Architectural Support for Inter-Thread Synchronization
in SIMT Architectures

by

Ahmed Mohammed ElTantawy

BSc, Electronics and Electrical Communications, Cairo University, 2009
MSc, Electronics and Electrical Communications, Cairo University, 2011

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

Doctor of Philosophy

in

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

The University of British Columbia
(Vancouver)

January 2018

© Ahmed Mohammed ElTantawy, 2018
Abstract

Single-Instruction Multiple-Threads (SIMT) architectures have seen widespread interest in accelerating data parallel applications. In the SIMT model, small groups of scalar threads operate in lockstep. Within each group, current SIMT implementations serialize the execution of threads that follow different paths, and to ensure efficiency, revert to lockstep execution as soon as possible. These thread scheduling constraints may cause a deadlock-free program on a multiple-instruction multiple-data architecture to deadlock on a SIMT machine. Further, fine-grained synchronization is often implemented using busy-wait loops. However, busy-wait synchronization incurs significant overheads and existing CPU solutions do not readily translate to SIMT architectures. In this thesis, we tackle these challenges. First, we propose a static analysis technique that detects SIMT deadlocks by inspecting the application control flow graph (CFG). We further propose a CFG transformation that avoids SIMT deadlocks when synchronization is local to a function. The static detection has a false detection rate of 4%–5%. The automated transformation has an average performance overhead of 8.2%–10.9% compared to manual transformation. We also propose an adaptive hardware reconvergence mechanism that supports MIMD synchronization without changing the application CFG. Our hardware approach performs on par with the compiler transformation but avoids key limitations in the compiler only solution. We show that this hardware can be further extended to support concurrent multi-path execution to improve the performance of divergent applications. Finally, We propose a hardware warp scheduling policy informed by a novel hardware mechanism for accurately detecting busy-wait synchronization on GPUs. When employed, it depriortizes spinning warps achieving a speedup of 42.7% over Greedy Then Oldest scheduling.
Lay Summary

This thesis proposes techniques to ease the programmability of, General Purpose Graphics Processing Units, a widely used class of general purpose accelerators while maintaining their performance and energy efficiency. This enables leveraging the power of these accelerators in wider application domains.
Preface

The following is a list of my publications during the PhD program in chronological order.


The preceding publications have been included or used in this thesis as follows:

- Chapter 1 uses motivational elements from [C3] and [C4].
- Chapter 2 uses background section material from [C2], [C3], and [C4].
• Chapter 3 presents a version of a subset the material published in [C3] that is related to SIMT-induced deadlock definition and detection. I performed the research, interpreted the data and wrote the manuscript with guidance and input from Professor Tor M. Aamodt.

• Chapter 4 presents a version of a subset the material published in [C3] that is related to SIMT-induced deadlock elimination using compiler techniques. I performed the research, interpreted the data and wrote the manuscript with guidance and input from Professor Tor M. Aamodt.

• Chapter 5 presents a version of a subset the material published in both [C2] and [C3] that is related to a hardware mechanism that enables flexible thread scheduling in SIMT architectures. I performed the research, interpreted the data and wrote the manuscript with guidance and input from Professor Tor M. Aamodt.

• Chapter 6 presents a version of a subset the material submitted in [C4]. I performed the research, interpreted the data and wrote the manuscript with guidance and input from Professor Tor M. Aamodt. The tool proposed in [C1] is used in this chapter for energy evaluation.

• Chapter 7 presents a version of the material submitted in [C2]. I performed the research, interpreted the data and wrote the manuscript with guidance and input from Mike O’Conner and Professor Tor M. Aamodt. Jessica Wenjie Ma ideas have significantly influenced the final scoreboard design presented in the chapter and she also performed the scoreboard area analysis. The tool proposed in [C1] is used in this chapter for energy evaluation.

• Chapter 8 uses the related work sections in [C2], [C3], and [C4].

• Chapter 9 uses conclusion text from [C2], [C3], and [C4].

• Chapter A presents a version of [TR1]. I developed the proof and drafted the technical report under the guidance of Professor Tor M. Aamodt.
# Table of Contents

Abstract .............................................. ii
Lay Summary ............................................ iii
Preface .................................................. iv
Table of Contents ...................................... vi
List of Tables .......................................... xi
List of Figures ......................................... xii
Glossary .................................................. xvi
Acknowledgments ....................................... xviii

1 Introduction .......................................... 1
   1.1 SIMT execution Model Potential .................... 2
   1.2 SIMT Model Interaction with Thread Synchronization ... 3
   1.3 Thesis Statement ..................................... 7
   1.4 Methodology .......................................... 9
   1.5 Contributions ....................................... 10
   1.6 Organization ....................................... 11

2 Background .......................................... 13
   2.1 Baseline SIMT Architectures ....................... 13
| 2.2 | The SIMT Programming Model | 14 |
| 2.3 | Thread Scheduling in SIMT Architectures | 16 |
| 2.3.1 | Threads from The same Warp | 16 |
| 2.3.2 | Threads from Different Warps | 18 |
| 3 | SIMT-Induced Deadlocks: Definition and Detection | 20 |
| 3.1 | SIMT Scheduling Constraints | 21 |
| 3.1.1 | SIMT-Induced Deadlocks | 22 |
| 3.1.2 | Causes of SIMT Deadlocks | 23 |
| 3.2 | SIMT Deadlocks Impact on Programmability | 24 |
| 3.2.1 | Ease of Programmability | 25 |
| 3.2.2 | Programming Language Abstraction | 30 |
| 3.2.3 | OpenMP Case Study: Synchronization Primitives Library | 31 |
| 3.3 | SIMT Deadlock Detection | 33 |
| 3.3.1 | SIMT Deadlock Detection Algorithm | 34 |
| 3.3.2 | SIMT Deadlock Detection Limitations | 37 |
| 3.4 | Methodology | 39 |
| 3.5 | Evaluation | 40 |
| 3.6 | Related Work | 41 |
| 3.7 | Summary, Conclusion and Future Directions | 42 |
| 4 | SSDE: Static SIMT-induced Deadlock Elimination | 44 |
| 4.1 | Safe Reconvergence Points Identification | 45 |
| 4.2 | SSDE: Static SIMT Deadlock Elimination | 49 |
| 4.2.1 | Elimination Algorithm | 50 |
| 4.2.2 | Compatibility with Nvidia GPUs | 52 |
| 4.2.3 | SSDE Limitations | 53 |
| 4.3 | Methodology | 55 |
| 4.4 | Evaluation | 56 |
| 4.4.1 | Static Overheads | 56 |
| 4.4.2 | Dynamic Overheads | 57 |
| 4.4.3 | OpenMP support | 60 |
| 4.5 | Related Work | 62 |
4.6 Summary, Conclusion and Future Directions .......................... 62

5 AWARE: Adaptive Warp Reconvergence .............................. 64
  5.1 Decoupled SIMT Tables ............................................. 66
  5.2 Warp Splits Scheduling .............................................. 69
  5.3 Nested Divergence .................................................. 70
  5.4 Using AWARE to avoid SIMT Deadlock ............................ 72
    5.4.1 Handling Divergent Barriers: ................................. 73
    5.4.2 Delayed Reconvergence: ....................................... 74
    5.4.3 Timed-out Reconvergence: ..................................... 75
  5.5 AWARE Implementation .............................................. 77
    5.5.1 AWARE Basic Implementation ................................. 77
    5.5.2 AWARE Virtualized Implementation ........................ 79
  5.6 Methodology .......................................................... 82
  5.7 Evaluation ............................................................ 82
  5.8 Related Work ........................................................ 86
  5.9 Summary, Conclusion and Future Directions ..................... 87

6 BOWS: Back-Off Warp Spinning ........................................ 89
  6.1 Sensitivity to Warp Scheduling .................................... 91
  6.2 BOWS: Backoff Warp Spinning .................................... 92
    6.2.1 BOWS scheduling policy ...................................... 92
  6.3 DDOS: Dynamic Detection of Spinning ............................ 95
    6.3.1 DDOS Operation ................................................ 98
    6.3.2 DDOS Design Trade-offs ..................................... 101
    6.3.3 DDOS integration with BOWS ................................. 104
  6.4 Methodology .......................................................... 105
  6.5 Evaluation ............................................................ 106
    6.5.1 Sensitivity to Back-off Delay Limit Value ................. 107
    6.5.2 Sensitivity to Detection Errors .............................. 110
    6.5.3 Sensitivity to Contention .................................... 111
    6.5.4 Pascal GTX1080Ti Evaluation ................................. 112
    6.5.5 Implementation Cost .......................................... 112

viii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6 Related Work</td>
<td>113</td>
</tr>
<tr>
<td>6.7 Summary, Conclusion and Future Directions</td>
<td>115</td>
</tr>
<tr>
<td>7 MP: Multi-Path Concurrent Execution</td>
<td>117</td>
</tr>
<tr>
<td>7.1 Stack-Based Reconvergence Performance Limitations:</td>
<td>117</td>
</tr>
<tr>
<td>7.2 Multi-Path IPDOM (MP IPDOM)</td>
<td>119</td>
</tr>
<tr>
<td>7.2.1 Warp Split Scheduling</td>
<td>120</td>
</tr>
<tr>
<td>7.2.2 Scoreboard Logic</td>
<td>122</td>
</tr>
<tr>
<td>7.3 Opportunistic Early Reconvergence</td>
<td>125</td>
</tr>
<tr>
<td>7.4 Methodology</td>
<td>126</td>
</tr>
<tr>
<td>7.5 Experimental Results</td>
<td>129</td>
</tr>
<tr>
<td>7.5.1 SIMD Unit Utilization</td>
<td>130</td>
</tr>
<tr>
<td>7.5.2 Thread Level Parallelism</td>
<td>130</td>
</tr>
<tr>
<td>7.5.3 Idle Cycles</td>
<td>132</td>
</tr>
<tr>
<td>7.5.4 Impact on Memory System</td>
<td>132</td>
</tr>
<tr>
<td>7.5.5 Overall Performance</td>
<td>134</td>
</tr>
<tr>
<td>7.5.6 Implementation Complexity</td>
<td>134</td>
</tr>
<tr>
<td>7.6 Related Work</td>
<td>135</td>
</tr>
<tr>
<td>7.7 Summary, Conclusion and Future Directions</td>
<td>136</td>
</tr>
<tr>
<td>8 Related Work</td>
<td>137</td>
</tr>
<tr>
<td>8.1 Enabling Thread Synchronization in GPGPUs</td>
<td>137</td>
</tr>
<tr>
<td>8.2 Warp Scheduling Policies in GPGPUs</td>
<td>142</td>
</tr>
<tr>
<td>8.3 Alternate GPGPU Execution Models</td>
<td>144</td>
</tr>
<tr>
<td>8.4 Verification Tools for SIMT architectures</td>
<td>146</td>
</tr>
<tr>
<td>9 Conclusions and Future Work</td>
<td>147</td>
</tr>
<tr>
<td>9.1 Conclusion</td>
<td>147</td>
</tr>
<tr>
<td>9.2 Potential Areas of Impact</td>
<td>149</td>
</tr>
<tr>
<td>9.3 Directions of Future Work</td>
<td>150</td>
</tr>
<tr>
<td>9.3.1 SIMT Synchronization APIs</td>
<td>150</td>
</tr>
<tr>
<td>9.3.2 Runtime Livelock Detection</td>
<td>152</td>
</tr>
<tr>
<td>9.3.3 Reconvergence adequacy prediction</td>
<td>152</td>
</tr>
</tbody>
</table>
List of Tables

Table 3.1 Evaluated Kernels ........................................... 39
Table 3.2 Detection Pass Results on CUDA and OpenCL Kernels .... 40
Table 4.1 Code Configuration Encoding ............................... 56
Table 4.2 Static Overheads for the Elimination Algorithm .......... 56
Table 4.3 OpenMP Kernels (Normalized Execution Times) .......... 60
Table 5.1 AWARE vs SSDE ........................................... 66
Table 5.2 Storage Cost in Bits per Hardware Warp. ................. 82
Table 5.3 GPGPUSim Configuration ................................... 83
Table 6.1 Spin Detection Sensitivity to Design Parameters .......... 102
Table 6.2 DDOS and BOWS Implementation Costs ................. 114
Table 7.1 GPGPUSim Configuration ................................. 128
Table 7.2 Studied Benchmarks ........................................ 129
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>SIMT-Induced Deadlock</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Fine-grained Synchronization in current GPGPUs. Both CPU and GPU versions are compiled with NVCC-6.5 -O3. Overheads are measured on Pascal GTX1080 using Nvidia profiler (nvprof) launching 120 blocks each of 256 threads (74.88% occupancy of the pascal GPU).</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Baseline architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>CUDA Programming Model. (This diagram reproduces Figure 7 and Figure 8 from The CUDA Programming Guide [111].)</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Divergent code example</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Stack-Based Reconvergence</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Barriers in divergent code</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Critical Section Implementation</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>HashTable Insertion - Version 1 [129].</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>HashTable Insertion - Version 2</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Modified Critical Section Implementation</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>HashTable Insertion Versions Comparison (2M Random Insertion in 1024 HashTable Entries). Recent GPUs are to the right.</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Synchronization patterns used in multi-threaded applications. Examples are from the graph analytics API in the Cloud suite [40].</td>
<td>29</td>
</tr>
<tr>
<td>3.9</td>
<td>PTX output of compiling the code in Figure 3.5 using NVCC 6.5 and NVCC 8.0 with default optimizations enabled.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>OpenMP Clang 3.8 Frontend Implementation for OpenMP 4.0</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>OpenMP Microbenchmark.</td>
<td></td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>addByte function in Histogram256 [120].</td>
<td></td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>SIMT-induced deadlock scenarios.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>SIMT-Induced Deadlock Elimination Steps.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Normalized Accumulative GPU Execution Time.</td>
<td></td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Normalized Kernel Execution Time</td>
<td></td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Normalized Dynamic Instruction Count</td>
<td></td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Average SIMD Utilization</td>
<td></td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Execution with AWARE</td>
<td></td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Example of Multi-Path IPDOM execution with nested divergence</td>
<td></td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Handling Barriers</td>
<td></td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Delayed Reconvergence</td>
<td></td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Timed-Out Reconvergence</td>
<td></td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>AWARE implementation</td>
<td></td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>AWARE Virtualized Implementation</td>
<td></td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Normalized Kernel Execution Time</td>
<td></td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Normalized Accumulated GPU Execution Time</td>
<td></td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>Normalized Dynamic Instruction Count</td>
<td></td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>Average SIMD Utilization</td>
<td></td>
</tr>
<tr>
<td>Figure 5.12</td>
<td>Sensitivity to the TimeOut value (in cycles). &quot;inf+DR&quot; refers to a time-out that is infinity but with delayed reconvergence.</td>
<td></td>
</tr>
<tr>
<td>Figure 5.13</td>
<td>Effect of AWARE Virtualization on Performance</td>
<td></td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Synchronization Status Distribution. Bars from left to right: LRR, GTO, and CAWA. GPGPU-Sim with a GTX480 configuration. See Section 7.4 for details.</td>
<td></td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Software Backoff Delay Performance in GPUs. *omp_set_lock GPU implementation for OpenMP 4.0 [15].</td>
<td></td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>BOWS scheduling Policy</td>
<td></td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Adaptive Back-off Delay Limit Estimation.</td>
<td></td>
</tr>
</tbody>
</table>

