In the SHADOW of GhOSTs: Cross-Domain Lessons in Parallelism

Ishita Chaturvedi

a dissertation
presented to the faculty
of princeton university
in candidacy for the degree
of Doctor of Philosophy

recommended for acceptance
by the Department of
Computer Science

Adviser: David I. August

May 2025

© Copyright by Ishita Chaturvedi, 2025. All rights reserved.

Abstract

Modern processors struggle to balance instruction-level parallelism (ILP) and thread-level parallelism (TLP) efficiently. GPUs rely on massive TLP to hide latency but suffer in workloads with low occupancy or control divergence due to limited ILP. CPUs, optimized for ILP, become bottlenecked in memory-bound workloads where TLP could improve resource utilization. This thesis challenges the rigid separation of ILP and TLP by introducing **GhOST** and **SHADOW**, two architectures that extend ILP in GPUs and TLP in CPUs to improve execution efficiency across diverse workloads.

GhOST introduces lightweight out-of-order (OoO) execution to GPUs, enabling warp-level instruction reordering without speculative execution or register renaming. Its instruction buffer-based reordering mechanism allows instructions to execute as soon as operands are ready, significantly reducing stalls in low-occupancy scenarios and mitigating control divergence overhead. GhOST achieves a *6.9% geometric mean speedup* (up to 36%) with only *0.007% area overhead*, demonstrating the feasibility of ILP-aware GPU architectures.

SHADOW is the first asymmetric simultaneous multithreading (SMT) core that dynamically balances ILP and TLP by executing OoO and in-order (InO) threads simultaneously on the same core. By leveraging deep ILP in the OoO thread and high TLP in lightweight InO threads, SHADOW maximizes CPU utilization without sacrificing single-thread performance. Unlike conventional SMT, which uniformly shares resources, SHADOW dynamically reallocates execution resources based on workload behavior. It achieves *up to 3.16× speedup and 1.38× average improvement* over an OoO CPU, with just *1% area and power overhead*, demonstrating its ability to accelerate memory-bound workloads efficiently.

This thesis demonstrates how cross-domain architectural techniques can improve execution efficiency by integrating ILP into GPUs and TLP into CPUs. GhOST and SHADOW reduce execution inefficiencies, enhancing performance in workloads that challenge conventional architectures.

To my parents, my brother, and Vinayak.

Acknowledgments

I am deeply grateful to my advisor, Prof. David I. August, for taking a chance on me and guiding me through every step of this journey. He not only taught me how to think critically and define the problems I want to solve but also helped me find my voice in research. His unwavering belief in me has been a constant source of strength. Through every challenge and rejection, he reminded me that setbacks only make my work stronger and that one day, we would look back and laugh at the tough moments. David encouraged me to pursue research I am passionate about while holding me to the highest standards, pushing me to meet them with rigor and integrity. His mentorship extends far beyond research—he has taught me resilience, the value of collaboration, and the confidence to trust my instincts. No matter how difficult things got, he always had my back, reminding me that research is a marathon, not a sprint. For his guidance, support, and belief in me, I will always be grateful.

I would also like to thank my examiners, Prof. Margaret Martonosi and Prof. David Wentzlaff, whose thoughtful input has not only improved the quality of this dissertation but has also shaped my research as a whole. I am also grateful to my readers, Prof. Sharad Malik and Prof. Niraj K. Jha, for their time and effort in serving on my committee and for their invaluable feedback, which has significantly strengthened this thesis.

Throughout my PhD, I have been incredibly fortunate to work with collaborators

who have pushed my research forward in ways I could not have done alone. Prof. Tyler Sorensen has been both a collaborator and a mentor since my second year, and I am truly grateful for his unwavering support, generosity, and kindness. His insights have strengthened my research, and his perspective on work and life has had a lasting impact on me. His patience, thoughtful feedback, and willingness to share his time have made me a better researcher and person. I am incredibly fortunate to have worked with him and deeply appreciate his support. I am also extremely grateful to Prof. Tor M. Aamodt for his invaluable insights and feedback. His deep expertise in microarchitecture has not only strengthened these works but has also shaped my understanding of microarchitecture research. He has always made time to discuss ideas, provide thoughtful guidance, and challenge assumptions, pushing my research further.

I have been incredibly fortunate to be part of the Liberty Research Group. I am especially grateful to Bhargav for always being there whenever I needed help—both professionally and personally—and for hosting me for countless dinners, making Princeton feel a little more like home. I would also like to thank Sotirios, Nayana, Susan, and Ziyang for being kind and supportive senior students, Yebin for always being helpful, and Yucan and Sophia for bringing fresh energy and enthusiasm to the lab. I am also grateful to my friends from the Wentzlaff Lab for their support and camaraderie, including Rohan, August, Haiyue, and Kaifung. My friends at the LSD Lab at UCSC have been incredibly generous, with Yanwen always willing to share remote systems and offer help. I also appreciate Reese, Devon, and Gurpreet for their time, insightful discussions, and support in preparing for talks.

I am deeply grateful to Bobbie Manne at AMD for her mentorship and generosity. Her kindness, strength, and guidance continue to push me to grow as both a researcher and a person. I am also extremely thankful to Biswabandhan Panda. His blogs have provided invaluable perspective on research, reminding me when to step

back and view the bigger picture and how to approach problems systematically.

Outside of research, I have been incredibly lucky to find friends who became my family. Rachna, Shanaya, and Shashank have been my constants since high school, always reminding me to laugh, listening to endless rants, and keeping me grounded. Their unwavering support has meant the world to me. In Princeton, Sophia, Parth, Vishwak, Sneha, and Bhishma became my home away from home. They have stood by me every step of the way, making time to listen, lifting me up through every challenge, and filling this journey with kindness, patience, and an endless amount of joy and laughter. I am also incredibly thankful for the friends I made along the way—Anvitha, Krishi, and Anish—for all the laughter, lighthearted moments, and much-needed fun. Simon and Fatima brought important perspectives into my life, reminding me to look beyond engineering and embrace a broader view of the world.

I am nothing without my family. I am indebted to my parents for everything they have done for me, sacrifices I can never fully put into words. They have always put my needs above their own, been my fiercest supporters, and believed in me even when I didn't believe in myself. They taught me the value of hard work, empathy, honesty, and integrity—not through words, but through the way they live their lives. They showed me that every mistake is just one less I will make in the future, that failure is a lesson, and that I will always have a home to return to no matter where life takes me. Their unwavering love and guidance have been my greatest strength.

My brother is the smartest person I know, my best friend since childhood through every city we moved to. Every argument and debate with him has sharpened my thinking (and tested my patience), but his relentless curiosity and razor-sharp reasoning push me to question everything more deeply. Watching him navigate the world with kindness and brilliance reminds me to work harder, to never stop learning, and to always strive to be better. I am endlessly grateful for my family—for

their love, their wisdom, and for being my rock through every step of this journey. The completion of this PhD would not have been possible without Vinayak.

Contents

Abstract **iii**

1 Introduction **9**

2 GhOST: A GPU Out-of-Order Scheduling Technique for Stall Reduction **15**

2.1 Introduction 15

2.2 Motivation 18

2.3 Baseline Execution Model 22

2.4 GhOST Design 25

2.5 Evaluation Methodology 37

2.6 Evaluation of GhOST 39

2.7 Related Work 47

2.8 Conclusion 49

3 SHADOW: Simultaneous Multi-Threading Architecture with Asymmetric Threads **50**

3.1 Background and Motivation 53

3.2 SHADOW 58

3.3 Evaluation of SHADOW 71

3.4 Related Work 80

3.5 Conclusion 81

4 Future Work **82**

4.1 Extending GhOST 82

4.2 Extending SHADOW 82

5 Conclusion **84**

References **96**

List of Tables

2.1 Related Work Comparison 22

2.2 Decode Checker Configuration 29

2.3 GPU Microarchitectural Parameters 38

2.4 Evaluated Benchmarks 39

2.5 Effect of Various OoO Optimizations on GhOST on the Simulated RTX 2060S GPU. 42

3.1 CPU Microarchitectural Parameters 70

3.2 Benchmarks Used to Evaluate SHADOW 71

3.3 Performance of SHADOW on Diverse Workloads 73

3.4 Hardware Overhead of SHADOW 80

List of Figures

2.1 Percentage of reordered instructions with idealized OoO execution (perfect branch prediction, register renaming and load-store queues) and efficient OoO execution which elides these techniques. 17

2.2 SASS example of a typical basic block that appears in GPU programs. 19

2.3 Histogram showing the variable load latency during program execution from Figure 2.2. 20

2.4 Idealized performance of OoO execution on GPU. 21

2.5 The GPU pipeline before and after modifications for GhOST. 24

2.6 Architectural diagram for GhOST modifications at the decode stage. 26

2.7 Dependence checking of new instruction with one instruction updating its corresponding IB-dep bit in IB-dep vector. 27

2.8 The time-shared GhOST scheduler. 32

2.9 Register dependence checking hardware for the GhOST scheduler. . 33

2.10 Oldest instruction selection hardware for the GhOST scheduler. . . . 34

2.11 GhOST-Precise does not issue non-idempotent instructions until prior load instructions hit in the TLB. 36

2.12 Execution of a small program on GhOST. 36

2.13 The performance of GhOST and the reduction in scheduler stalls with OoO execution plotted against the occupancy of applications on the simulated RTX 2060S GPU. 39

2.14 The performance variance of GhOST with different scheduling policies on the simulated RTX 2060S GPU. 40
2.15 The performance variance of GhOST with changing IsB size on simulated Nvidia RTX 2060S and RTX 3070 GPUs. 41
2.16 The performance variance of GhOST-Precise with changing IsB size on the simulated RTX 2060S GPU. 42
2.17 Comparison of unbounded register renaming in hardware with bounded register renaming (255 registers) in software. 44
2.18 The performance of GhOST and LOOG on the simulated RTX 2060S GPU. 45
2.19 Performance of GhOST and LOOG against in-order execution on the simulated RTX 2060S GPU configuration. 45

3.1 SHADOW dynamically redistributes work as IPC changes. High ILP skews execution toward the OoO thread, while work distributes equally among threads when the ILP is low. SHADOW adapts to the application without software intervention. 52
3.2 Change in the IPC and L2 cache misses of the SpMM application with varying sparsity. 54
3.3 Speedup of a 90% sparse SpMM with varying ROB/RS size for a 1 OoO configuration. Area overhead of a larger OoO window does not translate into proportional speedup. 54
3.4 Breakdown of IPC contributions from all threads for 90% sparse SpMM. 55
3.5 Microarchitecture design of SHADOW configured with 1 OoO and 4 inO threads. 58
3.6 Microarchitecture of the fetch and decode stage configured with 1 OoO and 4 inO threads. 60

3.7 Microarchitecture of the Rename and Wakeup+Select stages configured with SHADOW having 1 OoO and 4 inO threads. 61
3.8 Register File partitioning in SHADOW. 62
3.9 Impact of adding an InO thread to an OoO system: An illustrative assembly example. 69
3.10 Geometric mean performance of SHADOW configurations on high D-cache miss rate benchmarks from Table 3.2, normalized to a single-threaded OoO core. 71
3.11 Geometric mean performance of SHADOW configurations on low D-cache miss rate benchmarks from Table 3.2, normalized to a single-threaded OoO core. 72
3.12 Performance of SHADOW with varying degrees of sparsity for an 8-wide CPU over 1 OoO thread. 75
3.13 Performance of SHADOW with varying degrees of sparsity for a 4-wide CPU over 1 OoO thread. 75
3.14 Breakdown of IPC contribution from each thread with varying degrees of sparsity over 1 OoO thread. 76
3.15 Change in miss rate for different thread configurations. 77
3.16 Performance of SHADOW with varying matrix size for 90% sparsity for an 8-wide CPU over 1 OoO thread. 78
3.17 Performance of SHADOW with varying matrix size for 90% sparsity for a 4-wide CPU over 1 OoO thread. 78
3.18 Distribution of work across threads with dynamic work-stealing for various degrees of sparsity. 79

List of Abbreviations

Abbreviation	Definition
ILP	Instruction-Level Parallelism
TLP	Thread-Level Parallelism
OoO	Out-of-Order
InO	In-Order
SMT	Simultaneous Multithreading
GPU	Graphics Processing Unit
CPU	Central Processing Unit
GPGPU	General-Purpose GPU
IB	Instruction Buffer
MPS	Multi-Process Service
SIMT	Single Instruction, Multiple Threads
SSSP	Single-Source Shortest Path
CTAs	Cooperative Thread Arrays
SM	Streaming Multiprocessors
ICache	Instruction Cache
RAW	Read-After-Write
WAW	Write-After-Write
WAR	Write-After-Read
GTO	Greedy-Then-Oldest
PC	Program Counter
CU	Collector Unit
SPU	Scalar Processing Units
SFU	Special Function Units

TCU	Tensor Core Units
MEM	Memory Unit
DC	Dependence Checker
ITab	Instruction Table
IB-Calc	IB-dep Calculation Logic
FIFO	first-in, first-out
WWb	Warp-WriteBack
ITE	ITabEmpty
RDC	Register Dependence Check
OSel	Oldest Instruction Selection
SpMM	Sparse Matrix Multiplication
ROB	Reorder Buffer
RS	Reservation Stations
ARF	Architectural Register File
PRF	Physical Register File
RAS	Return Address Stacks
PCB	Process Control Block
PR	Physical Registers
RAT	Register Alias Table
CPSR	Current Program Status Register
TLB	Translation Lookaside Buffer
TCB	Thread Control Block

Chapter 1

Introduction

Modern computer architecture continually evolves to improve latency hiding and resource utilization, two critical factors in sustaining performance. To achieve this, processors rely on two fundamental forms of parallelism: instruction-level parallelism (ILP) and thread-level parallelism (TLP) [42]. ILP extracts parallelism within a single thread by executing multiple instructions concurrently using techniques like pipelining, superscalar execution, and out-of-order (OoO) execution. TLP, in contrast, increases parallelism across multiple threads by allowing execution streams to share resources through simultaneous multithreading (SMT), multicore architectures, and large-scale SMT systems.

While ILP and TLP both serve as latency-hiding mechanisms, modern architectures rigidly separate them. CPUs are optimized for ILP, using deep pipelines and complex scheduling, but offer limited TLP. GPUs prioritize TLP, executing thousands of lightweight threads to mask memory latency but lack mechanisms to efficiently extract ILP [16]. This strict division leaves hardware underutilized in workloads that require a balance of both forms of parallelism.

This separation of ILP and TLP is the result of decades of architectural evolution. Early processors executed instructions sequentially before ILP and TLP

emerged as distinct paradigms. As workloads grew more complex, the need for architectures that seamlessly combine ILP and TLP has never been greater. This thesis introduces GhOST, which enhances ILP in GPUs, and SHADOW, which enables CPUs to exploit TLP efficiently, together unlocking new performance opportunities across a wide range of applications.

The evolution toward ILP and TLP was driven by the limitations of early computing systems. Initially, processors could only execute one instruction at a time, a model that was simple but increasingly inefficient as computational demands grew. Over time, architects introduced techniques to extract more parallelism, first within a single thread and later across multiple threads. This transition laid the foundation for modern CPUs and GPUs, each designed to maximize a specific form of parallelism. Understanding this progression highlights both the strengths and constraints of existing architectures and the motivation for integrating ILP and TLP in a unified framework.

Sequential Processors: The Starting Point. Early processors followed a strictly sequential execution model, where each instruction completed before the next began. Systems like EDVAC (1949) and IBM 701 (1952) [42] ensured simplicity and reliability but underutilized hardware. As workloads grew, this model became a bottleneck—processors like PDP-8 (1965) and IBM System/360 (1964) executed one instruction at a time, leaving functional units idle and increasing latency. While higher clock speeds provided temporary gains, purely sequential execution fundamentally limited throughput, driving the need for architectural innovations to exploit parallelism.

Pipelining: The First Step Toward ILP. The IBM Stretch (7030) project pioneered pipelining, a technique that divided instruction execution into stages (fetch, decode, execute, memory access, write-back) so multiple instructions could progress

simultaneously. This approach, later refined in the IBM System/360 Model 91, became standard by the 1970s and 1980s, significantly improving processor throughput. However, pipelining introduced pipeline hazards—data hazards required forwarding, control hazards necessitated branch prediction, and structural hazards demanded better resource allocation. Deep pipelines, such as in Intel's Pentium 4 (NetBurst), increased clock speeds but made branch mispredictions costly. Modern architectures, like Intel Core and AMD Zen, adopt balanced pipeline depths to optimize performance, power, and complexity. Pipelining was the first major step toward ILP and remains fundamental in modern processors.

Wide-Issue and Superscalar Processors. By the late 1980s, single-issue pipelines underutilized available transistors, leading to superscalar execution, where multiple instructions issued per cycle to parallel execution units. Inspired by designs like the CDC 6600 (1964) and IBM System/360 Model 91 (1967) [42, 113], superscalar processors—HP PA-RISC 7100 (1992), DEC Alpha 21064 (1992), and Intel Pentium (1993)—improved throughput by resolving dependencies in hardware. However, as issue widths expanded, increasing instruction window sizes, decoders, and hazard detection added power and design complexity, yielding diminishing returns [116]. These scaling challenges drove the need for OoO, SMT, and multicore architectures to further enhance performance.

Transition to Out-of-Order Execution. As superscalar processors widened issue widths, in-order execution became a bottleneck, stalling on unresolved dependencies. To overcome this, OoO execution, inspired by Tomasulo's algorithm [113, 111], enabled instructions to execute as soon as operands were available, improving hardware utilization. By the mid-1990s, OoO became mainstream in MIPS R10000, Alpha 21264, and Intel Pentium Pro, and remains fundamental in modern architectures like Intel Skylake, AMD Zen, and Apple M1. While OoO execution pushed

ILP closer to its limits, its complexity and power demands necessitated further advancements, leading to SMT and multicore architectures.

SMT and Large SMT Designs. As superscalar and OoO execution matured, underutilized functional units remained a bottleneck. SMT [45, 123, 115, 114, 26] addressed this by enabling multiple threads to issue instructions in the same cycle, improving utilization without major hardware modifications. Early implementations, such as Intel's Hyper-Threading (Pentium 4), ran two threads per core, while large-scale SMT designs like UltraSPARC T1 (2005) scaled to 32+ threads per chip, maximizing throughput in server workloads.

However, SMT introduced resource contention, as threads competed for caches, execution units, and memory bandwidth, sometimes degrading single-thread performance. In-order SMT favored many threads but struggled with latency-sensitive workloads, while OoO SMT preserved single-thread speed at the cost of scalability. Despite these trade-offs, SMT became a standard feature in processors such as IBM POWER, Intel Hyper-Threading, and modern cloud-scale CPUs, shifting architectural focus from clock speeds to parallel execution.

From SMT to GPUs: The Rise of TLP. Initially designed for fixed-function graphics, GPUs evolved in the 1990s to support programmable shaders, increasing flexibility for complex 3D rendering. By the mid-2000s, frameworks like CUDA (2007) and OpenCL enabled general-purpose GPU (GPGPU) computing, expanding their role into scientific computing and machine learning.

Unlike CPUs, which extract ILP through OoO execution, GPUs maximize TLP, managing stalls by switching between thousands of lightweight threads instead of reordering instructions. Using the SIMT model, GPUs execute threads in warps, efficiently masking memory latency. This approach enables massive parallelism but introduces challenges such as control-flow divergence and memory access inefficien-

cies, limiting performance in irregular workloads.

Balancing ILP and TLP: Benefits and Trade-Offs. Modern processors rely on ILP and TLP to maximize performance, each with distinct trade-offs. ILP, used in CPUs, boosts single-thread performance through OoO execution, branch prediction, and dynamic scheduling, making it ideal for latency-sensitive tasks. However, it requires complex hardware and high power consumption. TLP, dominant in GPUs, achieves performance by running thousands of lightweight threads, masking stalls without speculative execution. While energy-efficient, TLP does not optimize per thread but instead relies on high thread occupancy to sustain throughput. When occupancy is low or control-flow divergence occurs, execution units sit idle, reducing efficiency.

Evolving Toward Hybrid Models. GPUs excel at TLP but struggle with ILP, making them inefficient for irregular workloads. CPUs, optimized for ILP, underperform in TLP-heavy applications. This thesis introduces GhOST and SHADOW, two architectures that dynamically integrate ILP and TLP to bridge this divide.

GhOST brings warp-level OoO execution to GPUs, allowing instructions within a warp to execute as soon as operands are ready, reducing stalls in low-occupancy and control-divergent workloads (see Section 2).

SHADOW enables CPUs to mix OoO and in-order execution, running a single OoO thread alongside multiple lightweight in-order threads, improving utilization while maintaining strong single-thread performance (see Section 3).

Authorship and Research Contributions. GhOST, which has been published [16], was developed with contributions from Bhargav Reddy Godala, Yucan Wu, Ziyang Xu, and David I. August from the Princeton Liberty Research Group; Simone Campanoni from Northwestern University; Tyler Sorensen from the University of Cali-

fornia, Santa Cruz; and Tor M. Aamodt from the University of British Columbia.

SHADOW, currently in preparation for publication, was developed with contributions from Bhargav Reddy Godala and David I. August from the Princeton Liberty Research Group; Abiram Gangavaram and Daniel Flyer from Princeton University; Tyler Sorensen from the University of California, Santa Cruz; and Tor M. Aamodt from the University of British Columbia.

Chapter 2

GhOST: A GPU Out-of-Order Scheduling Technique for Stall Reduction

2.1 Introduction

Graphics Processing Units (GPUs) have significantly enhanced the efficiency of computationally intensive applications, including medical imaging [102], machine learning [60], and computer vision [70], achieving orders of magnitude improvement. This efficiency is attributed to the GPUs' capability to (1) support a massively parallel programming model and (2) reduce latency through concurrency, allowing rapid switching between concurrent warps. Some Nvidia devices, for instance, can handle up to 64 warps per core [83, 120, 118].

