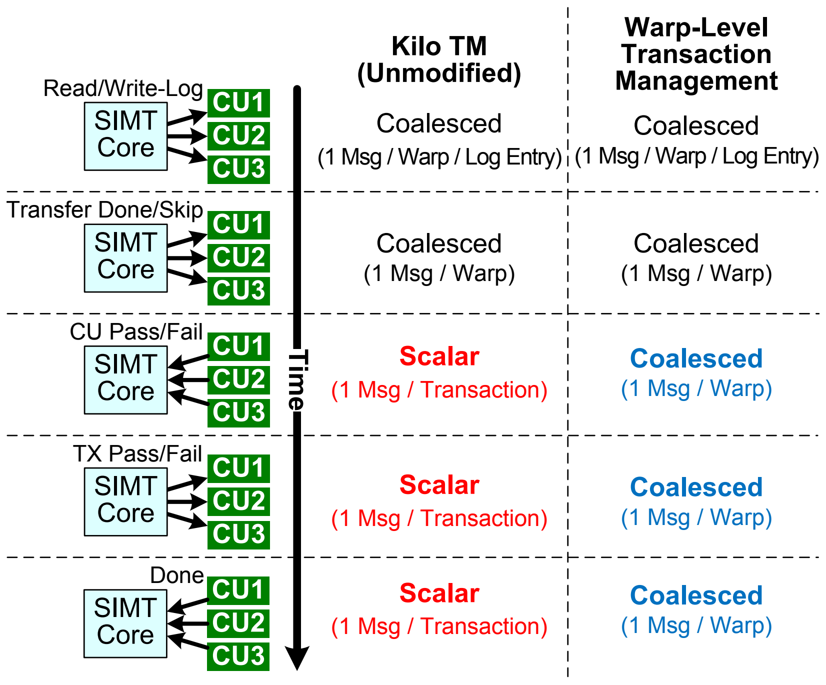


Figure 6.3: Kilo TM Protocol Messages.

trying to append leaf nodes to a small tree.

Aggregate Control Messages. Figure 6.3 illustrates the protocol messages that are sent between the SIMT core and the commit units to commit a transaction. Notice that the original proposal of Kilo TM already aggregates the read-set and write-set messages and the done-fill messages from multiple transactions in a warp. However, it does not aggregate the remaining protocol messages.

With WarpTM, in the absence of potential intra-warp conflicts, the commit unit

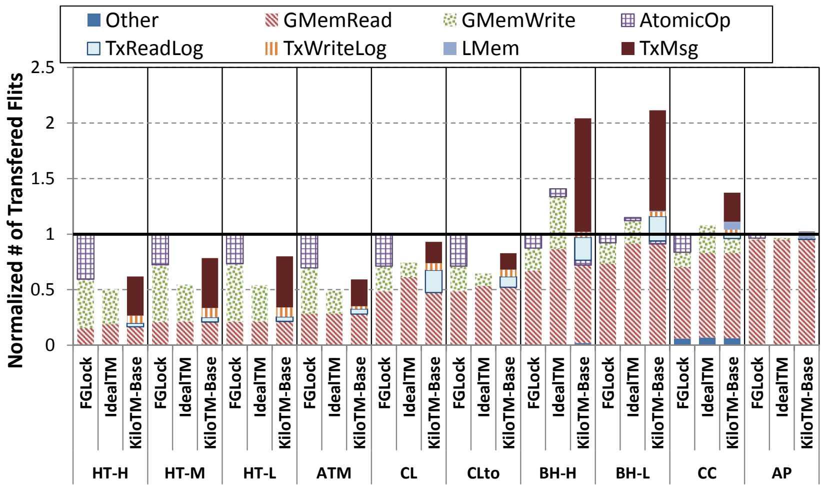


Figure 6.4: Interconnection Traffic Breakdown.

can wait until all transactions have finished validation and combine their outcomes (pass/fail) into a single message. After receiving replies from all the commit units, the SIMT core can also combine the final outcomes of the entire warp into a single message that is broadcast to the commit units.

Figure 6.4 shows the interconnection traffic breakdown for our workload with fine-grained locks, Ideal TM, and Kilo TM. The protocol messages for Kilo TM (TxMsg) on average account for 36% of the interconnection traffic.

Validation and Commit Coalescing. While applications with irregular parallelism tend to exhibit less spatial locality among threads within a warp, coalescing memory accesses from the same warp can still significantly reduce the number of accesses. With the original Kilo TM, the memory accesses performed by threads during transaction execution are already coalesced just as the non-transactional memory accesses. The commit units, however, generate scalar memory accesses for validation and commit of transactions to avoid explicitly handling intra-warp conflicts. This simplifies the design of the commit units. With WarpTM, each commit unit knows *a priori* that all transactions from the same warp are free of intra-warp conflicts. Consequently, the value-comparison outcome for the validation of one transaction will not be changed after another transaction in the same warp has committed. As a result, the commit unit can always merge the scalar

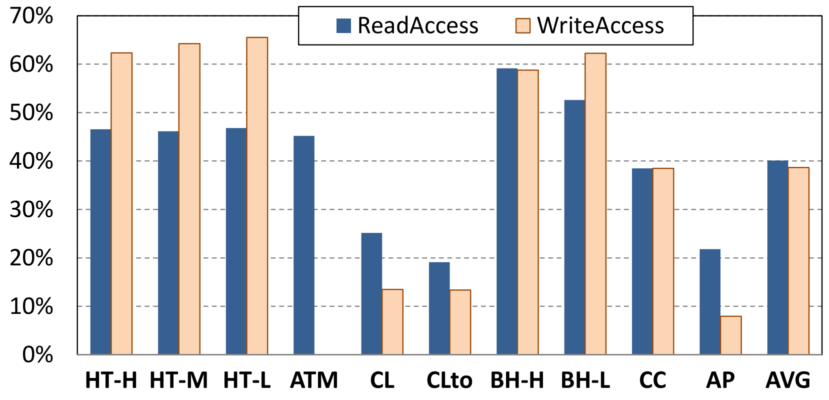


Figure 6.5: Reduction in L2 cache accesses from the commit units via validation and commit coalescing.

memory accesses for the validation of multiple transactions in the same warp into wider accesses. We call this *validation coalescing*. Similar reasoning permits the commit unit to merge scalar memory writeback accesses for the commit as well. We call this *commit coalescing*.

Figure 6.5 shows the amount of L2 cache access from the commit units that can be reduced through validation and commit coalescing. On average, coalescing can reduce the number of validation requests and memory writeback requests by 40% and 39% respectively. Without coalescing, scalar accesses that exhibit spatial locality tend to hit the L2 cache. However, they still waste L2 cache bandwidth supplied by the wide cache ports, and they will consume more L2 cache miss-status holding registers (MSHR) that track accesses waiting for in-flight requests from DRAM.

Validation and Commit coalescing can benefit any GPU TM system as long as the GPU still employs the SIMT execution model and accesses memory in large contiguous chunks. Even with wide-channel 3D DRAMs, there are tangible benefits in amortizing SRAM and DRAM control logic by accessing data in large chunks, as long as the applications contain sufficient memory access spatial locality. Most existing GPU applications do.

6.2.2 Hardware Modification to Kilo TM

In addition to implementing intra-warp conflict resolution (described in Section 6.3), WarpTM requires modification to the commit units. Each commit unit contains a small *read/write buffer* that caches the read-logs and write-logs of committing transactions. In the original Kilo TM, the commit unit only accesses one word from the read-log or write-log of one transaction in each cycle. The read/write buffer can supply this bandwidth with a narrow (4-Byte) port. WarpTM requires this read/write buffer to have a wide (64-Byte) port. The wide port allows the read-sets and write-sets from multiple transactions in the same warp to be retrieved in a single cycle. Each commit unit also needs to be extended with a memory coalesce logic unit to merge multiple scalar accesses that head to the same cache block into a wider access.

6.3 Intra-Warp Conflict Resolution

Developing a low overhead mechanism to detect conflicts among transactions within a warp is the key challenge in enabling WarpTM. Each transaction in Kilo TM stores its read-set and write-set as linear logs in the local memory space. The logs are organized physically such that each transaction may only access one word in its logs per cycle. Detecting conflicts between two transactions naively requires traversing the linear logs repeatedly, once for each word in the read- and write-log, for a full comparison of the logs. Even if the logs are fully cached in the L1 data cache, a full pair-wise comparison among T transactions still requires $O(T^2 \times N)$ traversals, where N is the combined size of the read- and write-logs of a transaction. The overhead might negate any performance and energy benefit from WarpTM.

While it is possible to detect and resolve intra-warp conflicts with the last writer history units employed in the commit units to boost commit parallelism of value-based conflict detection, each last write history unit can only detect conflicts for one transaction at a time. It is only as effective as the sequential conflict resolution (SCR) introduced in Section 6.3.2. The 2-phase parallel conflict resolution introduced in Section 6.3.3 allows multiple transactions in the warp to resolve their intra-warp conflicts in parallel.

6.3.1 Multiplexing Shared Memory for Resolution Metadata

We noticed that many applications that require irregular communication between threads in different SIMT cores make little use of shared memory (the on-chip scratchpad memory). This observation is exploited in NVIDIA Fermi GPUs by allowing part of the shared memory storage to be configured as the L1 data cache [121]. The non-configurable part of the shared memory remains unused in most of these applications. Given that intra-warp conflict resolution only involves communication within a warp, we propose to use this underused storage as temporary buffers for intra-warp conflict resolution. When a warp has finished executing its transactions, it allocates a buffer in the shared memory to perform the intra-warp conflict resolution. The warp then uses this buffer to store metadata for its intra-warp conflict resolution, and releases the buffer after the resolution is done. This allows the buffer to be time-shared by multiple warps – a technique known as shared memory multiplexing [168]. The shared memory storage can also be partitioned into multiple buffers to allow multiple warps to interleave their intra-warp conflict resolution to hide access latency for read/write-log accesses that miss the L1 data cache.

To support applications that use shared memory for other computations, the GPU command unit can be extended to launch fewer thread blocks on each SIMT core according to amount of metadata storage reserved by the programmer. Similar strategy can be used to reserve metadata buffers for intra-warp conflict resolution on future GPU architectures that feature unified storage for shared memory, data cache and registers [64]. We leave to future work exploration of trade offs between intra-warp conflict detection metadata capacity and the ability to run more concurrent threads with more register storage.

6.3.2 Sequential Conflict Resolution with Bloom Filter (SCR)

We propose to store a bloom filter [26, 111, 170] in the shared memory. Using this bloom filter, we have developed a sequential conflict resolution (SCR) scheme that always prioritizes the transactions executed by the lower lanes in the warp. Threads with lower thread ID are assigned to the lower lanes in the warp. In SCR, the transaction with the lowest lane in the warp first populates the bloom filter with its write-set. Each subsequent transaction in the warp first checks to see if its read-set or write-set hits in the bloom filter. If so, this transaction conflicts with one of the transactions in the prior lanes, and it is aborted. Otherwise, the transaction adds its write-set to the bloom filter to make its write-set visible to the subsequent transactions in the warp. The accumulative nature of the bloom filter allows each transaction to compare its read and write-set against the write-set of all transactions in the prior lanes. Prior TM proposals that uses bloom filter for conflict detection [26, 111, 170] did not exploit this accumulative effect of bloom filter to allow a transaction to detect conflicts with multiple transactions via a single query. While SCR does reduce the number of transaction log traversals from $O(T^2 \times N)$ to $O(T \times N)$, its sequential nature makes poor use of the bandwidth provided by the L1 cache and shared memory.

6.3.3 2-Phase Parallel Conflict Resolution with Ownership Table (2PCR)

In SCR, each transaction in the warp is essentially matching its read-set and write-set with the aggregated write-set of all the transactions in the prior lanes. This matching is inherently parallel if each lane has a pre-constructed record of the aggregated write-set from its prior lanes. Also, the priority among lanes is known in advance, so that multiple conflicting lanes can resolve the conflicts unanimously in parallel without extra communication. From these two insights, we have developed *two-phase parallel intra-warp conflict resolution* (2PCR). First, the transactions in the warp collaboratively construct an *ownership table* [118] in parallel from the write-logs of every transaction in the warp. Each transaction then checks this ownership table for conflicts with another transaction in a prior lane. If such a conflict exists, the transaction aborts itself.

Each entry in the ownership table represents a region in global memory. Its value contains the lane ID (5-bit) of the lowest lane that intends to write to the region and an extra null-bit to indicate if none of the lanes intends to write to the region. Each entry, padded with two unused bits, occupies a byte in shared memory. When a warp starts performing 2PCR, it first initialize the ownership table by setting the null-bit in every entry through a burst of shared memory writes (each write can initialize 128 Bytes, so a 2K-entry table only takes 16 writes). To construct the ownership table, each transaction traverses through its write-log to read out the locations in its write-set. For each location, the transaction calculates the index of the corresponding entry in the ownership table hashing the location's address. It then updates the corresponding entry with its own lane ID if the existing value in the entry has a higher lane ID. The lockstep nature of a warp and the memory pipeline allows this to occur in parallel: At each step, every transaction reads one entry from its write-log, reads the existing value from the corresponding ownership table entry, compares the value against its own lane ID and updates the entry if its lane ID is lower. A hardware mechanism that implements atomic operations for shared memory [32] is used to prevent two transactions from racing to update the same entry at the same step.

After constructing the ownership table, every transaction traverses through its read-log and write-log. For each location in the read-log, the transaction retrieves the lane ID from the corresponding ownership table entry. If the retrieved lane ID is not null and it is *lower* than the transaction's own lane ID, a conflict exists between this transaction and an earlier transaction. For each location in the write-log, the transaction also retrieves the lane ID from the corresponding ownership table entry. However, a conflict exists only if the retrieved lane ID value does *not equal* the transaction's own lane ID. The different lane ID indicates that another transaction may overwrite the same location as this transaction. Notice that in this case, the retrieved lane ID will always be smaller, because the ownership table construction mandates that every entry contains the lane ID of the lowest lane intending to write to the corresponding region. Every transaction with any detected conflict aborts itself. The remaining transactions in the warp can then proceed and benefit from the optimizations enabled by WarpTM.