xiii
Figure 6.5  Examples of Inter-Thread Synchronization Patterns used in GPUs  
(See Section 7.4 for more details)  96
Figure 6.6  Warp History Registers and SIB-PT Operation (Figure 6.7 shows 
the units locations in the pipeline)  97
Figure 6.7  Operation of BOWS with DDOS  104
Figure 6.8  Performance and Energy Savings on GTX480 (Fermi)  106
Figure 6.9  Normalized Execution Time at Different Back-off Delay Limit 
Values (using DDOS)  107
Figure 6.10  Distribution of Warps at the Scheduler. From left to right, GTO 
without BOWS, GTO with BOWS with delay limit in cycles 0, 
500, 1000, 3000, 5000, Adaptive  108
Figure 6.11  Distribution of Warps at the Scheduler. From left to right, GTO 
without BOWS, GTO with BOWS with delay limit in cycles 0, 
500, 1000, 3000, 5000, Adaptive  109
Figure 6.12  BOWS Impact on Dynamic Overheads  109
Figure 6.13  Overheads Due to Detection Errors  110
Figure 6.14  Performance and Energy Savings on Pascal  111
Figure 6.15  Sensitivity to Contention  113

Figure 7.1  Divergent code example  118
Figure 7.2  Execution with the Stack-Based Reconvergence Model. The 
Figure refers to the stack as a Single Path Stack to distinguish 
it from latter proposals that support dual path execution [123]  118
Figure 7.3  Fraction of running scalar threads while varying maximum 
warp splits and assuming IPDOM reconvergence  120
Figure 7.4  Execution with Multi-Path IPDOM  121
Figure 7.5  Changes required to the scoreboard logic to support Multi-Path 
IPDOM  123
Figure 7.6  MP IPDOM scoreboard example  124
Figure 7.7  Unstructured control flow  126
Figure 7.8  Operation of the Multi-Path with the Opportunistic Reconver-
gence (OREC) enabled  127
Figure 7.9  SIMD units utilization  131
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.10</td>
<td>Warp splits to warp ratio</td>
<td>131</td>
</tr>
<tr>
<td>7.11</td>
<td>Average breakdown of threads’ state at the scheduler</td>
<td>132</td>
</tr>
<tr>
<td>7.12</td>
<td>Idle cycles</td>
<td>133</td>
</tr>
<tr>
<td>7.13</td>
<td>Inst. cache misses (16KB I$)</td>
<td>133</td>
</tr>
<tr>
<td>7.14</td>
<td>L1 data cache misses (32KB D$)</td>
<td>133</td>
</tr>
<tr>
<td>7.15</td>
<td>Overall speedup</td>
<td>134</td>
</tr>
<tr>
<td>9.1</td>
<td>Illustration to the use of named locks</td>
<td>151</td>
</tr>
<tr>
<td>A.1</td>
<td>Visualization of T</td>
<td>171</td>
</tr>
</tbody>
</table>
Glossary

AMD  Advanced Micro Devices
AWARE  Adaptive Warp Reconvergence
ALU  Arithmetic Logic Unit
BOWS  Back-off Warp Spinning
CUDA  Compute Unified Device Architecture
CPU  Central Processing Unit
DDOS  Dynamic Detection of Spinning
DRAM  Dynamic Random Access Memory
GPGPU  General Purpose Graphics Processing Units
GPU  Graphics Processing Units
GTO  Greedy-Then-Oldest
I-BUFFER  Instruction Buffer
I-CACHE  Instruction-cache
IBM  [International Business Machines]
IPC  Instructions Per Cycle
IPDO  Immediate Post Dominator
L1  Level One

L2  Level Two

LRR  Loose Round Robin

MIMD  Multiple Instructions Multiple Data

MP  Multi-Path

OPENCL  Open Computing Language

OPENMP  Open Multi-Processing

PC  Program Counter

PDOM  Post Dominator

RT  RTReconvergence Table

SIMT  Single Instruction Multiple Threads

SIMD  Single Instruction Multiple Data

SM  Streaming Multiprocessor

ST  Splits Table

SSDE  Static SIMT Deadlock Elimination

TLP  Thread-Level Parallelism
Acknowledgments

"All praise and thanks are for God, the One who, by His blessing and favor, perfected good works are accomplished.", Prophet Mohammed.

It has been a long journey full of ups and downs. But, there are people who made this journey worth taking. First, I shall start by thanking my parents, my mother Elham and my father Mohammed. None of this would have been possible without your unconditional love throughout my life. I hope I can be one day the one you brought me to be. I want also to thank my sisters Hanan, Samah, and Amal and my brother Hossam. It was hard to be away from such a lovely family all these years with a single visit to home during my PhD but I wish to make this up for you in the coming years. There is one person, though, with whom I literally shared all the joy and pain during this journey, my wife. I could not do it without you. You were always of a great support. There were times where I put tremendous pressure on you, while you yourself were also doing your PhD. But, you were always there for me. I love you. I can not forget Mazin, my little monkey, you brought all the joy of life with you in my last couple of years in the PhD journey. I was also blessed by having a great community in Vancouver, especially in UBC campus. Thanks for every member in this community. Thank you for all my friends who made this journey enjoyable.

I was very blessed to join a world class research group led by Professor Tor. I would like to thank professor Tor for all the time and support that I always found during my PhD journey. This work was not possible without such support. I would like to also thank all my lab colleagues, Wilson, Andrew, Tayler, Tim, Ayoub, Hadi, Shadi, Myrice, Dave, Ali, Inder, Jimmy, Mahmoud, and Amruth. I learned a lot from all of you.
Chapter 1

Introduction

Over the last few decades, semiconductor process technology has been advancing according to Moore’s Law, which states that the density of transistors on integrated circuits doubles about once every two years. This increase in the number of transistors has been utilized to improve the single thread performance in general purpose CPUs. However, CPUs have hit a major challenge known as the power wall [98] which limits the increase in single thread performance. Therefore, computer architects have moved towards energy efficient parallel architectures. Massively multithreaded architectures, such as graphic processing units (GPUs), mitigate the power problem by running thousands of threads in parallel at lower frequencies, and amortizing the cost of fetching, decoding and scheduling instructions by executing them in a single instruction multiple data (SIMD) fashion.

These properties have motivated the computer industry to transform GPUs from merely fixed function accelerators for graphics into programmable compute accelerators. For such a transformation to happen, there was a need to develop adequate programming models for GPUs that allow non-graphics applications to utilize the computing power of GPUs without using graphics-oriented APIs. This led to the development of general purpose programming models for GPUs such as CUDA [111] and OpenCL [8]. The resultant programming model is referred to as the Single Instruction Multiple Thread (SIMT) model. The SIMT model has seen widespread interest and similar models have been adopted in CPU architectures with wide-vector support [64].
Using such programming models, software developers have demonstrated that SIMT architectures have significant potential in cost-effective computing for data-parallel applications with regular control-flow and regular memory access patterns [110]. However, it is quite challenging to obtain similar results on applications that have a significant portion of their instruction streams common across threads yet feature non-uniform control behaviour, irregular memory access patterns, and/or inter-thread synchronization [25]. This motivated the computer architecture community to study modifications to the graphics-based SIMT architectures to allow for efficient acceleration of wider scope of general purpose applications [11, 41, 91, 99, 124]. This thesis is part of this ongoing work.

1.1 SIMT execution Model Potential

Traditional Single Instructions Multiple Data (SIMD) architectures are hard to program. The underlying hardware provides little support for arbitrary memory access and control flow divergence [63, 128]. Thus, it is essential for the code running on such machines to be explicitly vectorized. The vectorization task is fully or partially performed by programmers [61]. Compilers can perform automatic vectorization but they fail on simple cases due to uncertainty about loop iterations’ dependencies or non-uniform memory access stride [62]. Thus, traditional SIMD systems are either hard to program and/or limited in scope.

This has significantly changed with the introduction of the Single Instructions Multiple Threads (SIMT) architectures. In SIMT architectures, the hardware with minimal help from the compiler supports arbitrary memory accesses and control flow divergence. This abstracts away the complexity of the underlying SIMD hardware allowing for much simpler programming models. The single-instruction multiple-thread (SIMT) programming model was originally introduced and popularized for graphics processor units (GPUs) along with the introduction of CUDA [111] but it has seen widespread interest and similar models have been adopted in CPU architectures with wide-vector support [64].

Arguably, a key reason for the success of this model is its abstraction of the underlying SIMD hardware. In SIMT-like execution models, scalar threads are combined into groups that execute in lockstep on single-instruction multiple-data
(SIMD) units. These groups are called warps by NVIDIA [111], wavefronts by AMD [4], and gangs by Intel [64]. The SIMT programming model divides the burden of identifying parallelism differently than traditional approaches of vector machines. The programmer, who is armed with application knowledge, identifies far-flung outer-loop parallelism and specifies the required behaviour of a single thread in the parallel region. The hardware implicitly handles control flow and memory divergence within threads of the same warp. Thus, with this abstraction, programmers can leverage the underlying SIMD hardware without having to deal with explicit vectorization.

However, current implementations for this desired abstraction are still far from perfect. In situations that involve inter-thread synchronization, the SIMD nature of the underlying hardware induces special types of deadlocks that would not happen otherwise. Further, recent SIMT implementations still suffer excessive performance overheads under non-uniform control behaviour, irregular memory access patterns, and/or inter-thread synchronization. This negatively impacts the programmability of SIMT architectures on irregular applications as it forces programmers to be aware of the details of the SIMT implementation to write functionally correct and optimized code.

1.2 SIMT Model Interaction with Thread Synchronization

On current hardware the SIMT model is implemented via predication, or in the general case using stack-based masking of execution units [5, 18, 29, 64, 77]. This mechanism enables threads within the same warp to diverge (i.e., follow different control flow paths). To do this, the hardware forces divergent threads to serialize their execution and then restores SIMD utilization by forcing divergent threads to reconverge as soon as possible (typically at the immediate postdominator point of the divergent branch) [29, 60, 64]. This mechanism creates implicit scheduling constraints for divergent threads within a warp which leads to programmability implications. For example, when a GPU kernel code is written in such a way that the programmer intends divergent threads to communicate, these scheduling constraints can lead to surprising (from a programmer perspective) deadlock and/or
Figure 1.1: SIMT-Induced Deadlock

livelock conditions. Thus, a multi-threaded program that is guaranteed to terminate on a MIMD architecture may not terminate on machines with current SIMT implementations \[47\].

Figure 1.1 shows a typical MIMD implementation of a spin lock guarding a critical section. On a SIMT machine, this code deadlocks. In particular, a thread that acquires the lock is indefinitely blocked at the loop exit waiting to reconverge with lagging threads from the same warp. However, lagging threads never exit the loop because they wait for the lock to be released by the leading thread. Similar scenarios occur with fine-grained synchronization. We refer to a case where the forward progress of a diverged thread is prevented due to the implicit SIMT scheduling constraints as \textit{SIMT-induced deadlock} or briefly \textit{SIMT deadlock}.

The possibility of SIMT-induced deadlocks is a challenge, given the increasing interest in using SIMT architectures for irregular applications \[25, 54, 58, 78, 90, 92, 96, 152\]. Moreover, parallel algorithms developed for MIMD execution can serve as starting points for GPU kernel development provided SIMT deadlock can be avoided. For complex applications writing functionally correct code can be challenging as programmers need to reason about how synchronization interacts with the SIMT implementation. Further, the code is vulnerable to compiler optimizations that may modify the control flow graph (CFG) assumed by programmers. SIMT deadlocks also present challenges to emerging OpenMP support for SIMT architectures \[7, 15, 71, 101, 117\] and to the transparent vectorization of

\[1\] We use the term “MIMD machine” to refer to any architecture that guarantees loose fairness in thread scheduling so that threads not waiting on a programmer synchronization condition make forward progress.
multi-threaded code on SIMD CPU architectures [64, 140].

Note that during the writing of this dissertation, on May 10th 2017, Nvidia revealed some details about their newest GPGPU architecture; Volta [103]. Volta supports “independent thread scheduling” to avoid thread synchronization deadlocks on earlier architectures and to enable interleaving the execution of divergent control flow paths. To the best of our knowledge, this is the first SIMT architecture in the market that supports such independent thread scheduling. No details were provided about how such independent thread scheduling is implemented. However, this dissertation proposes one way how such independent thread scheduling can be implemented in hardware to avoid synchronization deadlocks (Chapter 5) and to interleave divergent control flow paths (Chapter 7) while maintaining SIMT efficient execution. In this dissertation, we use the term current or recent SIMT architectures to refer to SIMT architectures prior to Volta.

Aside from these functional limitations imposed by current SIMT implementations, there are also performance implications. Let us consider a manual workaround for the SIMT deadlock problem shown in Figure 1.2a. In Figure 1.2a, the code is structured such that the atomicExch release statement (line 10) is contained within the spin loop to avoid SIMT-induced deadlocks. In this code threads that successfully acquire the lock are guaranteed to be able to make forward progress to the lock release code. This workaround handles a simple case where there is a single lock acquire statement with a single lock release statement that postdominates the lock acquire. Thus, the required code transformation is relatively simple to reason about using high level semantics provided the programmer is aware of the details of the reconvergence mechanism. This is, however, not necessarily true for more complex synchronization patterns (more details in Chapter 3). Next we illustrate the performance implications of the SIMT model on this code even after SIMT deadlocks are avoided.