Despite these capabilities, many GPU applications still encounter significant stall cycles [7, 63], during which none of the warps can issue instructions due to data, control, or structural hazards. Many techniques to address GPU stall cycles focus on thread-level parallelism. They either focus on (1) increasing the number of

warps on the GPU may improve the probability of finding a non-stalled warp. However, prior work has shown that doing so may result in worse performance due to increased resource contention [67, 52, 98]; or (2) re-prioritizing warps to enhance TLP and/or cache locality [82, 98, 30, 122, 50, 65]. However, all techniques focusing on TLP are limited, in part, because they only consider the next in-order instruction per warp. Consequently, the GPU core will stall when the oldest instructions for all warps encounter stalls.

Recent efforts have employed OoO execution, a technique successfully applied to CPUs, to enhance instruction-level parallelism on GPUs [46, 34, 56, 127]. While OoO GPU approaches show promising initial results, they still suffer from inefficiencies, including re-issuing instructions, saving state resources such as registers or memory for more complicated warp switches, and making ISA modifications.

LOOG [46] is the current state-of-the-art that enables complete OoO execution for GPUs. However, LOOG requires expensive features such as load-store queues for reordering memory instructions, result broadcasting, and register renaming.

While previous approaches aim to enable CPU-style OoO mechanisms on GPUs, GPU applications exhibit markedly different characteristics. Historically, the first dynamically scheduled processor, the CDC6600 [112], enabled OoO execution without renaming or speculation back when the number of devices in a system was far more constrained than today. We see merit in revisiting such simpler, yet not fully-featured, designs in the context of today's throughput architectures where efficiency is paramount.

We observed that having instructions from independent warps in the instruction buffer (IB) provides a high level of flexibility, leading to significant OoO performance without the need for register renaming, bookkeeping, or speculation. Figure 2.1 demonstrates that GPUs offer significant reordering opportunities to hide stalled instructions without requiring speculation, register renaming, and reordering

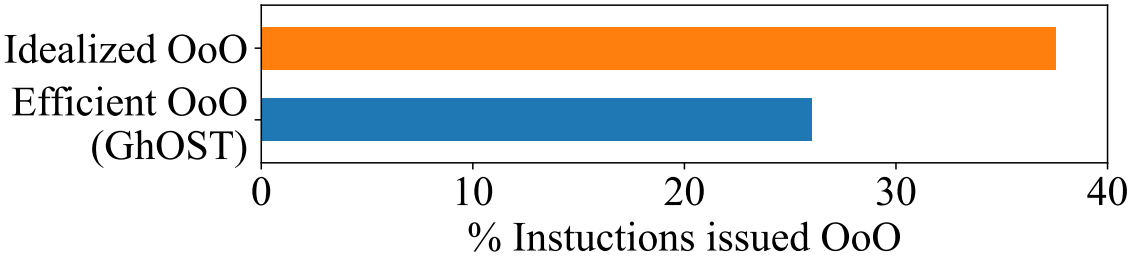


Figure 2.1: Percentage of reordered instructions with idealized OoO execution (perfect branch prediction, register renaming and load-store queues) and efficient OoO execution which elides these techniques.

of load and store instructions across a comprehensive benchmark suite by issuing instruction OoO from the IB.

Building on this insight, this chapter introduces GhOST: a low-overhead, hardware-only OoO technique for GPUs designed to leverage the unique characteristics of GPU architecture to achieve minimal hardware overhead. GhOST is the first work that utilizes the issue buffer as a large pool of instructions, thereby increasing the probability for GhOST to find a ready instruction without the need for the expensive techniques required by prior work. GhOST splits the instruction scheduler between the GhOST scheduler and the warp scheduler, leaving the complexity of the warp scheduler unmodified. This results in a simple and effective non-speculative OoO engine, which resides almost entirely in the decode stage.

GhOST outperforms LOOG [46], the state-of-the-art prior OoO technique, while using significantly less area and power. GhOST achieves a maximum speedup of up to 37%, with a geometric mean speedup of 6.9% across 28 applications, exhibiting a notable performance improvement in low-occupancy applications with an area increase of 0.007% and a 1.1707 mW increase in power. GhOST does not cause performance degradation in any evaluated workloads. While GhOST is a hardware-only technique, we show that compiler techniques like loop unrolling and software renaming work well with GhOST, bringing the OoO performance closer to idealized OoO execution.

2.1.1 Contributions of GhOST

GhOST is a lightweight hardware-only approach to OoO execution on GPUs. Unlike existing approaches that rely on register renaming, speculative execution, and reorder buffers, GhOST achieves comparable or superior performance with minimal hardware overhead. GhOST achieves this by leveraging the GPU's IB as a flexible pool of instructions, enabling the scheduler to issue instructions OoO without complex hardware changes. The simplicity of the GhOST design allows for seamless integration with existing GPU pipeline stages, while still offering significant performance improvements in workloads with low thread occupancy and high divergence.

The key contributions of GhOST are as follows:

1. A comprehensive study that measures the performance impact of various OoO optimizations on GPU workloads, emphasizing the minimal returns provided by traditional OoO components.

2. The design of GhOST, a cost-effective hardware-only OoO technique for GPUs. GhOST is predominantly contained within the decode stage, providing ample scheduling freedom with minimal changes to the interface with other stages.

3. An extensive experimental study of GhOST compared to typical state-of-art GPUs and LOOG, the best performing prior GPU OoO mechanism. The study shows that GhOST outperforms LOOG while using a fraction of its area and power overhead.

2.2 Motivation

GPU's performance comes from massively parallel programming and using TLP to hide latencies. However, some modern GPU workloads lack sufficient TLP to fully utilize GPU resources. Nvidia introduced Multi-Process Service (MPS) [86]

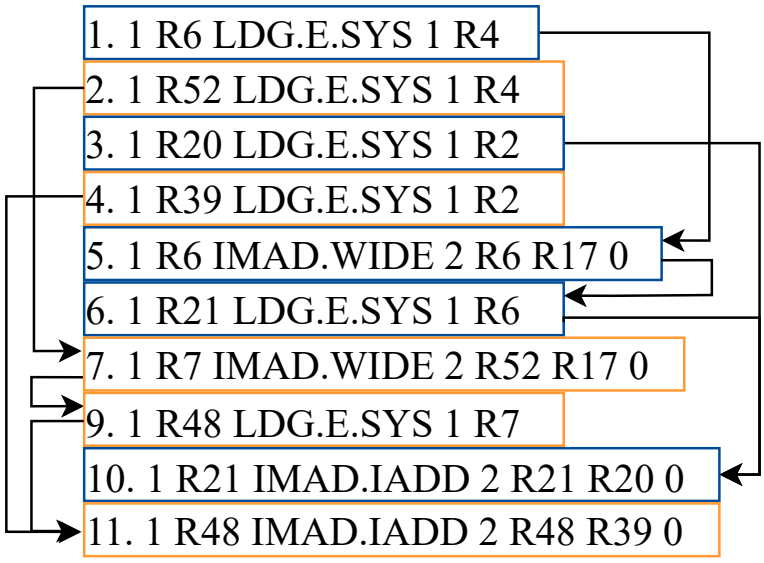


Figure 2.2: SASS example of a typical basic block that appears in GPU programs.

to enhance TLP through concurrent execution of multiple processes on the GPU. However, it fell short in improving the performance of individual applications, underscoring the importance of ILP enhancements. Despite the significant hardware support for TLP in GPUs, there remains untapped potential for exploiting ILP in these devices. Previous studies have explored techniques such as OoO execution [46, 56, 127, 34] and thread coarsening [76, 10, 106] to enhance ILP, with the goal of improving stall hiding.

To illustrate opportunities for ILP, consider the SASS code snippet from the single-source shortest path (SSSP) benchmark in the Pannotia suite [17], as shown in Figure 2.2; some details have been omitted for clarity. This code pattern, found in GPU workloads, involves multiple load instructions from different load-use chains. The compiler interleaves these chains to increase the distance between loads and their use. This basic block repeats several times in the execution of the SSSP benchmark and is the critical loop. Figure 2.3 displays the histogram of latencies for the load instructions shown in Figure 2.2. The histogram was plotted by collecting load latencies for every instance of the load instructions in Figure 2.2 being executed

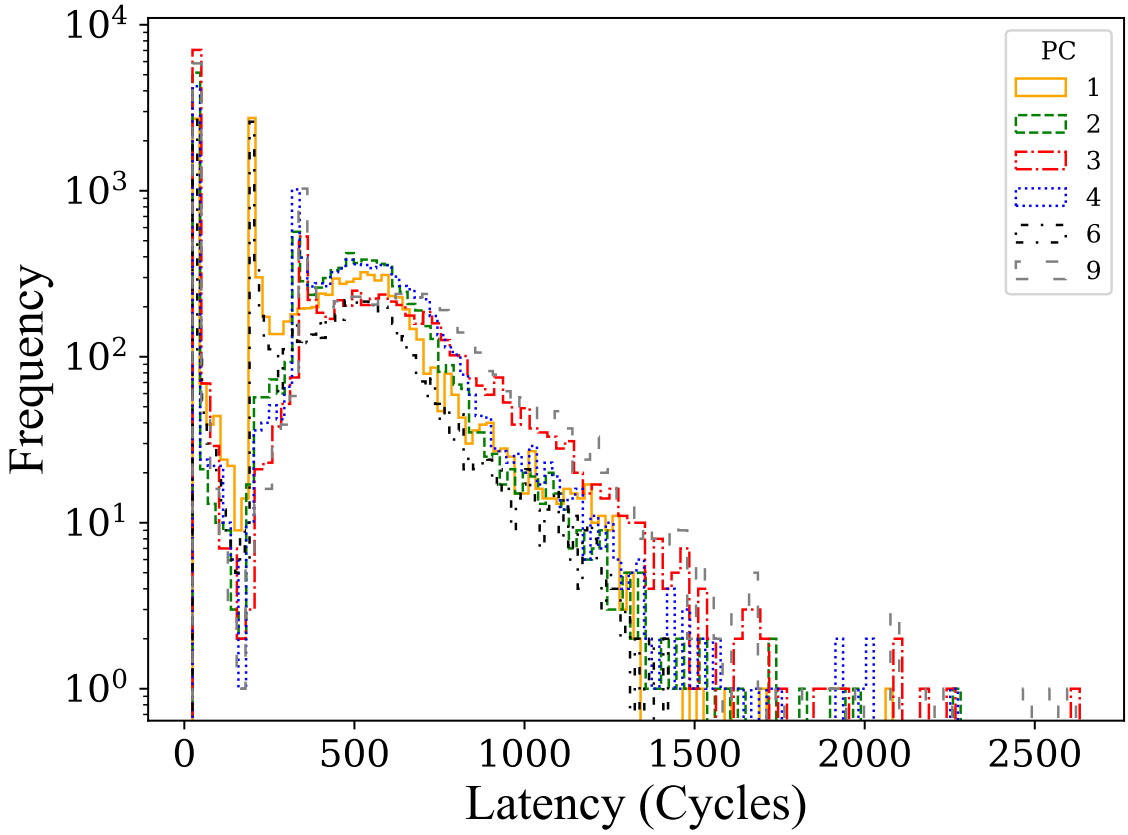


Figure 2.3: Histogram showing the variable load latency during program execution from Figure 2.2.

using the AccelSim [53] simulator.

The wide range of latencies poses a challenge for statically placing loads to achieve optimal latency hiding. While compilers attempt static scheduling to hide these latencies, achieving a universally optimal schedule for every basic block execution remains challenging. In contrast, OoO execution dynamically issues ready instructions as operand values become available, offering a more adaptable approach.

2.2.1 Idealized OoO performance on GPUs

We measured the idealized performance of OoO execution on GPUs, leveraging (1) perfect branch prediction, (2) perfect memory aliasing, (3) unbounded register renaming, and (4) a deep reorder queue. Figure 2.4 shows a 22% geomean performance improvement using these optimizations. OoO identifies independent chains

at runtime and achieves performance gains by adapting to the latency experienced per warp. There is a significant performance gain to be had with OoO execution; however, it must be optimized for the GPU architecture. This speedup sets an upper limit to the performance that can be obtained by OoO execution.

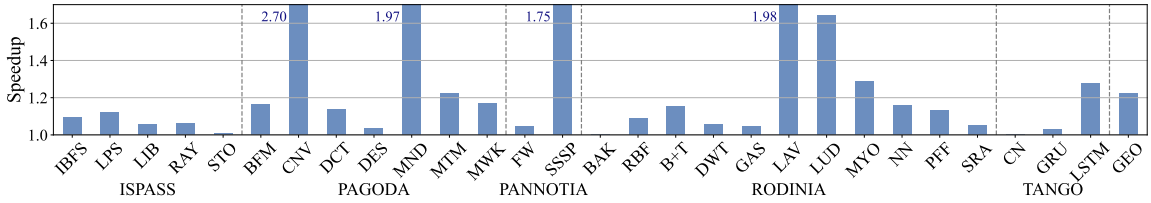


Figure 2.4: Idealized performance of OoO execution on GPU.

2.2.2 Existing GPU OoO Approaches

Previous research has demonstrated the effectiveness of OoO execution on GPUs, but these techniques have come with a high hardware cost to improve GPU performance [46, 34, 56, 127]. Register renaming was a common technique used to eliminate false dependencies in all prior work approaches, while some also implemented memory instruction reordering using alias checking or speculative execution, adding considerable hardware overhead [46, 34]. Additionally, some techniques required two-phase execution, switching between in-order and OoO modes, necessitating extensive hardware to save the state of the GPU during mode switches [34, 127, 56]. A detailed comparison of these techniques and their associated costs is summarized in Table 2.1.

The best-performing proposal, i.e., LOOG [46], while effective in PTX, shows a 16.5% slowdown when adapted to SASS. This research uses the SASS ISA, the native execution code in Nvidia GPUs [121]. The compiled SASS code was already optimized for target GPUs and has shown more ILP than PTX [53]. Our approach thus navigates a more complex optimization landscape, contrasting with previous studies that relied on PTX. Additionally, LOOG would increase the area by at least

1.29% compared to a baseline GPU architecture that employs operand collectors, and an unknown (but possibly significant) impact on clock frequency due to the introduction of long wires for result broadcast. Moreover, LOOG requires repurposing operand collector units, originally intended to enable a multiported register file [69, 74], to enable OOO execution. However, recent GPUs instead employ a software-managed operand reuse cache [30, 31, 36, 49, 1] likely making overheads for LOOG even higher.

Table 2.1: Related Work Comparison

Technique	No ISA change	No in-order to OoO mode switching	Low hardware overhead	Speedup evaluated with SASS
Perfect OoO				22% with perfect branch prediction, unbounded register renaming, and perfect memory alias checking
Warped Pre-Execution [56]	✓	✗	✗	No SASS reported
MIPSGPU [127]	✓	✗	✗	No SASS reported
LOOG [46]	✓	✓	✗	16.5% slowdown (SASS), 16% PTX. LOOG shows significant slowdowns after optimized compilation with SASS.
HAWS [34]	✗	✗	✓	No SASS reported
GhOST [This Work]	✓	✓	✓	6.9%

2.3 Baseline Execution Model

This section provides an overview of the GPU programming model and describes the micro-architecture of the baseline GPU pipeline modeled in Accel-Sim, which serves as the basis for the GhOST model.

GPU programming model: A GPU program consists of host code that executes on the machine's CPU and device code that executes on the GPU. The device code, called a kernel, is executed in a Single Instruction, Multiple Threads (SIMT) manner [58, 1]. A thread is the basic unit of computation. Threads are grouped in disjoint sets, called warps, that execute in lock-step: they synchronously execute the same instruction and share a program counter. Warps are grouped into disjoint sets called *blocks* or *Cooperative Thread Arrays* (CTAs).

The SIMT stack, which is private for each warp, handles the serialization of the execution of divergent control paths. SIMT stack handling and thread convergence of Nvidia GPUs were changed after the Volta generation [88]; for the ease of presentation, we describe GhOST using the older mechanism.

GPU architecture: The GPU architecture consists of many streaming multiprocessors (SMs). Each SM has up to four scheduling units, with each warp scheduling unit selecting one warp to issue an instruction. It utilizes concurrency to hide latencies; if one warp stalls, the SM can efficiently use fine-grained multi-threading to switch to another warp to execute an instruction. The ratio of the number of warps active on an SM to the maximum warps supported by an SM is called warp *occupancy*. While high occupancy is almost always desirable, very high occupancy can lead to slowdowns in some cases due to increased cache misses and memory contention [98, 67, 52].

The GPU offers various on- and off-chip memory structures to support program execution. Each SM has its own L1 cache, shared memory, and various specialized memory regions [90]. All SMs share a common L2 cache, and misses to the L2 are serviced by the off-chip DRAM. Each thread is assigned a set of registers for the entirety of its execution, allowing for faster context switching in the GPU since register file states do not need to be saved. Due to the large number of threads each SM can support, each SM has a large number of registers (64,000 32-bit registers

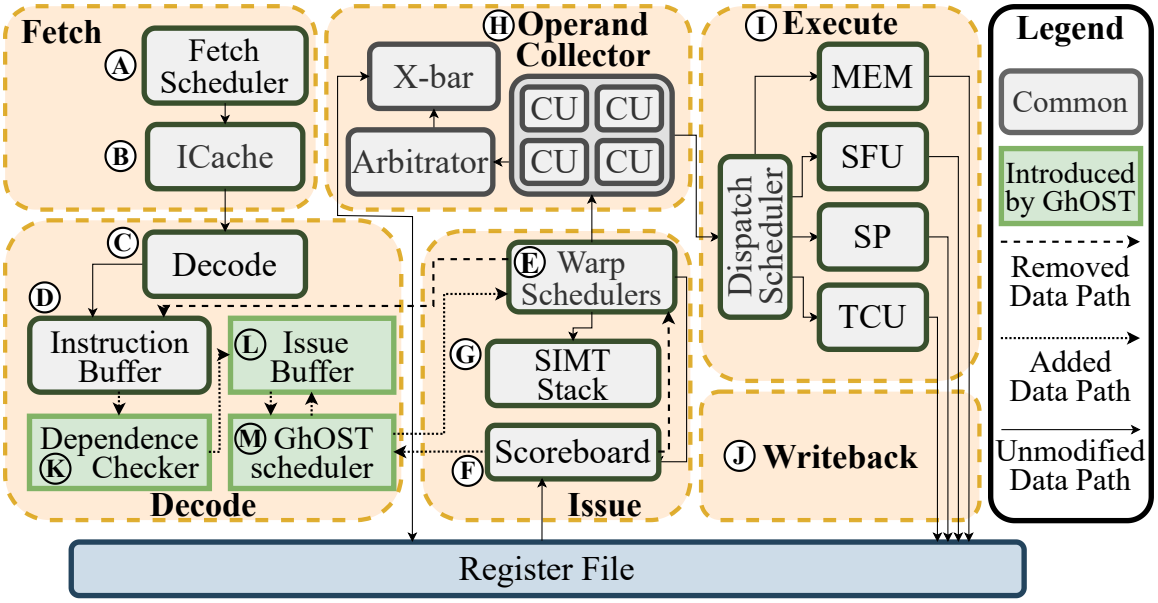


Figure 2.5: The GPU pipeline before and after modifications for GhOST.

for Nvidia RTX 2060S).

GPU Streaming Multiprocessor (SM) Micro-Architecture: We assume as our baseline SM micro-architecture the one in Accel-Sim [53] when configured to model the NVIDIA RTX-2060 Super. The baseline SM pipeline model consists of six stages: fetch, decode, issue, read operands, execute, and writeback. Figure 2.5 shows the baseline SM.

Fetch and Decode: The fetch scheduler Ⓐ selects warps waiting to access the instruction cache (ICache) Ⓑ and executes its fetch request; in the next cycle, the fetched instructions are decoded Ⓒ. A dependence bit-vector marks any read-after-write (RAW) or write-after-write (WAW) dependencies between the registers of this instruction and any in-flight instructions or older instructions in the IB [21]. The decoded instruction, along with the dependence bit-vector, is placed in the IB, and the valid bit for the instruction is set. The IB has space to hold two instructions per warp.

Issue: Each SM has multiple warp scheduling units Ⓔ. In each cycle, the warp

scheduler selects one warp from its assigned warps and issues an instruction from it. The warp schedulers can be instantiated with various policies. The RTX 2060S warp schedulers as modeled by AccelSim use the greedy-then-oldest (GTO) scheduling policy, where instructions from one warp are prioritized in every cycle until the warp stalls and the scheduler moves on to the next warp.

An instruction is issued when: (1) its program counter (PC) matches the PC expected by the SIMT stack; (2) it has no data dependencies; and (3) it secures available issue-pipeline registers (addressing structural hazards). In recent GPUs, warp schedulers issue one instruction per cycle from the selected warp. As such, this work models the RTX 2060S device to issue only one instruction per cycle. After issuing the instruction, the SIMT stack Ⓖ [27], and the scoreboard are updated.

Read Operands: After issue, instructions are allocated space in the Collector Unit (CU) Ⓗ for operand collection and are kept in the CU till all operands are collected.

Execute: The GPU has different types of execution units Ⓘ. Compute instructions are sent to the scalar processing units (SPUs), special function units (SFUs), or tensor core units (TCUs). Memory operations are forwarded to the memory unit (MEM).

Writeback: After executing, instructions issue a write request to the Register File arbitrator and exit the pipeline Ⓙ. Finally, the scoreboard entry for the instruction is cleared and the dependence vector bits for instructions waiting in the IB for the released registers are cleared.