Figure 6.6 contains an example of the two-phase parallel intra-warp conflict resolution. The example consists of four transactions (X_1, X_2, X_3, X_4), each reading and writing to two locations in memory (except X_4, which only reads from one location). These memory accesses are stored in the read-log and write-log shown in the upper left corner of Figure 6.6. Each column in both the read-log and the write-log shows the read-set and write-set of a particular transaction respectively. Each entry in the read-log is labeled according to its position in the log (RL1, RL2); entries in the write-log of each transaction is labeled similarly (WL1, WL2). Entries at the same position in read-logs and write-logs of the four transactions

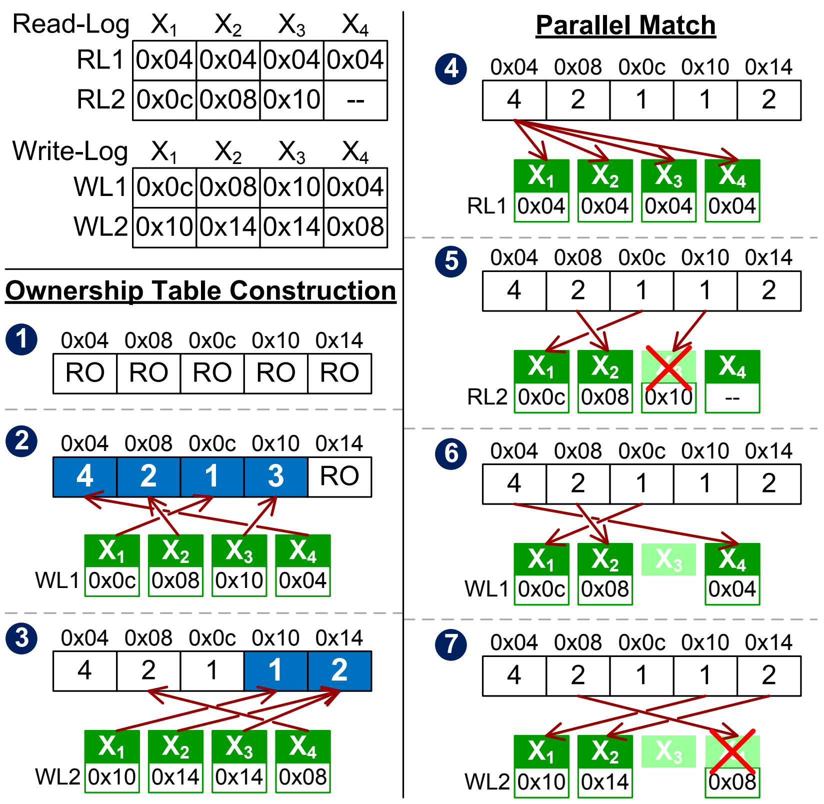


Figure 6.6: Two-phase parallel intra-warp conflict resolution. Each step shows the content of the ownership table and accesses from the transactions in the warp. RO = Read-Only

are grouped into a wide entry, shown as a row in the logs, that can be read out in parallel. Here are the steps in this example indicated by the numbers in Figure 6.6:

1. Every entry in the ownership table is initialized to read-only (RO).
2. Every transaction updates the ownership table according to entry 1 in its own write-log (WL1).
3. The transactions proceed to entry 2 in their write-log (WL2). The ownership entry for 0x08 is not updated to X_4 because it is already owned by X_2, which has a higher priority. Meanwhile, the ownership of 0x10 is updated from X_3 to X_1.
4. The transactions proceed to entry 1 in their read-log (RL1) for parallel match-

ing. Since all transactions read a single location 0x04, they check the ownership table in parallel for the first lane that writes to 0x04, which is X_4. None of the transactions aborts at this point since their lane IDs are smaller or equal to X_4.

5. Every transaction proceeds to checking entry 2 in its own read-log (RL2). X_3 is aborted since 0x10 is already owned by X_1.
6. Each remaining transaction has ownership to address in entry 1 of its write-log.
7. After checking entry 2 in its write-log, X_4 is aborted due to a WAW conflict with X_2 at 0x08.

Finally, X_1 and X_2 are conflict free and can be validated and committed together via the commit units. Notice that this relatively narrow warp will take 15 steps with SCR. With 32-wide warps and larger transaction footprints in real workloads, the difference is even greater.

The accuracy of 2PCR depends on the size of the ownership table. Our evaluation shows that a 4K entry ownership table (requiring 4kB of storage in shared memory) performs comparably to an ownership table with infinite capacity. With a fixed-size ownership table, the accuracy of the intra-warp conflict resolution decreases for transactions with larger read/write-set. The average per-transaction footprint in our workloads spans between 3 to 36 words. Since GPU applications usually decompose larger input data into more threads, we believe that the per-transaction footprints in future GPU TM applications should not grow significantly beyond the footprints found in our workloads.

Notice that 2PCR tends to be less accurate than SCR. Since the ownership table is constructed in parallel assuming that every transaction in the warp will be committed, a transaction may unnecessarily abort due to a conflict with another aborted transaction. Nevertheless, our evaluation shows that the benefit from 2PCR outweighs the overhead from its additional false conflicts.

6.4 Temporal Conflict Detection (TCD)

While warp-level transaction management can help reduce the extra interconnect traffic introduced by the Kilo TM protocol and improve L2 cache bandwidth utility, a fundamental overhead for Kilo TM still exists: Even without conflicts among transactions, each transaction has to reread its entire read-set for value comparisons prior to updating memory. To reduce this overhead, we propose *temporal conflict detection*, a low overhead mechanism that uses a set of globally synchronized timers to detect conflicts for read-only transactions.

A read-only transaction can occur dynamically when the transaction only conditionally writes to memory, or it can be explicitly introduced by programmers to ensure that code within the transaction can safely read from a shared data structure which may be updated occasionally by other transactions. The latter use of read-only transaction is required even for TM systems with strong isolation if the application expects to read multiple pieces of data from the shared data structure. For example, a thread that periodically computes the ratio between two measured quantities should read both quantities simultaneously for every computed ratio. If a transaction commits, updating both quantities atomically, after the thread has read one quantity but before it reads the other quantity, the computed ratio may be erroneous. Strong isolation does not protect the application from this error; the only way to prevent this error is to include both reads in a read-only transaction.

A read-only transaction differs from a read-write transaction in that it can commit silently and locally as long as it has observed a consistent memory state – a memory state that does not contain partial memory updates from other committing transactions. Although the original design of Kilo TM can dynamically detect a read-only transaction at commit by observing an empty write-log, it does not exploit this information. Among the GPU TM workloads we created for evaluation, read-only transactions account for $\sim$40% of the transactions in CL/CLto and $\sim$85% of the transactions in BH-L/BH-H. In CL/CLto, the read-only transaction occurs dynamically because the transaction only conditionally applies forces to two vertices in a mesh if they are sufficiently far away. In BH-L/BH-H, we have added the read-only transaction to the octree traversal to ensure that a freshly inserted branch node has been properly initialized by the inserting thread before other threads may traverse through the branch node. Being able to commit these read-only transactions silently without rereading their read-set can significantly reduce their energy and performance overhead.

TCD is a form of eager conflict detection that complements Kilo TM. Using a set of globally synchronized timers, it checks the accessed location as a transaction reads from global memory to ensure that the loaded data has not been modified since the transaction first reads from global memory. To do so, the system records when each word in memory was last written. Each transaction maintains the time of its first load, and each subsequent load in the transaction retrieves the time when the loaded word is last written. A retrieved last written time that occurs later than the time of the transaction's first load indicates a potential conflict, because the value at the loaded location has been modified since the first load. If none of the words loaded by the transaction has been written since the first load, the value of every word read by the transaction coexists in a instantaneous snapshot of global memory that existed at the time of the first load. A read-only transaction satisfying this condition has effectively obtained all of its input values from this snapshot, and appears to have executed instantly with respect to other transactions. Therefore, the read-only transaction can commit directly without further validation.

Notice that the instantaneous snapshot observed by the transaction via TCD may occur in the midst of a memory writeback from a committing transaction. This causes the snapshot to contain partial updates from a transaction, which is not a consistent view of memory. We have not observed this issue in the workloads we evaluated. Nevertheless, it is possible to extend TCD to detect if a transaction is loading from a location in the write-set of another committing transaction. The overhead for such detection mechanism involves a hardware buffer that conservatively records the last transaction that has written to each memory location, and extra protocol messages and hardware for maintaining a conservative set of committing transactions. We leave the exploration of this and other potential solutions for future work.

6.4.1 Globally Synchronized Timer

While timestamp-based conflict detection has been used in existing software TM systems [37, 148, 167], each of these systems uses a global version number (a software counter in memory) that is explicitly updated by software at transaction commit. The globally synchronous timers used by temporal conflict detection are different from these software maintained counters. *Once initialized, each of these*

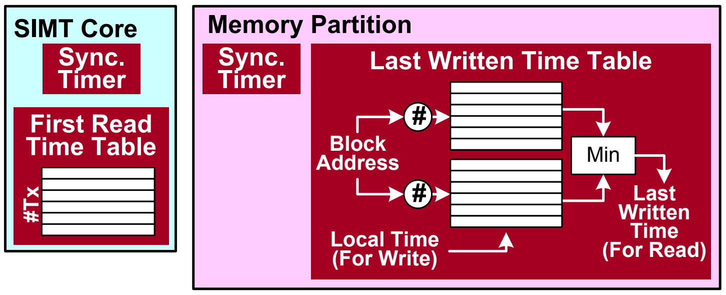


Figure 6.7: Hardware extensions for temporal conflict detection.

on-chip timers runs locally in its microarchitecture module, increments every cycle, and does not communicate with other timers. Since the timers run at the same frequency and are synchronized initially, one can compare a timestamp captured from one of the timers against another timestamp captured from another timer to determine their respective order in real time. This hardware mechanism eliminates the bottleneck of updating and accessing a centralized global version number. As pointed out by Singh et al. [146], existing hardware already implements timers synchronized across components [79, Section 17.12.1] to provide efficient timer services. Ruan et al. [135] also proposed extending Orec-based STM systems with synchronous hardware timers. Their approach embeds timestamps in the ownership record of each transaction variable in memory, whereas we use a small on-chip storage to conservatively record when each word is last written.

6.4.2 Implementation

Figure 6.7 shows the hardware modification to implement TCD: A 64-bit globally synchronized timer in each SIMT core and memory partition, a first-read time table in each SIMT core recording when each transaction sends its first load, and a last written time table in each memory partition that conservatively records the last written time of each 128-Byte block in the partition. This time table in conservative in that aliasing in the time table may cause it to return a timestamp for a memory location that is more recent than the actual time when the location was last written. We implement the last written timetable with a *recency bloom filter*. Kilo TM also use it for hazard detection in Chapter 4. This variant of the recency bloom filter consists of multiple sub-arrays of timestamps. Each 128-Byte block in the memory partition maps to an entry in each sub-array of the recency bloom filter via a different hash function. Whenever a word in the 128-Byte block is updated by a committing transaction in the L2 cache, the corresponding entries in every sub-array of the recency bloom filter are updated with the value from the synchronized timer. Each transactional load served by the L2 cache retrieves a timestamp from each sub-array in the recency bloom filter and returns the minimum of those timestamps along with the data to the SIMT core. This timestamp is compared against the time of the transaction's first load to detect conflicts.

At 700MHz, 64-bit timers only roll over every few hundred years. In the event that it happens, the TM system can handle the rollover by validating the read-set of all running transactions through value-based conflict detection. For the transactions that remain valid, the TM system resets their first read time to zero. Although not needed for correctness, it should also reset the last written time table so that the table will not report overly conservative last written times.

6.4.3 Example

Figure 6.8 walks through how TCD detects an inconsistent view of memory during transaction execution. The example consists of two committing transactions, X_A and X_B, and one read-only transaction X_C. At time T=1, X_C starts execution and issues its first load to location 0x10. X_A then commits and updates the value at 0x10 at time T=7. At the same time, X_A also updates the timestamps corresponding to location 0x10 in the recency bloom filter (consists of 2 sub-arrays with 2 timestamp each). X_C issues its second load to location 0x30 at T=10, and the recency bloom filter returns with T_W[0x30]=0. Notice that even though the original value at 0x10 is overwritten by X_A, the updated value is not visible to X_C and does not constitute a conflict. X_B commits and updates the value at 0x20 at time T=15, and updates the recency bloom filter. X_C issues its third load to location 0x20 at T=18, and the recency bloom filter returns with T_W[0x20]=15, which is later than the first load from X_C. This is a conflict for X_C because the memory value loaded from 0x20 at T=18 is not the same value at 0x20 at T=1; it has been updated by X_B at T=15. Hence, the memory state observed by X_C does not correspond to an actual global

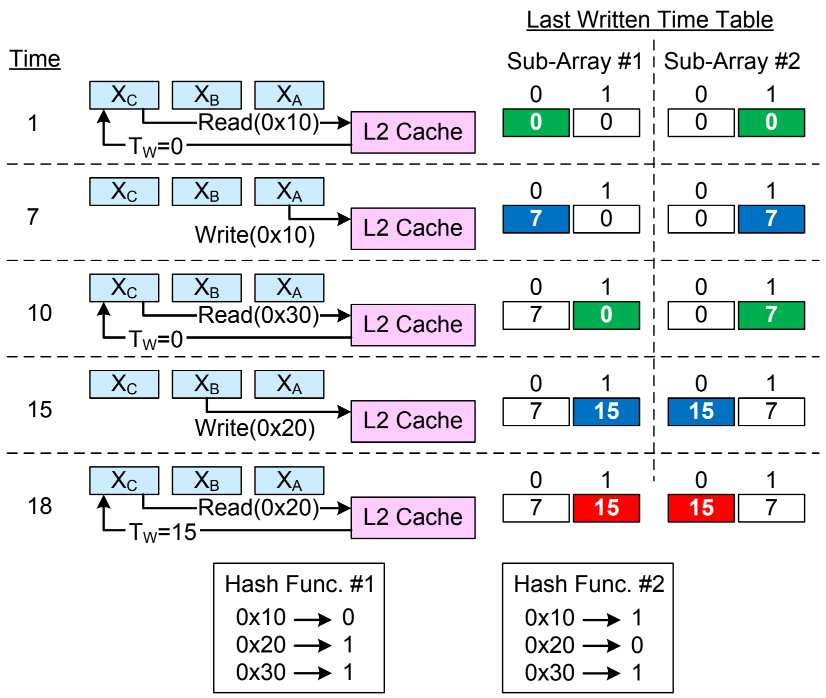


Figure 6.8: Temporal conflict detection example.

memory state at any time – an invalid snapshot.

6.4.4 Integration with Kilo TM

In this work, Kilo TM uses TCD to allow read-only transactions to commit silently in the absence of detected conflicts. The recency bloom filter does not perfectly record the time when each word is last written. Aliasing of timestamps in the filter can lead to false positives in TCD. To reduce the penalty of falsely detected conflicts, read-only transactions with detected conflict are given a second chance to commit through the commit units as read-write transactions in Kilo TM. In this way, we can use a relatively small filter with coarse granularity (128-Byte chunk maps to the same entry) to allow most conflict-free read-only transaction to commit silently, and use value-base conflict detection for the situations that require finer granularity detection. Similar hierarchical validation schemes are used in NOrec [37] and the software GPU TM system by Xu et al. [167].