The code in Figure 1.2a is an implementation of a critical section in a hashtable implementation in CUDA. Hashtables in GPUs are used in key-value store applications [54], text mining [157, 158], state space exploration [146], DNA alignment [159], and others [1]. The implementation we study here is an optimized version of NVIDIA’s CUDA by Example [129] (more details in Chapter 3). Figure 1.2b compares the execution time of 26.2 million insertions of random keys to
5. unsigned int key = keys[tid];
6. size_t hashValue = hash(key, table.count);
7. Entry *location = &(table.pool[tid]);
8. location->key = key;
9. location->value = values[tid];
10. bool done = false;
11. while(!done){
12. if(atomicCAS(lock[hashValue].mutex, 0, 1) == 0 ){
13. __threadfence();
14. location->next = table.entries[hashValue];
15. table.entries[hashValue] = location;
16. done = true;
17. __threadfence();
18. atomicExch(lock[hashValue].mutex,0);
19. }
20. }

(a) Critical Section in Hashtable Insertion

(b) GPU performance vs CPU.

c) Dynamic Instructions Overheads,

(d) Memory Traffic Overheads.

(e) Divergence Overheads.

Figure 1.2: Fine-grained Synchronization in current GPGPUs. Both CPU and GPU versions are compiled with NVCC-6.5 -O3. Overheads are measured on Pascal GTX1080 using Nvidia profiler (nvprof) launching 120 blocks each of 256 threads (74.88% occupancy of the pascal GPU).
the hashtable on a GPU versus on a CPU while varying the number of hashtable buckets. The smaller the number of the buckets, the larger the contention. The figure shows the execution time on two different generations of NVIDIA GPUs—a Tesla C2050 (Fermi), and GeForce GTX 1080 (Pascal)—and an Intel Core i7 CPU running a serial CPU version of the same hashtable code [129]. The CPU version outperforms the older Tesla C2050 for all sizes considered while the GTX 1080 outperforms the CPU starting from 512 hashtable buckets. At 4096 buckets the GTX 1080 is 9.77× faster than the serial CPU version (more details in Chapter 6).

Nevertheless, Figures 1.2c, 1.2d, and 1.2e show significant synchronization overheads that are still persistent in the Pascal architecture. Figures 1.2c shows that instruction count overhead ranges from 61.0% at low contention to 98.3% at high contention. Similarly, Figure 1.2d shows that 41.5% to 95.6% of memory operations are due to synchronization. A significant portion of both overheads are due to failed lock acquire attempts. Another source of synchronization overhead, unique to GPUs, is control-flow divergence. Figure 1.2e shows that if the code is executed by a single warp, the SIMD utilization (fraction of active lanes) ranges between 87.1%-98.6% but drops to 16.4%-47.1% when executing multiple warps. This is due to inter-warp lock conflicts, which can be impacted by warp scheduling.

1.3 Thesis Statement

This dissertation primarily explores methods that enable reliable and efficient support of inter-thread synchronization on SIMT architectures. The dissertation first highlights the functional implications of the thread scheduling constraints imposed by current SIMT implementations. It shows that a special type of deadlock (SIMT-induced deadlocks) can be produced by programmers and/or compilers that are oblivious to these constraints. The dissertation then proposes a novel static analysis technique to conservatively detect the presence of SIMT deadlocks in parallel kernels. The analysis builds on the following observation: the forward progress of a thread is prevented indefinitely (i.e., SIMT deadlock occurs) if a thread enters a loop for which the exit condition depends on the value of a shared memory location and that location will only be set by another thread that is blocked due to the SIMT scheduling constraints. This analysis conservatively reports potential SIMT deadlocks.
deadlocks. In practice it has a small positive false detection rate.

The dissertation also proposes two independent novel solutions for SIMT deadlocks: (1) SSDE: Static SIMT-induced Deadlock Elimination algorithm, which alters the CFG of the application to avoid SIMT deadlocks, and (2) AWARE: an Adaptive Warp REconvergence hardware mechanism, which has the flexibility to delay or timeout reconvergence without changing the application’s CFG. In SSDE, a static analysis is done first to identify code locations where reconvergence of loop exits should be moved to allow for inter-thread communication that is otherwise blocked by SIMT Constraints. For functional correctness, we can choose any postdominator point as the reconvergence point including the kernel exit. However, from a SIMD utilization perspective, it is preferable to reconverge at the earliest postdominator point that would not block the required inter-thread communication. We call these postdominator points safe postdominators. The static analysis is then followed by a CFG transformation algorithm that enforces the recommended safe postdominators as reconvergence points for loops.

The second solution is AWARE; a MIMD-Compatible reconvergence mechanism that avoids most limitations inherent in a compiler-only approach. AWARE relaxes the thread scheduling constraints imposed by current stack-based SIMT implementations. To this end, AWARE decouples the tracking of reconvergence points from the divergent control flow paths using two separate SIMT tables. Instead of the hardware stack structure, AWARE uses a hardware FIFO structure to reorder the execution of divergent control flow paths. This replaces the unfair depth-first execution of the kernel control flow graph by a fair breadth-first execution. Further, the decoupling of reconvergence points from divergent paths gives AWARE the flexibility to delay reconvergence (using the recommended safe postdominator points from the SSDE analysis pass) and/or to timeout reconvergence after a certain threshold period.

To address the overheads of busy-wait synchronization on current SIMT architectures, the dissertation proposes Back-Off Warp Spinning (BOWS); a novel hardware mechanism to dynamically detect spinning warps and modify warp scheduling. BOWS’ spin detection mechanism employs a path history register to identify repetitive execution (i.e., loops). To distinguish busy-wait synchronization loops from other loops BOWS employs a value history register to track the values of
registers used in the computation of the loop exit conditions. In loops not associated with busy-waiting, at least one of these registers typically holds the value of a loop induction variable that changes each iteration. In busy-wait loops these registers typically maintain the same values as long as the warp is spinning. For the workloads we study this simple mechanism accurately identifies busy-wait loops. The spin detection results then guides a scheduling policy, BOWS, that is designed to discourage spinning warps from competing for scheduler issue slots. BOWS efficiently approximates software back-off techniques used in multi-threaded CPU architectures while overcoming their limitations when applied to GPUs. In BOWS, warps that are about to execute a busy-wait iteration are removed from competition for scheduler issue slots until no other warps are ready to be scheduled and a certain time has passed since the previous iteration.

Finally, the dissertation explores an orthogonal application of the SIMT tables used in AWARE construction. In particular, along with additional microarchitectural modifications, the SIMT tables can be used to enable concurrent multi-path execution as a performance optimization for divergent applications. We refer to this mechanism as the Multi-Path (MP) execution model. In the MP execution model, an arbitrary number of parallel warp splits can interleave their execution while still maintaining immediate postdominator reconvergence. To enable this, the MP model proposes modifications to the scoreboard logic used to track data dependencies between registers in SIMT architectures. The modifications assure that the scoreboard can correctly handle dependencies for concurrent multi-path executions and avoids declaring false data dependencies between orthogonal parallel paths. Further, MP extends the operation of the SIMT tables to enable opportunistic early reconvergence at run-time which improves the SIMD units utilization of applications with unstructured control flow.

1.4 Methodology

We describe our methodology in details before the evaluation section of each chapter. However, in general, we rely on direct hardware measurements when possible (e.g., in Chapters 3, 4, and motivation part of chapter and 6). If a change in current hardware is proposed, we use cycle accurate simulation to evaluate the impact of
our proposal on a baseline that simulates recent architectures. This is the standard approach for computer architecture research in both industry and academia. We use GPGPU-Sim [11] simulator for performance simulation and GPUWattch [75] for power simulation. Both simulators report high correlation with recent hardware and are widely used in the research community (with more than 1000 citation for GPGPU-Sim and 300 citations for GPUWattch at the time of writing this thesis). The evaluated kernels usually run for hundred thousands to millions of cycles on these simulators (which correspond to hours to days in simulation time).

In our evaluation, we focus on the accelerated portion of the applications (i.e., the GPGPU kernels). Our focus is to improve GPU support for synchronization, but we make sure to evaluate the impact of our proposals on kernels that does include such synchronization. To represent these types of kernels, we used well-known benchmark suites (e.g., [26], [11]). It was a challenging task to find existing GPGPU kernels that utilize synchronization, arguably, because of the challenges discussed earlier in porting and/or developing such kernels on current GPU accelerators. However, we secured a number of kernels that show different patterns of synchronization (e.g., spin locks, nested spin locks, wait and signal, data-flow, small and large critical sections). The kernels used represent different application domains (e.g., graph analysis, key-value store, transactions, physics simulation). We describe our benchmarks in the methodology sections of each chapter.

1.5 Contributions

This thesis makes the following key contributions:

- It formalizes a special type of deadlocks that is imposed by current SIMT implementations and can be introduced by programmers and/or compilers that are oblivious to the thread scheduling constraints.

- It proposes a static analysis technique that conservatively detects potential SIMT-induced deadlocks in parallel kernels.

- It proposes a static analysis that identifies safe locations to reconverge threads divergent at loop exits while allowing for inter-thread communication under divergence.
• It proposes a code transformation algorithm that modifies the CFG to eliminate SIMT-induced deadlocks guided by the safe reconvergence analysis. It also identifies the limitations of relying on a compiler-only approach to eliminate SIMT-induced deadlocks.

• It proposes a MIMD-Compatible hardware reconvergence mechanism for SIMT architectures that avoids key limitations of the compiler-only approach.

• It proposes a low cost dynamic spin detection mechanism for SIMT architectures.

• It proposes an inter-thread synchronization aware warp scheduling policy that reduces busy-wait synchronization overheads in SIMT architectures.

• It proposes a multi-path execution model that enables concurrent execution of parallel control flow paths while maintaining reconvergence at immediate postdominators.

1.6 Organization

The rest of this dissertation is organized as follows:

• Chapter 2 details the background GPU architecture used in this dissertation.

• Chapter 3 introduces the definition of SIMT-induced deadlocks and the static analysis used for their detection.

• Chapter 4 introduces SSDE; a Static SIMD-induced Deadlock Elimination algorithm.

• Chapter 5 introduces AWARE; an Adaptive Warp Reconvergence Mechanism.

• Chapter 6 introduces BOWS; a warp scheduling policy for busy-wait synchronization in SIMT architectures.

• Chapter 7 introduces MP; a Multi-Path execution model for concurrent execution of divergent paths in SIMT architectures.
• Chapter 8 discusses related work.

• Chapter 9 concludes the dissertation and discusses future work.

• Chapter A is an appendix that presents a semi-formal proof to the correctness of the SSDE transformation.

The chapters are ordered to maintain a coherent flow of ideas. Nevertheless, each chapter is self-contained as it includes a brief introduction, a presentation of the new ideas, an evaluation section, a related work discussion, a conclusion, and future work directions. Chapters 8 and 9 expands over the related work discussion, conclusion and future work directions included in each chapter.
Chapter 2

Background

This chapter reviews the necessary background material for the rest of the dissertation. Section 2.1 describes the architecture of contemporary SIMT accelerators which acts as our baseline throughout the thesis. Section 2.2 provides a brief overview on current SIMT programming models. Section 2.3 provides more details on thread scheduling schemes in current SIMT accelerators as it is significantly relevant to the rest of the thesis.

2.1 Baseline SIMT Architectures

We study modifications to the SIMT accelerator architectures as shown in Figure 2.1. This architecture is the one deployed in contemporary General Purpose Graphics Processing Units (GPGPUs) [104, 108]. Current GPGPUs consist of multiple processing cores. Each core consists of a set of parallel lanes (or SIMD units). Initially, an application begins execution on a host CPU, then a kernel is launched on the GPU in the form of a large number of logically independent scalar threads. These threads are split into logical groups operating in lockstep in a SIMD fashion (referred to as warps by Nvidia and wavefronts by AMD)\(^1\). Each SIMT core interleaves a number of warps on a cycle-by-cycle basis. The Instruction Buffer unit (I-Buffer) contains storage to hold decoded instructions and register dependency information for each warp.

\(^1\)In the rest of this thesis, we adopt Nvidia’s naming convention.
The scoreboard unit is used to detect register dependencies. A branch unit manages control flow divergence. The branch unit abstracts both the storage and the control logic required for divergence and reconvergence.

The issue logic selects a warp with a ready instruction in the instruction buffer to issue for execution. Based on the active mask of the warp, threads that should not execute, due to branch divergence, are disabled. The issued instructions fetch their operands from the register file. It is then executed on the corresponding pipeline (ALU or MEM).

The SIMT architecture achieves its energy efficiency by amortizing the front end costs (i.e., fetching, decoding, and scheduling instructions) across the large number of threads within the same warp executing synchronously the same instruction. Further, it lowers the operating frequency and relaxes the latency requirements of the memory system and functional units compared to contemporary CPUs. To hide this latency, it relies on efficient warp scheduling policies that allows for a net high instruction throughput per cycle.

2.2 The SIMT Programming Model

The SIMT programming model divides the burden of identifying parallelism differently than traditional approaches of vector parallelism. The programmer, who is armed with application knowledge, identifies far-flung outer-loop parallelism and
Figure 2.2: CUDA Programming Model. (This diagram reproduces Figure 7 and Figure 8 from The CUDA Programming Guide [111].)

specifies the required behaviour of a scalar thread in the parallel region. Figure 2.2 shows a typical flow of a CUDA program. The host side (i.e., the CPU) executes the serial portion of the code, allocates the required memory on the device side (i.e., the GPU) and copy the required data from/to the host to/from the device. The programmer decides the number of logical threads required to execute the parallel region. These threads are typically organized as a grid of thread blocks. Threads in the same thread block can communicate through a low latency scratchpad memory referred to as shared memory. Threads in different thread blocks (and even different grids) can still communicate through a slower global memory.

The programming model provides some essential primitives that are required to manage inter-thread communication. These primitives include thread-block scope barriers to synchronize threads within the same thread block, thread-block scope and device scope memory fences to control the observable ordering of shared and
global memory reads and writes, and atomic functions that perform read-modify-write atomic operations (e.g., Compare and Swap, Add, Min, Max) where the operation is guaranteed to be performed without interference from other threads. The programmer writes the kernel code from the perspective of a scalar thread and leverages these primitives to manage inter-thread communication across the logically independent threads.