2.4 GhOST Design

This section presents GhOST, a non-speculative, lightweight, hardware-only OoO instruction scheduling technique designed to reduce GPU stalls. GhOST works with

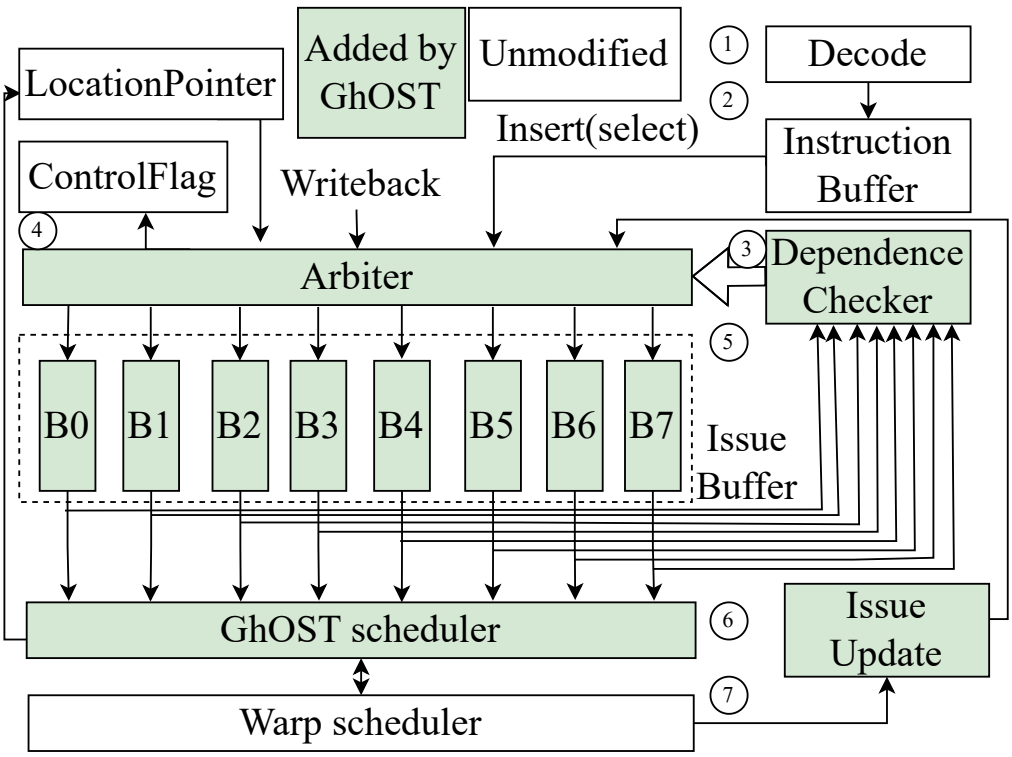


Figure 2.6: Architectural diagram for GhOST modifications at the decode stage.

existing warp-scheduling techniques by selecting an unblocked instruction from each warp to be considered for issue by the warp scheduler.

For an instruction to be issued OoO, it must not have any dependence conflicts with issued in-flight instructions, as well as older instructions that are still waiting to be issued in the Issue Buffer (IsB). Figure 2.6 shows the GPU pipeline modifications at the decode stage made by GhOST to enable OoO execution. GhOST selects instructions in-order from the IB ② after they are decoded by the decoder ① and places them in a time-shared Dependence Checker (DC) ③. The DC marks any dependence of this instruction with in-flight instructions and instructions currently in the IsB ⑤ to stop any incorrect OoO execution. Additionally, it ensures no reordering of load instructions against stores and the correct execution of control instructions.

After dependence checking, the instructions are placed in the IsB. The time-shared GhOST scheduler ⑥ selects and stores 2 instructions per warp, which do

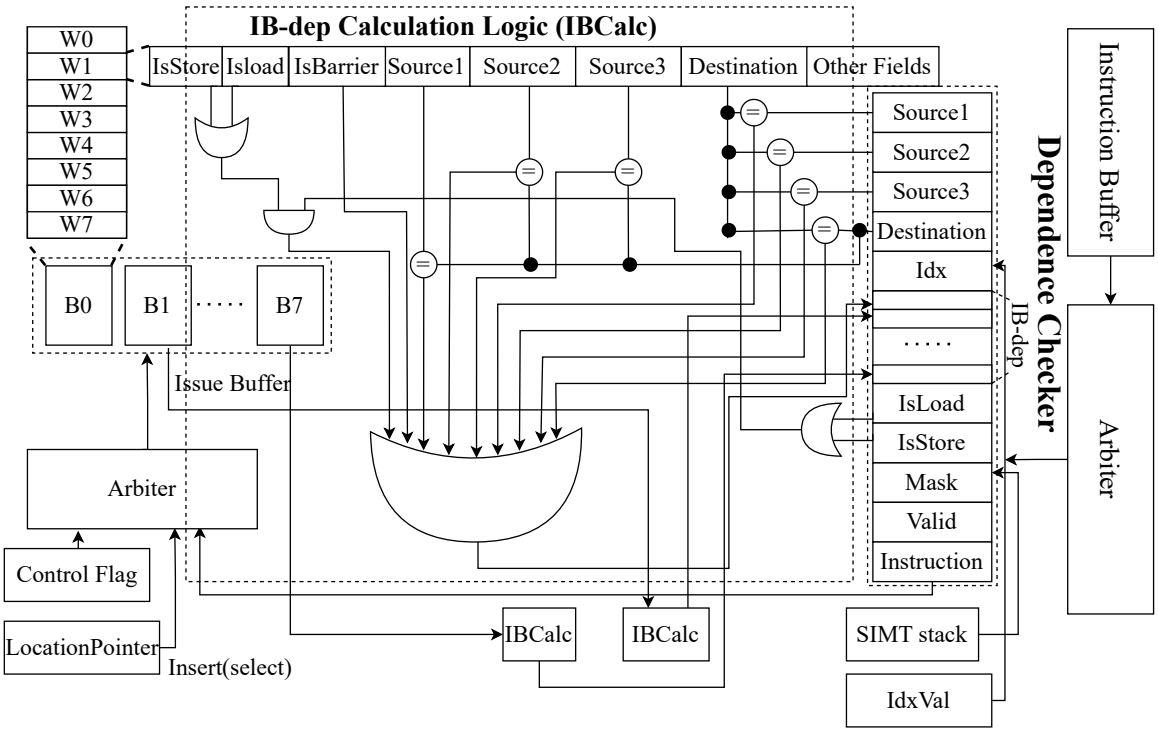


Figure 2.7: Dependence checking of new instruction with one instruction updating its corresponding IB-dep bit in IB-dep vector.

not have any dependence conflicts with in-flight instructions and older instructions in the IsB in the Instruction Table (ITab) which is contained within the GhOST scheduler. The warp scheduler ⑦ remains unmodified and looks at one instruction per warp in the ITab to be considered for issue. Details of these structures are explained below.

2.4.1 Dependence Checker

Figure 2.7 shows the hardware diagram of the DC, which is a time-shared structure for warps in a scheduler. Each scheduler has four DCs, equal to the decode throughput. The DC comprises of 8 *IB-dep Calculation Logic* (IB-Calc) structures. Each IB-Calc parallelly checks for any write-after-read (WAR) dependence on a register being used by this instruction against registers used by instructions of the same warp which are in the IsB, in addition to the RAW and WAW dependence

check with in-flight and older IsB instructions done in the baseline, to stop incorrect OoO issue of instructions from the IsB. If any dependence is found between the instruction in the DC and any instruction in the IsB, the IB-dep bit-vector entry corresponding to the instruction in the IsB is set to 1. Additionally, it prevents the reordering of loads against stores and stops OoO execution till control instructions are resolved. Details of this are discussed below. When the DC has space, the oldest instruction in the IB from a warp, based on the decode scheduler, is placed in the DC. The DC sets the following bits for an instruction before it is placed in the IsB:

- Valid: Validity of instruction.
- Mask: The mask for the threads in the warp is copied from the SIMT stack.
- IB-dep: Bit-vector where each bit corresponds to an instruction position in the IsB and is set by an IBCalc and a bit for each register to check if it depends on an in-flight instruction (discussed later).
- Instruction: The decoded instruction.
- Idx: Bits to track the order in which instructions were placed in the IsB for a warp.

Table 2.2 shows the breakdown of the total bits per instruction in the DC.

Each IB-Calc unit performs the following checks against the corresponding IsB instruction before setting its bit in the IB-dep vector:

1. **Handling data dependencies:** In addition to keeping track of any RAW and WAW dependencies with in-flight and pending IsB instructions as in the baseline, it also checks for any WAR dependencies of this instruction with any older instruction in the IsB. To keep track of dependencies in the IsB, the

Table 2.2: Decode Checker Configuration

Field	Bits	Usage
Valid	1	Instruction has no control hazard
Index	3	Tracks age of instructions
Mask	32	Tracks active threads for the instruction
IB-dep	11	Checks for dependence with an older instruction in the IsB
Instruction	64	Decoded instruction to be executed
Combined	111	

read registers of the instruction in the DC are checked against the write registers of the instruction in the IsB to identify RAW hazards. Similarly, the write register of this instruction is checked against the read and write registers of the instructions in the IsB to identify WAW and WAR hazards. If any dependence is found, the bit in the IB-dep vector corresponding to the hazard-causing instruction in the IsB is set to 1.

2. **Enforcing order for memory instructions:** GhOST enforces memory instructions to execute in-order except for load-load pairs. This implies that load instructions to the same addresses can also be reordered, which is often not allowed in memory consistency models. However, the only way a thread can observe if two load instructions from the same address have been reordered is if another thread writes to the location. However, this would be a data race [75], and the load re-ordering is allowed. GhOST does not reorder instructions marked as atomics by the compiler as this would violate memory model semantics. The *IsLoad* and *IsStore* bits are used to determine if the instruction is a memory instruction. If the instruction in the DC is a load or a store instruction, and the instruction in the IsB it is being compared

against is a store instruction or an atomic memory instruction, then the DC IB-dep bit vector entry corresponding to the IsB memory instruction is set to 1. Since instructions are issued only if their IB-dep bit vector is 0, this ensures that only load-load re-orderings are allowed.

Handling Control Instructions: GhOST does not implement branch prediction. Perfect branch prediction leaves the performance of GhOST virtually unmodified (see Table 2.5). Additionally, speculative execution comes with the additional complexity of enabling rollback for thousands of threads on mis-speculation.

GhOST allows the reordering of branch instructions with older instructions in the IsB while respecting all dependencies. Instructions after a branch instruction are not issued until the branch is resolved, avoiding any speculation. The Mask for the branch instruction is popped from the SIMT stack and placed in the DC, which is copied to the IsB when the instruction is moved. This avoids the serialization of instructions against branch convergence points.

Handling Synchronization Instructions: When a synchronization instruction enters the DC, it is not moved to the IsB until all the instructions from the IsB have been issued. When the synchronization instruction (e.g., `syncthreads`) is moved to the IsB, the control flag is raised, and new instructions are not moved to the IsB. After the synchronization instruction has been issued and the branch has been resolved, the control flag is unset, and instructions can be moved from the IB to the DC.

Mask: The thread mask for the instruction is picked up from the SIMT stack and the entry is popped from the SIMT stack.

Idx: The IdxVal is copied as the Idx in the DC, and the IdxVal counter for the warp is incremented.

2.4.2 Issue Buffer

Instructions are moved from the DC to the IsB from where they are selected for issue using the GhOST scheduler. When an instruction is moved into the IsB, it is placed in an empty slot pointed to by the Location Pointer for the warp, which is a first-in, first-out (FIFO) queue of locations of empty IsB slots for the warp. The Location Pointer then points to the next empty location or marks the IsB as full, so new instructions are not moved from the DC to the IsB. Index bits are used to track the order of instructions. The IsB is modeled as 8-way banked structure, and each entry of a given bank corresponds to a different warp. Instructions are read from the IsB by the GhOST scheduler to find valid instructions with no data dependence, which can be considered for issue by the warp scheduler. When an instruction is issued from a warp, the IB-dep bits corresponding to the issued instruction are set to 0 for the instructions in the IsB for that warp.

2.4.3 GhOST scheduler

GhOST introduces the GhOST scheduler, which is a time-shared structure across warps that serves two functionalities: 1) It selects up to two instructions per warp from the IsB that do not have a data dependence on any older instructions in the IsB and in-flight instructions. 2) It buffers these instructions in the ITab for the warp scheduler to consider during the issue stage scheduling. The selection logic involves choosing instructions for a given warp at any given time. The GhOST scheduler selects a warp to place instructions in the ITab if 1) there was a write-back to the warp or 2) the warp has empty space in the ITab. The time-shared nature of the GhOST scheduler reduces the area and power overhead of implementing GhOST. Additionally, it leaves the interface to the warp scheduler unmodified.

Selecting Warp for Dependence Checking The GhOST scheduler maintains a *Warp-WriteBack* (WWb) bit-vector and an *ITabEmpty* (ITE) bit vector, where each bit corresponds to a warp. When a warp has a write-back, the corresponding bit in the WWb is set. When a warp has empty space in the ITE, the corresponding ITE bit is set to one. The ITE bit is unset for a warp after instructions is placed in the ITab.

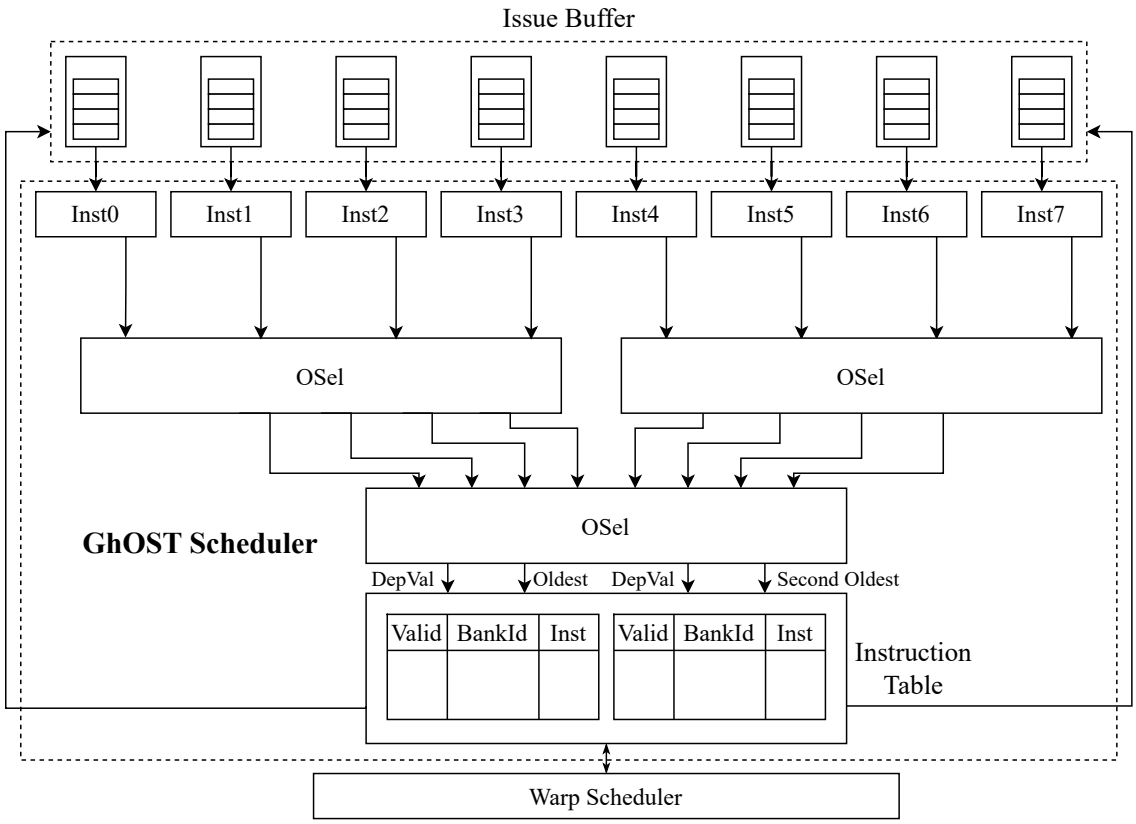


Figure 2.8: The time-shared GhOST scheduler.

Selecting Ready Instructions The hardware implementation of the GhOST scheduler is shown in Figure 2.8. When the GhOST scheduler selects a warp (as described above), the Register Dependence Checker (RDC) is used to check for data dependencies (See Figure 2.9). The IB-Dep vector of each instruction in the warp is checked for data hazards with another instruction in the IsB or any in-flight instructions.

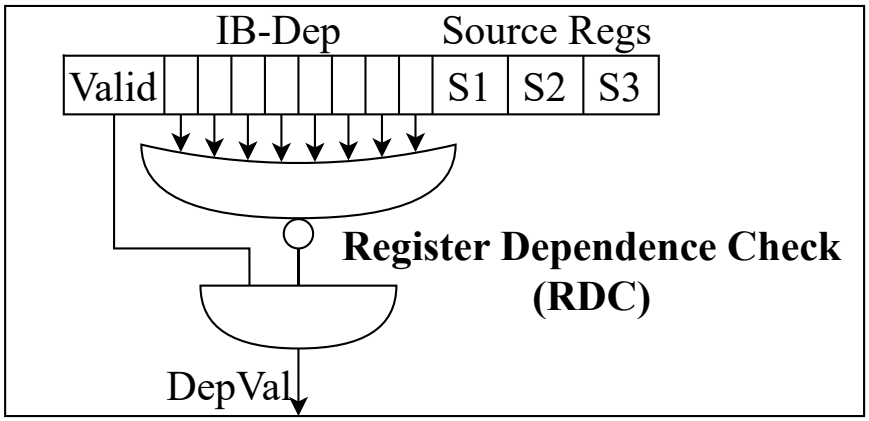


Figure 2.9: Register dependence checking hardware for the GhOST scheduler.

Additionally, it is checked if the instruction is valid (See Figure 2.9). If the instruction has no data dependencies and is valid, the RDC gives an output of 1. This output is fed into the OSel (see Figure 2.10). The OSel selects the two oldest instructions with no data dependencies. It places the instructions in the ITab, sets the bit of the placed instructions as 1, and stores the bank id of the location of this instruction in the IsB.

Issuing Instructions The warp scheduler selects instructions for issue from the ITab. When an instruction is issued from the ITab, for all the instructions in the IsB belonging to the same warp as the issued instruction, the IB-dep bit corresponding to the issued instruction is set to 0. This marks all WAR dependencies of the issued instruction on newer instructions in the IsB as resolved. RAW and WAW dependencies of in-flight and older IsB instructions are checked against the dependence vector before issue (same as baseline), ensuring correct execution. Additionally, for all instructions from the warp with an *idx* value greater than the *idx* of the issued instruction, the *idx* value is reduced by one. The entry of this instruction in the IsB is marked as invalid, and the bank id is added to the Location Pointer queue for the warp.

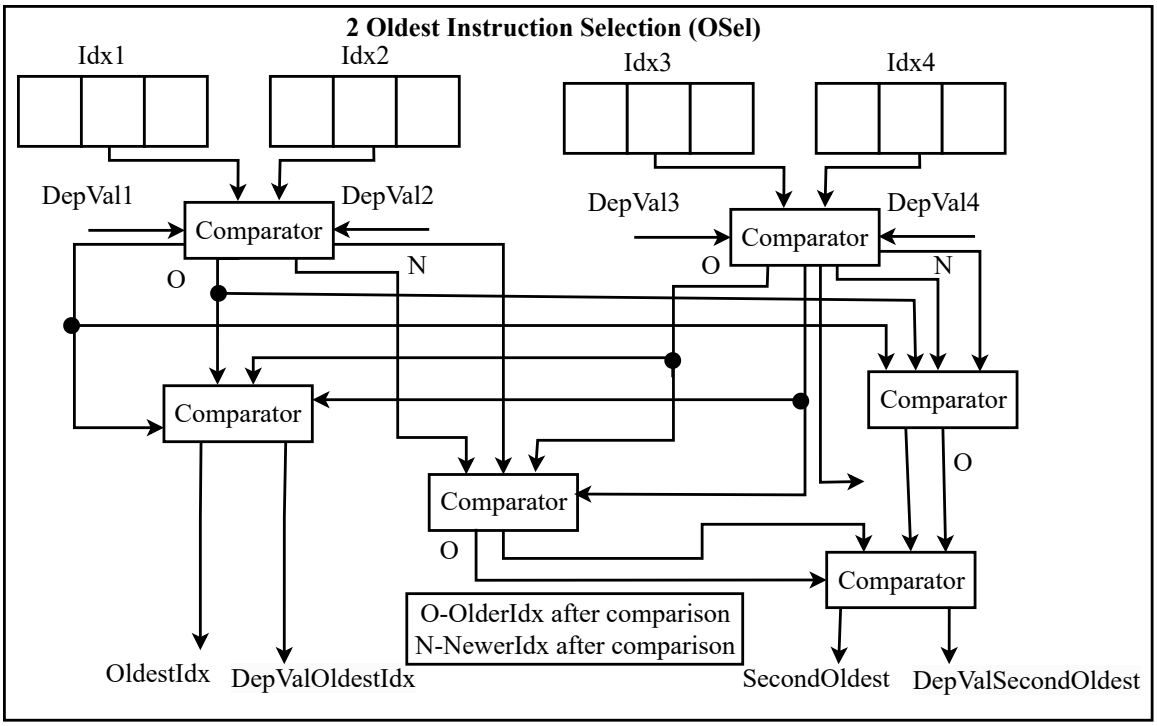


Figure 2.10: Oldest instruction selection hardware for the GhOST scheduler.

2.4.4 Warp scheduler

The warp scheduler is unmodified by GhOST. The warp scheduler considers the oldest ready instruction pointed to by the GhOST scheduler. If the GhOST scheduler has not marked any instruction as ready for issue for a warp, then the warp scheduler considers the oldest instruction for the warp. The warp scheduler checks for any structural hazards before issuing an instruction for operand collection.

2.4.5 Software Optimizations for OoO execution

GPUs boast a vast register space, offering up to 255 registers per thread [92]. However, this potential is often not fully harnessed during program execution [54, 48]. By leveraging these underutilized register files, the compiler can enhance OoO performance through strategic loop unrolling and register renaming, thereby mitigating false dependencies. This optimization broadens the scope for reordering in GhOST, leading to a marked improvement in OoO performance, as detailed in Section 2.6.5.

2.4.6 Exception Handling:

Nvidia GPUs have very limited support for exception handling [84], and the process running on the GPU is terminated on encountering errors [85]. When a warp encounters a page fault, the page fault is treated as a long stall on Nvidia GPUs [89]. Since GhOST is non-speculative, it can handle page faults in GPUs without any modifications. Like current GPUs, GhOST can return the error code for an exception that happens during execution. However, below we describe how to add support for precise exceptions, if needed.