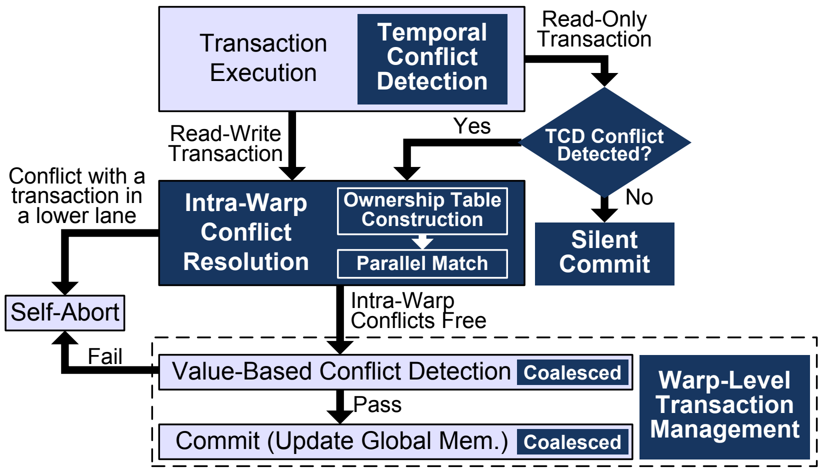

Transaction Execution
Temporal Conflict Detection
Read-Only Transaction
Read-Write Transaction
Yes
TCD Conflict Detected?
No
Silent Commit
Conflict with a transaction in a lower lane
Intra-Warp Conflict Resolution
Ownership Table Construction
Parallel Match
Self-Abort
Intra-Warp Conflicts Free
Fail
Value-Based Conflict Detection
Coalesced
Pass
Commit (Update Global Mem.)
Coalesced
Warp-Level Transaction Management

Figure 6.9: Kilo TM enhanced with warp-level transaction management and temporal conflict detection.

6.5 Putting It All Together

Our two proposed enhancements to Kilo TM, warp-level transaction management and temporal conflict detection, can work together to further improve performance. Figure 6.9 shows the overall design of Kilo TM with both enhancements enabled. In this enhanced Kilo TM, each transaction uses TCD to eagerly detect conflicts for each global memory read during its execution. Writes to global memory are buffered in the write-log as in the original Kilo TM. After the transaction has completed execution, if it is a read-only transaction (i.e. containing an empty write-log) and TCD has not detected any conflict, it can commit silently. The remaining transactions take part in the intra-warp conflict resolution to resolve all conflicts within the same warp. WarpTM then processes the still-active transactions in the warp with the assurance that they do not have conflict among each other. Our evaluation in Section 6.7 compares the performance and energy consumption of this combined TM system to the original Kilo TM.

6.6 Methodology

For our evaluation, we started with the version of GPGPU-Sim [10] from Chapter 4. It extends GPGPU-Sim version 3.1.2 with support for transactional memory,

Table 6.1: GPGPU-Sim configuration for enhanced Kilo TM evaluation

# SIMT Cores	15
Warp Size	32
SIMD Pipeline Width	16×2
# Threads / Core	1536
# Registers / Core	32768
Branch Divergence Method	PDOM [56]
Warp Scheduling Policy	Greedy-then-oldest [132]
Shared Memory / Core	16KB
L1 Data Cache / Core	48KB, 128B line, 6-way assoc. (transactional+local mem. access only)
L2 Unified Cache	128KB/Memory Partition, 128B line, 8-way assoc.
Interconnect Topology	1 Crossbar/Direction
Interconnect BW	32 (Bytes/Cycle) (288GB/s/Dir.)
Interconnect Latency	5 Cycle (Interconnect Clock)
Compute Core Clock	1400 MHz
Interconnect Clock	1400 MHz
Memory Clock	924 MHz
# Memory Partitions	6
DRAM Req. Queue	32 Requests
Memory Controller	Out-of-Order (FR-FCFS)
GDDR5 Memory Timing	Hynix H5GQ1H24AFR
Total DRAM BW	177GB/s
Min. L2 Latency	330 Cycle (Compute Core Clock)
DRAM Scheduler Latency	200 Cycle (Compute Core Clock)

Kilo TM	
Commit Unit Clock	700 MHz
Validation/Commit BW	1 Word/Cycle/Memory Partition
# Concurrent TX	1, 2, 4, 8 Warps/Core or No Limit (480, 960, 1920, 3840 or Unlimited # TX Globally)
Last Writer History Unit	5kB

Intra-Warp Conflict Resolution	
Shared Memory Metadata	4kB/Warp (3 Concur. Resolution/Core)
Default Mechanism	2-Phase Parallel Conflict Resolution

Temporal Conflict Detection	
Last Written Time Table	16kB (2048 Entries in 4 Sub-Arrays)
Detection Granularity	128-Byte

and includes the performance model for Kilo TM. We incorporated GPUWattch [97] into this version of GPGPU-Sim, and extended it to model the timing and power of our proposed enhancements. This version of GPGPU-Sim with all the modifications is available online [60]. We configured the modified GPGPU-Sim to simulate a GPU similar to Geforce GTX 480 (Fermi), with 16kB of shared memory storage per SIMT core. Table 6.1 lists the major microarchitecture configurations.

We used the GPU TM workloads from Chapter 4 to evaluate the proposed

Table 6.2: GPU TM workloads for performance and energy evaluations.

Name	Abbr.	Description
Hash Table (CUDA)	HT-H	Populate an 8000-entry hash table.
	HT-M	Populate an 80000-entry hash table.
	HT-L	Populate an 800000-entry hash table.
Bank Account (CUDA)	ATM	Parallel transfer between 1M accounts.
Cloth Physics [20] (OpenCL)	CL	Cloth physics simulation of 60K edges.
	CLto	Optimized version of CL.
Barnes Hut [23] (CUDA)	BH-H	Build an octree with 30K bodies.
	BH-L	Build an octree with 300K bodies.
CudaCuts [162] (CUDA)	CC	Segmentation of a 200×150 pixel image.
Data Mining [3, 88] (CUDA)	AP	Data mining 4000 records.

improvements to Kilo TM. In addition to the original input, we also added new inputs that varies the amount of contention for BH and HT. We also created an optimized version of CL (CLto). In this version, each thread loads read-only data into its register file before entering the transaction to reduce the read-set of the transaction. Table 6.2 summarizes each of our workloads.

6.6.1 Power Model

We modeled the power overhead of Kilo TM by estimating the access energy of the various major structures in the commit units implemented in the 40nm process with CACTI 6.5 [115]. We multiplied the access energies with the operating frequency, conservatively assuming that the structures are accessed every cycle, to estimate their power overhead. As observed by Shah [143], CACTI provides conservative area and energy estimates for small memory arrays as it automatically partitions the array into sub-arrays when a single array is sufficient. Similarly, we modeled the power overhead of TCD with the full activity power to the last written time buffer in each memory partition and the first read timetable in each SIMT core. Table 6.3 shows the estimated power for each component in Kilo TM and temporal conflict detection. The Kilo TM specific hardware consumes 0.9W in total, and extending it to support WarpTM increases the consumption to 2.5W. The power increase is introduced by having a wider port to the read-write buffer (See Section 6.2.2). The hardware that implements TCD consumes 0.7W. Kilo TM with WarpTM and TCD consumes a total of 3.2W.

Table 6.3: Power component breakdown for the added hardware specific to Kilo TM, warp-level transaction management, and temporal conflict detection.

Commit Unit			
	Size	Area (mm^2)	Power (mW)
Last Writer History - Look Up Table	3kB	0.010	6.3
Last Writer History - Recency Bloom Filter	2kB	0.010	6.6
Commit Entry Array	19kB	0.094	57.5
Read-Write Buffer	32kB	0.128	82.5
Per-Unit Total		0.242	153
All Units Total		1.454	918
Commit Unit (Warp-Level Transaction Management)			
Read-Write Buffer (Warp-Level TM)	32kB	0.731	260
Per-Unit Total		0.846	419
All Units Total		5.074	2512

Temporal Conflict Detection			
	Size	Area (mm^2)	Power (mW)
First Read Timetable (One per SIMT core)	12kB	0.034	25.5
Last Written Time Buffer (One per Mem. Part.)	16kB	0.078	52.3
All Units Total		0.979	696

For parts of the GPU microarchitecture not specific to Kilo TM, we used GPUWattch [97] to estimate the average dynamic power consumed by each workload with the different synchronization mechanisms. This captures the difference in microarchitecture activity between fine-grained locks and Kilo TM (with and without the proposed enhancements). This includes extra L1 cache accesses for the transaction logs, extra L2 accesses for value-based conflict detection, extra interconnection traffic for Kilo TM protocol messages, and accesses to shared memory for intra-warp conflict resolution. To compute the total power, we added 59W for leakage power (obtained by Leng et al. [97] via measuring the idle power of the hardware GPU), 9.8W of constant clock power (reported by GPUWattch) and the dynamic power of the Kilo TM specific hardware to the average dynamic power reported by GPUWattch to obtain the total power. Finally, we multiplied this total power by the execution time to obtain the total energy required to execute each workload.

We assumed that the GPU extended with Kilo TM, WarpTM and TCD runs at the same frequency as the unmodified GPU. We did not evaluate the impact of Kilo TM and the two enhancements on the GPU cycle time. We leave this evaluation as

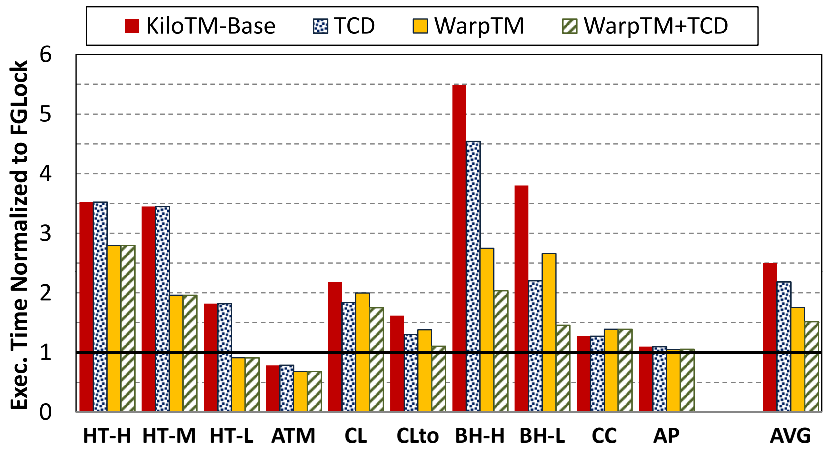


Figure 6.10: Execution time of GPU-TM applications with enhanced Kilo TM. Lower is better.

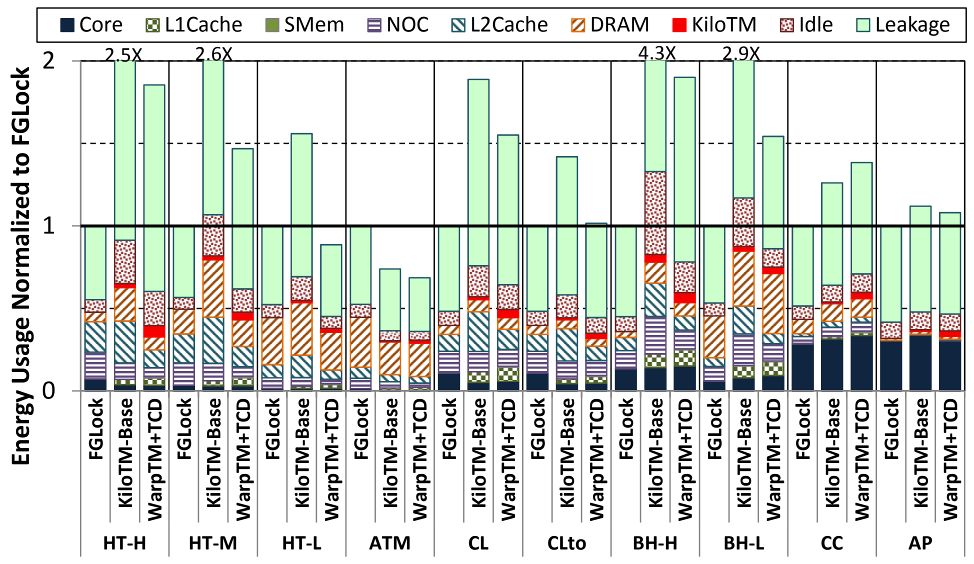
future work.

6.7 Experimental Results

In this section, we evaluate the performance and energy efficiency of our proposed enhancements to Kilo TM: warp-level transaction management (WarpTM) and temporal conflict detection (TCD). We also analyze the benefit of each optimization that is enabled by WarpTM, compare the two intra-warp conflict resolution approaches, and investigate the sensitivity of TCD to available hardware resources. Finally, we study the performance impact of L2 cache port width and number of SIMT cores on both fine-grained locks and Kilo TM.

6.7.1 Performance and Energy Efficiency

Figure 6.10 compares the execution time of the original Kilo TM (KiloTM-Base), Kilo TM with TCD enabled (TCD), Kilo TM with WarpTM (WarpTM) and a configuration with both enhancements enabled (WarpTM+TCD). We evaluate the performance of each configuration with different limits on the number of concurrent transactions, and select a limit for each workload that yields the optimal performance (See Table 6.4). The performance of each configuration with this optimal limit is normalized to the execution time of an alternative version of the application



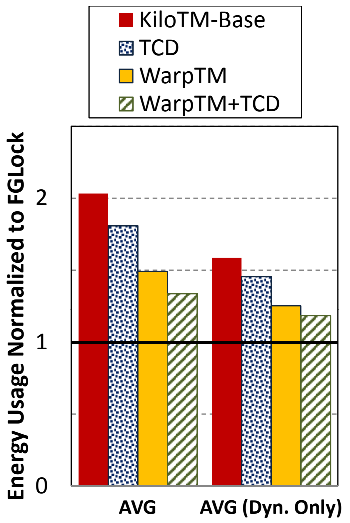


Figure 6.11: Energy consumption breakdown of GPU-TM applications. Lower is better.

using fine-grained locking (FGLock) to illustrate their overhead with respect to a pure software effort. Figure 6.11 breaks down the energy consumption of the same set of Kilo TM configurations. Each breakdown is normalized to the total energy used by the fine-grained locking version of the same workload.