The SIMT programming model, in its essence, does not expose the SIMD nature of the hardware to programmers. It also does not expose the mapping and the scheduling of the logical threads on the available hardware resources. Thus, from a programming model perspective, individual threads progress independently unless otherwise determined by programmers through the use of explicit synchronization primitives. This makes SIMT architectures both easier to program and suitable for a larger set of applications.

2.3 Thread Scheduling in SIMT Architectures

Thread scheduling in SIMT architectures is done by the hardware. The hardware is responsible for mapping thread blocks to the different processing cores, allocating the required resources for different threads and mapping individual threads to the SIMD units. Throughout the execution of the kernel, SIMT architectures greedily attempts to synchronize threads within the same warp to maximize SIMD units utilization, coalesce their memory accesses to reduce memory traffic, and optimize warp scheduling to efficiently hide long latency operations and harness the existing data locality. This section overviews the current policies used in thread scheduling.

2.3.1 Threads from The same Warp

As discussed in Section 2.2, SIMT architectures, unlike traditional vector machines, allow for arbitrary control flow divergence. Threads are split into groups (warps) that execute in lockstep on the underlying SIMD units. The warp size in recent GPUs is typically the number of available parallel SIMD lanes. With no divergence, threads within the same warp share the same program counter. However, upon a divergent branch, threads in a warp are allowed to follow different control flow paths. Current implementations achieve this by serializing the execution of
different control-flow paths while restoring SIMD utilization by forcing divergent threads to reconverge as soon as possible (typically at an immediate postdominator point) [29, 60, 64].

Figure 2.3 illustrates a simple example of divergent code and its corresponding control flow graph (CFG). The bit mask in each basic block of the CFG denotes which threads of a single warp containing four threads will execute that block. The rightmost bit represents thread with thread ID=0. All threads execute basic block A. Upon executing divergent branch BR\textsuperscript{A}_{B\rightarrow C}, warp A\textsuperscript{1111} diverges into two warp splits [91] B\textsuperscript{0101} and C\textsuperscript{1010}. In our notation, branches are abbreviated as BR with a superscript representing the basic block containing the branch and a subscript representing the successor basic blocks. Each warp split is represented by a letter representing the basic block that the split is executing with a subscript indicating the active threads.

The immediate postdominator (IPDOM) of the branch BR\textsuperscript{A}_{B\rightarrow C}, basic block D, is the earliest point where all threads diverging at the branch are guaranteed to execute. We say an execution mechanism supports IPDOM reconvergence if it guarantees all threads in the warp that are active at any given branch are again active (executing in lockstep) when the immediate postdominator of that branch is next encountered. IPDOM reconvergence is favorable because the immediate postdominator is the closest point at which all threads in a warp are guaranteed to reconverge\textsuperscript{2}. A mechanism for supporting IPDOM reconvergence using a stack of

\textsuperscript{2}Likely convergence [44] and thread frontiers [33] identify earlier reconvergence points that can occur dynamically in unstructured control flow if a subset of paths between branch and IPDOM are
active masks [77] was introduced by Fung et al. [41]. However, there are different possible implementations that can support IPDOM reconvergence as defined above. On current hardware the SIMT model is implemented via predication for simple branches, or in the general case using hardware and/or software managed stack-based masking of execution units [5, 18, 29, 64, 77].

In these stack-based execution models, the divergent paths are serialized. Thus, in this example, warp split C_{1010} may execute first until executing threads reach basic block D. Then, execution switches to warp split B_{0101}. Once the latter threads reach basic block D as well, the four threads reconverge and execute basic block D in lockstep.

### 2.3.2 Threads from Different Warps

Each cycle, one or more warp schedulers select one of their assigned active warps to be issued for execution. Typically, a scheduling heuristic needs to be used with the objective of efficiently hiding long latency operations and to harness the existing data locality [99, 124]. A simple scheduling policy would be to round robin across the available warps, if the next warp in the round robin order is not pending on data dependency or synchronization dependency (e.g., barrier) and the hardware required to execute its next instruction is available, the warp is issued for execution. This policy is referred to as Loose Round Robin (LRR). LRR guarantees fairness in scheduling different warps. However, this proves to be inefficient to hide long latency operations, as it encourages warps to progress at similar rates reaching high latency code portions at the same time which limits their ability to hide each other’s latencies. Further, LRR has negative impact on intra-warp temporal locality, as it allows other warps to evict the data brought to cache by a warp before the warp is rescheduled [124]. Greedy then Oldest (GTO) is another warp scheduling policy that typically outperform LRR. A greedy-then-oldest scheduler consistently selects the same warp for scheduling until it stalls then it moves the oldest ready warp. Older warps are those who are assigned earlier to the hardware resources. In case different warps were assigned at the same cycle (e.g., in the same thread block), warps with the smallest threads IDs are considered older. Compared to

---

executed by a warp.
LRR, GTO has the advantage of maintaining intra-warp locality as well as getting warps to progress at a different rate which typically allows for better latency hiding. There are, however, numerous research papers on different warp scheduling policies that uses different heuristics based on different optimization goals (e.g., improving latency hiding [99], improving locality [124], reducing barrier synchronization overheads [83], reducing load imbalance overhead across warps from the same CTA [72]).
Chapter 3

SIMT-Induced Deadlocks: Definition and Detection

This chapter studies the impact of current single-instruction multiple-thread (SIMT) implementations on programmability. We show that the scheduling constraints imposed by current SIMT implementations lead to surprising (from a programmer perspective) deadlocks when executing code that contains inter-thread synchronization. This type of deadlocks is unique to SIMT architectures and does not exist in traditional multi-threaded architectures. We refer to the cases where the forward progress of diverged threads is prevented due to the implicit scheduling constraints imposed by the SIMT implementation as SIMT-induced deadlocks or briefly as SIMT deadlocks. In this chapter, we precisely define the inter-thread synchronization patterns that leads to SIMT-induced deadlocks. We show that SIMT-induced deadlock can be produced by (1) programmers and (2) compilers that are oblivious to the subtle details of SIMT implementations.

In this chapter, we also discuss the implications of SIMT-deadlocks on SIMT architectures regarding both the ease of programmability and the acceptance to different programming models. We present an algorithm that conservatively detects potential SIMT-induced deadlocks in parallel kernels. We evaluate an implementation of the algorithm that relies solely on static analysis using a large number of CUDA and OpenCL kernels. The results show that the static detection successfully flags true SIMT-deadlocks with a false detection rate of 4%–5%. We discuss the
limitations of our current SIMT-deadlock detection implementation. Finally, we summarize related work and conclude with some pointers to future work.

3.1 SIMT Scheduling Constraints

Conceptually, in a stack-based SIMT implementation, a per-warp stack, as shown in Figure 3.1 is used to manage divergent control flow. Each entry contains three fields that represent a group of scalar threads executing in lock-step: (1) a program
counter (PC) which stores the address of the next instruction to execute, (2) a reconvergence program counter (RPC) which stores the instruction address at which these threads should reconverge with other threads from the same warp and (3) an active mask that indicates which threads have diverged to this path. Initially, the stack has a single entry. Once a divergent branch is encountered, the PC field of the divergent entry is replaced with the RPC of the encountered branch and the branch outcomes’ entries are pushed into the stack. Only threads at the top of the stack entry are eligible for scheduling. Once executing threads reach reconvergence, their corresponding entry is popped out of the stack. As noted earlier, in some implementations the stack is implemented and/or manipulated in software [5, 18, 29, 77].

Figure 3.1 records the change in the reconvergence stack after executing a divergent branch in two different cases. In Figure 3.1a, an if-else branch is executed at the end of basicblock A (BB_A) 1. The PC of the first stack entry changes to the RPC of the branch (i.e., BB_D), and two new entries are pushed into the stack representing the branch taken and not-taken paths 2. Only the top of the stack (TOS) entry is eligible for scheduling. Therefore, this warp starts executing threads that diverged to BB_C 3. After these threads reach the reconvergence point (i.e., BB_D), their entry is popped out of the stack and the execution of BB_B starts. Eventually, all threads reconverge at BB_D 4. This sequence allows the stack to track multiple reconvergence points in case of more complex CFGs that contain nested branches.

In Figure 3.1b, a loop branch is executed at the end of BB_B 1. One thread exits the loop while others continue iterating. Similar to the first example, the PC of the first stack entry changes to the RPC of the branch (i.e., BB_C). Only a single entry is pushed into the stack representing threads that diverged to the loop header 2. The warp keeps iterating through the loop until all threads eventually exit the loop and start executing BB_C. This mechanism enables to reconverge diverged threads improving SIMD units utilization. However, to achieve this, it imposes thread scheduling constraints that we discuss next.

3.1.1 SIMT-Induced Deadlocks

Conventionally, in simultaneous multi-threaded MIMD environment, programmers do not worry about specifics of thread schedulers to write functionally correct code.
It is assumed that the hardware and/or the operating system guarantees “loose” fairness in thread scheduling [47]. SIMT programming models attempt to provide similar guarantees [111]. However, recent SIMT implementations fail to do so. In current SIMT implementations, thread scheduling is constrained by the CFG of the executed application such that: if a warp $W$ encounters a branch $BR_{T,NT\rightarrow R}$ with two possible successor basic blocks $T$ and $NT$ and reconvergence point $R$, it may diverge into two splits [91]: $WP_{T\rightarrow R}$ and $WP_{NT\rightarrow R}$. $WP_{T\rightarrow R}$ contains threads that diverge to the taken path and $WP_{NT\rightarrow R}$ contains threads that diverge to the not-taken path. On current SIMT implementations execution respects the following:

**Constraint-1:** Serialization. If $WP_{T\rightarrow R}$ executes first then $WP_{NT\rightarrow R}$ blocks until $WP_{T\rightarrow R}$ reaches $R$ (or vice versa).

**Constraint-2:** Forced Reconvergence. When $WP_{T\rightarrow R}$ reaches $R$, it blocks until $WP_{NT\rightarrow R}$ reaches $R$ (or vice versa).

Collectively we refer to these two constraints as the stack-based SIMT reconvergence scheduling constraints. A SIMT-induced deadlock occurs when a thread is indefinitely blocked due to a cyclic dependency between either of these constraints and a synchronization operation in the program.

### 3.1.2 Causes of SIMT Deadlocks

We categorize SIMT-induced deadlocks into two types according to their cause:

**Barrier Induced:** Figure 3.2 illustrates code that uses barriers to synchronize communication between threads. Following MIMD execution semantics, the programmer’s intention is that every thread should reach either the first or second barrier before any thread continues. Placing barriers in such diverged code is used to implement producer-consumer communication using named barriers and the `.sync` and `.arrive` directives as suggested by Nvidia’s *PTX ISA Manual* [105]. Recent work has proposed employing such code to enable warp specialization [12][13]. However, for this code to run correctly on current SIMT implementations, the condition on line 2 in Figure 3.2 has to evaluate identically through all threads within the same warp every time this line is executed and for all warps executing the code. Otherwise this code will either lead to deadlock (if barrier arrival is counted
per scalar thread) or hard to predict and/or implementation dependent behavior (if barrier arrival is counted per warp \([147]\)). Prior work studied detection of barrier divergence \([17, 79, 131]\). Therefore, in this chapter, we focus more on the second type of SIMT-induced deadlocks.

**Conditional Loop Induced:** Figure 3.3 shows a typical “MIMD implementation” of a spin lock guarding a critical section. To simplify the example all threads attempt to acquire the same lock \((mutex)\). The issue illustrated by this example also affects fine-grained locks when more than one thread contends for the same lock. To acquire the lock, each thread repeatedly executes an atomic compare and swap. The loop condition evaluates to false for the single thread that successfully acquires the lock. We call this the leading thread. The remaining threads fail to acquire the lock. We call these lagging threads. The lagging threads continue to iterate through the loop waiting for the leading thread to release the lock by executing \(atomicExch(...)\). However, in current SIMT implementations the leading thread never reaches \(atomicExch(...)\) as it is blocked by Constraint 2 - the leading thread is waiting at the reconvergence point for the lagging threads. On the other hand, the lagging threads cannot make forward progress because the leading thread owns the lock. This issue is known among GPU application developers \([43, 47, 114, 115, 121, 151]\).

### 3.2 SIMT Deadlocks Impact on Programmability

In SIMT architectures, the hardware with minimal help from the compiler supports arbitrary memory accesses and control flow divergence. This enables abstracting
away the complexity of the underlying SIMD hardware allowing for much simpler programming models. The CUDA Programming Guide [111], for example, states that “For the purposes of correctness, the programmer can essentially ignore the SIMT behavior, ...” and suggests that SIMT behavior is primarily relevant for performance tuning purposes.

However, with current SIMT implementations, dealing with SIMT deadlocks is still a problem. In the presence of inter-thread synchronization, it is still necessary for programmers and/or compilers to be aware of the interaction between inter-thread communication, the control flow graph (CFG) of the parallel kernel and the SIMT divergence implementation to avoid SIMT deadlocks.

To the best of our knowledge, prior to our publication [38], there has not been any general algorithms or approaches to either detect or eliminate SIMT deadlocks. This affects both the ease of programmability of SIMT architectures using SIMT specific languages (e.g., CUDA) as well as a robust support of SIMT architectures to new programming models with higher levels of abstractions (e.g., OpenMP).

### 3.2.1 Ease of Programmability

With the presence of SIMT deadlocks, current SIMT implementations lack reliable support for fine-grained inter-thread synchronization that is essential for efficient implementations of many irregular applications. The inconvenience caused by SIMT-induced deadlocks is evident by the larger number of programming discussion boards on the issue. It also shows that the challenge exists for both general purpose programmers (e.g., in CUDA and OpenCL [114, 115]) and graphics programmers (e.g., in GLSL and HLSL [112, 113, 139]).

Therefore, it is currently up to programmers to work around the SIMT hard-
Figure 3.4: HashTable Insertion - Version 1 [129].

ware scheduling constraints when implementing algorithms that involve inter-thread synchronization. For example, Figure 3.4 shows a GPU implementation for hashtable insertion proposed in Nvidia’s “CUDA by Example” [129]. We highlight in bold the lines of code of interest. The example uses a spin lock implementation (lines 1-11 in Figure 3.4) similar to the one described in Figure 3.3. However, when the lock function is called, it is guaranteed that only one thread from each warp is executing it at a time which eliminates SIMT deadlocks possibilities.