GhOST-Precise: GhOST can be extended to support OoO execution with precise exception handling for virtual memory in the event kernel-level context switching [109, 94] on page faults is desirable. Prior work, iGPU [79], explored how to support demand paging exceptions in GPUs by exploiting idempotent code regions in the context of *in-order* instruction execution. An idempotent instruction sequence is one that can be re-executed without changing the result. Building on the insights of iGPU, GhOST-Precise utilizes compiler support to mark non-idempotent instructions and adds two additional constraints to GhOST:

- Load instructions are not reordered against each other
- Non idempotent instructions are not issued till all prior load instructions have hit the TLB.

The additional hardware required to support GhOST-Precise is described below:

- LoadCounter: Each warp has a load counter to maintain the number of loads in the pipeline which have not done a TLB check. When a load instruction enters the DC unit it increments the *LoadCounter*, and decrements it after the TLB check.

- InstLoadCounter: When an instruction enters the DC, the value of the *Load-Counter* is copied into the *InstLoadCounter* and placed with the instruction in the IsB. When a load instruction for a warp has a TLB hit the *InstLoad-Counter* for all entries in the IsB for the warp are right-shifted by 1 bit.

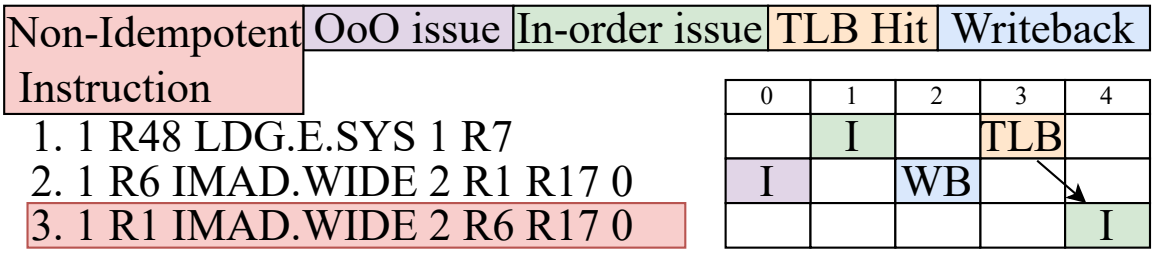


Figure 2.11: GhOST-Precise does not issue non-idempotent instructions until prior load instructions hit in the TLB.

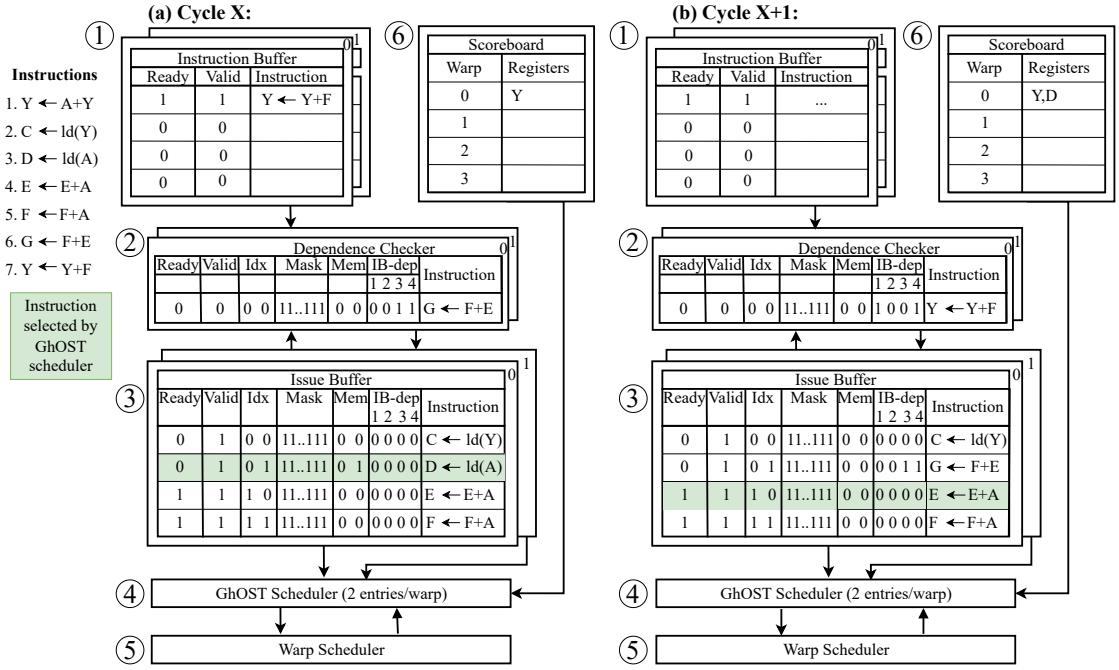


Figure 2.12: Execution of a small program on GhOST.

If an instruction is marked as non-idempotent by the compiler, it can be issued only after the *InstLoadCounter* value for that instruction is 0. Figure 2.11 shows how instruction 3 which is non-idempotent is not issued OoO against instruction 1 even though there is no dependence between them till instruction 1 hits in the TLB.

If instruction 1 missed in the TLB, GhOST-Precise can start re-execution from instruction 1, re-issue instruction 2 as it is idempotent and then issue instruction 3.

2.4.7 GhOST in action

Figure 2.12 shows the movement of instructions between the IB, DC, and IsB as a small program executes OoO using GhOST. Figure 2.12(a) shows an execution cycle where instruction 1 has been issued. Instruction 2 is stalled as it has a RAW dependence on instruction 1. The GhOST scheduler has marked instruction 2 as the oldest instruction ready for issue, and instruction 4 as the next oldest ready for issue in the previous cycle. The warp scheduler issues instruction 3 OoO after checking for structural hazards. Warp 0 goes to the GhOST scheduler to get the new oldest ready instructions.

Figure 2.12(b) shows the next cycle, where instruction 6 is moved from the DC to the IsB, and instruction 7 is moved into the DC. The GhOST scheduler is pointing to instruction 4 as the oldest ready instruction, which is issued OoO from the warp scheduler.

2.5 Evaluation Methodology

This section describes the experimental context used to evaluate GhOST.

Modeling GhOST: GhOST is implemented on top of the simulated Nvidia RTX 2060S and RTX 3070 GPUs, which are included with the Accel-Sim [53] simulator; Table 2.3 shows the microarchitectural parameters of the GPU.

To implement GhOST in the simulator, a dependence checker stage was added between the IB and the IsB. The GhOST scheduler was connected to IsB, the scoreboard and the warp scheduler. OoO control logic was added for the GhOST scheduler. The depth of the decoder, IB and IsB was parameterized to study the effect of

Table 2.3: GPU Microarchitectural Parameters

Parameter	**Nvidia RTX 2060S**
Simulator	Accel-Sim
Generation	Turing
SM clock Frequency	1905 MHz
SMs/GPU	34
Warp Schedulers/SM	4
Warp Scheduling Policy	Greedy-Then-Oldest (GTO)
SIMT lane width	32
Max #warps/SM	32
Max #threads/SM	1024
Max Registers/SM	65536
Register File Size	256 KB, 34k 32bit Registers
L1 Cache Size	64KB
IB Size	4 inst/warp
Inst. Issue/Warp	1
Parameter	**Nvidia RTX 3070**
Simulator	Accel-Sim
Generation	Ampere
SM clock Frequency	1132 MHz
SMs/GPU	46
Warp Schedulers/SM	4
Warp Scheduling Policy	Greedy-Then-Oldest (GTO)
SIMT lane width	32
Max #warps/SM	48
Max #threads/SM	1536
Max Registers/SM	65536
Register File Size	256 KB, 34k 32bit Registers
L1 Cache Size	64KB
IB Size	4 inst/warp
Inst. Issue/Warp	1

increasing IsB size on GhOST.

Benchmarks: 28 GPU applications from standard benchmark suites, namely Rodinia [18], GPGPU-Sim [53], Pannotia [17], and Tango [51] were used to evaluate GhOST.

The benchmarks cover various GPU domains, including irregular graph processing and large-scale neural networks, and are classified in Table 2.4. SASS traces were collected on the Nvidia RTX 2060S GPU for these applications to feed into

Table 2.4: Evaluated Benchmarks

Benchmark Suite	Benchmarks
Ispass [53]	bfs (IBFS), LPS, LIB, RAY, STO
Pagoda [125]	beamformer (BFM), DCT, DES, convolution (CNV), mandelbrot (MND), matrixMul (MTM), multiwork (MWK)
Pannotia [17]	SSSP, Floyd-Warshall (FW)
Rodinia 3.1 [18]	b+tree (B+T), dwt2d (DWT), Gaussian (GAS), lavaMD (LAV), LUD, NN, bfs (RBF), myocyte (MYO), particlefilter-float (PFF), srad-v1 (SRA), backprop (BAK)
Tango [51]	CN, GRU, LSTM

the simulator. Inputs were selected to maximize the input size while ensuring that the GPU did not run out of memory during SASS collection. We could not collect SASS traces for all benchmarks as the GPU ran out of memory when running these applications, the NVbit tool [87] which is used for collecting SASS traces crashed or the application ran for over six days.

Best Performing Prior Out-of-Order Approach: We compare GhOST with LOOG [46], which implements a CPU-style OoO mechanism for the GPU, containing expensive OoO components, such as register renaming and load-store queues. The LOOG Accel-Sim implementation provided with the publication has been used.

2.6 Evaluation of GhOST

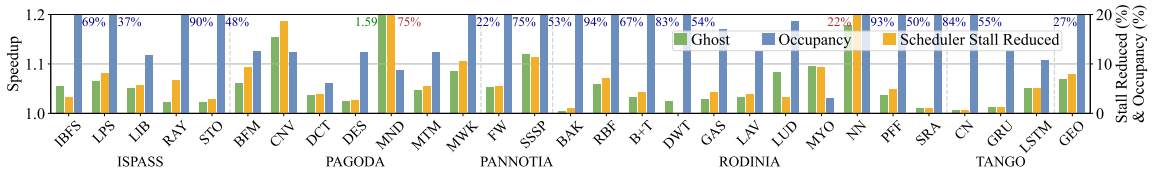


Figure 2.13: The performance of GhOST and the reduction in scheduler stalls with OoO execution plotted against the occupancy of applications on the simulated RTX 2060S GPU.

2.6.1 Performance of GhOST and Sensitivity to applications' occupancy

Figure 2.13 shows the speedup of GhOST over the SASS baseline, the reduction in scheduler stalls with OoO execution and the average occupancy for each benchmark on the simulated RTX 2060S GPU. GPU occupancy measures the ratio of active warps to the maximum possible warps on an SM, reflecting the utilization of computational resources. Both GhOST and the baseline have 4 instructions in the IB. GhOST gives a speedup of 6.9% over the SASS baseline on the simulated RTX 2060S GPU. The reduction in the scheduler stalls when a scheduler cannot find any instruction ready for issue due to data and/or structural hazards reduces in proportion to the speedup lent by GhOST using OoO execution. Although SASS offers the most optimized binary for the hardware it runs on, GhOST can further improve the runtime of the programs by dynamically reordering instructions, even for applications with high occupancy, while significantly improving the performance of low occupancy workloads.

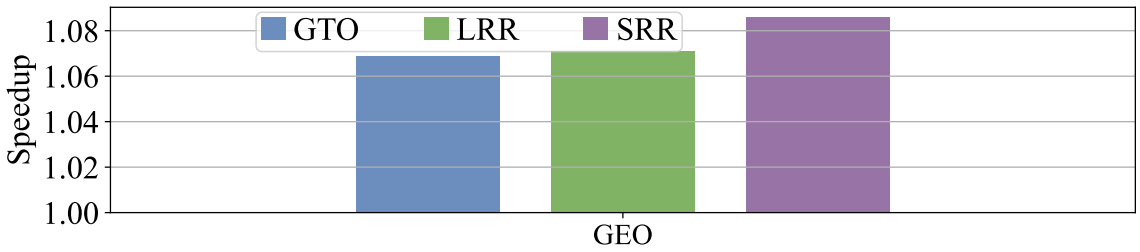


Figure 2.14: The performance variance of GhOST with different scheduling policies on the simulated RTX 2060S GPU.

2.6.2 GhOST with various warp scheduling policies

Figure 2.14 shows the performance of GhOST with three warp scheduling policies on the simulated RTX 2060S GPU: GTO which selects the same warp for scheduling until it stalls, then moves to the oldest ready warp; Lose round robin (LRR)

which round robins over warps to find a ready warp for issue; and strong round robin (SRR) which only looks at the next warp to be considered for scheduling. As the scheduling policy becomes less aggressive (SRR), the performance of GhOST improves as OoO execution has a greater impact on finding instructions that are ready for issue.

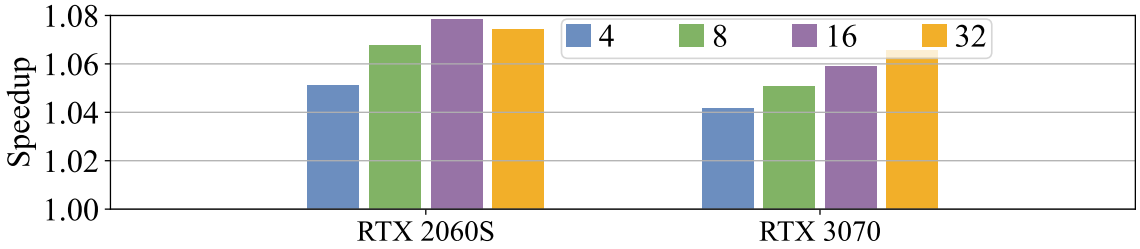


Figure 2.15: The performance variance of GhOST with changing IsB size on simulated Nvidia RTX 2060S and RTX 3070 GPUs.

2.6.3 Sensitivity of GhOST to Issue Buffer size

Figure 2.15 shows that the size of the IsB influences the performance of GhOST's OoO execution, while the size of the IB remains constant on the Nvidia RTX 2060S and RTX 3070 GPU models belonging to the Turing and Ampere architectures, respectively. GhOST consistently outperforms in-order execution, and the geomean speedups for an IsB size of 8 is 6.9% on RTX 2060S and 5% on RTX 3070. As the depth increases further to 16 and 32 IsB entries per warp, the performance of GhOST further improves to 7.9% and 7.5% for RTX 2060S and to 5.9% and 6.5% for RTX 3070 (for 16 and 32, respectively). While configurations 16 and 32 outperform IsB size 8, they exponentially increase the control logic required to enable GhOST. We chose an IsB size of 8 for GhOST, considering the trade-off between OoO performance and hardware overhead. GhOST gives 31% of limit study speedup on RTX 2060S.

Furthermore, performing a sensitivity analysis around IB size reveals that increasing this size to 8 results in a negligible performance improvement of 0.09%.

2.6.4 GhOST-Precise

Figure 2.16 shows that GhOST-precise has a small slowdown due to the constraints added to support precise exceptions for virtual memory handling. GhOST-precise reduces the geomean performance to 6.3% on the simulated RTX 2060S GPU. Increasing the IsB size improves the performance of GhOST-Precise.

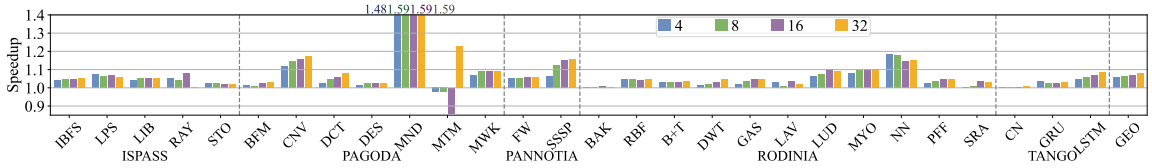


Figure 2.16: The performance variance of GhOST-Precise with changing IsB size on the simulated RTX 2060S GPU.

2.6.5 Effect of various OoO optimizations on GhOST

Table 2.5: Effect of Various OoO Optimizations on GhOST on the Simulated RTX 2060S GPU.

Unbounded Register Renaming	✓	✓	✓	✗	✓	✗	✗	✗
Perfect Memory Alias Check	✓	✓	✗	✓	✗	✓	✗	✗
Perfect Branch Prediction	✓	✗	✓	✓	✗	✗	✓	✗
Speedup %	15.1%	14.4%	13.7%	8%	10.1%	7.4%	7.3%	6.9%

Table 2.5 delves into a limit study analyzing various optimizations applied to OoO execution on GPUs with a very deep reorder queue, considering perfect branch prediction, unbounded register renaming, and perfect memory address alias checking. The results highlight that perfect branch prediction and reordering of load operations against store operations offer limited performance benefits. Notably, register renaming enhances performance significantly by expanding the number of instructions for OoO execution. The performance benefits from these optimizations are discussed in detail below.

Perfect Branch Prediction: Perfect branch prediction improves OoO execution by enabling extensive instruction reordering without stalling on branches. It ensures correct branch direction decisions, even in cases involving long-latency operations for branch resolution. Branch prediction has a limited impact on the OoO performance of GPUs. Additionally, implementing perfect branch prediction is prohibitively expensive, requiring the rollback of all threads in a warp in case of misprediction. Due to this high overhead, there has yet to be prior implementation of branch prediction on GPUs to the best of our knowledge.

Perfect Memory Alias Checking: Memory alias checking is crucial for reordering load and store instructions. Implementing perfect alias checking, where address alias information is known at the time of instruction issue, has limited impact on OoO performance, as shown in Table 2.5. This implementation necessitates load-store queues or speculative execution of load instructions, resulting in high hardware overhead with limited performance benefits.

Register Renaming: Aggressive register renaming can help the performance of OoO execution by removing false dependencies. In the perfect register renaming implementation, a new register is assigned to an instruction at the issue stage in case of an anti or output dependence from an unbounded list of free registers. In many GPUs, all warps share a single physical register file. In Table 2.5, we do not consider the potential negative impact of reduced warp occupancy that may be required to free up registers for renaming.

GPUs have a large register space per thread with a maximum limit of 255 registers per thread [92]. The GPU register file is not fully utilized through the execution of the program [54, 48]. The compiler can also assist in removing false dependencies by unrolling loops and using more registers during the program's execution. Figure 2.17 compares the performance gains of unbounded register renaming in hardware against aggressive unrolling and renaming in software with a limited set

of registers (255 registers per thread) for the lavaMD and convolution benchmarks which show significant performance gains by unbounded register renaming. The results show that while unbounded renaming does outperform renaming and unrolling done by the compiler, this performance gap is not significant. This suggests that more aggressive unrolling and higher register utilization to reduce name dependencies can enable more aggressive OoO execution with GhOST without implementing register renaming in hardware.

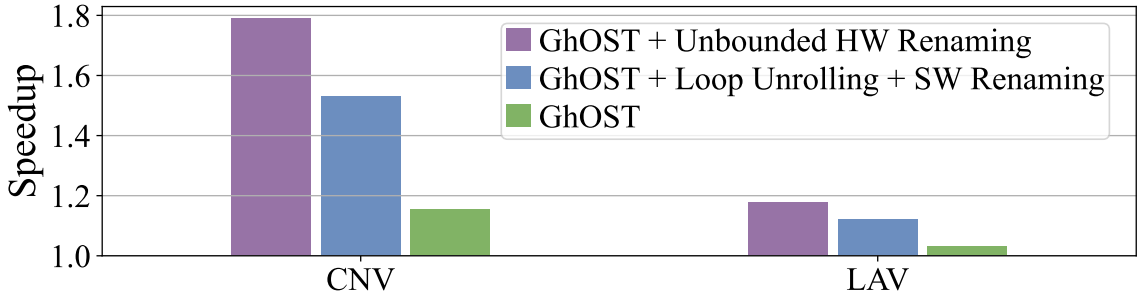


Figure 2.17: Comparison of unbounded register renaming in hardware with bounded register renaming (255 registers) in software.

Despite minimal impacts from individual GPU OoO optimizations, their collective implementation significantly boosts performance. Issues like false dependencies stemming from name conflicts limit gains from alias checking and branch prediction by hindering OoO instruction issuance. Integrating register renaming with these optimizations overcomes these limitations, enabling more aggressive OoO execution and resulting in significant performance improvements.

2.6.6 Area and power efficiency

We modeled the design of the Dependence Checker, Issue Buffer, and the GhOST scheduler in RTL and used Synopsys [32] to calculate the area and power of the components added by GhOST on the 45 nm technology node. The GhOST units add 1276 um^2 of area per scheduler, or 0.173536 mm^2 area per GPU. The Nvidia RTX 2060S GPU was developed on the 12nm technology node and has an area of

445mm^2 [110]. To get the area estimate for GhOST, we use a multiplicative factor of 0.17 to convert the area analysis to a 14nm technology node to get an upper limit estimate for the area increase with GhOST [107]. GhOST increases the area of the GPU by 0.007% with a 1.1707 mW increase in power.

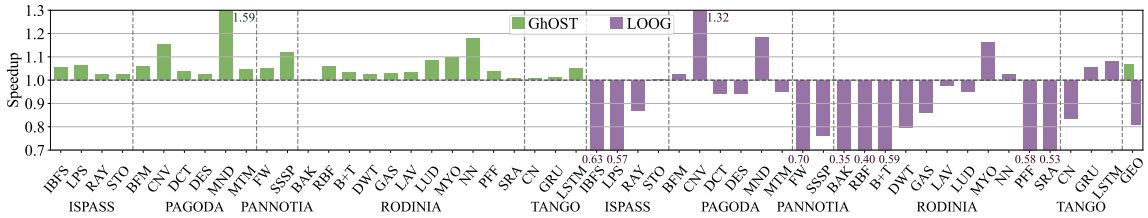


Figure 2.18: The performance of GhOST and LOOG on the simulated RTX 2060S GPU.