Without WarpTM and TCD, Kilo TM performs 2.5× slower[8]than FGLock on average. The performance of BH-H is particularly poor. Our detailed investigation shows that the major slowdown occurs near the start of the octree-building kernel, where every thread tries to append its node to only a few branches in the octree. This behavior also stresses one memory partition in the GPU, working against the distributed design of Kilo TM. This effect slowly disappears as the octree grows to a point that conflicts become rare. As a result, BH-L, which scales the number of nodes in the octree by 10×, exhibits less slowdown. A similar attempt to reduce transaction contention does not work as well between HT-H and HT-M. The energy-per-operation penalty of Kilo TM is relatively lower (2× energy used vs. 2.5× performance slowdown) due to its lower activity from the poor performance.

Enabling temporal conflict detection for Kilo TM improves the performance of workloads that contain read-only transactions (CL, CLto, BH-H and BH-L). By allowing non-conflicting read-only transactions to commit silently, TCD reduces contention at the commit units and the memory subsystem. This performance improvement translates directly to energy savings as well. Across workloads with read-only transactions, TCD improves performance of Kilo TM by 37% while reducing energy per operation by 30%.

The optimizations enabled by warp-level transaction management (WarpTM) apply to a broader class of applications. In particular, coalescing of memory accesses from the commit units alleviates the bottlenecks at the L2 cache banks. Without this bottleneck, the reduction in transaction contention from HT-H to HT-M now leads to performance improvement for Kilo TM + WarpTM. Section 6.7.3 analyzes how each optimization from WarpTM contributes to speeding up Kilo TM in different applications. WarpTM does slow down CC, due to overhead from intra-warp conflict resolution (See Section 6.7.4). Overall, WarpTM speeds up Kilo TM by 42% and reduces energy per operation by 27%.

[8]Notice that this is slower than the 59% slowdown reported in Chapter 4. This change is introduced by a change in the baseline GPU architecture as explained in Section 6.1.

Table 6.4: Performance-optimal concurrent transaction limit and abort-commit ratio. Base = KiloTM-Base. NL = No Limit. – = Identical to no TCD.

	Concurrent Transaction Limit (#Trans. Warps/SIMT Core)				Aborts per 1000 Committed Trans.			
	Base	TCD	WarpTM	TCD+WarpTM	Base	TCD	WarpTM	TCD+WarpTM
HT-H	2	–	2	–	50	–	107	–
HT-M	2	–	8	–	7	–	84	–
HT-L	4	–	8	–	2	–	63	–
ATM	1	–	4	–	1	–	27	–
CL	2	2	2	2	84	55	149	97
CLto	2	2	2	4	85	49	102	99
BH-H	2	2	4	4	23	23	53	56
BH-L	8	4	8	8	7	5	17	20
CC	NL	–	NL	–	6	–	6	–
AP	1	–	1	–	264	–	318	–

Kilo TM with both enhancements enabled shows greater benefit than with either enhancement alone. The combined benefits, together with software optimizations, allow CLto to perform within 90% of the FGLock version while using about the same amount of energy. The original version of this cloth simulation workload performs 2.2× worse than the FGLock version on Kilo TM. This kind of performance improvement indicates that a well designed TM system can complement optimization efforts from the software developer to produce efficient TM applications comparable to FGLock. Overall, Kilo TM with both TCD and WarpTM enabled achieves 66% of the performance of FGLock with only 34% energy overhead.

Concurrency Limit. Table 6.4 shows the limit on number of concurrent transactions that yields the best performance for each workload with the different Kilo TM configurations. Enabling WarpTM generally increases the optimal limit for each workload, as a result of reduced congestion at the memory subsystem and interconnection network compared to KiloTM-Base. The exceptions include high-contention workloads (HT-H), workloads that do not benefit from WarpTM (CC, AP), and workloads that may overflow the L1 cache with higher limits (CL). We find that enabling TCD has little effect on the optimal concurrency limit for our workload. Enabling TCD increases the limit for CLto, but the actual speedup from

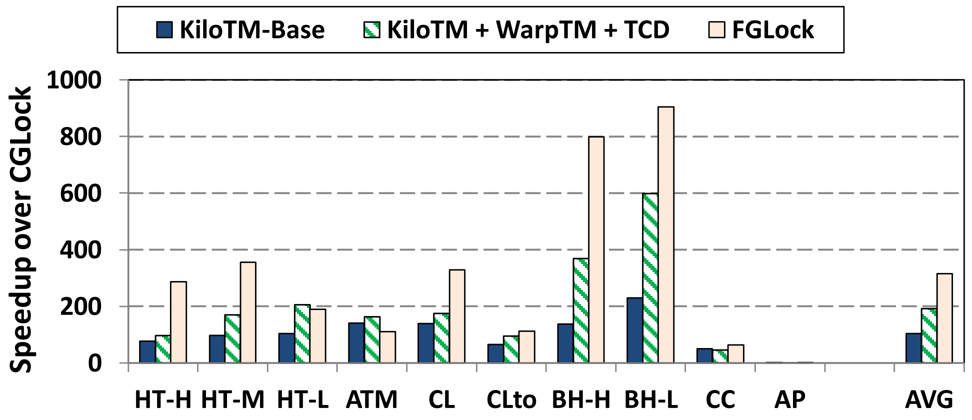


Figure 6.12: Performance comparison with coarse-grained locking. Higher is better.

the increased limit is $< 5\%$.

Impact on Abort-Commit Ratio. Table 6.4 also shows the number of aborts per 1000 committed transactions for the different Kilo TM configurations. Enabling TCD reduces the number of aborted transactions for CL and CLto by shortening the execution time span for each read-only transaction. This lowers the probability of another transaction overwriting a location in the read-set of the read-only transaction. Enabling WarpTM introduces significantly more aborted transactions due to false conflicts from intra-warp conflict resolution. Nevertheless, our evaluation has shown that WarpTM leads to an overall speedup and a net energy saving.

Comparison with Coarse-Grained Locking. Figure 6.12 compares the performance of our GPU TM applications running on Kilo TM against coarse-grained lock versions of the applications (CGLock). The performance of coarse-grained lock is estimated by serializing all transaction executions via a single global lock. On average, the applications running on Kilo TM outperforms coarse-grained locking by $104\times$ on average. Enabling WarpTM and TCD for Kilo TM increases this speedup to $192\times$. Finally, fine-grained locking versions of the applications (FGLock) runs $315\times$ faster than its coarse-grained locking analogs.

6.7.2 Energy Usage Breakdown

The energy usage breakdown in Figure 6.11 illustrates the relative contributions from different overheads of Kilo TM to its overall energy usage. Across our workloads, leakage and idle power contributes to $> 50\%$ of the total energy consumption. Both leakage power and idle power (consisting mostly of clock distribution power) persist throughout the program execution, so their contributions increase as execution time lengthens. Removing the contribution from leakage reduces the overall energy overhead of KiloTM-Base from 103% to 59%. Similarly, the pure dynamic energy overhead of KiloTM with WarpTM and TCD enabled is only 18% (versus 34% with leakage).

Aside from leakage and idle power, the memory subsystem (L2 Cache and DRAM) and the interconnection network (NoC) dominate the remaining portion of the dynamic energy usage. On average, the two combined contribute to $\sim$70% of the dynamic energy. For some workloads, the L2cache energy with KiloTM-Base is $> 2\times$ that of FGLock. WarpTM and TCD have essentially eliminated this overhead, and on average, reduce the combined energy of L2 Cache, DRAM and NoC by 29%. Energy consumed by the SIMT cores only contributes to $\sim$25% of the dynamic energy usage. With Kilo TM, energy consumption by the core is lower than FGLock. This illustrates how an effective transaction concurrency control mechanism can substantially cut down the energy overhead for transaction re-execution. L1 accesses for transaction logs contribute to 5% of the dynamic energy usage. Adding intra-warp conflict resolution to support WarpTM increases this overhead by 23% (i.e. $< 2\%$ increase to the overall energy usage), because the intra-warp conflict resolution generate extra accesses to the transaction logs. Finally, Kilo TM-specific hardware only contributes to 3% of the dynamic energy usage. Inclusion of hardware to support TCD and WarpTM only increases this to 8%.

6.7.3 WarpTM Optimizations Breakdown

Figure 6.13 breaks down the performance impact of each optimization introduced by WarpTM by enabling the optimizations one by one. In this analysis, we have used an ideal version of intra-warp conflict resolution that has perfect accuracy and

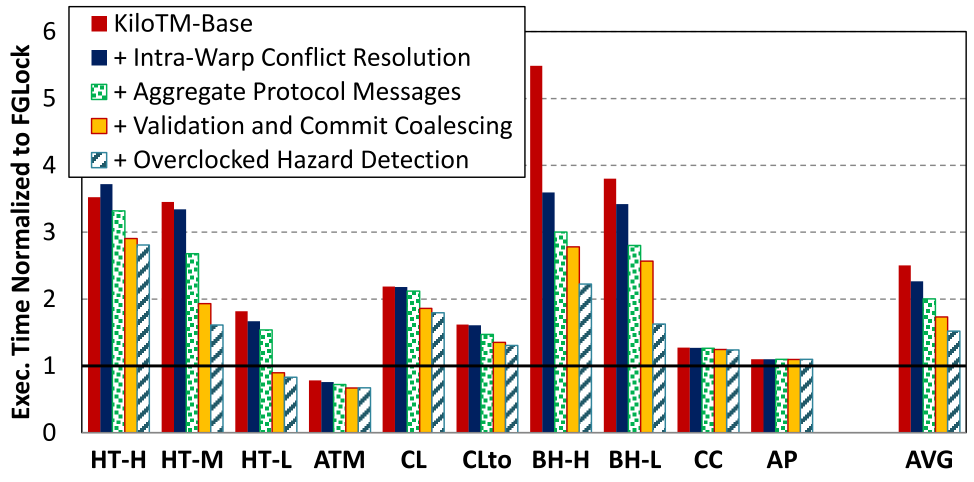


Figure 6.13: Performance impact from different optimizations enabled by WarpTM.

no overhead. This isolates our analysis from performance issues that may arise from the particular conflict resolution scheme.

With only intra-warp conflict resolution enabled, WarpTM only impacts performance for applications that exhibit intra-warp transaction conflicts. While detecting such conflicts and resolving them within the warp benefits BH, it slows down HT-H. This is because intra-warp conflict resolution is prematurely aborting transactions that could have been committed in HT-H. In resolving a conflict within a warp, it is possible that a transaction that could eventually commit is aborted while the conflicting transaction is in turn aborted in the global phase of the commit.

Aggregating protocol messages speeds up applications by a varying amount. Without further optimizations, the reduction in interconnection traffic via the aggregation simply exposes the memory subsystem bottleneck. Nevertheless, the average performance of Kilo TM is improved by 30% and the energy consumption is reduced by 17%.

Coalescing the memory accesses from the commit units allows the L2 cache bandwidth to be better utilized. This reduces the stress on the memory subsystem and improves performance for most of the applications. This optimizes Kilo TM by another 14% on average over aggregating protocol messages. BH-H and BH-L do not benefit from this optimization. This is surprising given our measurements

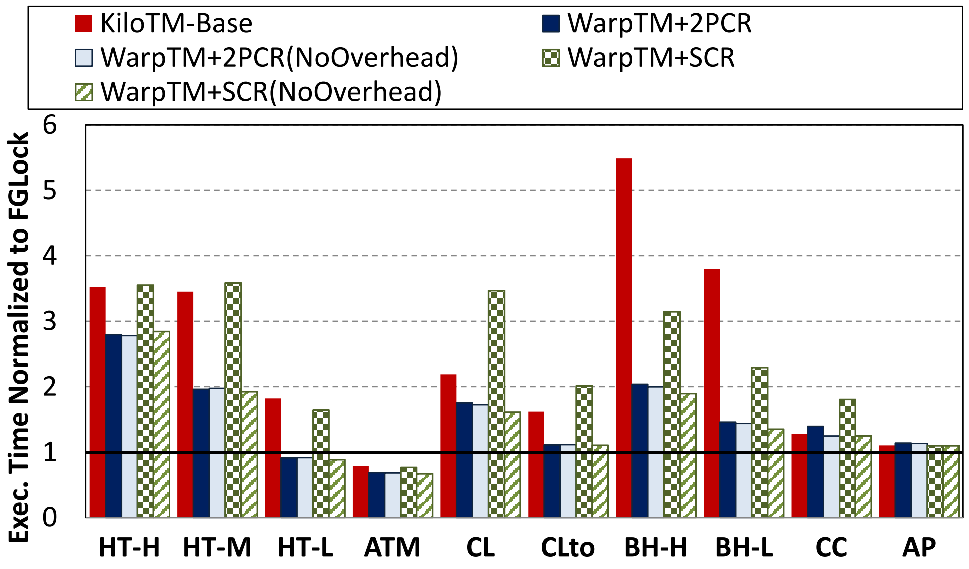


Figure 6.14: Comparison between different intra-warp conflict resolution mechanisms. SCR = Sequential Conflict Resolution. 2PCR = 2-Phase Parallel Conflict Resolution.

have shown that validation and commit coalescing can reduce the number of L2 cache accesses from the commit units by greater than 50% for both workloads (See Figure 6.5). The reason is that the serial hazard detection in each commit unit becomes a bottleneck. If the hazard detection hardware were running at twice the clock frequency, BH-H and BH-L would benefit from validation and commit coalescing. With all the optimizations enabled (including the overclocked hazard detection), WarpTM with ideal intra-warp conflict resolution can speed up Kilo TM by 49%.

While we could parallelize the hazard detection in the commit units with additional hardware, we find that temporal conflict detection removes this bottleneck in BH-H and BH-L by allowing most of their read-only transactions to commit silently.