However, the serialization of threads within the same warp significantly under-
utilizes the SIMD hardware regardless of the granularity of the synchronization. For the simple case as in Figure 3.3, there is another well-known workaround on Nvidia GPUs shown in Figure 3.6 [121] which avoids underutilizing the SIMD hardware. In this modified code, the while loop body includes both the lock acquisition and release. Hence, the reconvergence point of the while loop does not prevent the required communication between threads. Threads that fail to acquire the lock in the if statement condition wait at the end of the if statement to recon-
verge with threads that acquired the lock. Threads that acquired the lock proceed to execute the critical section and then release the lock. All threads check if they are eligible for another iteration of the while loop, any thread that acquired the lock will have its local variable `done` set to true and thus exits the while loop. Other threads proceed into the loop body. Eventually, all threads within a warp reconverge at the exit of the while loop and continue execution.

We apply this workaround on the hashtable insertion code in Figure 3.5 to compare between both workarounds in performance. Figure 3.7 shows the execution time of the `add_to_table` kernel on different GPU generations with 1024 entries in the hashtable and 2M random insertions. Except from Fermi’s TeslaC2050, all recent GPU devices perform better with version-2 with a speedup of 56% on the Pascal GTX-1080. On Fermi, version-1 performs better as it reduces contention on the atomic operations which used to perform poorly on earlier architectures. However, even the second workaround is not reliable for the following reasons:

1. Lack of general solution: Both workarounds in Figures 3.4 and 3.5 handle a simple case where there is a single lock acquire statement with a single lock release statement that postdominates the lock acquire. Thus, the required code transformation is relatively simple to reason about using high level semantics given that the programmer is aware of the details of reconvergence mechanism. However, both in principle and in practice, more complex synchronization patterns (could) exist. Figure 3.8 shows a couple of such examples where the code could have mul-

---

**Figure 3.7:** HashTable Insertion Versions Comparison (2M Random Insertion in 1024 HashTable Entries). Recent GPUs are to the right.
Figure 3.8: Synchronization patterns used in multi-threaded applications. Examples are from the graph analytics API in the Cloud suite [40].

Multiple interleaved locks and/or multiple lock releases that may not postdominate the lock acquire statement(s). With such patterns, it is difficult even for programmers who are aware of the hardware details to reason about a safe code transformation. Thus, there is a need for a general algorithm that enables to both detect and eliminate SIMT deadlocks regardless of the synchronization pattern in place.

2. Unstandardized SIMT Behavior: Different GPU vendors have their own implementation of the SIMT execution model. As a consequence, the order in which divergent paths are executed and the specific locations of reconvergence points is not standardized and often undocumented. Given that these nuances may impact programs’ correctness, programmers need to be aware of the exact implementation on the targeted hardware. Such solutions, when they exist, lack forward compatibility as well as portability. Therefore, a pure compiler and/or hardware solution is preferable.

3. Vulnerability to Compiler Optimizations: Manual workarounds as in Figure 3.5 assume that the compiler maintains the same control flow described by the high level language. However, optimizations that either modify the CFG or move statements across basicblocks can conflict with the intended CFG. We found that this problem exists in default optimization passes in both open source compilers (e.g., LLVM) and propriety compilers (e.g., Nvidia’s NVCC).

For example, the code in Figure 3.5 works fine if compiler optimizations are
disabled. However, if default compiler optimizations are enabled it depends on the compiler and/or the version. Figure 3.9 shows the PTX assembly resulting from compiling the hashtable insertion code in Figure 3.5 with two versions of Nvidia’s NVCC compiler. NVCC versions 6.5 and lower maintain the CFG specified by the programmer for this example. However, we found that starting from NVCC 7.0, the CFG is altered in a way that induces SIMT deadlocks and on existing hardware. More of these examples are available in the author’s github repository [37].

LLVM showed similar behaviour to NVCC 8.0 (we have used LLVM 3.6). In LLVM, we identified two control flow optimizations that induces SIMT deadlocks even when the original code was SIMT deadlock free; namely jump-threading and simplifycfg. Both optimizations attempt to simplify control flow graphs by removing unnecessary branches [86].

This observation motivates the need for compiler writers for SIMT backends to find methodologies to restrict compiler optimizations that induces SIMT deadlocks or at least to avoid their SIMT-incompatible side effects. Without the work presented in this thesis, we believe the most likely fix for the compiler would be to disable all optimizations that could potentially cause SIMT deadlocks in the compilation of any SIMT kernel. This would include all optimizations that alter the CFG or that move instructions across basic blocks. A more robust methodology would be to detect whether a certain application is prone to SIMT deadlocks given a certain transformation. This work enables this methodology using variations of Algorithm 1.

3.2.2 Programming Language Abstraction

SIMT deadlocks limit the level of abstraction that can be supported by SIMT architectures. For example, current multi-threaded MIMD architectures support a wide range of general purpose programming languages (e.g., MPI, Java, OpenMP, pThreads, OpenCL, OpenACC, TBB, etc). Each of these languages meet certain criteria such as development time, simplicity, efficiency, and readability for the applications of interest. For the expanding adoption of SIMT architectures to continue, SIMT architectures need to be flexible enough to robustly support wider

\footnote{We have tested these codes on a variety of Nvidia’s GPUs; TeslaC2090 (Fermi), Tesla K20C (Kepler), and GeForce GTX 1080 (Pascal)}
range of programming models. Below, we provide a case study where SIMT deadlock is an obstacle to support of higher levels of abstraction.

### 3.2.3 OpenMP Case Study: Synchronization Primitives Library

The OpenMP 4.0 standard and its successors support the offloading of a parallel region to an accelerator (e.g., a GPU). The OpenMP programming model is appealing because of both its abstraction and portability across architectures. Thus, it helps accelerators reach to a broader set of developers [66]. Currently, there is a coordinated effort by many technology companies such as IBM, Intel, AMD, Nvidia and others to add an implementation of OpenMP 4.0 support for accelerators in Clang and LLVM [15, 93, 117].

To support a synchronization API in OpenMP such as `omp_set_lock(...)`, the front-end replaces the API call with a spin lock code with back-off delay as shown in Figure 3.10. Given the structure of this code, a valid OpenMP program that

---

**Figure 3.9:** PTX output of compiling the code in Figure 3.5 using NVCC 6.5 and NVCC 8.0 with default optimizations enabled.
Figure 3.10: OpenMP Clang 3.8 Frontend Implementation for OpenMP 4.0 Runtime Library Synchronization Calls `omp_set_lock` and `omp_unset_lock`.

executes properly on CPUs may not terminate on GPUs due to SIMT deadlocks.

Consider the example in Figure 3.11, the compiler front-end replaces the the `omp_set_lock(...)` call at line 8 with the code in Figure 3.10. Successful threads in acquiring the lock wait at the exit of the while loop at line 22 in Figure 3.10 to reconverge with lagging ones. Thus, they do not proceed execution to the `omp_unset_lock(...)` statement at line 11 in Figure 3.11. Therefore, if other threads are waiting on locks held by these leading blocked threads, a deadlock occurs.

Note that the current OpenMP clang front-end correctly handles “`#omp critical`”, which is used to enclose global critical code sections that should be executed by one thread at a time. When a critical section surrounded by `#omp critical` is found, an extra loop around the section is added to serialize the execution of threads within the same warp in addition to an `omp_set_lock(...)` with a global lock that serializes execution of different warps (i.e., similar to Figure 3.4 but using a global lock to
force different warps to serialize their execution as well). However, this solution does not generalize because it requires global locking and also requires the lock release to happen in a single known location that postdominates the lock acquisition (as the case with the “#omp critical”). This is not always true when using `omp_(un)set_lock(..).

This obstacle is not unique to OpenMP support. In fact, even SIMT-specific languages such as CUDA could make use of an optimized synchronization primitive library for both performance and productivity. For example, recent work [78] has proposed promising efficient fine-grained synchronization primitives on GPUs that can boost performance up to 4× compared to current spin lock implementation. However, a main limitation that work describes is SIMT deadlock which rules out library based implementations for these primitives. These challenges encourage coming up with a robust technique to eliminate SIMT deadlocks without intervention from programmers.

### 3.3 SIMT Deadlock Detection

This section proposes an algorithm to detect the presence of SIMT deadlocks in a parallel kernel. It also presents the limitations of a static implementation of such algorithm.
Listing 1 Definitions and Prerequisites for Algorithms 1, 2, 4

- \( BB(I) \): basic block which contains instruction (i.e., \( I \in BB(I) \)).
- \( P_{BB1 \rightarrow BB2} \): union set of basic blocks in execution paths that connect \( BB1 \) to \( BB2 \).
- \( IPDom(I) \): immediate postdominator of an instruction \( I \). For non-branch instructions, \( IPDom(I) \) is the instruction that immediately follows \( I \). For branch instructions, \( IPDom(I) \) is defined as the immediate common postdominator for the basic blocks at the branch targets \( BB_T \) and \( BB_{NT} \) (i.e., the common postdominator that strictly postdominate both \( BB_T \) and \( BB_{NT} \) and does not postdominate any of their other common postdominators.).
- \( IPDom(arg1,arg2) \) is the immediate common postdominator for \( arg1 \) and \( arg2 \); \( arg1 \) and \( arg2 \) could be either basic blocks or instructions.
- \( LSet \): the set of loops in the kernel, where \( \forall L \in LSet \).
- \( BBs(L) \): the set of basic blocks within loop \( L \) body.
- \( ExitConds(L) \): the set of branch Instructions at loop \( L \) exits.
- \( Exits(L) \): the set of basic blocks outside the loop that are immediate successors to a basic block in the loop.
- \( Latch(L) \): loop \( L \) backward edge, \( Latch(L).src \) and \( Latch(L).dst \) are the edge source and destination basic blocks respectively.
- Basic block \( BB \) is reachable from loop \( L \), iff there is a non-null path(s) connecting the reconvergence point of \( Exits(L) \) with basic block \( BB \) without going through a barrier. \( ReachBrSet(L,BB) \) is a union set of conditional branch instructions in all execution paths that connects the reconvergence point of \( Exits(L) \) with \( BB \).
- Basic block \( BB \) is parallel to loop \( L \), iff there is one or more conditional branch instructions where \( BBs(L) \subset P_{T \rightarrow R} \) and \( BB \in P_{NT \rightarrow R} \) or vice versa, where \( T \) is the first basic block at the taken path of the branch, \( NT \) is the first basic block at the not-taken path, and \( R \) is the reconvergence point of the branch instruction. \( ParaBrSet(L,BB) \) is a union set that includes all branch instructions that satisfy this condition.

3.3.1 SIMT Deadlock Detection Algorithm

This section proposes a compiler analysis algorithm that conservatively detects SIMT-induced deadlocks due to conditional loops. In the remainder of this section, we consider the case of a single kernel function \( K \) with no function calls (e.g., either natively or through inlining) and with a single exit. The single exit assumption is required to guarantee the convergence of the safe reconvergence identification algorithm (Chapter 4) and not by the detection algorithm itself. It is achieved by merging return statement(s) into a single return (e.g., using -mergereturn pass in LLVM).

To focus on SIMT deadlocks due to conditional loops, we assume that \( K \) is guaranteed to terminate (i.e., is a deadlock and livelock free) if executed on a MIMD machine. This means that any loosely fair thread scheduler that does not indefinitely block a thread (or a group of threads) from execution always leads to
Algorithm 1 SIMT-Induced Deadlock Detection

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>Inputs: LSet.</td>
</tr>
<tr>
<td>2:</td>
<td>Outputs: RedefWrites(L) for each L ∈ LSet and if L induces a potential SIMT deadlock.</td>
</tr>
<tr>
<td>3:</td>
<td>for each loop L ∈ LSet do</td>
</tr>
<tr>
<td>4:</td>
<td>ShrdReads(L) = ∅, ShrdWrites(L) = ∅, RedefWrites(L) = ∅</td>
</tr>
<tr>
<td>5:</td>
<td>for each instruction I, where BB(I) ∈ BBs(L) do</td>
</tr>
<tr>
<td>6:</td>
<td>if I is a shared memory read ∧ ExitConds(L) depends on I then</td>
</tr>
<tr>
<td>7:</td>
<td>ShrdReads = ShrdReads ∪ I</td>
</tr>
<tr>
<td>8:</td>
<td>end if</td>
</tr>
<tr>
<td>9:</td>
<td>end for</td>
</tr>
<tr>
<td>10:</td>
<td>for each instruction I do</td>
</tr>
<tr>
<td>11:</td>
<td>if BB(I) is parallel to or reachable from L then</td>
</tr>
<tr>
<td>12:</td>
<td>if I is a shared memory write then</td>
</tr>
<tr>
<td>13:</td>
<td>ShrdWrites(L) = ShrdWrites(L) ∪ I</td>
</tr>
<tr>
<td>14:</td>
<td>end if</td>
</tr>
<tr>
<td>15:</td>
<td>end if</td>
</tr>
<tr>
<td>16:</td>
<td>end for</td>
</tr>
<tr>
<td>17:</td>
<td>for each pair (IR, IW), where IR ∈ ShrdReads(L) and IW ∈ ShrdWrites(L) do</td>
</tr>
<tr>
<td>18:</td>
<td>if IW does/may alias with IR then</td>
</tr>
<tr>
<td>19:</td>
<td>RedefWrites(L) = RedefWrites(L) ∪ IW</td>
</tr>
<tr>
<td>20:</td>
<td>end if</td>
</tr>
<tr>
<td>21:</td>
<td>end for</td>
</tr>
<tr>
<td>22:</td>
<td>if RedefWrites(L) ≠ ∅ then Label L as a potential SIMT-induced deadlock.</td>
</tr>
<tr>
<td>23:</td>
<td>end if</td>
</tr>
<tr>
<td>24:</td>
<td>end for</td>
</tr>
</tbody>
</table>

K termination.

We also assume that K is barrier divergence-free [17] (i.e., for all barriers within the kernel, if a barrier is encountered by a warp, the execution predicate evaluates to true across all threads within this warp). This assumption excludes the possibility of SIMT deadlocks due to placing barriers in divergent code assuming thread level barrier arrival detection. Such barrier divergence detection techniques and the issues arising from placing barriers in divergent code have been extensively studied in prior work [17, 79, 131]. For brevity, we refer to all memory spaces capable of holding synchronization variables as shared memory (i.e., including both global and shared memory using CUDA terminology). Listing 1 summarizes definitions used in Algorithms 1, 2, 4. We discuss these definitions as we explain each algorithm.