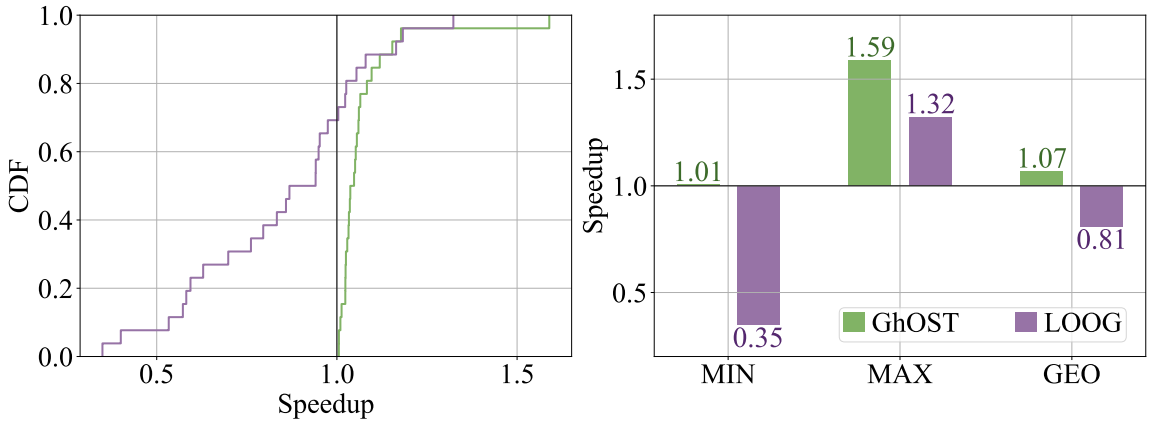


Figure 2.19: Performance of GhOST and LOOG against in-order execution on the simulated RTX 2060S GPU configuration.

The evaluation depicted in Figure 2.18 and Figure 2.19 demonstrates GhOST's superior performance compared to the state-of-the-art prior work, LOOG [46] on the RTX 2060S GPU model. Figure 2.18 shows that GhOST does not negatively affect the performance of any applications, unlike LOOG. Results for the LIB and multiwork benchmarks are missing as LOOG crashed for the applications. In Figure 2.19, GhOST shows a higher maximum performance than LOOG with no slowdowns on any applications, while LOOG shows a maximum slowdown of 0.35×. The cumulative frequency distribution of LOOG shows a very high variability in

the performance of LOOG for various applications, while GhOST has a more uniform, non-negative performance impact on applications.

Although LOOG outperforms GhOST in specific applications by efficiently renaming registers and reordering memory instructions, it experiences substantial slowdowns in all other scenarios. This is due to LOOG's reliance on the operand collection stage for OoO execution, where instructions wait in the reservation station until the write-back phase. As a result, the operand collector may become congested with stalled instructions due to true dependencies. This congestion prevents ready-to-issue instructions from proceeding due to structural hazards, leading to significant performance degradation. Furthermore, LOOG lacks a specific mechanism for congestion-aware CU allocation, rendering it highly susceptible to the warp scheduler's scheduling policy. In contrast, GhOST does not negatively affect the performance of any application and is independent of the scheduling policy. Despite efforts to improve performance by increasing the operand collector's size fourfold to 32, it remains smaller than GhOST's IsB capacity for holding instructions.

LOOG introduces significant complexity to the pipeline through a register alias table linked to all collector units for register renaming, load-store queues for memory address disambiguation across threads, and a result broadcast bus. These structures need to manage information for every thread in a warp per instruction, necessitating substantial buffers for storage and broadcast. LOOG modifies the issue, operand collector, and writeback stages. In contrast, GhOST avoids this complexity by making limited modifications to the decode stage with the DC, IsB, and the GhOST scheduler, eliminating the need to store per-thread information for each instruction in a warp.

These findings deviate from their previously published work, which employed the PTXPlus flag on GPGPU-Sim 3 to simulate SASS, which offers limited support for an older version of SASS (10+-year-old Nvidia GT200 GPU) [35] and is incompati-

ble with the newer Pascal architecture they claim to use for their evaluation. As a result, performance numbers in the LOOG paper were overestimated due to the unoptimized version of SASS modeled by the simulator. Moreover, we identified and rectified issues related to synchronization instructions, branches, and dependence checking in their SASS model implementation in the simulator. We have confirmed our results for LOOG when using trace-driven SASS simulation with the LOOG authors.

2.7 Related Work

Prior work has explored both software and hardware approaches to enhance GPU utilization.

2.7.1 Scheduling Policies

Prior work explores different warp interleavings using various warp scheduling techniques to improve GPU resource utilization and reduce stalling. Narasiman *et al.* [82] propose a Two Level Scheduler (TLS) combined with a Large Warp Microarchitecture (LWM). TLS helps distribute long-latency stalls over time, while LWM focuses on creating SIMD-width dynamic subwarps to improve execution units' utilization in warp divergence. RLWS [8] proposes a reinforcement learning-based warp scheduler that can learn and adapt to various workloads. MASCAR [101] implements a memory-aware warp prioritizing scheduler with a cache-access re-execution mechanism to overlap more computation with memory accesses. These schedulers are typically tailored for specific workloads, such as cache-sensitive or compute-heavy.

NVIDIA's Shader Execution Reordering (SER)[91] is a scheduling technique that regroups threads on the fly to reduce execution and data divergence. GhOST can

be used with SER to improve the performance of GPUs further.

Twin-Kernel (TK) [33] optimizes GPU memory access by compiling altered kernels with varied memory patterns, pairing them for efficient execution. However, TK's static nature requires creating two versions per kernel, that is not always possible. GhOST offers a dynamic alternative without workload assumptions.

HyperQ [13] by NVIDIA allows empty SMs to start running subsequent independent kernels. This enables multiple kernels to run simultaneously on the GPU, thus reducing the total run time of the program. However, this does not help address the stalls each kernel will face. GhOST is compatible with HyperQ.

Breathing Operand Windows [28] bypasses register file accesses, passing values directly between instructions within the same window, improving the rate of instructions being issued from the OC, and can be used with GhOST to further improve performance.

2.7.2 Warp Throttling Techniques

Many works have studied how multiple warps executing on an SM affect the performance of the L1D cache, network bandwidth utilization, and DRAM; and observe that multiple warps sharing the L1D cache can lead to cache thrashing and performance degradation. Studies have been done on reducing the maximum number of active warps at a time on one SM (TLP throttling) or prioritizing specific warps' access to shared resources. Rogers *et al.* proposed CCWS [98], which uses a scoring system to identify warps that re-reference their data in the L1D cache and prioritize their memory accesses.

Dyncta [52] uses a CTA scheduling mechanism based on the application's characteristics to limit the number of CTAs assigned to an SM. This reduces the contention for shared resources between warps. OWL [50] develops a two-level warp scheduler with CTA prioritization. LCS [64] develops warp scheduling techniques

to throttle TLP and better utilize memory resources. PCAL [67] showed that while TLP throttling improves L1D footprint, it can lead to under-utilization of other resources and proposed a system to prioritize a subset of warps to use caches while allowing remaining warps to use other GPU resources.

Poise [25] uses machine learning combined with a run-time inference engine to make warp scheduling decisions, optimizing both the TLP and the memory system performance. Virtual threads [126] suggest that increasing the maximum number of CTAs on an SM can favor some applications. GhOST GPU improves the performance of applications by exploiting the hidden ILP in applications at the micro-architectural level using OoO. It thus improves the performance of workloads with regular and irregular memory access patterns. This complements the existing TLP models.

2.8 Conclusion

This chapter introduced GhOST, a novel and lightweight OoO technique for GPUs primarily residing in the decode state for instruction reordering. GhOST utilizes the decode stage for OoO execution, leveraging its stored pool of decoded instructions for warps, thereby expanding available instruction choices and eliminating the need for expensive OoO techniques to identify reordering opportunities. GhOST introduces instruction reordering by presenting an alternative instruction to the warp scheduler through the existing interface rather than offering the oldest instruction in each warp. The result is a simple and efficient OoO engine that operates predominantly in the decode stage. Experimental results on the simulated Nvidia RTX 2060S and RTX 3070 GPUs using SASS demonstrate that GhOST outperforms previous GPU OoO techniques, enhancing GPU performance by 6% and 4.6% across 28 benchmarks, with only a 0.007% increase in area.

Chapter 3

SHADOW: Simultaneous Multi-Threading Architecture with Asymmetric Threads

Memory-bound applications, such as deep learning workloads [41] and high-performance computing kernels [77], often exhibit irregular memory access patterns and data sparsity. These characteristics cause fluctuations between ILP and TLP during execution [57]. Traditional CPUs are typically optimized for either ILP or TLP, with OoO cores focusing on ILP and lean InO cores on TLP. This specialization can lead to underutilization of resources and performance inefficiencies when handling applications with dynamic parallelism requirements.

To address CPU inefficiencies, domain-specific accelerators have been developed to improve memory-bound performance [93, 117, 95, 29, 124]. While these accelerators achieve high throughput, they often sacrifice programmability, making them unsuitable for general-purpose workloads. GPUs provide greater flexibility but struggle with irregular memory access patterns, limiting their efficiency in sparse workloads [22]. Meanwhile, CPUs remain the most programmable option but strug-

gle to adapt dynamically to shifting ILP and TLP demands. While OoO CPUs effectively exploit ILP by reordering instructions, their performance deteriorates in sparse and memory-bound workloads, where ILP opportunities diminish. This results in inefficiencies when handling high-latency workloads such as sparse matrix-vector multiplication (SpMV) in iterative solvers [99], graph traversal for social network analysis [66], database query processing [96], stencil computations in high-performance computing [23], and deep learning workloads with large embeddings for recommendation systems [38].

Evolving Beyond Traditional SMT SMT improves execution unit utilization by allowing multiple threads to share resources. However, conventional SMT architectures trade off between ILP and TLP, either supporting a few OoO threads for ILP [71] or many InO threads for TLP [43]. For example, Intel's SMT designs prioritize ILP by supporting only two OoO threads per core. Prior work has explored adaptive SMT execution. MorphCore [108] switches between a 2-thread deep OoO mode and an 8-wide InO mode but lacks simultaneous ILP-TLP execution, leading to underutilization when workloads do not fit neatly into one mode. FIFO Shelf [103] speculatively routes instructions between OoO and InO paths, but at the cost of increased hardware complexity.

This chapter introduces SHADOW, a fundamentally different approach to balancing ILP and TLP. SHADOW enables the concurrent execution of OoO and InO threads within a single core, dynamically redistributing work without speculative instruction steering. Unlike prior approaches that switch execution modes, SHADOW allows simultaneous deep ILP and wide TLP execution, maximizing resource utilization.

Figure 3.1 illustrates how SHADOW dynamically adjusts work distribution. At low sparsity, minimal cache misses allow OoO threads to achieve high IPC, greed-

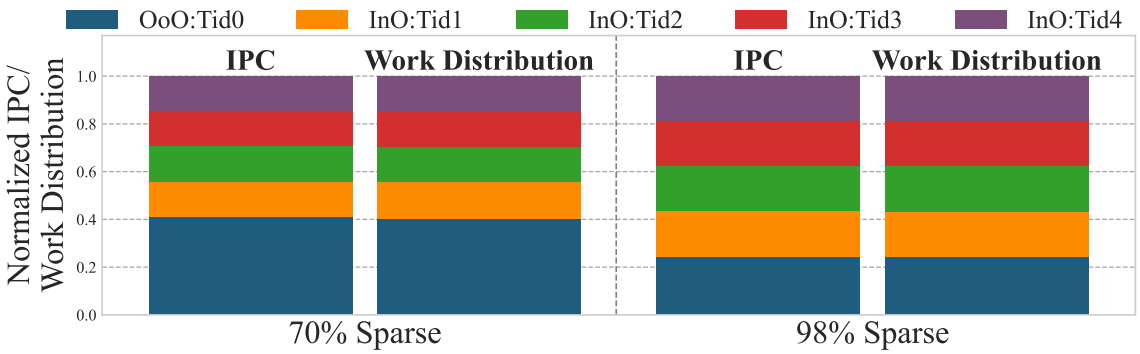


Figure 3.1: SHADOW dynamically redistributes work as IPC changes. High ILP skews execution toward the OoO thread, while work distributes equally among threads when the ILP is low. SHADOW adapts to the application without software intervention.

ily taking more iterations via software-driven work-stealing. As sparsity increases and L1 D-cache misses rise, OoO efficiency drops. When memory stalls limit OoO throughput, InO threads, free from speculative overhead, continue stealing work to sustain execution. This adaptation occurs without explicit hardware coordination, threads independently steal work as it becomes available, naturally distributing tasks based on ILP and TLP demands. While SHADOW, like MorphCore, partitions the physical register file among threads, a key distinction is that MorphCore operates in either OoO or InO mode at a given time, whereas SHADOW enables the simultaneous execution of both (see Section 3.2.3). This co-execution allows SHADOW to dynamically balance ILP and TLP within a single core, adapting more effectively to workload demands.

A key advantage of this hybrid approach is its efficiency in terms of area and power. Adding additional InO threads to an existing core increases area by only 1%, whereas adding a separate core would incur a significant increase in area overhead. Furthermore, the base core itself is often underutilized, making it more cost-effective to fully utilize existing resources before introducing additional cores. By enabling InO and OoO execution within a single core, SHADOW maximizes utilization while minimizing power and area costs, offering an efficient, general-purpose so-

lution for memory-bound workloads without the complexity of speculative instruction steering.

Contributions of This Work

1. A comprehensive study of ILP-TLP tradeoffs in modern CPUs, demonstrating the inefficiencies of traditional SMT architectures and highlighting the need for asymmetric multithreading (Section 3.1).

2. Through SHADOW, we demonstrate that an asymmetric SMT core can efficiently balance ILP and TLP by enabling simultaneous execution of InO and OoO threads, maximizing performance with minimal area and power overhead (Section 3.2).

3. An extensive evaluation of SHADOW across diverse workloads, spanning compute-bound and memory-intensive applications, demonstrates its adaptability to varying cache pressures, with a detailed study on Sparse Matrix Multiplication (SpMM) (Section 3.3).

3.1 Background and Motivation

Modern CPUs struggle to balance ILP and TLP, leading to underutilization. OoO execution extracts ILP but stalls on cache misses in memory-bound workloads, while InO execution exploits TLP but cannot hide long-latency memory operations. This section examines SMT's limitations in adapting to shifting ILP/TLP demands and introduces SHADOW, an asymmetric SMT architecture that integrates OoO and InO threads within a single core, dynamically redistributing execution resources without speculative overhead.

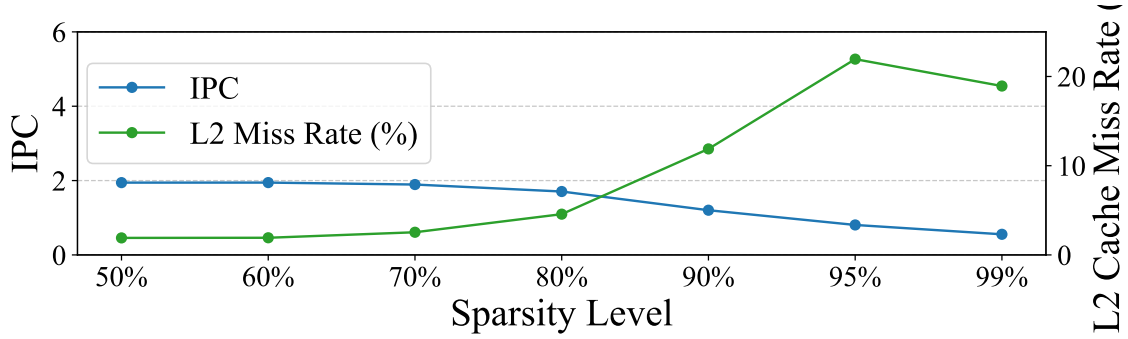


Figure 3.2: Change in the IPC and L2 cache misses of the SpMM application with varying sparsity.

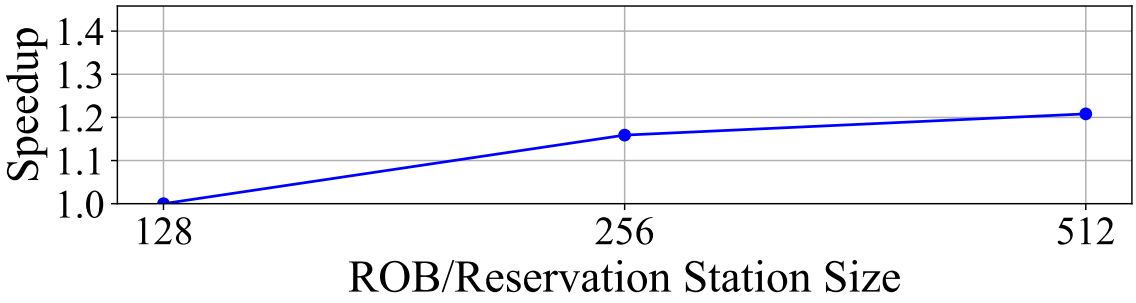


Figure 3.3: Speedup of a 90% sparse SpMM with varying ROB/RS size for a 1 OoO configuration. Area overhead of a larger OoO window does not translate into proportional speedup.

3.1.1 CPU Underutilization in Memory-Bound Workloads

Memory-bound applications fluctuate between ILP- and TLP-friendly phases, leading to CPU underutilization when execution resources sit idle due to limited ILP or TLP extraction.

Challenges in OoO Execution: OoO processors mitigate memory stalls using a large instruction window via the reorder buffer (ROB) and reservation stations (RS). However, these resources deplete quickly in memory-bound workloads. Figure 3.2 shows that in SpMM, increasing sparsity leads to rising L2 cache misses, severely degrading IPC as instruction windows remain underutilized.

Expanding the ROB and RS increases the instruction window, potentially improving performance in memory-bound workloads [72]. Figure 3.3 isolates this effect by scaling only the ROB and RS while keeping other structures unchanged. A

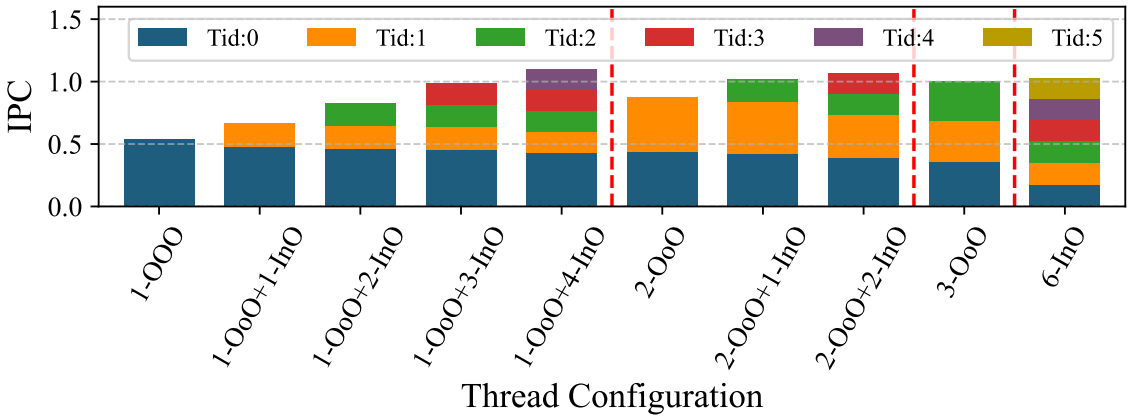


Figure 3.4: Breakdown of IPC contributions from all threads for 90% sparse SpMM.

larger ROB yields a modest 20% IPC gain but significantly increases wakeup-select CAM complexity, leading to high area and power overhead. The diminishing returns suggest that ROB scaling alone is insufficient for mitigating stalls in memory-bound scenarios. While increasing the instruction window improves performance in ILP-heavy workloads, it does not effectively leverage the TLP present in memory-bound workloads like SpMM. To fully exploit available parallelism, CPUs must also incorporate mechanisms to distribute work across multiple threads.

Challenges in Thread-Level Parallelism: Memory-bound workloads like SpMM exhibit inherent TLP, as matrix rows can be processed independently. However, traditional SMT struggles with resource allocation, causing contention among threads. Figure 3.4 isolates the impact of increasing thread count while keeping the ROB, LSQ, and physical registers fixed (Table 3.1). The results show that adding OoO threads initially improves IPC, but gains diminish beyond two threads due to ROB saturation and register file pressure.

The results indicate that while adding more OoO threads initially boosts IPC, the benefits diminish beyond two OoO threads due to ROB saturation and increased register file pressure. In contrast, the 1 OoO + 4 InO configuration achieves higher IPC than a 3 OoO setup by leveraging InO threads, which rely solely on the archi-

tectural register file (ARF) without adding speculative execution overhead. However, a fully InO configuration with 6 InO threads struggles in workloads requiring high ILP, highlighting the importance of SHADOW's hybrid execution model, which dynamically balances ILP and TLP to optimize performance.

3.1.2 Limitations of Traditional SMT Architectures

Existing SMT implementations do not dynamically adjust ILP/TLP execution balance, leading to suboptimal utilization. Two major prior SMT approaches have attempted to address this issue:

MorphCore [108] (Mode-Switching SMT): MorphCore allows a CPU core to switch between two execution modes based on the number of threads used by the application: (1) A deep OoO mode (up to 2-wide SMT) optimized for ILP. (2) A wide InO mode (up to 8-wide SMT) optimized for TLP. However, this approach does not allow ILP and TLP execution to coexist, leading to underutilization when workloads do not fit neatly into one execution mode.

FIFO Shelf [103] (Speculative SMT): FIFO Shelf augments OoO execution with an InO FIFO queue (the 'shelf') to improve SMT efficiency. Instructions predicted to execute in program order are steered to the shelf, bypassing the OoO structures such as the issue queue and reorder buffer, reducing contention. However, this requires sophisticated hardware mechanisms to ensure correctness, including dependency tracking across both execution paths, speculative wakeup, and misprediction recovery. Unlike FIFO Shelf, which speculatively partitions instructions from a single thread across execution modes, SHADOW assigns entire software threads to either InO or OoO execution. This eliminates the need for speculation and dependency tracking between execution paths, reducing complexity while preserving ILP-TLP balancing.

3.1.3 Why a Hybrid SMT Core is Needed Over Separate OoO and InO Cores

An alternative to asymmetric SMT is using separate InO and OoO cores instead of merging them into a single hybrid core. However, this approach is inefficient due to the following reasons:

(1) **Area Efficiency:** As we will show, when done judiciously, adding extra InO threads to an existing core increases area by only 1%, whereas adding a separate core incurs a significant area overhead.

(2) **Power Efficiency:** Hybrid SMT allows better resource sharing within a single core, reducing redundant hardware replication.

(3) **Execution Resource Utilization:** The base core is often underutilized, so SHADOW maximizes its available resources instead of relying on additional cores. By integrating OoO threads and multiple InO threads within the same core, SHADOW achieves higher utilization at minimal cost.