6.7.4 Intra-Warp Conflict Resolution Overhead

Figure 6.14 compares performance of the two intra-warp conflict resolution mechanisms proposed in Section 6.3: sequential conflict resolution (SCR) in Section 6.3.2

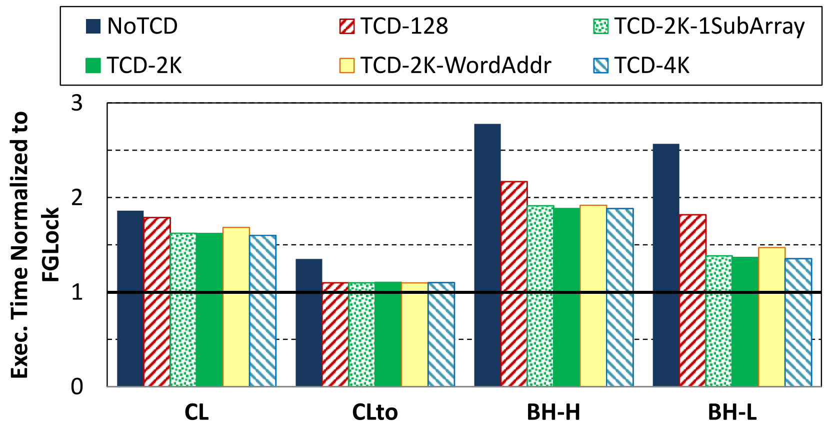


Figure 6.15: Performance of temporal conflict detection with different last written timetable organizations. Lower is better.

and 2-phase parallel conflict resolution (2PCR) in Section 6.3.3. We evaluate the performance of each mechanism with and without modeling the overhead of conflict resolution. In this study, we enable both WarpTM and TCD in configurations other than the baseline Kilo TM (KiloTM-Baseline). Without modeling the overhead (NoOverhead), each warp finishes intra-warp conflict resolution instantaneously, and does not generate traffic to the memory pipeline in the SIMT core when it traverses its logs or when it accesses the metadata in shared memory. This allows us to discern between two sources of performance overhead: transaction aborts due to inaccurate conflict resolution and the extra operations that implement the resolution itself. The no-overhead configurations of both SCR and 2PCR perform almost identically across all of our workloads, indicating that the accuracy of both mechanisms are roughly equivalent. However, the serial nature of SCR introduces significant overhead (an average 60% slowdown) to WarpTM, to the extent that most of its performance benefits are negated. 2PCR, on the other hand, can deliver similar accuracy as SCR with a much lower overhead (∼2% on average). The overhead in most cases is minor compared to the benefits from WarpTM, except for CC, where it causes 11% slowdown.

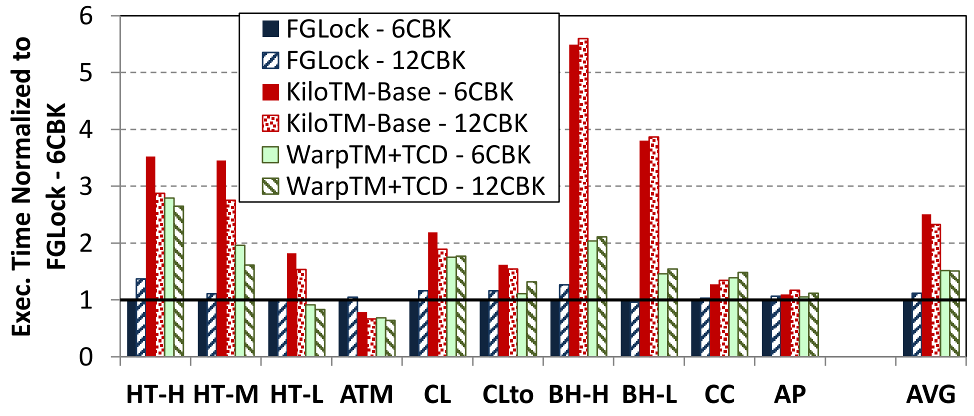


Figure 6.16: Performance impact with different L2 cache port widths. 6CBK = 6 L2 cache banks with 64-Byte ports. 12CBK = 12 L2 cache banks with 32-Byte ports. Lower is better.

6.7.5 Temporal Conflict Detection Resource Sensitivity

Figure 6.15 shows the performance sensitivity of TCD to different last written time table organizations. While our default configuration uses 2048 entries in each memory partition (TCD-2K), a lower cost organization using 128 entries (TCD-128) can capture a significant portion of the performance benefit from TCD-2K. Doubling the size of the last written time table (TCD-4K) shows no further improvement over TCD-2K, indicating that the 2048-entry table is sufficient. Even though the data shows that a single 2048-entry sub-array (TCD-2K-1SubArray) performs comparably to our default organization, we do notice having multiple sub-arrays can reduce the effect of aliasing as transaction concurrency increases. Finally, contrary to our intuition, we notice that reducing the detection granularity of TCD from 128-Byte blocks to 4-Byte words (TCD-2K-WordAddr), while keeping the same last written time table capacity, decreases performance. We believe that reducing the granularity causes more entries in the recency bloom filter to be populated and the aliasing effect dominates.

6.7.6 Sensitivity to L2 Cache Port Width

In this section, we study the performance impact of L2 cache port width on FGLock, the original KiloTM (KiloTM-Base), and the enhanced KiloTM with both WarpTM and TCD (WarpTM+TCD). The L2 cache in our baseline GPU architecture is partitioned into 6 banks, each with a 64-Byte port (6CBK). In this study, we further divide each L2 cache bank into two subbanks with a 32-Byte port, resulting in 12 L2 cache banks across the system (12CBK). Other parts of the GPU architecture, including the total L2 cache bandwidth, remain identical between the two configurations. Figure 6.16 compares the performance between these two configurations.

Overall, the FGLock workloads with 32-Byte ports run 12% slower than with 64-Byte ports. HT-H and BH-H suffer a higher degree of load imbalance among the L2 cache banks. The more congested L2 cache bank forms a tighter bottleneck with 32-Byte ports than with 64-Byte ports. While we did not observe this imbalance for CL and CLto, we did notice more atomic accesses from increased lock acquisition failures. The extra failures are caused by higher memory latency due to lower DRAM efficiency.

In this study, we also increased the number of commit units with the number of L2 cache banks due to their tightly-coupled design. The increased number of commit units can increase interconnection network traffic for Kilo TM, because each transaction needs to communicate with more commit units for validation and commit. We notice this extra traffic impacting the performance of Kilo TM for BH-H, BH-L, CC, and AP.

In other workloads, switching to 32-Byte ports improves performance for the original Kilo TM, because the commit units can use the port bandwidth more effectively with just scalar (4-Byte wide) accesses. In turn, the narrower L2 cache ports reduces the benefit provided by validation and commit coalescing. This lowers the average speedup of WarpTM over the original KiloTM from 43% to 40%. In particular, WarpTM does not speedup CLto at all. Nevertheless, in a GPU architecture with narrower L2 cache ports, KiloTM-Base obtains 49% of the FGLock performance. Kilo TM with WarpTM and TCD captures 76% of the FGLock performance, up from 66% with the wider L2 cache ports.

6.7.7 Sensitivity to Core Scaling

In this section, we explore the impact of increasing the number of SIMT cores on the overhead of Kilo TM. Specifically, we evaluate the performance of our work-

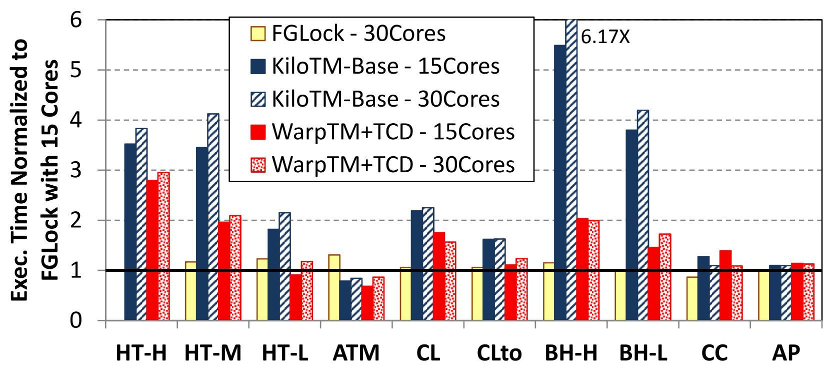


Figure 6.17: Performance impact from doubling the number of SIMT cores. Lower is better.

loads on a scaled up GPU architecture with 30 SIMT cores, doubled from our baseline configuration. Figure 6.17 compares the performance overhead of Kilo TM (and WarpTM+TCD) over FGLock with 15 and 30 SIMT cores. While doubling the SIMT cores increases concurrency in the GPU architecture, it slows down our FGLock applications by 9% on average. For this study, we have not scaled the memory subsystem with the core counts. The increased concurrency generates extra memory-level parallelism, but these extra concurrent memory accesses introduce more L2 cache misses, resulting in a net performance loss [132]. We also noticed more atomic accesses from increased lock acquisition failures in HT-M, CL, and BH-H. Only CC can take advantage of the extra cores for a 13% speedup. Moreover, the FGLock version of HT-H runs into a livelock, so its performance is not shown in Figure 6.17. In comparison, the Kilo TM version is only slowed down by 6-9% with 30 cores.

Aside from HT-H, Kilo TM and WarpTM+TCD mostly follow the performance trends of FGLock with the scaled up GPU architecture. For HT-M, HT-L and ATM, the overhead of Kilo TM and WarpTM over FGLock remains approximately the same with more cores. WarpTM works more effectively for CL with 30 cores because the extra cores provide extra L1 cache capacity for transaction logs, allowing extra transaction concurrency without the L1 cache overflow penalty. WarpTM+TCD works less effectively for CLto and BH-L with 30 cores, whereas enabling each enhancement alone with 30 cores is just as effective as with

15 cores. A detailed investigation reveals that enabling both enhancements with Kilo TM boosts the transaction concurrency in these workloads, generating significantly more transaction conflicts. CC and AP are not affected by the increased concurrency limit, and hence, the extra cores do not impact the overhead of Kilo TM and WarpTM+TCD for these two workloads.

6.8 Summary

In this chapter, we proposed two enhancements to Kilo TM, an existing hardware TM proposal for GPU architectures. *Warp-level transaction management* exploits the spatial locality among transactions within a warp to enable a set of optimizations. These optimizations allow Kilo TM to exploit the wide memory subsystem in GPU architectures. *Temporal conflict detection* complements WarpTM by allowing read-only transactions to commit silently in the absence of a conflict. Our evaluation shows that these two enhancements can improve Kilo TM performance by 65% while reducing its energy per operation by 34%. Kilo TM with the two enhancements outperforms coarse-grained locking by 192$\times$ and achieves 66% of the performance of fine-grained locking, while only requiring 34% more energy per operation. Moreover, software optimizations that reduce transaction footprints and contention can further close this gap.

While this chapter presents WarpTM and TCD as enhancements to Kilo TM, the insights behind these mechanisms extend well beyond GPU TM systems. WarpTM demonstrates the effectiveness of aggregating multiple transactions to amortize their management overheads in a TM system. This principle applies to other novel data synchronization/inter-thread communication mechanisms [66, 169]. The 2-phase parallel conflict resolution that enables WarpTM illustrates how transactions with predetermined order can resolve conflicts in parallel with low overhead. This insight may be readily applied to thread-level speculation on multi-core systems. We believe that TCD's ability to cheaply verify that a thread has observed an instantaneous global memory snapshot has wider uses beyond TM. For example, one may use TCD to accelerate runtime data-race detection on parallel computing systems without relying on any cache coherence protocol.

Finally, as newer commodity CMP systems start to add hardware support for

TM [70], more software developers will start using transactions in their applications. GPUs that support TM will have higher interoperability with these future software applications. This will be an important design consideration for future heterogeneous processors with tightly integrated CMP and GPU.

Chapter 7

Related Work

This chapter discusses related work for this dissertation. Section 7.1 discusses various proposals to improve the handling of branch divergence on GPUs. Section 7.2 discusses prior and concurrent transactional memory system proposals that are related to Kilo TM, as well as various microarchitecture/algorithmic techniques that are related to the implementation of Kilo TM.

7.1 Related Work for Branch Divergence Handling on GPUs

We classify the various proposals to improve the handling of GPU branch divergence into four categories: software compaction, hardware compaction, intra-warp divergent path management and adding MIMD capability. Some works contain improvements that capture aspects from multiple categories, and thus are mentioned multiple times.

7.1.1 Software Compaction

On existing GPUs, one way to improve SIMD efficiency of an application is through software compaction – using software to group threads/work items according to their control flow behavior. The regrouping involves moving the thread and its private data in memory, potentially introducing a significant memory bandwidth overhead. Below we highlight several works on software compaction that were published before thread block compaction.

Conditional streams [84] applies this concept to stream computing. It splits a compute kernel for stream processors with potentially divergent control flow into multiple kernels. At a divergent branch, a kernel splits its data stream into multiple streams according to branch outcome of each data element. Each stream is then processed by a separate kernel, and merges back at the end of the control flow divergence.

Billeter et al. [14] proposed to use a parallel prefix sum to implement SIMD *stream compaction*. The stream compaction reorganizes streams of elements with assorted tasks into compact substreams of identical tasks. This implementation leverages the access flexibility of the GPU on-chip scratchpad to achieve high efficiency. Hoberock et al. [76] proposes a deferred shading technique for ray tracing that uses stream compaction to improve the SIMD efficiency of pixel shading in a complex scene with many material classes. Each material class requires its unique computation. A pixel shader combining the computation for every material class runs inefficiently on GPUs. Stream compaction groups the rays hitting objects with similar material classes, allowing the GPU SIMD hardware to execute the shader for these pixels efficiently.

Zhang et al. [172] proposes a runtime system that remaps thread into different warps on the fly to improve SIMD efficiency as well as memory access spatial locality. The runtime system features a pipelined system, with the CPU performing the on-the-fly remapping and the GPU performing computations on the remapped data/threads.

7.1.2 Hardware Compaction

Similar to thread block compaction (proposed in Chapter 3) and its precursor, dynamic warp formation [56], many proposals uses hardware to compact loosely populated warps to improve the performance of GPU applications that suffers from branch divergence. Unlike software compaction, which takes place in the global memory, the hardware compaction proposals usually takes place locally within a SIMT core. Since the compacted threads all located on the same core sharing the same register file, it is possible to perform compaction without moving their architectural states with a more flexible register file design [56]. Among the hardware compaction proposals discussed below, only dynamic micro-kernels proposed by Steffen and Zambreno [150] was published before TBC; the other proposals were published after TBC.

Steffen and Zambreno [150] improve SIMD efficiency of ray tracing on GPUs with *dynamic micro-kernels*. The programmer is given primitives to break iterations in a data-dependent loop into successive micro-kernel launches. This decomposition by itself does not improve parallelism, because each iteration depends on data from the previous iteration. Instead, the launch mechanism improves the load imbalance between different threads in the same core by compacting the remaining active threads into few warps. It also differs from the rest of the hardware compaction techniques (including TBC) in that the compaction migrates the threads with their architectural states, using the per-core scratchpad memory as a staging area.

Published after TBC, the large warp microarchitecture [117] extends the SIMT stack, similar to TBC, to manage the reconvergence of a group of warps. However, instead of restricting the compaction at branches and reconvergence points. LWM requires warps within the group to execute in complete lockstep, so that it can compact the group at every instruction. This reduces the available TLP even more so than TBC, but allows LWM to perform compaction with predicated instructions as well as unconditional jumps. Similar to TBC, LWM splits warps running on the same core into multiple groups, and restricts compaction to occur only within a group. It also opts for a more complex scoreboard microarchitecture that tracks register dependency at thread-granularity. This allows some warps in the group to execute slightly ahead of others compensate the lost TLP due to lockstep execution.