Under the barrier divergence and the MIMD deadlock and livelock freedom assumptions, the only way to prevent forward progress of a thread is a loop whose
exit condition depends upon a synchronization variable. Forward progress is prevented (i.e., SIMT deadlock occurs) if a thread enters a loop for which the exit condition depends on the value of a shared memory location and that location will only be set by another thread of the same warp but that thread is blocked due to Constraint 1 or 2.

To determine whether there is a dependency we consider the static backward slice of the loop exit condition considering both data and control dependencies. Algorithm 1 detects potential SIMT-induced deadlocks.

It is applied for each loop in a given kernel. If the loop exit conditions do not depend on a shared memory read operation that occurs inside the loop body then the loop cannot have a SIMT-induced deadlock. If a loop exit condition does depend on a shared memory read instruction \( I_{R} \), we add \( I_{R} \) in the set of shared reads \( \text{ShrdReads} \) on lines 4-7. A potential SIMT-induced deadlock exists if any of these shared memory reads can be redefined by divergent threads from the same warp. The next steps of the algorithm detect these shared memory redefinitions.

Lines 8-14 record, in set \( \text{ShrdW} \text{reads} \), all shared memory write instructions \( I_{W} \) located in basicblocks that cannot be executed, due to the reconvergence scheduling constraints, by a thread in a given warp so long as some of the threads within that warp are executing in the loop. These basic blocks fall into two categories (Listing 1): The first category we call reachable. In a structured CFG, the reachable blocks are those blocks that a thread can arrive at following a control flow path starting at the loop exit. The second category we call parallel. The parallel blocks are those that can be reached by a path that starts from a block that dominates the loop header but avoids entering the loop. The detection algorithm requires that reconvergence points can be precisely determined at compile time. In our implementation, reconvergence is at immediate postdominators. We limit reachable basicblocks to those before a barrier due to our assumption of barrier-divergence freedom.

Lines 15-20 check each pair of a shared memory read from \( \text{ShrdReads} \) and write from \( \text{ShrdW} \text{rites} \) for aliasing. If they do or “may” alias, then the write instruction might affect the read value and hence affect the exit condition. If such a case occurs, the loop is labeled as potential SIMT-induced deadlock and we add the write to the redefining writes set \( \text{RedefW} \text{rites} \).

For example, consider the application of Algorithm 1 to the code in Figure 3.3.
1. `uint count;`
2. `do{
3.    count = `WarpHist`[data] & 0x07FFFFFFU;
4.    count = uniqueThreadTag | (count + 1);
5.    `WarpHist`[data] = count;
6. }while(`WarpHist`[data] != count);

**Figure 3.12:** addByte function in Histogram256 [120].

and 3.6 In Figure 3.3, the loop exit is data dependent on the `atomicCAS` instruction. There is one shared memory write that is reachable from the loop exit, the `atomicExch` instruction. The two instructions alias. Hence, SIMT-induced deadlock is detected. In Figure 3.6, although the loop exit is control dependent on the `atomicCAS` instruction, there are no shared memory write instructions that are parallel to or reachable from the loop exit. Therefore, no SIMT deadlock is detected.

### 3.3.2 SIMT Deadlock Detection Limitations

Our current static implementation of the detection algorithm, Algorithm 1, has some limitations that we highlight next.

**Inter-procedural Dependencies:** The algorithm as stated assumes a single kernel function with no function calls to other procedures. Consequently, it can only detect SIMT deadlocks when synchronization is local to a function. It could be possible to extend our detection algorithm to handle function calls with an inter-procedural analysis that tracks dependencies across functions [23, 59]. We speculate this may lead to more conservative aliasing decisions and thus a higher false detection rate.

**Conservative Reachability Analysis:** Our implementation of Algorithm 1 solely relies on the static CFG layout to identify reachable and parallel basicblocks. However, in reality, not all static paths are executed at runtime.

**Conservative Alias Analysis:** We rely upon static alias analysis to identify redefining writes. Thus, it is possible to flag a loop causing SIMT deadlock where a programmer may reason such deadlock cannot occur. Further, relying on alias analysis leads to conservative dependency analysis when pointer chasing is involved in the dependency chain.
**Indirect Branches:** The static analysis can be extended to deal with indirect branches with known potential targets (as supported in Nvidia’s PTX 2.1). However, without clues about potential targets, the analysis would be conservative in labeling potential SIMT-induced deadlocks including indirect branches as they might form loops. This leads to significant overheads due to excessive potential false detections. This limitation can be seen as a special case of the conservative reachability analysis limitation.

**Warp-Synchronous Behaviour Legacy Codes:** With the lack of shared memory atomics in early GPU generations, some GPU applications were written relying on implicit warp synchronization\(^2\) to implement atomics. One example is Histogram256 from OpenCL SDK [120]. Figure 3.12 shows a code snippet for the main body of the core function in Histogram256: `addByte`. The `l_WarpHist` is a pointer to current warp sub-histogram. The `data` is the value being added to the histogram. The outcome of this piece of code should be that each thread increments the histogram value `l_WarpHist[a]`, where `data` might be the same or different across different threads. Hence, it is equivalent to atomic increment operation.

Let’s consider the case where two or more threads of the same warp receive the same `data` value. Each thread will increment the old `l_WarpHist[a]` and store the outcome into a private variable `count`, which is then tagged with a unique value for each thread. Finally, all threads attempt to store the `count` value in the same memory location. This race condition is handled by the hardware through write combining that effectively results in rejection of all but one colliding store [120]. The `while` loop exit condition guarantees that only the winning thread exits the loop, while others try again.

In the main histogram kernel, the `addByte` function is called four times to implement `addWord`. This creates write-write race between the four different occurrences of the store in line 5. Therefore, in a MIMD execution model, this code is broken and needs explicit barriers after loop bodies to avoid the write-after-write races between divergent threads executing different occurrences of the `addByte`. However, this code works fine on current GPUs, as it relies on the existing implementation of SIMT execution model. In particular, it assumes there is an implicit

\(^2\)These application do not follow the CUDA programming model and are dependent on a specific hardware platform. Hence, they lack forward compatibility.
Table 3.1: Evaluated Kernels

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HT [43]</td>
<td>CUDA</td>
<td>Chained Hash Table of 80K entries and 40K threads</td>
</tr>
<tr>
<td>ATM [43]</td>
<td>CUDA</td>
<td>ATM 122K transactions, 1M accounts, 24K threads</td>
</tr>
<tr>
<td>CP-DS [20,43]</td>
<td>OpenCL</td>
<td>Distance Solver in Cloth Physics simulation</td>
</tr>
<tr>
<td>BH-TB [24]</td>
<td>CUDA</td>
<td>Tree Building in BarnesHut (30,000 bodies)</td>
</tr>
<tr>
<td>BH-SM [24]</td>
<td>CUDA</td>
<td>Summarization kernel in BarnesHut</td>
</tr>
<tr>
<td>BH-ST [24]</td>
<td>CUDA</td>
<td>Sort kernel in BarnesHut</td>
</tr>
<tr>
<td>BH-FC [24]</td>
<td>CUDA</td>
<td>Force Calculation in BarnesHut</td>
</tr>
<tr>
<td>Rodinia 1.0 [26]</td>
<td>CUDA</td>
<td>Regular and Irregular Kernels with no fine-grained synchronization</td>
</tr>
<tr>
<td>OpenCL SDK 4.2 [106]</td>
<td>OpenCL</td>
<td>Regular Kernels with no fine-grained synchronization</td>
</tr>
</tbody>
</table>

Barrier outside the loop body (due to reconvergence) that synchronizes threads of the same warp.

In the absence of explicit barriers, our detection Algorithm 1 infers the writes to the shared memory array in latter loops as redefining writes that may change the loop exit condition of earlier loops and thus flags the loops as potential SIMT deadlocks. False detections in such situation are particularly harmful because they may trigger code transformations (described in Chapter 4) that would remove the implicit synchronization that was essential for correct functionality.

However, such codes are discouraged by Nvidia as they use undocumented and non-standardized features of the hardware. CUDA 8.0, the most recent version at the time of the thesis writing, has introduced the ability to perform warp level barriers using \texttt{sync(this,warp)} as a fast and safe way to replace the use of implicit warp synchronization \texttt{[50].}

3.4 Methodology

This section describes the implementation, methodology and evaluation of our detection algorithm. We implemented the detection algorithm as an analysis pass in LLVM 3.6 \texttt{[86]}. For alias analysis, we used the basic alias analysis pass in LLVM \texttt{[87]}.\footnote{We did not observe lower false detection rates using other LLVM alias analysis passes as they focus on optimizing inter-procedural alias analysis.} We ran passes that inline function calls and lower generic memory spaces into non-generic memory spaces before our detection pass. Note that this helps to limit the scope of detection to loops whose exist depend on shared...
or global memory and avoids unnecessarily including local memory. We used the approach described in [134] to identify irreducible loops.

We use CUDA and OpenCL applications for evaluation. OpenCL applications are compiled to LLVM Intermediate Representation (LLVM-IR) using Clang compiler’s OpenCL frontend [84] with the help of libclc library that provides LLVM-IR compatible implementation for OpenCL intrinsics [85]. For CUDA applications, we use nvcc-llvm-ir tool [35]. This tool allows us to retrieve LLVM-IR from CUDA kernels by instrumenting some of the interfaces of libNVVM library [102].

### Table 3.2: Detection Pass Results on CUDA and OpenCL Kernels

<table>
<thead>
<tr>
<th>Opt. Level</th>
<th>Kernels</th>
<th>N. Brs</th>
<th>N. Lps</th>
<th>Detections</th>
<th>T. Det.</th>
<th>F. Det.</th>
<th>F. Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>159</td>
<td>2751</td>
<td>277</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>5.05%</td>
</tr>
<tr>
<td>O2</td>
<td></td>
<td>1832</td>
<td>242</td>
<td>14</td>
<td>4</td>
<td>10</td>
<td>4.13%</td>
</tr>
</tbody>
</table>

3.5 Evaluation

First, we ran our detection on the CUDA and OpenCL applications mentioned in Section 3.4. These applications were written to run on current GPUs, so they are supposed to be free of SIMT-induced deadlocks. Table 3.2 summarizes the detection results from 159 kernels. With default compiler optimizations in LLVM 3.6 enabled, four true detections in four different kernels (ATM, HT, CP-DS and BH-ST) were detected (See Vulnerability to Compilers’ Optimizations in Section 2.2) [4]. These true detections were confirmed by both manual inspection of the machine code and by verifying that the code does not terminate when executed. False detections include four detections that occur in the histogram256 kernel from the OpenCL SDK and one false detection in the bucketsort kernel in MGST from Rodinia. In both cases, kernels are dependent on warp synchronous behaviour to implement an atomic increment operation. Other false detections exist in applications that involve inter-thread synchronization [24, 43]. No false negatives were observed (specifically, after using the detection results to apply static SIMT-induced deadlock elimination algorithm described in Chapter 3 no deadlocks were observed at runtime). As an additional check, we ran the detection pass on the

---

4Using Nvidia’s NVCC compiler, only Cloth Physics deadlocks after turning on compiler optimizations.
transformed kernels after applying the elimination pass which yielded no detections.

Let’s consider the O0 compilation to show the effect of the different filtration stages in the detection algorithm. Out of the 277 loops in O0 compilation, there are only 103 that are detected to be dependent on shared (or global) memory. Only 65 of those are dependent on a shared memory read that happens within the loop body. Only 14 of these are detected to be potentially redefined. Besides alias analysis, the main reasons for false detection include conservative reachability and dependence analysis that relies on the static CFG layout and instruction dependencies without considering the dynamic behaviour. For example, our analysis conservatively consider shared memory writes that are in basic blocks that are ancestors to a loop body whose exist dependent on shared memory read as potential redefining writes to the loop exit if these writes are reachable through an outer loop. Although, typically, a release or signal statement follow the lock or wait loop.

For example, one of BH-FC loops is control dependent on a shared array. The array is written to a basic block that is an ancestor the loop of interest but it is still reachable from the loop exit through an outer loop. Our analysis flags the load/store pair as potential reason of a SIMT induced deadlock. This particular example is also interesting because the code includes a more explicit hint to exclude SIMT deadlock possibility. In particular, the conditional check on this shared array uses the all() CUDA intrinsic that evaluates the same for all threads within a warp forcing all threads within the same warp to exit at the same iteration preventing divergence at runtime. If our algorithm incorporated such information, it could have eliminated this SIMT deadlock possibility because it will be able to infer that the loop exit in this case is dependent only on the code within the loop and not the code following the loop. This motivates leveraging runtime information and elaborate static analysis as a future work.

### 3.6 Related Work

The presence of SIMT deadlocks has been highlighted initially in different developers forums [114, 115]. The first research work to pay attention to this problem
and attempt to find a solution is Ramamurthy’s MASc thesis [121]. However, the thesis did not attempt to propose a way to detect SIMT deadlocks nor did propose a general solution. In [47], the authors provide formal semantics for NVIDIA’s stack-based reconvergence mechanism and a formal definition for the scheduling unfairness problem in the stack-based execution model that may lead to valid programs not being terminated. However, they do not attempt to provide ways to detect or prevent the problem.

There is also some recent work on verification of GPU kernels that focuses on detecting data-races and/or barrier divergence freedom in GPU kernels [17, 79, 131, 160]. However, none of the verification tools considered the problem of SIMT deadlocks due to conditional loops.

3.7 Summary, Conclusion and Future Directions

In this chapter, we explained the scheduling constraints imposed by current SIMT implementations and how it conflicts with inter-thread communication under divergence. We defined SIMT-induced deadlock, a type of deadlock that is unique to SIMT architectures. We showed that SIMT deadlock can be introduced by either programmers or compilers that are oblivious to the SIMT scheduling constraints. We explained how the presence of SIMT deadlocks affects the programmability of SIMT architectures. Finally, we presented an algorithm that can be used to statically identify potential SIMT deadlocks and highlighted the limitations of our current implementation of the algorithm. We implemented the algorithm in LLVM 3.6 and evaluated it on a number of CUDA and OpenCL applications showing a false detection rate of 4%–5%.

The increasing complexity of algorithms mapped to SIMT accelerators mandates the development of robust verification tools. To the best of our knowledge, none of the current verification tools considers the problem of SIMT deadlocks due to conditional loops. We can also use the outcome of the detection algorithm to eliminate SIMT deadlocks as we explain in the next chapters.