3.1.4 SHADOW: A Balanced ILP-TLP Execution Model

The limitations of prior SMT approaches highlight the need for a hybrid execution model that can simultaneously exploit ILP while efficiently scaling TLP. SHADOW achieves this through:

- OoO threads to extract ILP where possible.
- Multiple lightweight InO threads to exploit TLP without incurring OoO execution overhead.
- A simple, software-based work-stealing mechanism to dynamically redistribute work based on execution demand.

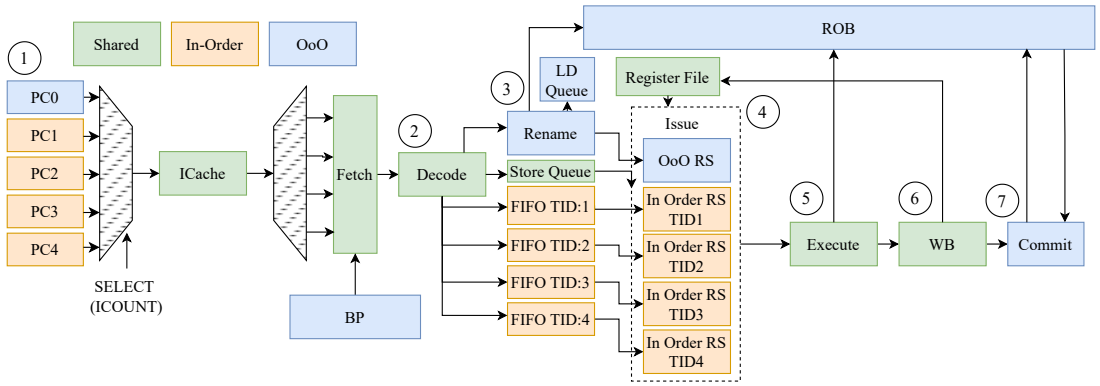


Figure 3.5: Microarchitecture design of SHADOW configured with 1 OoO and 4 inO threads.

By concurrently leveraging ILP and TLP, SHADOW achieves higher efficiency than existing SMT designs, making it a scalable solution for modern memory-bound workloads.

3.2 SHADOW

SHADOW is the first **asymmetric simultaneous multithreading (SMT) architecture** to integrate OoO and InO threads within a single core, dynamically balancing ILP and TLP without speculative instruction steering or execution mode switching. InO threads execute strictly sequentially without speculation or register renaming, allowing SHADOW to efficiently adapt to workload demands, particularly in memory-bound applications.

Figure 3.5 illustrates SHADOW's microarchitecture. Each cycle, a thread fetches ① and decodes instructions ②. OoO instructions are renamed ③ and placed in the RS④, while InO instructions bypass renaming and enter a FIFO queue. The oldest InO instructions are inserted into the RS. Ready instructions execute⑤, write back ⑥, and commit ⑦, though only OoO threads commit, as InO threads execute non-speculatively.

SHADOW is runtime-configurable, with configurations limited by to-

tal physical register count. InO threads use integer and floating-point registers, maximizing TLP without renaming or speculation, while OoO threads access all registers, including vector registers. Registers are split between the ARF and the physical register file (PRF). InO threads execute non-speculatively, each receiving a number of registers equal to the ARF. The remaining registers are shared among OoO threads, with architectural state tracking similar to the ARM Cortex-A9 [9], which uses a Register Alias Table to manage the subset of physical registers holding architectural state. With a 256 integer register and 192 floating-point register setup modeling the Grace ARM CPU core, SHADOW enables multiple configurations, including 1–3 OoO threads, up to 6 InO threads, and hybrid setups such as 1 OoO + 4 InO and 2 OoO + 2 InO.

SHADOW balances ILP and TLP through a software-based **dynamic work-stealing mechanism** that emerges from a **greedy scheduling strategy** on asymmetric hardware threads. Each thread independently steals work as soon as it becomes available, without centralized control. OoO threads, benefiting from a deep execution window, greedily take more iterations when ILP is high. When memory-bound stalls limit OoO throughput, InO threads, unaffected by speculative bottlenecks, continue stealing work to sustain execution. This adaptive redistribution occurs **without explicit intervention or complex scheduling logic**; instead, the software work-stealing mechanism allows threads to claim available work in a decentralized manner, ensuring efficient workload balancing as an emergent property of execution.

The specific modifications SHADOW introduces at each pipeline stage are described below.

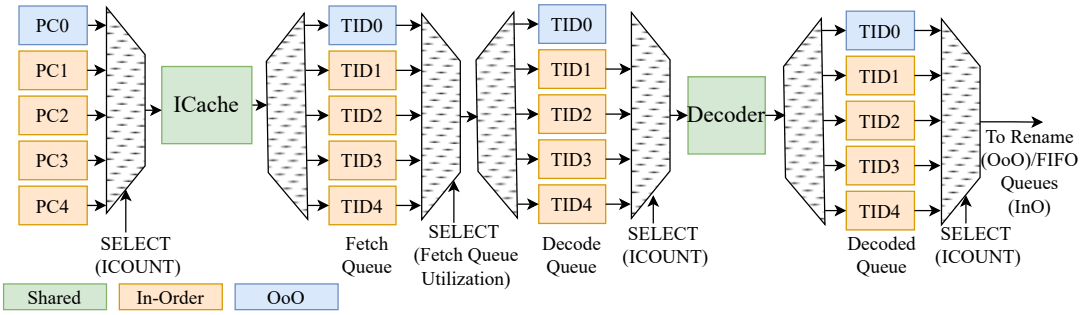


Figure 3.6: Microarchitecture of the fetch and decode stage configured with 1 OoO and 4 inO threads.

3.2.1 Fetch and Decode

The fetch ① and decode ② stages in SHADOW operate similarly to SMT-enabled cores (Figure 3.6). The architecture supports up to six SMT contexts, each with its own PC. Return Address Stacks (RAS) are exclusive to OoO threads for speculative execution, while InO threads execute without speculation, eliminating rollback mechanisms. Since InO threads do not predict branches, they stall fetch until branch resolution, but SHADOW offsets this with higher TLP from multiple InO threads.

Each cycle, a single thread accesses the ICache, selected via the ICOUNT policy [114]. Decode prioritizes threads with the most instructions in the fetch queue, ensuring a steady supply. Rename and issue selection follow ICOUNT, favoring OoO threads when occupancy is equal to maximize ILP and prevent starvation. This scheduling strategy dynamically balances resource allocation for efficient backend utilization.

3.2.2 Rename

Instruction Placement in the ROB: Figure 3.7 details SHADOW's rename ③, select, and wakeup ④ stages. After decoding, OoO instructions are renamed to eliminate false dependencies, checked for remaining dependencies, and

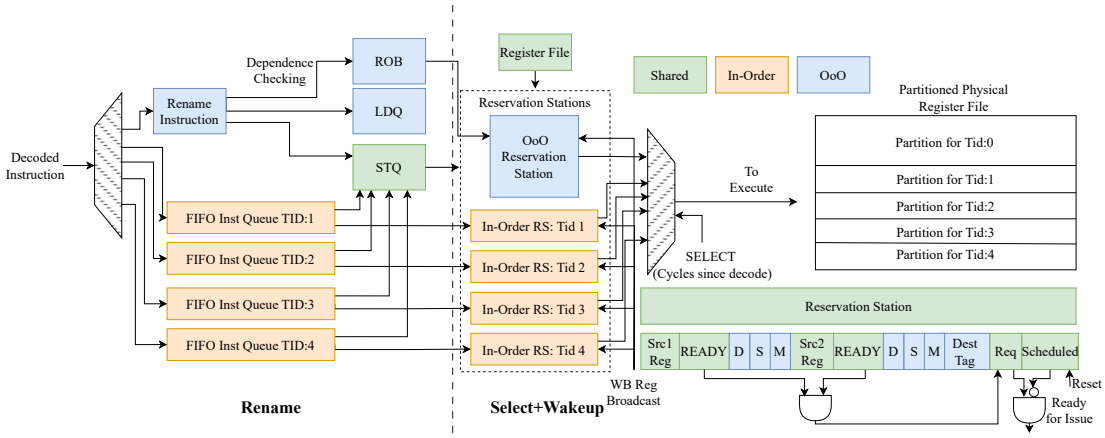


Figure 3.7: Microarchitecture of the Rename and Wakeup+Select stages configured with SHADOW having 1 OoO and 4 inO threads.

placed in the ROB and RS, statically partitioned when multiple OoO threads are active to prevent contention. InO instructions bypass renaming, entering per-thread circular FIFO queues that power down when inactive to save energy.

Instruction Placement in the RS: After renaming (for OoO threads) or static PRF mapping (for InO threads), instructions enter the RS. At runtime, the RS is partitioned between OoO and InO threads, with each InO thread allocated one entry, while the rest are evenly split among OoO threads. The oldest InO instructions check dependencies via a per-thread scoreboard, a multi-ported 448-bit table supporting two reads and two writes per thread per cycle. Dependencies are tracked on source registers, and if present, the RS entry waits until sources are ready.

3.2.3 Context Switching and Register File Allocation

SHADOW employs delegate threads for hardware reconfiguration, similar to prior OS-managed reconfigurable systems [5]. The OS stores an application's thread configuration alongside its process state in the process control block (PCB). When an application spawns pthreads, or is context-switched in, a delegate thread config-

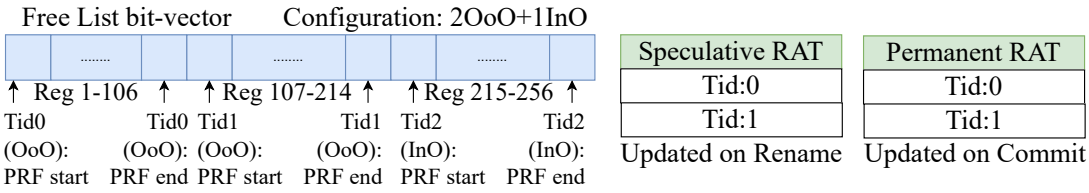


Figure 3.8: Register File partitioning in SHADOW.

ures the hardware using `shdw_cfg <#OoO>, <#InO>`, a new instruction which: (1) **Resets the Front End**, activating `<#OoO>` OoO and `<#InO>` InO PCs; (2) **Reconfigures Rename Tables**, assigning OoO threads to the first `<#OoO>` hardware threads and InO threads to the remaining ones, with each OoO thread maintaining an independent free list of physical registers (PRs); and (3) **Partitions Execution Resources**, configuring the ROB and RS based on Section 3.2.2.

Once reconfiguration completes, the OS restores application state, and execution resumes. To simplify scheduling, SHADOW supports either (1) multiple single-threaded applications or (2) one multi-threaded application. Only thread ID 0 configures hardware in multi-threaded applications, and changes occur only when pthreads spawn, ensuring a fixed configuration during execution.

Register file allocation: Similar to GPUs, which assign registers per thread to manage warp distribution [20], SHADOW dynamically allocates registers at context switches based on the application's configuration. As shown in Figure 3.8, Register allocation uses a bit-vector free list with a priority decoder, as in BOOM [14]. (1) Each I InO thread is allocated A architectural registers, totaling $A \cdot I$ across all active InO threads. (2) The remaining registers are divided among O OoO threads, with each receiving $(P - A \cdot I)/O$ registers. The free list for each OoO thread is determined using bitwise AND on the available bit vector for the range $[(P - A \cdot I)/O, (i+1)(P - A \cdot I)/O - 1]$.

OoO threads use a Pentium 4-style renaming approach [44], similar to MorphCore [108], storing both speculative and architectural data in the PRF. Each OoO thread main-

tains a per-thread Register Alias Table (RAT) with a Speculative-RAT for speculative execution and a Permanent-RAT for architectural state tracking. This design is preferred over rename checkpoints, as deeper and wider instruction windows (320 entries in this configuration) require more checkpoints—up to 48 for a 256-entry ROB—incurring high RAT restore latency and increased power and performance costs [81]. During context switches, the OS saves the register state and hardware configuration in the PCB, including the stack pointer and Current Program Status Register (CPSR) for each thread before executing the `shdw_cfg` operation.

3.2.4 Select and Wakeup

The wakeup logic ④ remains unchanged and operates exactly as in a traditional OoO core. Figure 3.7 shows the structure of the RS entry [105, 108].

OoO Wakeup: SHADOW's OoO wakeup process follows traditional OoO cores. An operand is marked ready (R bit set) once its dependencies are resolved. For OoO threads, this also occurs when the MATCH bit remains active for the number of cycles specified in the DELAY field. During execution, an instruction broadcasts its destination tag, which is compared against source tags in the RS. A match sets the MATCH bit and updates the DELAY field with the instruction's execution latency, stored in its RS entry. The DELAY value transfers to the SHIFT field associated with the source tag, decrementing each cycle while the MATCH bit stays active. Once the SHIFT field reaches zero, the R bit is set, marking the operand as ready. The RS entry then raises a request for execution once all operands are available.

OoO Select: The select logic scans the RS for the oldest instructions with Req Exec set, generating a Grant bit vector to mark entries for execution. The selected

instruction sets its SCHEDULED bit, preventing further competition in later cycles.

InO Wakeup: InO instructions issue sequentially, respecting all dependencies. Each InO thread has a single RS entry, reducing hardware complexity. If no dependencies exist, the instruction asserts Req to signal readiness; otherwise, it waits until dependencies resolve. When a dependent instruction completes, its destination register is checked against waiting instructions' sources—if matched, the ready bit is set. Execution proceeds once both sources are ready.

This wakeup mechanism is designed for efficiency, eliminating the need for costly broadcasts and tag comparisons across the RS. To maintain correctness, InO execution stalls and the load instruction entry is not removed from the RS until it has a hit in the translation lookaside buffer (TLB). This precaution prevents an incorrect register state due to potential load re-execution caused by TLB misses.

InO Select: With only one RS entry per InO thread, no additional selection logic is required.

Selecting Ready Instructions for Issue: From the ready instructions in the RS, instructions are selected for issue based on the cycles the instructions have spent in the pipeline since decode, and sent to the issue queues.

3.2.5 Execute, Writeback, and Commit

SHADOW's execute ⑤, writeback ⑥, and commit ⑦ stages operate like a traditional OoO core. Both OoO and InO threads read from the PRF, execute in the ALU, and use writeback bypass. However, only OoO threads commit, as InO threads execute sequentially without speculation. OoO instructions commit in order, updating the Permanent-RAT.

3.2.6 Branch Prediction

Only OoO threads perform branch prediction in SHADOW. Experiments show that ILP from the OoO thread and TLP from InO threads effectively mask branch resolution latencies, making branch prediction for InO threads unnecessary. Thus, SHADOW, like the Sun Niagara T1 [59], eliminates branch prediction for InO threads. Mispredictions in the OoO thread affect only its own instructions, leaving InO threads unaffected.

3.2.7 Load and Store Queues

SHADOW configures load and store queues at runtime. Each OoO thread has a dedicated partition and can execute loads speculatively. Loads update the load queue, check the store queue for forwarding, and enable speculative execution. The store queue detects violations by tracking speculative loads against pending stores.

In contrast, InO threads do not use the load queue, execute conservatively, and each receives five store queue entries. Loads check only the store queue for correctness, avoiding speculation. To prevent incorrect execution, InO threads stall until the issues load registers a TLB hit to avoid incorrect execution if a load needs to be replayed. The TLB is shared among threads.

3.2.8 Memory Consistency and Coherence

SHADOW targets data-parallel applications that use locks, providing a **sequential consistency for data-race-free (SC for DRF)** model [2]. Lock acquisitions and releases flush relevant pipelines, preventing unintended reordering across lock boundaries. With a shared L1 cache, coherence is maintained without explicit intra-core messaging. While SHADOW currently supports coarse-grained synchronization with SC for DRF, its design can naturally extend to finer-grained memory consis-

tency using ARM-style memory ordering if needed.

3.2.9 Dynamic Work Distribution and Stealing Mechanism

Algorithm 1: Dynamic Work-Stealing Mechanism in SHADOW

```
1  Initialize global counter index currentChunk ← 0 and mutex;
2  Define CHUNK_SIZE as the amount of work each thread steals at a time;
3  Spawn N worker threads;
4  foreach worker thread t in parallel do
5  |  while true do
6  |  |  Lock mutex;
7  |  |  Set startChunk ← currentChunk;
8  |  |  Update currentChunk ← currentChunk+CHUNK_SIZE;
9  |  |  Unlock mutex;
10 |  |  if startChunk ≥ TotalWork then
11 |  |  |  break;
12 |  |  end
13 |  |  Set endChunk ← min(startChunk + CHUNK_SIZE, TotalWork);
14 |  |  foreach iteration i in [startChunk, endChunk) do
15 |  |  |  Execute assigned work;
16 |  |  end
17 |  end
18 end
19 Synchronize and join worker threads;
20 Return completed workload;
```

SHADOW employs a software-based work-stealing mechanism to dynamically distribute workloads between OoO and InO threads. Algorithm 1 demonstrates its implementation using Pthreads. The user specifies `CHUNK_SIZE` to control the granularity of work stolen by each thread (line 3). Worker threads are then spawned (line 4), each executing in a continuous loop (line 5) until all work is completed. A thread acquires the lock (line 6) to safely access and update the global `currentChunk` counter, which tracks the next available workload index. It records its assigned starting index (line 7), updates `currentChunk` so other threads are aware of the remaining work (line 8), and then releases the lock (line 9). If the assigned index exceeds the total workload, the thread terminates (line 11) and waits for others to

complete (line 18). Otherwise, it determines the range of iterations it will process (line 13) and executes its assigned work (line 15). The thread then loops back to steal more work (line 5) until no work remains.

The software layer remains agnostic to SHADOW's asymmetric execution, ensuring compatibility with Pthreads-based workloads primarily relying on integer and floating-point operations. Once assigned, work ownership remains fixed, preserving deterministic execution. SHADOW is also compatible with high-level programming models like Intel TBB [97] and Cilk [12], which provide implicit workload distribution and work-stealing.

This evaluation assumes uniformly distributed workloads, where each chunk has the same amount of work. Work-stealing adapts to ILP-TLP characteristics: when cache misses are low, OoO threads achieve higher IPC and process more iterations, while at high miss rates, InO threads take on a larger workload, improving resource utilization. Contention on the lock used for work-stealing can arise when multiple threads steal work simultaneously, particularly with small `CHUNK_SIZE` or when many threads finish their assigned work around the same time. In our evaluation, OoO threads stall when no work remains, though implementations could allow them to reclaim unfinished iterations from InO threads for improved efficiency.

Work-stealing can affect cache performance, particularly in workloads with high L1 D-cache hit rates, due to increased data movement across threads (Section 3.3.2). While SHADOW is designed for a single core, its principles extend to multi-core architectures. For example, an ARM big core in a big.LITTLE configuration could be adapted into a SHADOW core, and dynamic work-stealing between big and little cores can be used to maximize resource utilization.

3.2.10 Impact on CPU Frequency

SHADOW is expected to have a similar impact on CPU core frequency as MorphCore [108], which estimates a 2.5% slowdown due to added multiplexers in critical pipeline stages. SHADOW introduces comparable multiplexers to support the selection between OoO and InO instructions in the rename stage, RS scheduling, and operand bypass logic. Additionally, SHADOW integrates per-thread fetch queues and InO wakeup/select logic, similar to MorphCore. Like MorphCore, SHADOW ensures that newly added InO scheduling logic is placed and routed to avoid extending the critical path beyond these minor multiplexer delays.

When configured with only InO threads, SHADOW does not boost core frequency as InO threads share execution resources with OoO threads, thus changing frequency would require decoupling their pipeline, adding complexity and undermining SHADOW's simplicity. Even with a frequency boost, 6 InO threads would underperform a single OoO thread. Using IBM POWER6 [62] (5 GHz, InO) and POWER7 [37] (4.25 GHz, OoO) as reference points, experiments show that 6 InO threads are 1.08× slower in high L1 D-cache miss workloads and 1.78× slower in ILP-heavy workloads.

3.2.11 Example of SHADOW's Impact on TLP and ILP Performance

In OoO execution, long-latency loads with high L2 miss rates stall execution by occupying the ROB for hundreds of cycles, preventing new instructions from entering. Completed instructions cannot retire due to InO commit, further filling the ROB. Figure 3.9 illustrates this bottleneck.

With (a) one OoO thread, in-flight loads are limited to four as the ROB fills, stalling instruction issue and underutilizing execution resources. Adding (b) an InO

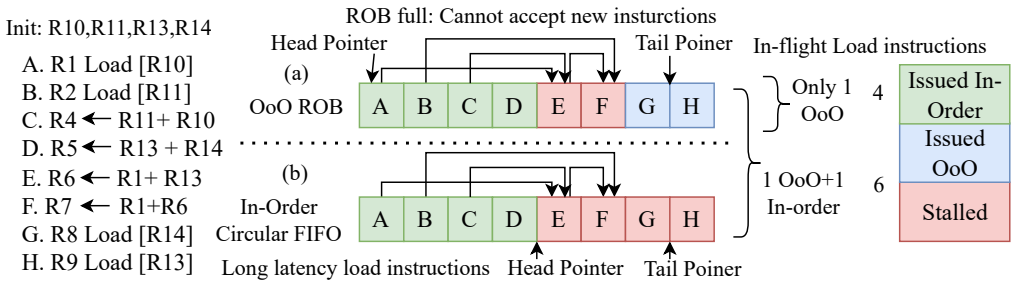


Figure 3.9: Impact of adding an InO thread to an OoO system: An illustrative assembly example.

thread introduces minimal overhead while increasing in-flight loads to six, boosting memory-level parallelism and backend utilization despite stalls. By integrating OoO and InO threads, SHADOW dynamically balances ILP and TLP, adapting to workload demands and mitigating pipeline stalls.

This section describes the experimental context used to evaluate SHADOW.

Modeling SHADOW: SHADOW is implemented in Gem5 [73, 11] using system emulation mode, modeling a high-performance ARM Grace core (Table 3.1), with several modifications to support its asymmetric multithreading model. SMT support was extended to enable heterogeneous execution within a single core, integrating OoO and InO threads. The rename stage was modified to let InO threads bypass register renaming while extending dependency tracking for false dependencies. Dispatch logic enforces strict InO execution for InO threads, prevents speculation, and prioritizes the oldest ready instruction across all threads. The commit stage was modified so that only OoO threads commit instructions, while InO threads execute strictly non-speculatively, eliminating rollback mechanisms. These changes ensure an accurate SHADOW model in Gem5.