Rhu and Erez [130] extend TBC with a compaction-adequacy predictor (CAPRI). The predictor identifies the effectiveness of compacting threads into few warps at each branch, and only synchronizes the threads at branches where the compaction is predicted to yield a benefit. This reclaims the TLP lost due to non-beneficial stall and compaction with TBC. Rhu and Erez [130] also show that a simple history-based predictor similar to a single-level branch predictor is sufficient to achieve high accuracy.

Vaidya et al. [159] propose a low-complexity compaction technique that benefits wide SIMD execution groups that executes multiple cycle on narrower hardware units. Their basic technique divides a single execution group into multiple subgroups that match the hardware width. SIMD execution group that suffers from divergence can run faster on the narrow hardware by skipping subgroups that are completely idle. To create more completely idle subgroups, they propose a swizzle mechanism that compacts elements into fewer subgroups at divergence.

Brunie et al. [21] propose simultaneous branch and warp interweaving (SBI and SWI). They extend the GPU SIMT front-end to support issuing two different instructions per cycle. They compensate this increased complexity by widening the warp to twice its original size. SWI co-issues an instruction from a warp suffering from divergence with instructions from another diverged warp to fill the gaps left by branch divergence.

7.1.3 Intra-Warp Divergent Path Management

While a SIMT stack with immediate post-dominator reconvergence points can handle branch divergence with arbitrary control flow, it can be further improved in various aspects, such as the likely-convergence points proposed in Chapter 3. Below we highlight several works that attempt to improve the SIMT stack. Among these works, only dynamic warp subdivision proposed by Meng et al. [107] was published before likely-convergence points; the other proposals were published afterwards.

Meng et al. [107] propose *dynamic warp subdivision* (DWS), which extends the SIMT stack with a warp-split table to subdivide a diverged warp into concurrent warp-splits. The warp-splits, each executing a divergent branch target, can execute in parallel to reclaim hardware idleness due to memory accesses. Warp-splits are also created at memory divergences – when only a portion of the threads in a warp hit in the L1 data cache. Instead of waiting for all threads to obtain their data, DWS split the warp and allow the warp-split that hits in the cache to execute ahead, potentially prefetching data for those who have missed the cache. DWS is orthogonal TBC. The block-wide SIMT stack in TBC can be extended with DWS to boost the available TLP.

Diamos et al. [43] depart from the SIMT stack altogether and instead propose to reconverged threads after divergence via thread frontiers. A compiler supporting thread frontiers sorts the basic blocks in a kernel according to their topological order. In this way, threads executing at an instruction at a higher PC can never jump to an instruction at a lower PC. Loops are handled by placing the loop exit at the end of the loop body. With this sorted code layout, a diverged warp will eventually reconverge by prioritizing threads with lower PCs (allowing them to catch up). Compared to SIMT stacks with immediate post-dominator reconvergence, reconvergence via thread frontiers yields higher SIMD efficiency for applications with unstructured control flow. The evaluation semantics of multi-expression conditional statements and the use of exceptions can both generate code with unstructured control flow. SIMT stacks extended with likely-convergence points can yield similar SIMD efficiency improvement on applications with unstructured control flow; however, each entry in the SIMT stack may only have a finite number of likely-convergence points, whereas the thread frontier approach has no such restriction.

Rhu and Erez [131] propose dual-path execution model, which addresses some of the implementation shortcomings of DWS by restricting each warp to execute only two concurrent warp-splits. They also extend the scoreboard to track the register dependency of each warp-split independently. This allows dual-path execution model to achieve greater TLP than DWS.

ElTantaway et al. [49] remove the dual-path limitation with a multi-path execution model. They further extend the multi-path execution hardware with opportunistic early reconvergence, boosting its SIMD efficiency for unstructured control flow without using sorted code layout as in thread frontiers.

7.1.4 Adding MIMD Capability

The following proposals improve GPUs' compatibility with divergent control flow by incorporating some limited amount of MIMD capability.

Published before both dynamic warp formation and thread block compaction, vector-thread (VT) architecture [91] is an architecture that combines aspects of both SIMD and MIMD architectures, with the goal of capturing the best of both approaches. A VT architecture features a set of lanes that are connected to a common L1 instruction cache. In SIMD mode, all lanes receive instructions directly from the L1 instruction cache for lockstep execution, but each lane may switch to a MIMD model, running at its own pace with instructions from its L0 cache. A recent comparison with traditional SIMT architectures (e.g. GPUs) by Lee et al. [96] shows that VT architectures have comparable efficiency with regular parallel applications, while performing much more efficiently with irregular parallel applications.

Published after thread block compaction, Temporal SIMT [85, 92] permits each lane to execute in MIMD fashion, similar to VT architecture. However, instead of running a warp across all lanes in lockstep, it time-multiplexes the execution of a warp through a single lane, and each lane runs a separate set warp. Temporal SIMT achieves the efficiency of SIMD hardware by fetching each instruction only once for the whole warp. This amortizes the control flow overhead across time, while the traditional SIMD architecture amortizes the same overhead across multiple lanes in space.

Brunie et al. [21] propose simultaneous branch and warp interweaving (SBI and SWI) after the publication of thread block compaction. They extend the GPU SIMT front-end to support issuing two different instructions per cycle. SBI co-issues instructions from the same warp when it encounters a branch divergence. Executing both targets of a divergence branch at the same time eliminates its performance penalty significantly.

7.2 Related Work for Kilo TM

In this section, we discuss the related work for Kilo TM. We classify the work based on the microarchitecture mechanisms of interest to Kilo TM.

7.2.1 GPU Software Transactional Memory

Other than Kilo TM, there are other proposals to support transactional memory on GPU architectures. These proposals implement software TM systems designed to work with existing GPUs.

Cederman et al [25] proposed a GPU software TM system that uses per-object version locks to detect conflicts. As each transaction executes, it records the version of the object it access. These version numbers are later checked during commit to detect conflicts with committed transactions. In their evaluations, while STM-based data structures scale well, they perform about 10$\times$ slower than lock-free data structures. Kilo TM uses value-based conflict detection to remove the storage overhead for version locks. With dedicated hardware to increase commit parallelism and limit concurrency, Kilo TM performs much closer to fine-grained locking (which should perform on-par with lock-free data structures).

Xu et al. [167] proposed a GPU software TM system after the publication of Kilo TM. Similar to Kilo TM, it also uses value-based conflict detection. For each transaction, it has a hash table to store the set of memory locations that needs to be locked during commit. The memory locations stored in this table are sorted, so that every transaction acquires its lock in the same order to prevent deadlocks. Without the extended SIMT stack in Kilo TM that handles the control flow divergence due to transaction aborts (see Section 4.2.1), programmers using this STM system need to handle the divergence explicitly – a significant programming burden. In their evaluations, this STM system performs up to 20$\times$ faster than coarse-grained locks. Kilo TM outperforms coarse-grained locks by 192$\times$ on average.

7.2.2 Hardware Transaction Memory with Cache Coherence Protocol

Many existing hardware and hybrid TM systems focus on leveraging the sharer/-modifier information maintained by cache coherency hardware for conflict detection and using thread-private caches for version management. Some of the HTMs propose to extend the cache coherence protocol with new coherence states for conflict detection and version management [27, 29, 44, 157]. The additional states extend the thread-private cache into a transaction cache, buffering the read-set and write-set accessed by a transaction. A conflict is detected when a cache line with transactional data receives an invalidation/read sharing request in the cache coherence protocol. Other HTMs just monitor the cache coherency traffic (of an existing protocol) for conflict detection, using per-transaction metadata stored in signatures [24, 111, 170], or extra bits added to the cache line [19, 113, 136].

7.2.3 Signature-Based Conflict Detection

Many hardware transactional memory (HTM) systems use signatures to create compressed representations of transaction footprints, with the aim to support unbounded transactions. In BulkTM [26], a committing transaction broadcasts its write-set in a bloom filter based signature, which is compared for conflicts against the signature and the L1 cache tags at each recipient transaction. SigTM [111] and LogTM-SE [170] eagerly detect conflicts with signatures by monitoring the address of each incoming cache coherence request. A software conflict resolution handler is invoked when the address hits the signatures. Other HTMs (TMACC [24], FlexTM [145]) use signatures to handle unbounded transactions that have exceeded L1 cache capacity.

Software transactional memory (STM) systems also use signatures for conflict detection. RingSTM [149] uses a ring of commit records that hold the write-signatures of recently committed transactions. Before each transactional load, the transaction compares its read-signature against the write-signatures of newly committed transactions added to the commit records since the transaction's last load. A match indicates a new conflict and the transaction is aborted. In InvalSTM [67], a committing transaction compares its write-signature against the read- and write-signatures of other running transactions. A contention manager is invoked upon a match.

In this work, we have investigated the viability of signature-based conflict detection for GPU TM workloads and find that they are ill-suited for representing thousands of small transactions. We do propose having each transactional read check a bloom filter for the presence of buffered transactional data in the write-log.

7.2.4 Value-Based Conflict Detection

Other than Kilo TM, there are a number of TM systems using value-based conflict detection. JudoSTM [124] allows parallel commits with a set of versioned locks each guarding a memory region. Each transaction checks/acquires the locks of all the regions that require protection during validation and commit. NORec [37] uses a single global versioned lock to offer fast checking with a small number of concurrent threads. To ensure opacity [68], an incremented global lock version informs a transaction to revalidate its read-set before reading from memory again. DPTM [153] is a cache coherence protocol based HTM. It uses value-based conflict detection to mitigate the false conflicts that are caused by false sharing of cache lines between transactions. Kilo TM uses value-based conflict detection, but the motivation is to eliminate global metadata for conflict detection with thousands of concurrent transactions. It uses special hardware to allow non-conflicting transactions to validate and commit in parallel.

7.2.5 Ring-Based Commit Ordering

Kilo TM and RingSTM [149] are conceptually similar in that both use a ring to order transaction commits and detect conflicts between read-set of a transaction and write-sets of committed transactions. Kilo TM uses value-based conflict detection to eliminate storage for committed write-sets, and uses a LWH unit to detect hazards (conflicts) among committing transactions. Each transaction in Kilo TM stores its read-set in exact address-value pairs. It also features multiple commit units, each with a separate ring, and maintains consistent commit order via a protocol similar to Scalable TCC [27]. Kilo TM enforces opacity via a watchdog timer, removing the need to validate before every transactional read.

7.2.6 Recency Bloom Filter

Recency bloom filters contribute greatly in boosting the commit parallelism in Kilo TM and allow hazard detection in each commit unit to directly support unbounded transaction. Recency bloom filters were proposed in Chapter 4 and used in both Chapter 4 and Chapter 6.

While the construction of a recency bloom filter resembles a time-out bloom filter [90], the recency bloom filter differs in how it uses the data stored inside the filter. The time-out bloom filter was proposed for use in network packet sampling, where the key result is the existence of a similar packet occurring within a given window. In contrast, the recency bloom filter provides more information. In the presence of a false conflict due to aliasing, it can still return the approximate identity of the conflict rather than simply a pass/fail result. Kilo TM can still use this approximate identity to determine the set of transactions that may commit in parallel with the transaction in query.

The store sequence bloom filter (SSBF) in store vulnerability window (SVW) [134] also uses a hash table of order numbers to approximately detect conflicts between concurrent memory operations. The recency bloom filter extends this construction to use multiple hash tables of order number and uses a minimum logic block to combine the order number returned from each table to provide the best approximation. This extension improves the recency bloom filter's tolerance of the effects of aliasing in hash functions and allows the filter to scale to much larger item sets. Also, a recency bloom filter assigns the same commit ID (order number) to represent multiple memory locations in a transaction, whereas a SSBF assigns a unique order number to each memory operation.

7.2.7 Transaction Scheduling

In Chapter 4, we describe an extension to the GPU hardware thread scheduler to control the number of concurrent transactions. It is effective for high-contention workloads. Others have proposed more adaptive transaction schedulers that dynamically adjust concurrency according to predicted contention or to support unbounded transaction. Yoo and Lee [171] use a system-wide queue to control the number of concurrent transactions based on a system-wide conflict rate threshold. CAR-STM [47] maintains per-core scheduling queues to serially execute transactions that are likely to conflict. Maldonado et al. [105] extend the OS thread scheduler to implement a similar capability. Both Blake et al. [15] and Dragojević et al. [48] propose mechanisms to predict transactions that are likely to conflict via runtime profiling. OneTM [17] simplifies the handling of unbounded transactions by serializing the execution of these transactions. We leave the exploration of such adaptive transaction schedulers on GPUs as future work.

7.2.8 Energy Analysis for Transaction Memory

Ferri et al. [52] analyzed the energy and performance of SoC-TM, their TM system proposal for embedded multi-core SoC. Their analysis shows that for workloads that scale to multiple threads, SoC-TM performs better than locking while consuming less energy. We perform similar analysis for Kilo TM, a TM proposal for

GPU architectures.

7.2.9 Intra-Warp Conflict Resolution

Similar to the intra-warp conflict resolution mechanisms explored in Chapter 6, Qian et al. [128] described a method to detect and resolve conflicts among threads running on SMT CPU cores, which usually have far fewer threads per core in comparison to GPU cores. The relatively small number of threads allows their design to dedicate explicit storage to record the dependency between transactions and extend each cache line to record read-sharer information. Such storage is impractical for GPU cores, which have hundreds of threads on each core sharing the L1 cache. This work focuses on detecting and resolving conflicts among transactions within a warp on GPU.

Yang et al. [168] proposed multiplexing shared memory storage among multiple concurrently running thread blocks by dynamically allocating the storage to each thread block for temporary use and freeing it immediately after. Our proposed intra-warp conflict resolution employs this strategy to allow each warp to use a ownership table larger than the capacity possible with static allocation.

Nasre et al. [118] proposed a probabilistic 3-phase conflict resolution that uses parallel passes to resolve conflicts among multiple threads. Similar to our proposed intra-warp 2-phase parallel conflict resolution in Chapter 6, their approach uses thread ID to prioritize among different threads. However, their approach focuses on obtaining exclusive access to modify shared data and does not permit read-sharing, which is key to TM system performance.

7.2.10 Globally Synchronized Timers in Memory Systems

Temporal Coherence [146] proposed by Singh et al., is a cache coherence framework for GPU architectures that uses a set of globally synchronized timers to eliminate invalidation messages. It uses timestamps to determine when cache blocks in local data caches will expire. Temporal conflict detection uses timestamps to detect if all the values read by a transaction can exist as a global memory snapshot. Ruan et al. [135] also proposed extending Orec-based STM systems with synchronous hardware timers found on existing CMP systems. Their approach embeds timestamps in the ownership record of each transaction variable in the main memory, whereas TCD uses a set of small on-chip buffers to conservatively record when each word in the global memory space was last written.