Future directions for this line of research should attempt to deal with the limitations mentioned in Section 3.3.2. For example, the static detection algorithm can be extended to leverage runtime information and/or more elaborate static analysis.
to perform less conservative reachability, dependence and alias analysis. Also, an inter-procedural analysis is needed to detect SIMT deadlock across function calls.

This work also makes the observation that SIMT compilers need to be aware of SIMT scheduling constraints to avoid generating SIMT deadlocks. This affects all optimizations that alter the CFG or that move instructions across CFG basicblocks. Our work provides a framework that could be extended to detect whether a certain application is prone to SIMT deadlocks given a certain transformation (using variations of our detection algorithm).
Chapter 4

SSDE: Static SIMT-induced Deadlock Elimination

In Chapter 3, we proposed a static detection algorithm for SIMT-induced deadlocks. This chapter proposes a SIMT-induced deadlock static elimination algorithm which we briefly refer to as “SSDE”. The purpose of this transformation is to allow code with traditional MIMD-style inter-thread synchronization to correctly terminate on current SIMT implementations. In simple terms, it can be viewed as a generalization and automation of the workaround in Figure 3.6.

The main goal of the elimination algorithm is to workaround the SIMT scheduling constraints mentioned in Section 3.1, namely, the forced reconvergence at immediate postdominators and the serialization of divergent paths. Our SIMT-induced deadlock elimination algorithm deals with these constraints as follows: (1) delays reconvergence to the earliest point where reconvergence would certainly not interfere with inter-thread communication, and (2) enforces through CFG transformation a breadth first traversal to basicblocks with inter-thread communication. The later step is essential to avoid the impact of the serialization constraint. To understand this better, let’s consider the code example in Figure 3.3. Assume that we delayed the reconvergence point of the lock acquire loop to a point beyond the lock release statement at line 4, `atomicExch(mutex, 0)`, instead of reconverging at the loop’s immediate postdominator. We would still suffer from a SIMT deadlock because the stack would also prioritize the execution of the taken path (i.e., the
lock acquire loop path) and would never get to execute threads that diverged to the not-taken path (i.e., the lock release path). Note that changing the current stack behaviour to prioritize the non-taken path first would fix the serialization problem on this example. However, this simple change fails to handle the general case. For example, consider the case of a producer-consumer relationship where threads diverged to a not-taken path wait in a loop for a signal from threads diverged to the taken path. If the stack prioritized the not-taken path, a SIMT deadlock would result because threads in the taken-path would never get a chance to execute. Furthermore, if the producer-consumer relationship between alternate paths following a branch is mutual, both paths may need interleave execution to guarantee forward progress. Thus, no static (i.e., compile time) prioritization of one path over another is sufficient to avoid SIMT deadlock. Our SSDE CFG transformation attempts to guarantee forward progress by achieving breadth-first execution of the application CFG instead of the current depth-first execution imposed by the stack.

SSDE is composed of two steps. The first step runs a static analysis to identify safe reconvergence points for loops that are detected to induce a SIMT deadlock (described in Chapter 3). The step stage runs a CFG transformation that maintains the original MIMD semantics of the program while eliminating SIMT-induced deadlocks when running on SIMT machines.

4.1 Safe Reconvergence Points Identification

This section describes the second stage of our static analysis, which is summarized in Algorithms 2 and 4. These algorithms require as inputs the results of Algorithm 1 from Chapter 3. This stage identifies program points, called safe postdominators (SafePDoms), that can be employed by our SSDE code transformation to eliminate SIMT deadlock. A key idea is that a reconvergence point of a branch is an arbitrary choice of all its postdominators. Thus, it is possible to delay reconvergence of loop exits to any of their postdominator points including the kernel exit. However, from a SIMD utilization perspective, it is preferable to reconverge at the earliest postdominator point that would not block the required inter-thread

\[1\] Due to the serialization constraint, delaying reconvergence is necessary but not sufficient to eliminate SIMT deadlocks on current SIMT implementations (more in Section 4.2.1).
communication. SafePDoms are used in both our compiler based SIMT deadlock elimination algorithm (Section 4.2.1) and/or our proposed adaptive hardware reconvergence mechanism (Chapter 5).

The potential benefit of delaying reconvergence to overcome SIMT-induced deadlocks is intuitive when one considers the second scheduling constraint (i.e., the forced reconvergence). A SIMT deadlock due to the forced reconvergence scheduling constraint, happens when threads that exit a conditional loop are blocked at the loop reconvergence point indefinitely waiting for looping threads to exit; the looping threads are waiting for the blocked threads to proceed beyond the current reconvergence point to release their lock(s). This cyclic dependency can be broken if the loop reconvergence point could be moved later such that even if threads were allowed, hypothetically, to pass the new delayed reconvergence point they could not affect the loop exit conditions (i.e., the threads could not reach a redefining write which indicates that the forced reconvergence at the delayed reconvergence point does not prevent necessary inter-thread communication). SafePDoms are computed such that they postdominate the loop exit branches and all control flow paths that lead to redefining writes from the loop exits (lines 4-9 in Algorithm 2).

It is perhaps less intuitive how delaying reconvergence helps overcome the first scheduling constraint (i.e., serialization). To understand this, it is necessary to understand how SSDE transforms the code after having identified a SafePDom. Due to the serialization constraint, threads iterating in a loop can be prioritized over the loop exit path and/or other paths that are parallel to the loop. A SIMT-induced deadlock occurs if these blocked threads must execute to enable the exit conditions of the looping threads. To avoid this, our compiler based SIMT deadlock elimination algorithm (explained in more detail below in Section 4.2.1) replaces the backward edge of a loop identified by Algorithm 1 with two edges: a forward edge towards the loop’s SafePDom, and a backward edge from SafePDom to the loop header. This control flow modification combined with the forced reconvergence constraint of the hardware, guarantees that threads iterating in the loop wait at the SafePDom for threads executing alternate paths that are postdominated by this SafePDom before attempting another iteration. Accordingly, a SafePDom should postdominate the original loop exit and points along control flow paths that could lead to redefining writes that are either reachable from the loop (lines 4-9 in
Algorithm 2 Safe Reconvergence Points

1: Inputs: \texttt{RedefWrite}(L) for each L \in \texttt{LSet}
2: Outputs: SafePDOM(L) for each L \in \texttt{LSet}
3: SafePDOM(L) = IPDom(Exits(L)) \forall L \in \texttt{LSet}
4: for each loop L \in \texttt{LSet} do
5:    for each I_W \in \texttt{RedefWrite}(L) do
6:       SafePDOM(L) = IPDom(SafePDOM(L), I_W)
7:       if BB(I_W) is reachable from L then
8:          for each branch instruction I_BR \in ReachBrSet(L, BB(I_W)) do
9:             SafePDOM(L) = IPDom(SafePDOM(L), I_BR)
10:       end for
11:    end if
12:    if BB(I_W) is parallel to L then
13:       for each branch instruction I_BR \in ParaBrSet(L, BB(I_W)) do
14:          SafePDOM(L) = IPDom(SafePDOM(L), I_BR)
15:       end for
16:    end if
17: end for
18: resolve SafePDOM\_conflicts()

Algorithm 2 or parallel to it (lines 10-14 in Algorithm 2).

Iterative refinement to the identified SafePDoms for nested or consecutive loops may be necessary. In particular, SafePDOM(L) should postdominate SafePDOM of any loop in the path between the exits of loop L and SafePDOM(L). Otherwise, it is not a valid reconvergence point. We resolve this by recalculating SafePDOM(L) for each loop to postdominate all SafePDOM of loops in the path between loop L exits and SafePDOM(L) as summarized in Algorithm 3 which executes on the outputs of Algorithm 2. The process is then iterated until it converges. Convergence is guaranteed because in the worst case, the exit node of the kernel is a common postdominator. Note that we force a single exit using a merge return pass that merges multiple return points (if they exist) into one. During this process, lines 10-17 in Algorithm 3 detect if the transformation of a nested loop with the selected reconvergence point would create new SIMT-induced deadlocks in dominating loops and if so calculate a more conservative reconvergence point.

To better understand Algorithm 2, we briefly walk through its application to the code in Figure 3.3. The initial reconvergence point of the loop is the instruction that immediately follows its exit edge. However, there is a redefining write (the \texttt{atomicExch} instruction) in a basicblock that is reachable from the \texttt{while} loop exit.
Algorithm 3 Resolve SafePDOM Conflicts

1: Inputs: initial SafePDOM(L) for each loop
2: Outputs: final SafePDOM(L) for each loop
3: do
4:   converged = true
5:   for each loop curL ∈ LSet do
6:     iSafePDOM(curL) = SafePDOM(curL)
7:     if curL causes potential SIMT-induced deadlocks then
8:       for each loop L, where BBs(L) ⊂ P_Exits(curL)→SafePDom(curL) do
9:         SafePDom(curL) = IPDom(SafePDom(curL), SafePDom(L))
10:     end for
11:   else
12:     for each loop L, where BBs(L) ⊂ BBs(curL) do
13:       if SafePDOM(L) dominates SafePDOM(curL) then
14:         if curL Exits are dependent on shared variable then
15:           Label curL as a potential cause for SIMT-induced deadlocks
16:         end if
17:       end if
18:       SafePDom(curL) = IPDom(SafePDom(curL), SafePDom(L))
19:     end for
20:   end if
21:   if iSafePDOM(curL) ≠ SafePDOM(curL) then
22:     converged = false
23:   end if
24: end for
25: while converged ≠ true

Thus, line 4 updates SafePDom of the loop to be the instruction that immediately follows the atomicExch. No further updates to SafePDom by the rest of the algorithm are performed. Figure 4.1 shows the appropriate choice of SafePDom for more complex scenarios. For example, the CFG to the bottom right resembles a scenario found in the sort kernel of the BarnesHut application [24] when compiler optimizations are enabled. Threads iterating in the self loop are supposed to wait for a ready flag to be set by other threads executing the outer loop. Threads executing the outer loop may need more than one iteration to set the ready flag of waiting threads. Thus, the filled (gray) basicblock is an appropriate choice of SafePDom as it postdominates all reachable paths to the redefining writes (i.e., leading threads may only wait for lagging one after they finish all iterations of the outer loop).
4.2 SSDE: Static SIMT Deadlock Elimination

This section proposes compiler transformation pass (Algorithm 4) that when combined with Algorithms 1, 2, and 3 enables the execution of MIMD code with inter-thread synchronization on stack-based SIMT machines. Note that very recently, Nvidia announced their newest GPU architecture, Volta, which replaces the stack-based execution model with a one that allows independent thread scheduling to enable inter-thread synchronization [103]. Volta’s execution model is in fact very...
Algorithm 4 SIMT-Induced Deadlock Elimination

1: Inputs: SafePDOM(L) for each L ∈ LSet
2: Output: SIMT deadlock free modified CFG.
3: SwitchBBs = ∅
4: for each loop L ∈ LSet do
5:   if L causes potential SIMT-induced deadlock then
6:     if SafePDOM(L) ∉ SwitchBBs then
7:       SwitchBBs = SwitchBBs ∪ SafePDOM(L)
8:       if SafePDOM(L) is the first instruction of a basicblock BB then
9:         Add a new basicblock BBs before the BB.
10:      Incoming edges to BBs are from BB predecessors.
11:      Outgoing edge from BBs is to BB.
12:   else
13:     Split BB into two blocks BB_A and BB_B.
14:     BB_A contains instructions up to but not including SafePDOM(L).
15:     BB_B contains remaining instructions including SafePDOM(L).
16:     BBs inserted in the middle, BB_A as predecessor, BB_B as successor.
17:   end if
18:   Insert a PHI node to compute a value cond in BBs, where:
19:     for each predecessor Pred to BBs
20:       cond.addIncomingEdge(0,Pred)
21:   end for
22:   Insert Switch branch swInst on the value cond at the end of BBs, where:
23:     swInst.addDefaultTarget(BBs successor)
24:   end if
25:   BBs is the basicblock immediately preceding SafePDOM(L)
26:   Update PHI node cond in BBs as follows:
27:     cond.addIncomingEdge(UniqueVal,Latch(L).src -unique to this edge
28:   Update Switch branch swInst at the end of BBs as follows:
29:     swInst.addCase(UniqueVal,Latch(L).dst)
30:   Set Latch(L).dst = BBs.
31: end if
32: end for

similar to our hardware proposal for an adaptive warp reconvergence mechanism in Chapter 5. We discuss Volta’s execution model more in Chapters 5 and 8.

4.2.1 Elimination Algorithm

To avoid SIMT-induced deadlocks, loose fairness in the scheduling of the diverged, yet communicating, threads is required. Constrained by the existing architecture, we achieve this by manipulating the CFG. Algorithms 2 and 3 pick the earliest point in the program that postdominates the loop exit and all control flow flow paths that can potentially affect the loop exit condition. Algorithm 4 modifies the CFG
by replacing the backward edge of a loop identified by Algorithm 1 by two edges: a forward edge towards SafePDom, and a backward edge from SafePDom to the loop header. This guarantees that threads iterating in the loop wait at the SafePDom for threads executing other paths postdominated by SafePDom before attempting another iteration allowing for inter-thread communication. Algorithm 4 is a generalization and automation of the manual workaround shown in Figure 3.6.

To help explain Algorithm 4, we examine how it is applied to the spin lock code from Figure 3.3 in Figure 4.2. The CFG on the left is the original and the one to the right is the CFG after the transformation. Applying Algorithm 2 and 3 results in identifying the SafePDom of the self loop at BB$_B$ 1 to be the first instruction after the atomicExch 2. Lines 11-14 in Algorithm 4 then introduce a new basicblock BB$_S$ 3. Next, lines 16-26 add a PHI node and switch branch instructions in BB$_S$ 4. Finally, line 28 modifies the destination of the loop edge to BB$_S$ 5. Thus, the end result is that the backward branch of the loop is replaced by two edges, a forward edge to BB$_S$ and a backward edge to the loop header. The new added basicblock acts as a switch that redirects the flow of the execution to an output basicblock according to the input basicblock. The PHI node is translated by the back-end to move instructions at predecessors of BB$_S$. The switch instruction is lowered into a series of compare instructions followed by direct branches such that it diverges first to the default not-taken path. This guarantees that BB$_S$ remains the IPDOM of subsequent backward branches forcing (potentially) multiple loops to interleave their execution across iterations.