Evaluating SHADOW: SHADOW is evaluated on SpMM (Gustavson's algorithm), tiled dense matrix multiplication, and selected CRONO [6] and Rodinia [18] benchmarks, summarized in Table 3.2. Benchmarks are classified by L1 D-cache

Table 3.1: CPU Microarchitectural Parameters

Parameter	Value
ISA	ARM
Simulator	Gem5
Reorder Buffer (ROB) Size and Reservation Station size [100, 19]	320
Circular FIFO Queue Size	20
OoO Thread Store Queue Size	68
OoO Thread Load Queue Size	72
In-Order Thread Store Queue Size	5
Integer Registers (ARF+PRF)	258
Floating-Point Registers (ARF+PRF)	192
Vector Registers (ARF+PRF)	192
L1 Data Cache	64 kB, 2-way set associative, 2 cycles pipelined, 24 MSHRs
L1 Instruction Cache	64 kB, 2-way set associative, 2 cycles pipelined, 12 MSHRs
L2 Cache	1 MB, 16-way set associative, 16 cycles pipelined, 24 MSHRs, Strided Prefetcher
Issue Width	8 / 4
Commit Width	8 / 4

miss rates as high ($\geq 25\%$) or low ($< 25\%$). The table also quantifies code modifications required to integrate dynamic work-stealing in Pthreads-based implementations, highlighting adaptation effort.

SpMM serves as the primary workload, with varying sparsity levels showcasing SHADOW's ILP-TLP balancing under different memory access patterns affecting OoO IPC. Additional benchmarks cover compute-bound (e.g., BC, NN, Pathfinder) and memory-intensive (e.g., APSP, Backprop, Heartwall) workloads for broad evaluation. Benchmarks were selected for Pthreads compatibility, with a subset used due to software-based dynamic work-stealing requiring well-defined workload distribution. No benchmarks were excluded based on performance.

SHADOW is compared to MorphCore [108], which switches between two-OoO-thread and six-InO-thread modes based on application thread count. Equivalent SHADOW configurations are evaluated for direct performance comparison.

Table 3.2: Benchmarks Used to Evaluate SHADOW

Benchmark	Miss Rate	Additional lines of code
Sparse Matrix Multiplication	High	-
APSP [6]	High	0
Backprop [18]	High	8
Heartwall [18]	High	8
Tiled Dense Matrix Multiplication	Low	-
BC [18]	Low	0
TSP [6]	Low	12
nn [6]	Low	10
Pathfinder [6]	Low	10

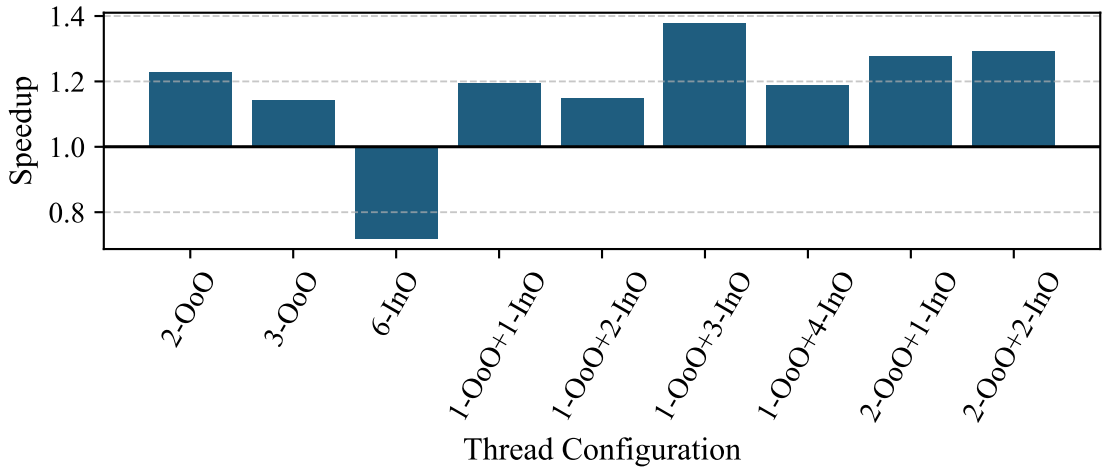


Figure 3.10: Geometric mean performance of SHADOW configurations on high D-cache miss rate benchmarks from Table 3.2, normalized to a single-threaded OoO core.

3.3 Evaluation of SHADOW

This section evaluates SHADOW across memory- and compute-bound workloads under various configurations.

3.3.1 Performance of SHADOW Across Workloads with Varying Cache Miss Rates

Impact of SHADOW Configurations on High-Miss-Rate Workloads: Figure 3.10 shows SHADOW's speedup over the 1 OoO thread baseline for APSP, Backprop, and Heartwall (high cache miss benchmarks). Two OoO threads im-

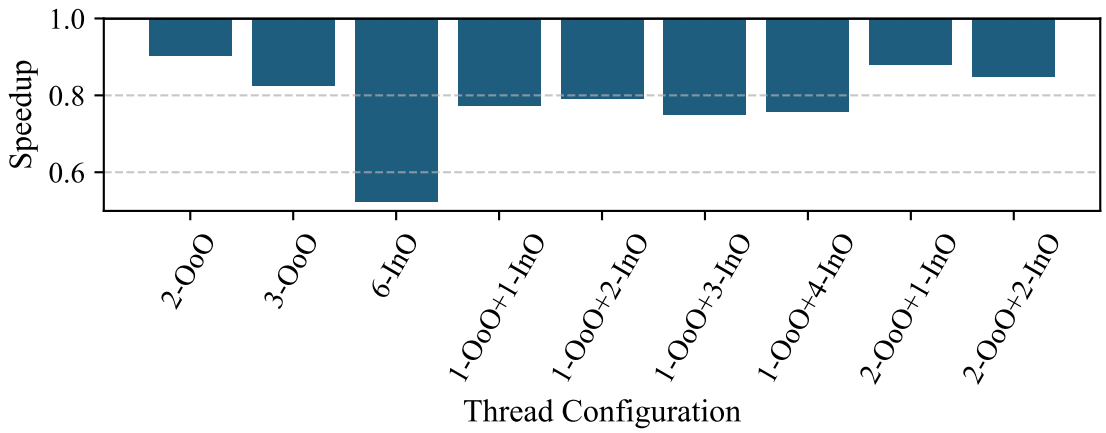


Figure 3.11: Geometric mean performance of SHADOW configurations on low D-cache miss rate benchmarks from Table 3.2, normalized to a single-threaded OoO core.

prove performance by 1.35×, but a third reduces it to 1.21× due to register file contention. The best configuration, 1 OoO + 3 InO threads, achieves 1.47× speedup by increasing TLP while preserving enough registers for ILP. Similarly, 2 OoO + 1 InO and 2 OoO + 2 InO outperform the 2 OoO baseline by leveraging TLP without sacrificing ILP. In contrast, 6 InO threads cause a 1.32× slowdown, as increased TLP fails to compensate for low ILP per thread.

Impact of SHADOW Configurations on Low-Miss-Rate Workloads: Figure 3.11 shows SHADOW's speedup over the 1 OoO thread baseline for Dense Matrix Multiplication, BC, TSP, NN, and Pathfinder, compute-bound benchmarks with low L1 DCache miss rates. These workloads favor single-threaded execution, as adding threads increases conflict misses, degrading performance. All configurations perform worse than the baseline due to higher cache contention.

3.3.2 Best Performing SHADOW Configurations Across Diverse Workloads

Table 3.3 shows the optimal SHADOW configuration per benchmark.

Table 3.3: Performance of SHADOW on Diverse Workloads

Benchmark	Best Configuration	Speedup
APSP [6]	2 OoO-2 InO	1.56×
Backprop [18]	1 OoO-3 InO	3.16×
Heartwall [18]	1 OoO	1×
Tiled Dense Matrix Multiplication	2 OoO	1.2×
BC [18]	2 OoO-1 InO	1.37×
TSP [6]	3 OoO	1.7×
nn [6]	1 OoO	1×
Pathfinder [6]	1 OoO	1×
Average	-	1.38×

High-Miss-Rate Benchmarks:

- **APSP:** Performance scales with thread count as added TLP complements ILP without harming cache performance. The 2 OoO-2 InO configuration outperforms 3 OoO by avoiding register file contention and store queue bottlenecks.
- **Backprop:** The 1 OoO-3 InO configuration best balances ILP and TLP, maximizing TLP while avoiding structural hazards like rename register shortages or ROB constraints, all while maintaining stable cache performance.
- **Heartwall:** The 1 OoO configuration achieves the best performance. Despite a high L1 D-cache miss rate, L2 misses remain low. Adding more threads significantly increases L2 misses, degrading performance.

Low-Miss-Rate Benchmarks:

- **Tiled Dense Matrix Multiplication, NN, and Pathfinder:** Performance declines as more threads increase cache contention. While TLP improves, conflict misses reduce ILP in the OoO thread. NN and Pathfinder are highly sensitive, with any extra thread causing slowdowns. In Dense Matrix Multiplication, 2 OoO threads achieve 1.2× speedup, balancing TLP gains against ILP loss, but adding more threads degrades performance.

- **BC:** The 2 OoO-1 InO configuration performs best, as BC maintains stable cache miss rates, preserving ILP while leveraging additional TLP.
- **TSP:** The 3 OoO configuration is optimal. While D-cache misses slightly increase with more threads, L2 misses drop significantly, improving ILP as TLP aids L2 prefetching, leading to performance gains.

Configuring SHADOW for Optimal Performance: There is no single best SHADOW configuration for all workloads. The effectiveness of additional InO threads depends on the ILP-TLP tradeoff of each benchmark. Some workloads benefit from more TLP (e.g., Backprop, APSP), while others rely heavily on ILP and suffer from cache contention when additional threads are added (e.g., Dense MM). The configurability of SHADOW allows users to select the optimal configuration for their specific workload, enabling adaptive performance tuning based on application characteristics.

3.3.3 Characterization of SHADOW on SpMM

Beyond evaluating SHADOW across diverse workloads, a detailed analysis is conducted on **sparse matrix multiplication (SpMM)**. SpMM serves as a controlled testbed to assess SHADOW's adaptability across varying cache miss rates by adjusting matrix sparsity and size. While matrices maintain uniform sparsity across rows, individual elements are randomly distributed, disrupting spatial locality and limiting the effectiveness of hardware prefetchers. This results in higher cache miss rates, making SpMM an ideal workload to evaluate SHADOW's ability to balance ILP and TLP under memory constraints. Using Gustavson's algorithm [40], we systematically vary sparsity and matrix size to characterize SHADOW's response to different memory access patterns.

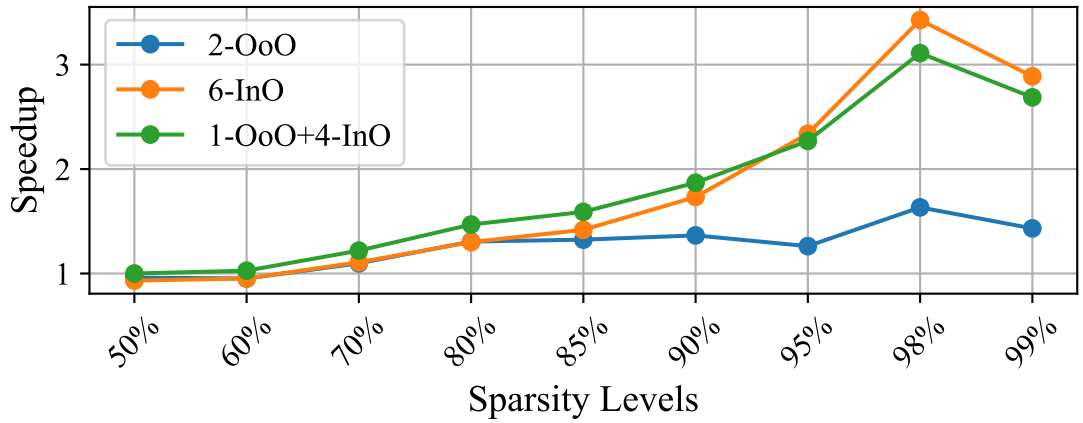


Figure 3.12: Performance of SHADOW with varying degrees of sparsity for an 8-wide CPU over 1 OoO thread.

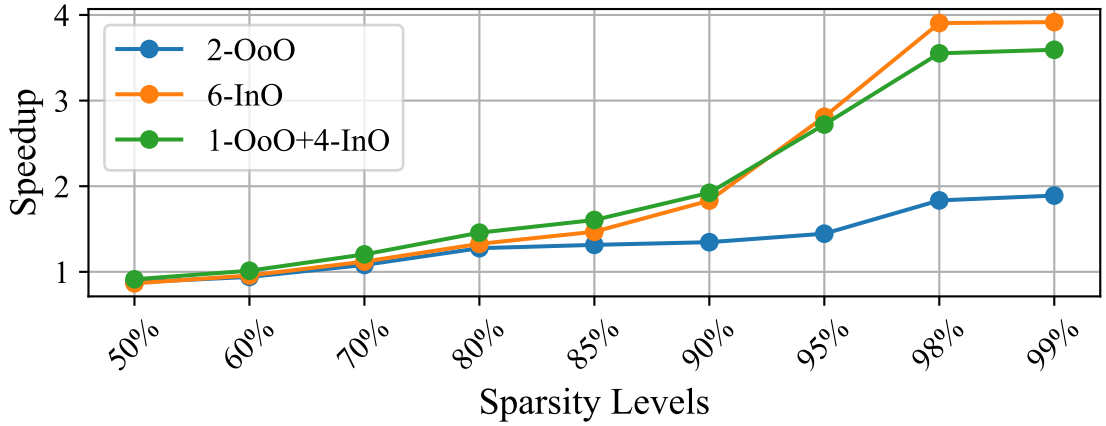


Figure 3.13: Performance of SHADOW with varying degrees of sparsity for a 4-wide CPU over 1 OoO thread.

We compare four SHADOW configurations: (1) 1 OoO, (2) 2 OoO, (3) 1 OoO + 4 InO, and (4) 6 InO. This analysis demonstrates how SHADOW dynamically balances ILP and TLP to optimize performance across different sparsity levels.

Varying Degrees of Sparsity

SHADOW's performance was evaluated on SpMM using 600 × 600 matrices with sparsity ranging from 50% to 99% (Figures 3.12 and 3.13 for 8-wide and 4-wide CPUs, respectively).

At lower sparsity, adding a second OoO thread improves performance over a

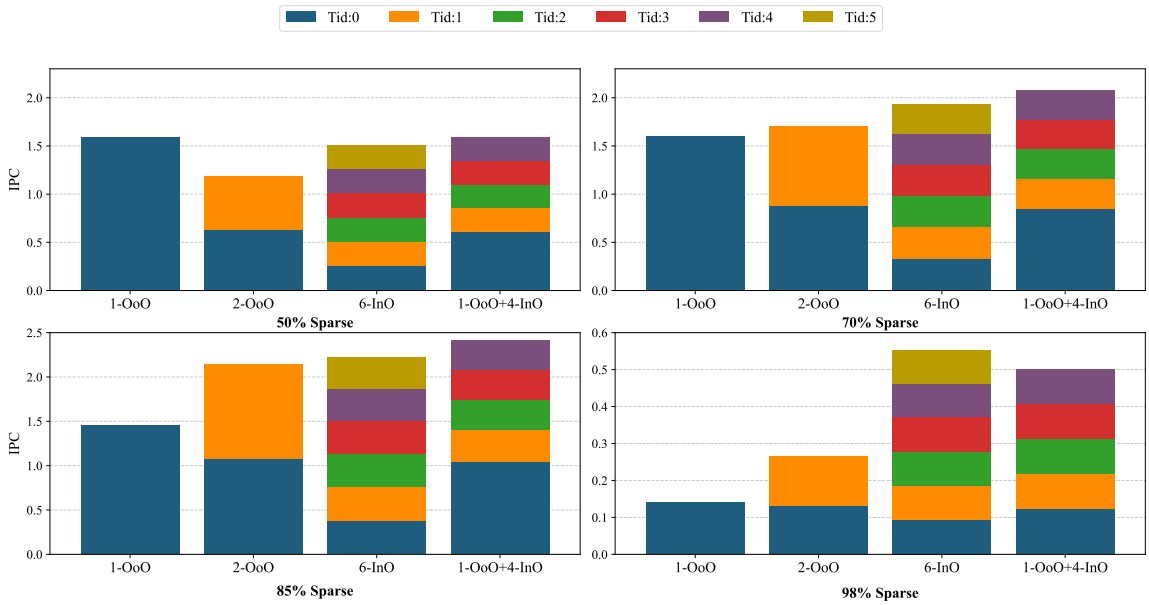


Figure 3.14: Breakdown of IPC contribution from each thread with varying degrees of sparsity over 1 OoO thread.

single OoO thread. However, as ILP diminishes with increasing sparsity, six InO threads surpass two OoO threads. SHADOW's 1 OoO + 4 InO configuration achieves the highest performance up to 95%, effectively balancing ILP and TLP.

Figure 3.14 shows IPC trends. At 70% sparsity, a single OoO thread reaches an IPC of 1.6, increasing to 1.7 with a second OoO thread. While InO threads have lower per-thread IPC, their combined throughput allows six InO threads to outperform two OoO threads. The 1 OoO + 4 InO configuration leverages the OoO thread's ILP while benefiting from TLP, yielding the highest overall IPC.

At 85% sparsity, OoO IPC drops due to increasing irregularity, yet 1 OoO + 4 InO configuration maintains a performance edge. Beyond 95%, ILP opportunities diminish, and six InO threads surpass 1 OoO + 4 InO. At extreme sparsity, InO-only configurations perform best.

On an 8-wide CPU, 1 OoO + 4 InO configuration achieves up to 3.1× speedup at 98% sparsity and 1.67× average improvement over a single OoO thread (not shown in figure). On a 4-wide CPU, it peaks at 3.6×, averaging 1.77×. Compared to two OoO threads, 1 OoO + 4 InO reaches 1.93× speedup (average 1.37×) and

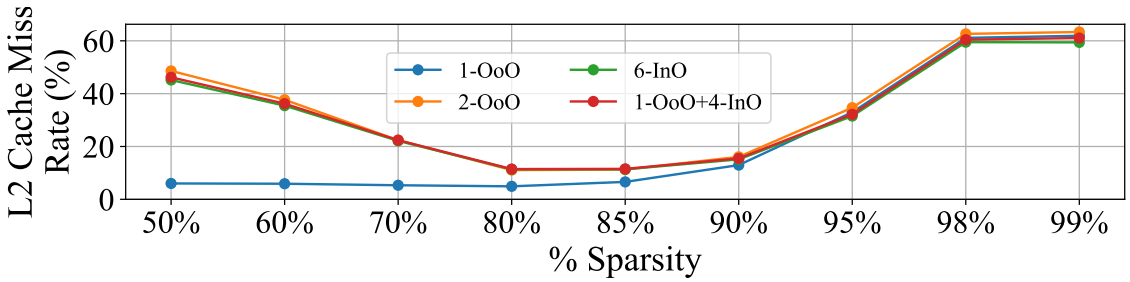


Figure 3.15: Change in miss rate for different thread configurations.

1.13× over six InO threads (average 1.04×) on an 8-wide core, with similar trends on a 4-wide CPU.

1 OoO + 4 InO configuration's advantage over OoO configurations grows with sparsity, where reduced ILP makes additional TLP from InO threads more beneficial. Conversely, its edge over InO-only designs is most pronounced at lower sparsity, where the OoO thread extracts more ILP.

L2 Miss Rate Across Various Thread Configurations

Figure 3.15 shows the L2 cache miss rate for different configurations: a single OoO thread, two OoO threads, six InO threads, and SHADOW (1 OoO + 4 InO). At 50% sparsity, the L2 miss rate is low (2%) but increases with additional threads. At 98% sparsity, it peaks at 60% for a single OoO thread due to the irregular access pattern. Despite a 98% sparse 600 × 600 matrix containing only 7200 elements, high L2 misses occur as the result matrix must be loaded from DRAM, the extreme sparsity prevents effective prefetching, limiting cache efficiency. Overall, while more threads lead to a small rise in L2 misses, the impact remains limited and does not significantly degrade cache performance across evaluated configurations.

Varying Matrix Sizes

Figures 3.16 (8-wide CPU) and 3.17 (4-wide CPU) show SHADOW's performance on 90% sparse SpMM across different matrix sizes.

Across both CPU widths, SHADOW (1 OoO + 4 InO) outperforms 2 OoO and 6

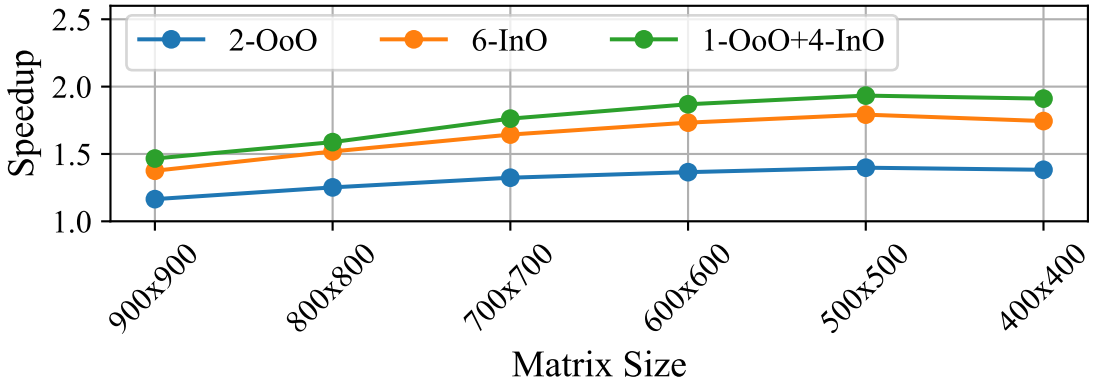


Figure 3.16: Performance of SHADOW with varying matrix size for 90% sparsity for an 8-wide CPU over 1 OoO thread.