7.2.11 Timestamp/Counter-based Conflict Detection

Many existing STMs use timestamps for conflict detection. We highlight two common use of timestamps as follows.

Spear et al. [148] propose using a global commit counter to represent the global memory version. The counter is incremented by every transaction commit. Transactions can check this counter to see if any transactions have committed between now and last around of validation. When a transaction observes an incremented counter, it performs another around of validation and buffers the updated counter value. If the counter remained unchanged, the transaction proceeds directly, skipping an expensive around of validation.

Lev and Moir [99] propose using a counter at each software object to represent the number of read-sharers to the object. Each transaction increments the counter as it reads the object and decrements it as the transaction commits. A transaction writing to the object can use this counter to detect conflicts – conflict exists if transaction is writing to an object with one or more read-sharers.

Notice that in both uses, the counters are updated explicitly by events. Temporal conflict detection (TCD) uses free-running timers that increment every cycle. Instead of using the timestamp to represent memory versions, TCD uses the timestamps representing the time when a set of memory locations were last written to determine the consistency of a read-only transaction.

7.2.12 Transactional Memory Verification

Concurrent to the development of the proof presented in Chapter 5, Lesani et al. [98] completed a machine proof framework to verify the correctness of a TM system design, and used to verify the design of NORec [37]. The framework uses I/O automata to specify the behavior of a naive TM system known to behave correctly, and to model the behavior of the TM system under test. They then use the PVS prover [125] to check the equivalence of the naive TM system and the tested TM system. The equivalence indicates that the TM system design correctly supports serializability.

Chapter 8

Conclusions and Future Work

This chapter concludes this dissertation and provides directions for future work.

8.1 Conclusions

The slowdown of Dennard Scaling has motivated the computing industry to look for more efficient alternatives in the forms of GPU computing. GPU features a highly parallel architecture that is designed to exploit fine-grained data-level-parallelism present in graphics rendering. While advances in GPU computing APIs such as CUDA and OpenCL had soothed many system-level challenges in using GPU as a computing device, GPU computing remains challenging.

To fully harness the computing power of GPUs, software developers need to decompose the implicit DLP in their applications into thousands of threads. This decomposition is relatively easy for applications with regular parallelism; decomposing the parallel computations in applications with irregular parallelism is much more challenging:

1. The microarchitectural behavior of an application with irregular parallelism is often sensitive to the application's input. This sensitivity can introduce uncertainty to the performance/energy benefit from GPU acceleration.

2. Applications with irregular parallelism often feature irregular, fine-grained communications between threads. Managing this communication with fine-grained locking (or other *ad hoc* synchronization solutions) is challenging

even for parallel programming experts. The development effort required to verify the use of fine-grained locking (or these *ad hoc* solutions) is highly unpredictable.

3. The resulting software has many hidden constraints that only expert programmers can understand. This makes maintenance and long-term development very expensive.

These challenges plague the development of GPU applications with irregular parallelism. Although many research studies have indicated the performance potential of GPU acceleration for these applications, software developers appear to show limited interest likely due to the risk involved in the development.

In this dissertation, we proposed two enhancements to the existing GPU architectures for reducing the risk in developing GPU applications with irregular parallelism: Thread block compaction (TBC) and Kilo TM.

Thread block compaction (Chapter 3) is a novel microarchitecture mechanism that improves the performance of GPU applications that suffer from branch divergence. TBC addresses the pathologies found in its precursor, dynamic warp formation [56]. It draws from our observation that most applications alternate between code regions that suffer from branch divergence and regions that do not. This observation had driven us to expand the per-warp SIMT stack in current GPU architecture to encompass the whole thread block. This block-wide stack tracks the entry and exit of the divergence regions, and ensuring that dynamic, compacted warps are restored to their aligned static warps at the exit. With simplified DWF hardware which restricts compaction are the entry to the divergence region, TBC can robustly speedup GPU applications that suffer from branch divergence. Our evaluation showed that it speeds up a set of divergent GPU applications by 22% on average, and maintains the performance of the GPU applications free of branch divergence.

Kilo TM (Chapter 4) is our proposal to support transactional memory (TM) on GPU. The TM programming model simplifies parallel programming replacing lock-protected critical regions with atomic code region called transactions. The programmer is only responsible for specifying the operations, the functionality, inside each transaction; an underlying TM system runs multiple transactions concurrently for performance, and automatically handles any data-race raised from the concurrent execution. This clean separation of functionality and performance makes TM software more maintainable than lock-based code.

Unlike existing TM proposals that focused on supporting tens of large-footprint transactions on existing CMP systems, Kilo TM aims to support thousands of concurrent small transactions. This focus, as well as the throughput-focused design of GPUs, had driven us towards a very different design space, with a strong emphasis on scalability. The initial design of Kilo TM used value-based conflict detection, which rereads data values from global memory to detect conflicts. It does not require any global metadata to work and eliminates all direct communication between transactions. However, a naive implementation requires serializing all transaction commits. In response, we extended Kilo TM with specialized hardware to boost the commit parallelism. Later, we further augmented Kilo TM with warp-level transaction management system to capture spatial locality between transactions; we proposed a low-overhead intra-warp conflict resolution mechanism to make this possible. Finally, we accelerated the execution of read-only transactions on Kilo TM with temporal conflict detection. In our evaluations via simulations with GPGPU-Sim [10] and GPUWattch [97], GPU TM applications with the enhanced version of Kilo TM overall runs $192\times$ faster than their coarse-grained locking analogs. This enhanced Kilo TM also captures 66% of fine-grained locking performance, while consuming only 34% more energy. If we only count the contributions from low-contention GPU TM workloads, Kilo TM performs on par with fine-grained locking in both execution speed and energy efficiency. This type of workload is what we envision to be found in future GPU TM applications.

We opted not to evaluate a combination of TBC and Kilo TM in this work. This is because we believe both solutions are still in their infancy. This is especially true for Kilo TM, which is our first attempt to design a TM system for GPUs. Many assumptions of the design came from a set of primitive GPU TM workloads we created ourselves. Many of these assumptions could be refined through the development of more realistic GPU TM workloads in the future.

8.2 Directions for Future Work

This section provides some directions for future research projects that extend the scope of the work presented in this dissertation.

8.2.1 Cost-Benefit-Aware Multi-Scope Compaction

One observation we have with the current techniques of compacting sparsely populated warps (due to divergence) into denser populated warps is that each technique employs only one way to compact the warps. In general, compaction that involves a larger scope of threads (within a thread block $\rightarrow$ within a core $\rightarrow$ the entire GPU) generally produces more benefit, but incurs a higher cost as well due to the communication and data migration overhead involved.

Through a model that estimates the cost and benefit trade offs behind each scope of compaction, a hybrid system can dynamically decide which scope of compaction should be applied. This hybrid may have the potential of capturing the combined benefits of all forms of compaction.

8.2.2 Extend Kilo TM to Support Strong Isolation

The current implementation of Kilo TM only supports weak isolation [16]. With weak isolation, transactions only appear atomic to other transactional memory accesses. Non-transactional memory accesses can observe the intermediate memory states in between transaction commits. Supporting only weak isolation simplifies the design of Kilo TM (or at least reduces its performance overhead), but it generally makes programming TM harder for reasons explained in Section 2.2.2.

To support strong isolation, Kilo TM will need to be extended to handle the interactions of non-transactional loads and stores that may conflict with a concurrent transaction. The goal is to avoid adding significant performance and energy overhead for non-transactional memory accesses. As clarified by Dalessandro and Scott [35], a TM system may handle non-transactional loads and stores separately while satisfying the requirement for strong isolation.

A naive approach to order non-transactional loads with respect to other transactions is to treat each load as a read-only transaction in Kilo TM, and to commit this read-only transaction via the commit unit in the corresponding memory partition to obtain its value. In this way, transactions that appear to be committed to an earlier non-transactional load also appears committed for a later non-transactional load from the same thread. This property can be maintained by simply returning the retired commit ID (RCID) from the memory partition that serviced the non-transactional load. If a later load from the same thread obtains an older retired commit ID than the earlier load, the later load is replayed to reread the memory until the RCID is updated. Notice that temporal conflict detection cannot be used in this case because it does not restrict order between read-only transactions (fine for weak isolation, but breaks strong isolation).

While Kilo TM should detect most conflicts caused by non-transactional stores with value-based conflict detection, extra effort is required to cover all cases. One corner case occurs when the store modifies a memory location that is read, modified and written by a committing transaction. The non-transactional store may occur right in between the validation and commit of a transaction, violating the atomicity of the transaction.

One approach to handle this corner case is to use a separate recency bloom filter to keep track of the read-set of all committing transactions. Every non-transactional store checks this filter at the memory partition, and if it hits in the filter, it performs the store through the commit unit as a write-only transaction.

8.2.3 Application-Driven Transaction Scheduling

In some GPU TM applications, we have observed that the level of contention between transactions changes dynamically at runtime. For example, BH starts out as a high-contention application where every transaction is trying to insert a leaf node near the root, and gradually relaxes into low-contention as the octree grows. A more intelligent concurrency control should be able to speed up this application significantly by restricting the concurrent only during the high-contention phase.

Instead of approaching this by just adopting one of the existing transaction schedulers, a limit study should be performed to estimate the potential gain. One can try to construct an oracle transaction schedule for each application from memory traces obtained via profiling. This idealized transaction schedule will serve as the gold standard. One way to approach this ideal schedule is to analyze the runtime input data prior to the transaction executions. We envision that some form of approximate graph coloring may be able to recreate a transaction schedule that approximates the idealized schedule. The key is that since the computation is still done via transactions, the schedule just needs to remove a majority of the transaction aborts, but does not need to be so conservative such that it culls away available thread-level-parallelism. The generated schedule can be implemented by using a software thread spawner similar to the ones used for dynamic load-balancing in ray tracing on GPUs [4].

8.2.4 Multithreaded Transactions

During the development of the various GPU TM workloads for this work, we have observed that some transactions, especially the larger ones, contain a good amount of parallelism (e.g., initializing entries in a node or updating all of its neighbours). With the current design of Kilo TM, this parallelism remains buried inside a single thread, because each transaction may only be executed by a single thread.

This observation made us realized that the current TM semantics need to be extended to support *multithreaded transactions*. This new TM would not bind the transaction context to a single thread. Instead, it would allow multiple threads to execute a transaction collaboratively. These threads should observe the intermediate work by each other within a transaction.

The implementation of this new TM model will need to address the following two challenges:

- How to enable multiple threads working on the same transaction to see each other's intermediate work? Should this be allowed at all?
- How does the new TM system detect conflicts between transactions of different scale? Can a single conflict detection system efficiently handle both single-thread transaction and thousand-threads transaction?

We believe that ideas proposed in this dissertation, such as warp-level transaction management and temporal conflict detection, can be further developed to overcome these challenges.

Bibliography

[1] NVIDIA Forums - atomicCAS does NOT seem to work. http://forums.nvidia.com/index.php?showtopic=98444. Accessed: March 15, 2011. → pages 77

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 185–196, New York, NY, USA, 2009. ACM. → pages 31

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Advances in Knowledge Discovery and Data Mining. chapter Fast Discovery of Association Rules. American Association for Artificial Intelligence, 1996. → pages 92, 94, 145

[4] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *HPG '09*, 2009. → pages 7, 15, 16, 63, 64, 179

[5] *R700-Family Instruction Set Architecture*. AMD, March 2009. → pages 41, 60

[6] *AMD Southern Islands Series Instruction Set Architecture*. AMD, 1.1 edition, December 2012. → pages 41

[7] ARM Holdings. Cortex-A9 NEON Media Processing Engine Technical Reference Manual (Revision r2p2), 2008. → pages 22

[8] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2007, pages 1–10. IEEE, 2007. → pages 6, 76

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. → pages 1

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE Symposium of Performance and Analysis of Systems and Software*, ISPASS'09, pages 163–174, 2009. → pages 63, 64, 92, 143, 176

[11] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM. → pages 55

[12] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 119–130, New York, NY, USA, 2014. ACM. → pages 55

[13] N. Bell and M. Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM. → pages 16

[14] M. Billeter, O. Olsson, and U. Assarsson. Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM. → pages 162

[15] G. Blake, R. G. Dreslinski, and T. Mudge. Bloom Filter Guided Transaction Scheduling. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 75–86, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9432-3. → pages 92, 108, 170

[16] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *the Annual Workshop on Deplicating, Deconstructing, and Debunking*, WDDD, 2005. → pages 10, 30, 31, 112, 113, 177

[17] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 24–34, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi:10.1145/1250662.1250667. → pages 34, 170

[18] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007. → pages 28, 32, 87

[19] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 127–138, Washington, DC, USA, 2008. IEEE Computer Society. → pages 167

[20] A. Brownsword. Cloth in OpenCL, 2009. → pages 92, 93, 145

[21] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 49–60, Washington, DC, USA, 2012. IEEE Computer Society. → pages 164, 166

[22] I. A. Buck, J. R. Nickolls, M. C. Shebanow, and L. S. Nyland. United States Patent #7,627,723: Atomic Memory Operators in a Parallel Processor (Assignee NVIDIA Corp.), December 2009. → pages 19, 44

[23] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. *Chapter 6 in GPU Computing Gems Emerald Edition*, 2011. → pages 7, 16, 76, 92, 93, 145

[24] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS XVI, pages 27–38, New York, NY, USA, 2011. ACM. → pages 167, 168

[25] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a Software Transactional Memory for Graphics Processors. In *EGPGV*, 2010. → pages 166

[26] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society. → pages 32, 33, 34, 80, 103, 133, 134, 168

[27] H. Chafi, J. Casper, B. D. Carlstrom, A. Mcdonald, C. Cao, M. W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society. → pages 9, 33, 34, 75, 81, 88, 167, 169

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. → pages 63, 64

[29] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi:10.1109/MICRO.2010.40. → pages 167

[30] B. W. Coon and J. E. Lindholm. United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008. → pages 39, 41, 60, 61

[31] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. United States Patent #7,434,032: Tracking Register Usage During Multithreaded Processing Using a Scorebard having Separate Memory Regions and Storing Sequential Register Size Indicators (Assignee NVIDIA Corp.), October 2008. → pages 39, 40

[32] B. W. Coon, J. R. Nickolls, L. S. Nyland, and P. C. Mills. United States Patent #8,375,176 B2: Lock Mechanism to Enabled Atomic Updates to Shared Memory (Assignee NVIDIA Corp.), December 2009. → pages 135

[33] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, October 1971. → pages 21

[34] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997. → pages 18, 19

[35] L. Dalessandro and M. L. Scott. Strong Isolation is a Weak Idea. In *Workshop on Transactional Memory*, TRANSACT'09, 2009. → pages 31, 177