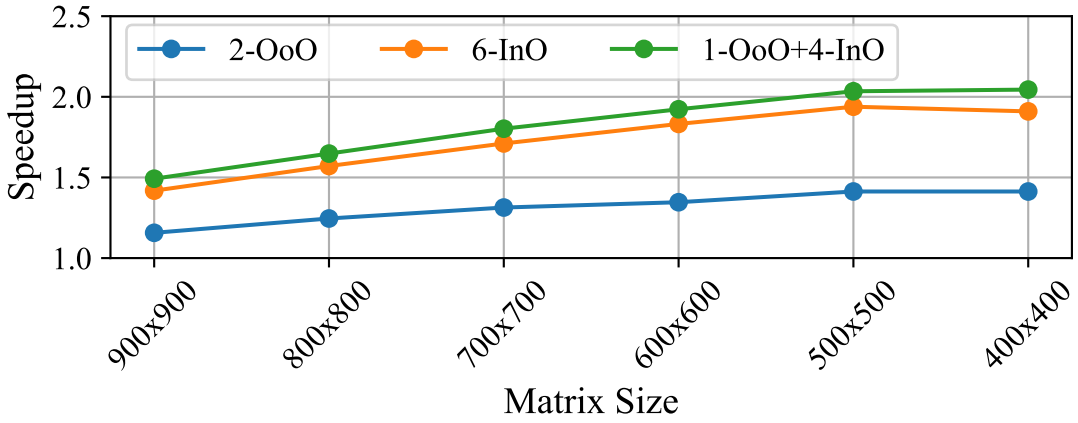


Figure 3.17: Performance of SHADOW with varying matrix size for 90% sparsity for a 4-wide CPU over 1 OoO thread.

InO configurations, demonstrating the effectiveness of its asymmetric multithreading. On an 8-wide CPU, it achieves up to 1.93× speedup over a single OoO thread, while on a 4-wide CPU, it reaches 2.04×, efficiently utilizing resources across varying matrix sizes.

Benefit of Dynamic Work-Stealing Across Threads

SHADOW dynamically balances workload distribution through work-stealing (Algorithm 1). Threads acquire more work upon completing assigned chunks, with the OoO thread taking a larger share when ILP is high, and distribution equalizing as ILP decreases.

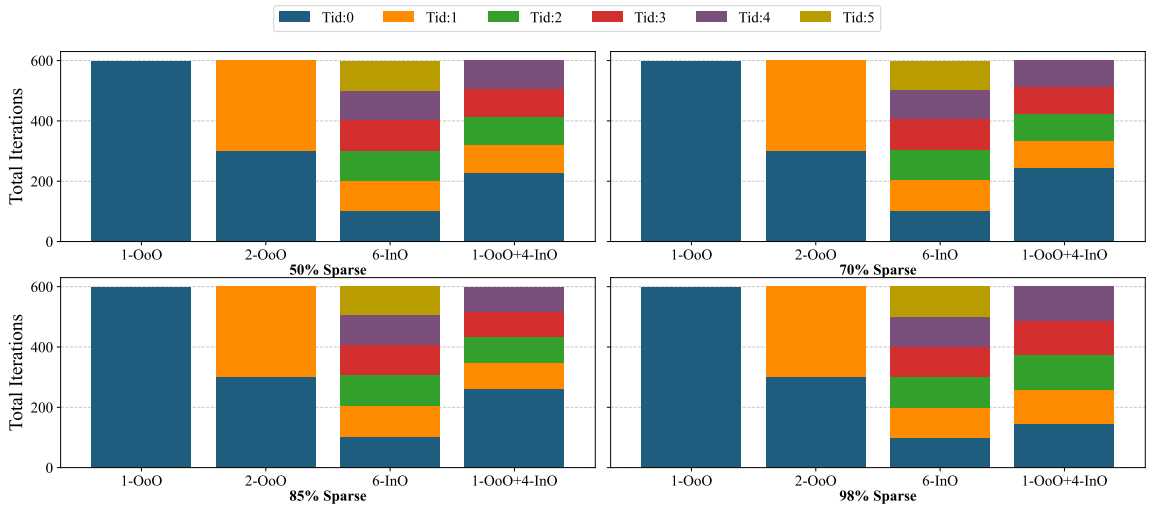


Figure 3.18: Distribution of work across threads with dynamic work-stealing for various degrees of sparsity.

Figure 3.18 shows how SHADOW adjusts workload allocation based on IPC. At 50% and 70% sparsity, the OoO thread's high IPC allows it to process more iterations. At 85% sparsity, declining IPC shifts more work to InO threads. By 98% sparsity, IPC differences narrow, leading to an even distribution. By adapting to workload characteristics, SHADOW effectively balances ILP and TLP, optimizing parallelism while simplifying workload management for programmers.

3.3.4 Area and Power Overhead of SHADOW

Table 3.4 summarizes SHADOW's hardware overhead. Despite supporting multiple threads, the PRF remains unchanged, with each thread adding minimal state via Thread Control State Registers and a Thread Control Block (TCB). Pipeline bandwidth is unaffected, as SHADOW employs per-thread fetch queues with simple multiplexers/demultiplexers. Unlike OoO threads, InO threads bypass renaming and ROB allocation, relying instead on lightweight FIFO queues. The reservation station size, CAM complexity, and ROB remain unchanged. Using a modified McPAT [68], we estimate SHADOW's area and power overhead to be just 1% over a high-performance core.

Table 3.4: Hardware Overhead of SHADOW

HW Structure	Components	Quantity / Size
Thread Control State Registers	General-purpose registers, Link register, Program counter and Current Program Status Register	68 Registers
Thread Control Block	Thread State Information, Thread Local Storage (TLS) Pointer, Stack Pointer, Thread ID, Priority Information, Pointers to resources	1664 Bits
Circular FIFO Queues	20 Entries per thread	5120 Bits
Multiplexers and Demultiplexers	Added in Fetch, Decode, Rename and Issue stages	4
Fetch Queues	Queue to store 8 fetched instructions	2048 Bits
Scoreboard	Scoreboard with six multi-ported tables, one dedicated per thread	2688 Bits

3.4 Related Work

Prior work has explored various techniques to improve ILP and TLP on CPUs. This section summarizes key approaches and contrasts them with SHADOW's design.

SMT enhances TLP by allowing multiple threads to share execution resources [45, 123, 115, 114, 26]. Traditional SMT treats all threads as microarchitecturally identical, limiting its ability to adapt to workload characteristics. Prior efforts introduced thread prioritization and resource partitioning [24, 78, 104], but these still rely on symmetric thread execution. SHADOW breaks this limitation by integrating OoO threads with multiple lightweight InO threads, enabling dynamic ILP-TLP balancing.

Simultaneous subordinate microthreading [15] improves single-threaded performance by spawning auxiliary microthreads to optimize branch prediction and cache behavior. While these threads assist execution, they remain symmetric OoO threads. SHADOW differs by using microarchitecturally asymmetric threads, leveraging

both deep ILP in OoO execution and wider TLP through InO threads to improve multi-threaded performance. Other approaches have reconfigured multi-core architectures to improve TLP, combining simple cores for higher ILP when needed [47, 39, 119, 55]. Unlike these inter-core techniques, SHADOW enhances TLP within a single core, retaining the full performance of an OoO core with only 1% area overhead. Similarly, prior work has proposed migrating applications across heterogeneous cores to optimize utilization [80, 4, 3, 61], while SHADOW improves intra-core parallelism without requiring core migration.

SHADOW achieves these benefits without increasing CPU area overhead, modifying the ISA, or altering the programming model. This preserves general-purpose flexibility while improving TLP and maintaining single-threaded performance.

3.5 Conclusion

SHADOW introduces an asymmetric SMT architecture that balances ILP and TLP by running OoO and InO threads concurrently. Unlike traditional SMT, it dynamically adapts to workload demands, leveraging deep ILP in the OoO thread and high TLP in lightweight InO threads. SHADOW is runtime-configurable, optimizing execution based on workload characteristics. Across nine diverse benchmarks SHADOW achieves up to 3.16× speedup and 1.38× average improvement over an OoO core, with just 1% area and power overhead.

Chapter 4

Future Work

While **GhOST** and **SHADOW** improve the balance between ILP and TLP, further enhancements can strengthen their scalability, efficiency, and adoption in real-world architectures.

4.1 Extending GhOST

- **Selective Register Renaming:** Introduce minimal renaming to reduce dependency stalls without high hardware overhead.
- **OoO-Aware Compilation:** Develop compiler optimizations that leverage GhOST's scheduling model to expose more ILP.
- **Adaptive Warp Occupancy:** Dynamically adjust warp allocation based on workload demands to improve execution efficiency.

4.2 Extending SHADOW

- **Dynamic Thread Allocation:** Adaptively adjust the mix of OoO and InO threads at runtime based on workload characteristics.

- **OoO Thread Work-Stealing:** Enable OoO threads to take over InO thread execution when no further work is available to steal.
- **DVFS-Aware Scheduling:** Integrate dynamic voltage and frequency scaling (DVFS) to optimize power efficiency based on workload phases.

Chapter 5

Conclusion

This thesis introduced **GhOST and SHADOW**, two novel architectures that bridge the gap between *ILP* and *TLP*. By integrating ILP-awareness into GPUs and TLP-aware execution into CPUs, these designs challenge the rigid divide between CPU and GPU architectures, improving performance across diverse workloads.

GhOST enables *warp-level out-of-order execution* for GPUs, achieving a *6.9% geometric mean speedup* with minimal area overhead. **SHADOW** introduces *asymmetric SMT* in CPUs, delivering *1.38× average speedup* by combining in-order and out-of-order execution. Together, these architectures demonstrate that *hybridizing ILP and TLP* unlocks new performance opportunities beyond traditional CPU-GPU boundaries.

As computing evolves, the rigid separation between CPU and GPU execution models must be reconsidered. This thesis demonstrates that *dynamic ILP-TLP integration* is both practical and beneficial, paving the way for more adaptive and efficient high-performance computing systems.

References

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. *ACM SIGARCH Computer Architecture News*, 23(2):2–13, 1995.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[5] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. Reconos: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2013.

[6] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55. IEEE, 2015.

[7] Johnathan Alsop, Matthew D. Sinclair, Rakesh Komuravelli, and Sarita V. Adve. GSI: A GPU stall inspector to characterize the sources of memory stalls for tightly coupled GPUs. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 172–182. IEEE, 2016.

[8] Jayvant Anantpur, Nagendra Gulur Dwarakanath, Shivaram Kalyanakrishnan, Shalabh Bhatnagar, and R. Govindarajan. RLWS: A reinforcement learning based GPU warp scheduler. *arXiv preprint arXiv:1712.04303*, 2017.

[9] Arm Ltd. Cortex-a9 technical reference manual: Register renaming. https://developer.arm.com/documentation/ddi0388/h/Functional-Description/About-the-functions/Register-renaming, 2010. Accessed: 2025-02-20.

[10] Prithayan Barua, Jun Shirako, and Vivek Sarkar. Cost-driven thread coarsening for GPU kernels. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. ACM, 1995.

[13] Thomas Bradley. Hyper-Q example. https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[14] Christopher Celio, David A. Patterson, and Krste Asanović. The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[15] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186–195, 1999.

[16] Ishita Chaturvedi, Bhargav Reddy Godala, Yucan Wu, Ziyang Xu, Konstantinos Iliakis, Panagiotis-Eleftherios Eleftherakis, Sotirios Xydis, Dimitrios Soudris, Tyler Sorensen, Simone Campanoni, et al. GhOST: A GPU out-of-order scheduling technique for stall reduction. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1–16. IEEE, 2024.

[17] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

[18] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. Ieee, 2009.

[19] Wikipedia contributors. Arm neoverse. https://en.wikipedia.org/wiki/ARM_Neoverse, 2025. Accessed: 2025-02-21.

[20] Brett W. Coon, John Erik Lindholm, Gary Tarolli, Svetoslav D. Tzvetkov, John R. Nickolls, and Ming Y. Siu. Register file allocation, December 2009.

[21] Brett W. Coon, Peter C. Mills, Stuart F. Oberman, and Ming Y. Siu. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators, October 7 2008. US Patent 7,434,032.

[22] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[23] Kaushik Datta and et al. Stencil computation optimization and auto-tuning on modern microprocessors. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[24] Gautham K. Dorai and Donald Yeung. Transparent threads: Resource sharing in smt processors for high single-thread performance. In *Proceedings International Conference on Parallel Architectures and Compilation Techniques*, pages 30–41. IEEE, 2002.

[25] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. Poise: Balancing thread-level parallelism and memory system performance in GPUs using machine learning. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 492–505. IEEE, 2019.

[26] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[27] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. A scalable multi-path microarchitecture for efficient GPU control flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 248–259. IEEE, 2014.

[28] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. Bow: Breathing operand windows to exploit bypassing in GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 996–1008. IEEE, 2020.

[29] Adi Fuchs and David Wentzlaff. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2019.

[30] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 235–246. IEEE, 2011.

[31] Mark Gebhart, Stephen W. Keckler, and William J. Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 465–476, 2011.

[32] Richard Goldman, Karen Bartleson, Troy Wood, Kevin Kranen, C. Cao, Vazgen Melikyan, and Gayane Markosyan. Synopsys' open educational design kit: Capabilities, deployment and future. In *2009 IEEE International Conference on Microelectronic Systems Education*, pages 20–24. IEEE, 2009.

[33] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David Kaeli. TwinKernels: An execution model to improve GPU hardware scheduling at compile time. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–49. IEEE, 2017.

[34] Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. HAWS: Accelerating GPU wavefront execution through selective out-of-order execution. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–22, 2019.

[35] GPGPU-Sim. Manual. http://gpgpu-sim.org/manual/index.php/Main_Page#PTXPlus_support, 2015.

[36] Scott Gray. Assembler for NVIDIA Maxwell architecture. https://github.com/NervanaSystems/maxas.

[37] Nigel Griffiths. POWER CPU Memory Affinity 3 - Scheduling Processes to SMT and Virtual Processors. https://www.ibm.com/support/pages/power-cpu-memory-affinity-3-scheduling-processes-smt-and-virtual-processors, 2019. Accessed: 2025-02-19.

[38] Dheevatsa Gupta and et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:2008.07678*, 2020.

[39] Shantanu Gupta, Shuguang Feng, Amin Ansari, and Scott Mahlke. Erasing core boundaries for robust and configurable performance. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 325–336. IEEE, 2010.

[40] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[41] Juan Gómez-Luna, Kaan Sari, Animesh Das, Saher Abulila, Sayantan Ghose, Rachata Ausavarungnirun, and Onur Mutlu. Evaluating machine learning workloads on memory-centric computing systems. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2022.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA, 5th edition, 2011.

[43] Sébastien Hily and André Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 64–67. IEEE, 1999.

[44] Glenn Hinton. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5:1–13, 2001.

[45] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, 1992.

[46] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. Repurposing GPU microarchitectures with light-weight out-of-order execution. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):388–402, 2021.

[47] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.

[48] Hyeran Jeon. *Resource underutilization exploitation for power efficient and reliable throughput processor*. PhD thesis, University of Southern California, 2015.

[49] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[50] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Owl: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 395–406, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 137–138. IEEE, 2019.

[52] Onur Kayıran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166. IEEE, 2013.

[53] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[54] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. Regmutex: Inter-warp GPU register time-sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 816–828. IEEE, 2018.

[55] Changkyu Kim, Simha Sethumadhavan, Madhu S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 381–394. IEEE, 2007.

[56] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 163–175. IEEE, 2016.

[57] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and memory level parallelism. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14. ACM, 2018.

[58] David B. Kirk and W. Hwu Wen-Mei. *Programming massively parallel processors: A hands-on approach*. Morgan kaufmann, 2016.

[59] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[60] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[61] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92. IEEE Computer Society, 2003.

[62] Hung Q. Le, William J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, Eric M. Schwarz, and Michael T. Vaden. IBM Power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[63] Jounghoo Lee, Yeonan Ha, Suhyun Lee, Jinyoung Woo, Jinho Lee, Hanhwi Jang, and Youngsok Kim. GCoM: A detailed GPU core model for accurate analytical modeling of modern GPUs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 424–436, 2022.

[64] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 260–271. IEEE, 2014.

[65] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 175–186, 2014.

[66] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[67] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 89–100. IEEE, 2015.

[68] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual IEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[69] John Erik Lindholm, Ming Y. Siu, Simon S. Moy, Samuel Liu, and John R. Nickolls. United States Patent #7,339,592: Simulating multiported memories using lower port count memories (Assignee NVIDIA Corp.), March 2008.

[70] NVIDIA LLC. Chipmaker Nvidia launches new system for autonomous driving. https://www.reuters.com/technology/chipmaker-nvidia-launches-new-system-autonomous-driving-2022-09-20, 2022.

[71] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems (TOCS)*, 15(3):322–354, 1997.

[72] Jason Loew and Dmitry Ponomarev. Two-level reorder buffers: Accelerating memory-bound applications on SMT architectures. In *2008 37th International Conference on Parallel Processing*, pages 182–189. IEEE, 2008.

[73] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[74] Samuel Lui, John Erik Lindholm, Ming Y. Siu, Brett W. Coon, and Stuart F. Oberman. United States Patent Application 11/555,649: Operand collector architecture (Assignee NVIDIA Corp.), May 2008.

[75] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 2019.

[76] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 455–466, 2014.

[77] Hadi Makrani, Sai Manoj Pudukotai Dinakarrao, Avesta Sasan, and Houman Homayoun. A comprehensive memory analysis of data intensive workloads on server class architecture. In *Proceedings of the International Symposium on Memory Systems*, pages 125–136. ACM, 2018.

[78] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing QoS and throughput for colocated server workloads on SMT cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–27. IEEE, 2019.

[79] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: Exception support and speculative execution on GPUs. *ACM SIGARCH Computer Architecture News*, 40(3):72–83, 2012.

[80] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198. IEEE, 2019.

[81] Andreas Moshovos. Checkpointing alternatives for high performance, power-aware processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 318–321, 2003.

[82] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale 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*, pages 308–317, 2011.

[83] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[84] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[85] NVIDIA. Proper CUDA Error Checking. https://leimao.github.io/blog/Proper-CUDA-Error-Checking/.

[86] NVIDIA. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2015.

[87] NVIDIA. NVIDIA Nvbit tool. https://research.nvidia.com/publication/2019-10_nvbit-dynamic-binary-instrumentation-framework-nvidia-gpus, 2015.

[88] NVIDIA. NVIDIA Volta White Paper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2015.

[89] NVIDIA. NVIDIA on demand paging. https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf, 2018.

[90] NVIDIA. NVIDIA Turing GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[91] NVIDIA. Shader execution reordering - developer.nvidia.com. https://developer.nvidia.com/sites/default/files/akamai/gameworks/ser-whitepaper.pdf, 2022.

[92] NVIDIA. NVIDIA ada-tuning-guide. https://docs.nvidia.com/cuda/ada-tuning-guide/index.html, 2023.

[93] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[94] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[95] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[96] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[97] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[98] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE, 2012.

[99] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[100] Mahyar Samani, Jason Lowe-Power, and Prajwal Hegde. Grace out-of-order CPU implementation. https://github.com/darchr/novoverse/blob/main/components/processors/grace/gracecore/grace_o3_cpu.py, 2025. Accessed: 2025-02-21.

[101] Ankit Sethia, D. Anoushe Jamshidi, and Scott Mahlke. Mascar: Speeding up GPU warps by reducing memory pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185. IEEE, 2015.

[102] Lin Shi, Wen Liu, Heye Zhang, Yongming Xie, and Defeng Wang. A survey of GPU-based medical image computing techniques. *Quantitative Imaging in Medicine and Surgery*, 2(3):188, 2012.

[103] Faissal M. Sleiman and Thomas F. Wenisch. Efficiently scaling out-of-order cores for simultaneous multithreading. *ACM SIGARCH Computer Architecture News*, 44(3):431–443, 2016.

[104] Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244. ACM, 2000.

[105] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–66, 2000.

[106] Nicolai Stawinoga and Tony Field. Predictable thread coarsening. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):1–26, 2018.

[107] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[108] M. Aater Suleman, Milad Hashemi, Chris Wilkerson, Yale N. Patt, et al. Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 305–316. IEEE, 2012.

[109] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News*, 42(3):193–204, 2014.

[110] TechPowerUp. NVIDIA GeForce RTX 2060 SUPER. https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441.

[111] James E. Thornton. *Design of a Computer: The Control Data 6600*. Scott Foresman & Company, 1970.

[112] James E. Thornton. The CDC 6600 project. *Annals of the History of Computing*, 2(4):338–348, 1980.

[113] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[114] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, 1996.

[115] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[116] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 176–188, 1991.

[117] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[118] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Towards efficient GPU sharing on multicore processors. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):119–124, 2012.

[119] Yasuko Watanabe, John D. Davis, and David A. Wood. Widget: Wisconsin decoupled grid execution tiles. *ACM SIGARCH Computer Architecture News*, 38(3):2–13, 2010.

[120] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 125–137. IEEE, 2020.

[121] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010.

[122] Qiumin Xu and Murali Annavaram. PATS: Pattern aware scheduling and power gating for GPGPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 225–236, 2014.

[123] Wayne Yamamoto, Mauricio J. Serrano, Adam R. Talcott, Roger C. Wood, and M. Nemirosky. Performance estimation of multistreamed, superscalar processors. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 1, pages 195–204. IEEE, 1994.

[124] Yifan Yang, Joel S. Emer, and Daniel Sanchez. Trapezoid: A versatile accelerator for dense and sparse matrix multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 931–945. IEEE, 2024.

[125] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. Pagoda: A GPU runtime system for narrow tasks. *ACM Transactions on Parallel Computing (TOPC)*, 6(4):1–23, 2019.

[126] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. *ACM SIGARCH Computer Architecture News*, 44(3):609–621, 2016.

[127] Chao Yu, Yuebin Bai, and Rui Wang. MIPSGPU: Minimizing pipeline stalls for GPUs With non-blocking execution. *IEEE Transactions on Computers*, 70(11):1804–1816, 2020.