[36] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 171–180, New York, NY, USA, 2012. ACM. → pages 30

[37] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM. → pages 9, 75, 81, 84, 87, 109, 125, 139, 142, 168, 172

[38] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004. → pages 94

[39] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM. → pages 34

[40] M. de Kruijf and K. Sankaralingam. Idempotent Processor Architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 140–151, New York, NY, USA, 2011. ACM. → pages 18, 85

[41] M. de Kruijf and K. Sankaralingam. Idempotent Code Generation: Implementation, Analysis, and Evaluation. In *Proceedings of the 2013*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO), CGO ’13, pages 1–12, Washington, DC, USA, Feb 2013. IEEE Computer Society. → pages 85

[42] R. H. Dennard, F. H. Gaensslen, and K. Mai. Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, October 1974. → pages 1

[43] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-convergence at Thread Frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 477–488, New York, NY, USA, 2011. ACM. → pages 165

[44] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience With a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi:10.1145/1508244.1508263. → pages 167

[45] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical report, Mountain View, CA, USA, 2009. → pages 34

[46] S. Doherty et al. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’04, pages 216–224, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi:10.1145/1007912.1007945. → pages 113

[47] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC ’08, pages 125–134, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-989-0. doi:10.1145/1400751.1400769. → pages 170

[48] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC

’09, pages 7–16, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-396-9. doi:10.1145/1582716.1582725. → pages 170

[49] A. ElTantaway, J. W. Ma, M. O’Connor, and T. M. Aamodt. A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA ’14, Washington, DC, USA, 2013. IEEE Computer Society. → pages 165

[50] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA ’11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi:10.1145/2000064.2000108. → pages 1

[51] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA ’11, pages 169–180, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9432-3. → pages 80

[52] C. Ferri, A. Marongiu, B. Lipton, R. I. Bahar, T. Moreshet, L. Benini, and M. Herlihy. SoC-TM: Integrated HW/SW Support for Transactional Memory Programming on Embedded MPSoCs. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’11, pages 39–48, New York, NY, USA, 2011. ACM. → pages 170

[53] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966. → pages 21, 22

[54] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA ’11, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society. → pages 9, 47

[55] W. W. L. Fung and T. M. Aamodt. Energy Efficient GPU Transactional Memory via Space-time Optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 408–420, New York, NY, USA, 2013. ACM. → pages 124

[56] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society. → pages 3, 7, 9, 41, 45, 46, 50, 51, 52, 58, 60, 65, 69, 72, 73, 94, 144, 162, 163, 175

[57] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 296–307, New York, NY, USA, 2011. ACM. → pages 9, 74, 115

[58] W. W. L. Fung, I. Singh, and T. M. Aamodt. KILO TM Correctness: ABA Tolerance and Validation-Commit Indivisibility. Technical report, University of British Columbia, 2012. http://www.ece.ubc.ca/~aamodt/papers/wwlfung.tr2012.pdf. → pages 109

[59] W. Fung et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):7:1–7:37, 2009. ISSN 1544-3566. → pages 41, 45, 46, 50, 51, 60

[60] W. W. L. Fung et al. http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpusim-micro2013.tgz, . Accessed: November 5, 2013. → pages 144

[61] W. W. L. Fung et al. http://www.ece.ubc.ca/~wwlfung/code/gpu-tm-tests.tgz, . Accessed: April 18, 2013. → pages 95

[62] W. W. L. Fung et al. http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpu_sim.tgz, . Accessed: April 18, 2013. → pages 95

[63] W. W. L. Fung et al. http://www.ece.ubc.ca/~wwlfung/code/tbc-gpgpusim.tgz, . Accessed: Feburary 19, 2013. → pages 63

[64] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *Proceedings of the 2012 45th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO-45, pages 96–106, Washington, DC, USA, 2012. IEEE Computer Society. → pages 133

[65] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *IEEE/ACM Supercomputing (SC 2010)*, 2010. → pages 63, 64

[66] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM. → pages 7, 16, 159

[67] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 101–110, New York, NY, USA, 2010. ACM. → pages 32, 168

[68] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi:10.1145/1345206.1345233. → pages 27, 29, 85, 168

[69] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 108–120, New York, NY, USA, 1997. ACM. → pages 44

[70] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *Micro, IEEE*, 34(2):6–20, Mar 2014. → pages 160

[71] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, March 2012. ISSN 0272-1732. → pages 19, 24, 26, 34

[72] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. *Chapter 39 in GPU Gems 3*, 2007. → pages 16

[73] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool, second edition, 2010. → pages 10, 24, 26, 27, 29, 30, 35, 74, 81, 83

[74] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 4 edition, 2008. → pages 13, 14, 16

[75] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. → pages 8, 24, 27, 31, 33, 74, 76

[76] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart. Stream Compaction for Deferred Shading. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 173–180, New York, NY, USA, 2009. ACM. → pages 162

[77] W.-m. W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. → pages 4

[78] IBM Corp. Power ISA Version 2.05, 2007. → pages 22

[79] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp., May 2012. → pages 140

[80] Intel Corp. Intel Architecture Instruction Set Extensions Programming Reference, March 2014. → pages 22, 24, 26, 34

[81] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society. → pages 24, 26, 34

[82] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987. → pages 18

[83] H. F. Jordan. Performance measurements on HEP - a pipelined MIMD computer. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212, 1983. → pages 23

[84] U. J. Kapasi et al. Efficient Conditional Operations for Data-Parallel Architectures. In *Proceedings of the 33rd IEEE/ACM International Symposium on Microarchitecture*, MICRO-33, pages 159–170, Washington, DC, USA, 2000. IEEE Computer Society. → pages 162

[85] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, Sept 2011. → pages 166

[86] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel. WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 99–110, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi:10.1145/1854273.1854291. → pages 80

[87] J. H. Kelm et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 140–151, 2009. → pages 19

[88] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, 2011. → pages 92, 94, 145

[89] Khronos Group. OpenCL. http://www.khronos.org/opencl/. Accessed: November 14, 2012. → pages 2, 36, 37, 44

[90] S. Kong et al. Time-Out Bloom Filter: A New Sampling Method for Recording More Flows. In *ICOIN*, 2006. → pages 169

[91] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-Thread Architecture. In *Proceedings. 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 52–63, June 2004. → pages 165

[92] R. M. Krashinsky. United States Patent Application #20130042090 A1: Temporal SIMT Execution Optimization, August 2011. → pages 166

[93] J. Laudon. Performance/watt: the new server focus. *SIGARCH Computer Architecture News*, 33(4):5–13, 2005. → pages 23

[94] E. A. Lee. The Problem with Threads. *Computer*, 39, May 2006. → pages 76

[95] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM. → pages 4

[96] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 129–140, New York, NY, USA, 2011. ACM. → pages 166

[97] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM. → pages 144, 146, 176

[98] M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 516–530, Berlin, Heidelberg, 2012. Springer-Verlag. → pages 172

[99] Y. Lev and M. Moir. Fast Read Sharing Mechanism for Software Transactional Memory (POSTER). In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, 2004. → pages 172

[100] A. Levinthal and T. Porter. Chap - A SIMD Graphics Processor. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 77–82, New York, NY, USA, 1984. ACM. → pages 41, 60

[101] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008. → pages 10, 38

[102] E. Lindholm et al. United States Patent Application #2010/0122067: Across-Thread Out-of-Order Instruction Dispatch in a Multithreaded Microprocessor (Assignee NVIDIA Corp.), May 2010. → pages 39

[103] A. Mahesri. *Tradeoffs in Designing Massively Parallel Accelerator Architectures*. PhD thesis, University of Illinois at Urbana-Champaign, 2009. → pages 63

[104] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 164–175, 2008. → pages 63, 64

[105] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling Support for Transactional Memory Contention Management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi:10.1145/1693453.1693465. → pages 170

[106] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006. → pages 30

[107] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM. → pages 50, 51, 52, 55, 164

[108] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM. → pages 6, 7, 16

[109] M. M. Michael. Practical Lock-Free and Wait-Free LL/SC/VL Implementations Using 64-Bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing*, DISC 2004, pages 144–158. Springer, 2004. → pages 109, 113

[110] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical report, Rochester, NY, USA, 1995. → pages 109

[111] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007. → pages 31, 33, 34, 80, 133, 134, 167, 168

[112] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965. → pages 1

[113] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, HPCA '06, Washington, DC, USA, 2006. IEEE Computer Society. → pages 31, 33, 167

[114] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmanns, 1997. → pages 41

[115] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society. → pages 145

[116] S. Naffziger, J. Warnock, and H. Knapp. When Processors Hit the Power Wall (or "When the CPU hits the fan"). In *Proceedings of the IEEE International Solid-State Circuits Conference*, ISSCC 2005, pages 16–17, 2005. → pages 1

[117] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM. → pages 163

[118] R. Nasre, M. Burtscher, and K. Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM. → pages 7, 16, 134, 171

[119] J. R. Nickolls and J. Reusch. Autonomous SIMD flexibility in the MP-1 and MP-2. In *SPAA '93: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 98–99, 1993. → pages 38

[120] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008. → pages 2, 36

[121] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. NVIDIA, 2009. → pages 44, 63, 76, 79, 80, 94, 106, 133

[122] *NVIDIA CUDA Programming Guide v3.1*. NVIDIA Corp., 2010. → pages 2, 5, 36, 38, 43, 44, 58, 73

[123] *NVIDIA Compute PTX: Parallel Thread Execution ISA Version 1.4*. NVIDIA Corporation, CUDA Toolkit 2.3 edition, 2009. → pages 55

[124] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 365–375, Washington, DC, USA, 2007. IEEE Computer Society. → pages 9, 30, 32, 75, 81, 84, 87, 109, 168

[125] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55602-8. → pages 172

[126] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 43–52, New York, NY, USA, 2011. ACM. → pages 24

[127] J. C. Phillips et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 2005. → pages 63, 64

[128] X. Qian, B. Sahelices, and J. Torrellas. BulkSMT: Designing SMT Processors for Atomic-Block Execution. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. → pages 171

[129] A. Ramamurthy. Towards Scalar Synchronization in SIMT Architectures. Master's thesis, University of British Columbia, 2011. → pages 6, 21, 77, 94

[130] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 61–71, Washington, DC, USA, 2012. IEEE Computer Society. → pages 69, 163

[131] M. Rhu and M. Erez. The Dual-path Execution Model for Efficient GPU Control Flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 591–602, Washington, DC, USA, 2013. IEEE Computer Society. → pages 165

[132] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society. → pages 144, 158

[133] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM. → pages 24

[134] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 458–468, Washington, DC, USA, 2005. IEEE Computer Society. → pages 170

[135] W. Ruan et al. Boosting Timestamp-based Tranasctional Memory by Exploiting Hardware Cycle Counters. In *TRANSACT*, 2013. → pages 140, 171

[136] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi:10.1109/MICRO.2006.9. → pages 167

[137] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society. → pages 81, 103

[138] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society. → pages 22

[139] M. Schatz et al. High-Throughput Sequence Alignment Using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007. → pages 63

[140] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 181–194, New York, NY, USA, 2008. ACM. → pages 31

[141] M. L. Scott. *Shared-Memory Sycnhronization*. Morgan and Claypool, first edition, 2013. → pages 17, 18, 21

[142] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM. ISBN 978-1-4503-0112-1. doi:10.1145/1399504.1360617. → pages 79

[143] T. A. Shah. FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration. Master's thesis, North Carolina State University, 2010. → pages 145

[144] P. Shivakumar and N. Jouppi. *CACTI 5.0. Technical Report HPL-2007-167*. HP Laboratories, 2007. → pages 76, 106

[145] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 139–150, Washington, DC, USA, 2008. IEEE Computer Society. → pages 31, 32, 33, 168

[146] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer*

Architecture (HPCA), HPCA '13, pages 578–590, Washington, DC, USA, 2013. IEEE Computer Society. → pages 140, 171

[147] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011. → pages 31

[148] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 179–193, Berlin, Heidelberg, 2006. Springer-Verlag. → pages 125, 139, 172

[149] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM. → pages 9, 32, 75, 84, 88, 117, 168, 169

[150] M. Steffen and J. Zambreno. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 237–248, Washington, DC, USA, 2010. IEEE Computer Society. → pages 163

[151] J. M. Stone. A Simple and Correct Shared-queue Algorithm Using Compare-and-swap. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 495–504, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. → pages 109

[152] *OpenSPARC^{TM} T2 Core Microarchitecture Specification*. Sun Microsystems, Inc., 2007. → pages 23

[153] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional Conflict Decoupling and Value Prediction. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 33–42, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi:10.1145/1995896.1995904. → pages 169

[154] S. T. Thakkar and T. Huff. Internet Streaming SIMD Extensions. *Computer*, 32(12):26–34, 1999. → pages 22

[155] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proceedings of Supercomputing*, pages 35–41, 1988. → pages 23

[156] J. E. Thornton. Parallel Operation in the Control Data 6600. In *AFIPS Proc. FJCC*, volume 26, pages 33–40, 1964. → pages 23

[157] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, New York, NY, USA, 2009. ACM. → pages 31, 32, 167

[158] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995. → pages 23

[159] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi. SIMD Divergence Optimization Through Intra-warp Compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 368–379, New York, NY, USA, 2013. ACM. → pages 163

[160] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. → pages 19

[161] J. D. Valois. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM. → pages 109

[162] V. Vineet and P. Narayanan. CudaCuts: Fast Graph Cuts on the GPU. In *CVPRW '08*, 2008. → pages 92, 93, 145

[163] G. WeiKum and G. Vossen. *Transactional Information Systems: Theory Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002. → pages 27, 28, 110, 111, 121, 122

[164] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM. → pages 16

[165] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the IEEE Symposium of Performance and Analysis of Systems and Software*, ISPASS'10, 2010. → pages 95

[166] S. Xiao and W. chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IPDPS'10, pages 1–12, April 2010. → pages 20

[167] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software Transactional Memory for GPU Architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 1:1–1:10, New York, NY, USA, 2014. ACM. → pages 125, 139, 142, 167

[168] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 283–292, New York, NY, USA, 2012. ACM. → pages 133, 171

[169] K. Yelick. Antisocial Parallelism: Avoiding, Hiding and Managing Communication. 2013. Keynote at HPCA-2013. → pages 159

[170] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society. → pages 31, 33, 80, 81, 133, 134, 167, 168

[171] R. M. Yoo and H.-H. S. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi:10.1145/1378533.1378564. → pages 92, 108, 170

[172] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM. → pages 162

[173] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi:10.1145/1854273.1854294. → pages 80

[174] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 285–294, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi:10.1145/1854273.1854311. → pages 108