Algorithm 4 preserves the MIMD semantics of the original code because the combination of the PHI node and branch in BB$_S$ guarantees the following: (1) each control flow path in the original CFG has a single equivalent path in the modified CFG that maintains the same observable behaviour of the original path by maintaining the same sequence of static instructions that affect the state of the machine (e.g., updates data registers or memory). For example, the execution path (BB$_{c1}$-BB$_S$-BB$_{c2}$) in the transformed CFG is equivalent to the path (BB$_C$) in the original CFG. The loop path (BB$_B$-BB$_B$) is equivalent to (BB$_B$-BB$_S$-BB$_B$) and (2) control flow paths in the modified CFG that have no equivalent path in the original CFG are paths and would not execute due to the way the switch statement added in BB$_S$ works. For example, the path (BB$_{c1}$-BB$_S$-BB$_B$) in the modified CFG would not
Figure 4.2: SIMT-Induced Deadlock Elimination Steps.

execute because the PHI node in BB₃ controls the switch condition such that if a thread is coming from BB₆ it branches to BB₇. The elimination algorithm does not reorder memory instructions or memory synchronization operations and therefore it does not impact the consistency model assumed by the input MIMD program. We add a more detailed semi-formal correctness discussion to the appendix of this thesis in Chapter A.

Algorithm 4 ensures a SIMT-induced deadlock free execution. For example, threads that execute the branch at the end of BB₄ wait at the beginning of BS₃ for other threads within the same warp to reach this reconvergence point. This allows the leading thread that acquired the lock to execute the \texttt{atomicExch} instruction and release the lock before attempting another iteration for the loop by the lagging threads.

4.2.2 Compatibility with Nvidia GPUs

There is no official documentation for Nvidia’s reconvergence mechanism, however, a close reading of Nvidia’s published patents \cite{18, 29} and examination of disassembly \cite{107} suggests that for a branch instruction to have a reconvergence point at its immediate postdominator, the branch must dominate its immediate post-
dominator and for loops the loop header must dominate the loop body (i.e., to be a single-entry/reducible loop). We accounted for the additional constraints as follows: first, in Algorithm 2 reconvergence points of divergent branches are not necessarily their immediate postdominators. If the branch’s immediate postdominator is not dominated by the branch basicblock then the reconvergence point of the branch is the post-dominator of the closest dominating basic block of the branch. Note that this affects the \( IPDom(I) \) definition in the prerequisites of Algorithm 1 defined in Listing 1 in Chapter 3. Second, in Algorithm 4 we have to guarantee that the new added basicblock is a valid reconvergence point. We guarantee this by forcing the new created loops to be single entry loops. In particular, If a loop has entries other than the header, it is converted to a single entry loop by adding a new loop header that merges all the loop entries and redistribute them to their original destinations. Furthermore, if the added switch branch instruction has more than two destinations, all backward edges are merged into one backward edge that jumps to the common dominator of all backward edges destination (this will be the final loop header) which controls divergence to the original backward edges destinations.

### 4.2.3 SSDE Limitations

The static elimination approach has some limitations. We highlighted the impact of some of these limitations on the detection algorithm scope and accuracy in Section 3.3.2. In this section, we focus on the implications of these limitations on the scope and efficiency of the elimination algorithm.

**Inter-procedural Dependencies:** Although, it is possible to extend our detection algorithm to handle function calls with an inter-procedural analysis [23, 59], extending the elimination algorithm is less straightforward. Algorithm 4 requires the redefinition statement to be inlined with the SIMT deadlock inducing loop in the same function. However, aggressive inlining can significantly increase instruction cache misses [32]. Also, in some cases, it may not be possible to perform inlining (e.g., recursive functions or calls to proprietary library code).

**CFG Modifications:** To avoid SIMT-induced deadlocks, the elimination algorithm modifies the original application’s CFG. This could make debugging the orig-
inal code a harder task [19, 52, 57]. The modified CFG adds more instructions that do not do useful work except to workaround the constraints of the current reconvergence mechanisms. These added instructions increase the static code size and dynamic instructions count. Finally, the modified CFG could result in increased liveness scope for some variables. This occurs because the modified CFG introduces new edges in the CFG that increase the connectivity between basic blocks. For example, in the original CFG in Figure 4.2, a variable that is defined in BB\textsubscript{A} and used only in BB\textsubscript{B} is live in basic blocks BB\textsubscript{A} and BB\textsubscript{B} only. However, in the modified CFG to the right, it is live in BB\textsubscript{A}, BB\textsubscript{B}, BB\textsubscript{C1}, and BB\textsubscript{S}. This increased liveness scope may lead to increased pressure on the register file leading to register spills to local memory thus causing increased memory traffic. Alternatively, by increasing the number of registers used per thread it could limit the maximum number of threads that could be launched. In either case, performance would be reduced.

**False Detections:** An increased number of false detections (loops that erroneously classified as involved in synchronization) directly amplifies the implications of the CFG modifications mentioned earlier by increasing the number of loops affected by the code transformation. It amplifies the unnecessary increase in dynamic instructions count and the register spills.

**Indirect Branches:** Directly applying the elimination algorithm on indirect branches would require some means of determining all potential targets. In the absence of complete information on potential branch targets, the CFG of the application is effectively unknown. One potential solution is to conservatively assume the same the safe postdominator point is the kernel exit.

**Barriers in Divergent Code:** SSDE does not handle divergent barriers and thus codes as shown in Figure 3.2 would still lead to undefined behaviour (refer back to Section 3.1.2 for more details).

**Warp Synchronous Behaviour:** As indicated in detail in Chapter 3, some GPU applications rely on the implicit warp synchronous behaviour. For such applications, it is necessary to disable our transformation, possibly using pragmas. In Chapter 5, we present a hardware reconvergence mechanism for avoiding SIMT deadlock. This mechanism, combined with the static analysis in Section 3.3, tackles most of the challenges associated with the compiler approach, but it is still
necessary to mark code relying on implicit warp synchronization.

4.3 Methodology

This section describes our methodology to evaluate SSDE. We implement the elimination algorithm as a transformation pass in LLVM 3.6 [86] that runs as the last transformation pass after the detection pass. For alias analysis, we use the basic alias analysis pass in LLVM [87].

In all experiments, we disabled optimizations in the back-end compilation stages from LLVM-IR to PTX by LLVM NVPTX back-end and from PTX to SASS by Nvidia’s driver\textsuperscript{2} to guarantee that there are no further alternation in our generated CFG. The detection pass runs as the last compiler pass to detect SIMT deadlocks because we found that SIMT deadlocks can be introduced by other transformations. Our LLVM code for both the detection and elimination passes has been posted online [37]. Note that, ideally, the safe postdominator analysis and SSDE would be implemented in the compiler back-end that generates the final hardware assembly. This back-end is not released by NVIDIA.

We use CUDA, OpenCL and OpenMP applications for evaluation. OpenCL applications are compiled to LLVM Intermediate Representation (LLVM-IR) using Clang compiler’s OpenCL frontend [84] with the help of the \textit{libclc} library that provides LLVM-IR compatible implementation for OpenCL intrinsics [85]. For CUDA applications, we use \textit{nvcc-llvm-ir} tool [35]. This tool allows us to retrieve LLVM-IR from CUDA kernels by instrumenting some of the interfaces of \textit{libNVVM} library [102]. OpenMP compilation relies on the recent support of OpenMP 4.0 in LLVM [15]. For hardware results, we use a Tesla K20C Kepler architecture GPU. Our benchmarks include OpenCL SDK 4.2 [106], Rodinia 1.0 [26], KILO TM [43], both CUDA and OpenMP versions of BarnesHut [24, 31], and two OpenMP microbenchmarks, Test Lock (TL) [22] and Array Max (AM) [94]. OpenCL SDK and Rodinia applications do not have synchronization between divergent threads, however, kernels from [22, 24, 31, 43, 94] require synchronization between divergent threads. Table 3.1 describes briefly all the kernels

\textsuperscript{2}PTX is NVIDIA’s virtual assembly and SASS is the machine assembly. SASS generation is currently done only by Nvidia’s drivers.
Table 4.1: Code Configuration Encoding

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>MIMD</td>
<td>-O0</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>SIMT</td>
<td>-O2</td>
<td>D Delayed Rec. (Ch. 5)</td>
</tr>
</tbody>
</table>

Table 4.2: Static Overheads for the Elimination Algorithm

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Tf. Loops</th>
<th>SASS-Static Inst.</th>
<th>SASS-Used Reg. + Stack Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>HT</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AFM</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CP-DS</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BH-TB</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>BH-SM</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BH-ST</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BH-FC</td>
<td>-</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

that are affected by our elimination algorithm.

Table 4.1 shows the abbreviation encoding we use in the results discussion. The first character indicates whether the original code accounts for the reconvergence constraints (S for SIMT) or not (M for MIMD). The second indicates the level of the optimization performed on the code before we run our passes. The last character indicates the type of analysis or transformation performed, it could be either applying the elimination algorithm (S), calculating the delayed reconvergence points without CFG transformations (D) or neither (-).

4.4 Evaluation

4.4.1 Static Overheads

We rewrite CUDA and OpenCL kernels that require inter-thread synchronization assuming MIMD semantics. Table 4.2 compares six different code versions in terms of static instructions, register and stack storage per thread. The M0- and M2- versions deadlock on current GPUs. Enabling default compiler optimizations in S2- lead all our four applications to deadlock. To address this, in configuration S2*-., we selectively enable passes that are experimentally found not to conflict with
the manual code transformation. This excluded all invocations of the `-simplifycfg` and `-jumpthreading` passes.

For the non-optimized versions (i.e., $S0$- and $M0S$), Table 4.2 shows that static instruction overhead is small in both manual and compiler based transformations ($S0$- and $M0S$ are comparable to $M0$-). Also, they have little to no overhead in terms of registers (e.g., CP) compared to the MIMD version when we consider the non-optimized versions. Turning on compiler optimizations generally reduces stack usage and increases the number of registers as a result of register allocation (via the `-mem2reg` pass in LLVM). This, however, has a negative impact for kernels with false positives when applying our detection Algorithm 1. For example, in BH-TB kernel in $M2S$, there is an increase in both registers and stack usage. This is due to a significant number of register spills as a result of increased liveness scope for registers after our CFG transformation.

### 4.4.2 Dynamic Overheads

We evaluate the run-time overheads using performance counters on Tesla K20C (Kepler architecture). Figure 4.3 shows the accumulated GPU time for all kernel launches averaged over 100 runs. HT and ATM have a single kernel, CP has four kernels and BH has 6 major kernels. $M0S$ has a 11% overhead on average compared to $S0$-. $M2S$ leads to a speedup of 45% compared to $S0$-. $M2S$ is also within 7.5% overhead compared to $S2^*$-. For some kernels (e.g., HT and ATM)
the benefit of enabling all compiler optimizations overcomes the overhead of the automated transformation leading to improvements compared to $S_2^*$. Figure 4.4 breaks down these results for kernels that are affected by our transformation. As shown in Figure 4.4, for kernels that have no false detections, $M_0S$ and $S_0$ versions have almost identical performance (e.g., HT and BH-ST). CP-DS is an exception. It has two consecutive loops that implement two nested spin locks. Both loops are detected as a potential cause for a SIMT-induced deadlocks. However, due to aliasing, the outer lock release is falsely classified as a redefining write for the inner lock. Hence, the SafePDom point for both loops is conservatively estimated to be after the outer lock release. This causes additional overhead versus the manual transformation. Threads try to acquire the outer lock even though the lock has been acquired by another thread within the same warp (one that acquired the outer lock but failed to acquire the inner lock). This leads to increase in dynamic instruction count seen in Figure 4.5.

For kernels that have false detections, the overhead is dependent on the runtime behavior. For example, although BH-FC has 3 false detections in its $M_0S$ version, they hardly impacts its performance. This is mainly because the kernel has very high utilization (Figure 4.6$^3$), hence it is barely affected by the transformations in its CFG. In other cases (e.g., ATM, BH-TB and BH-SM), false detec-

$^3$NVIDIA profiler does not measure SIMD efficiency for OpenCL applications. Thus, SIMD utilization for CP is not reported.
tions lead to significant performance overheads. We attribute these overheads to both the reduced SIMD utilization (Figure 4.6) and increased dynamic instruction count (Figure 4.5). In BH-TB, enabling compiler optimizations when SSDE is applied leads to an increase in execution time. We attribute this to the increase in memory traffic due to excessive register spills. We measured an increase of 28.5% in the DRAM requests per cycle for the \textit{M2S} version with respect to \textit{S0}- and \textit{M0S}. NVIDIA profiler also shows that \textit{M2S} has 38.3% less dram requests per cycle compared to \textit{S0-} for BH-SM. This explains the improved performance \textit{M2S} exhibits over \textit{S0-}. Finally, although \textit{M2S} performs poorly for both BH-TB and
BH-SM, the overall impact on the benchmark is not severe because BH-FC is the dominant kernel in the total execution time.

### 4.4.3 OpenMP support

The OpenMP 4.0 standard supports the offloading of a parallel region to an accelerator (e.g., a GPU). The OpenMP programming model is appealing due to both its abstraction and portability across architectures. Thus, it enables accelerators reaching to a broader set of developers [66]. Currently, there is an ongoing coordinated effort by many technology companies to add OpenMP 4.0 support for accelerators in LLVM [93]. Basic support is currently available [15, 117] while there are ongoing efforts to improve its performance [14, 16] compared to native GPU programming models. SIMT-induced deadlock is a fundamental challenge to correctly supporting synchronization on SIMT accelerators (e.g., it is unclear how to compile `omp_set_lock(...)` for GPUs). Our investigations have found that current OpenMP support for GPUs suffers from the SIMT deadlock problem as it generates code for a spin lock with back-off delay to translate `omp_set_lock(...)`. Hence, a valid OpenMP program that executes properly on CPUs may not terminate on GPUs due to SIMT deadlocks.

We evaluate the potential for our proposed compiler algorithms to address this by integrating the detection and elimination passes described in this chapter and Chapter 3 into the compilation chain of OpenMP.

At the time of writing we could not find any applications that make use of OpenMP support for GPUs and which employ fine-grained synchronization. This may partly be due to the recency of OpenMP support for GPUs and perhaps also the lack of support for generating efficient fine-grained synchronization code for GPUs in the current versions of OpenMP. Therefore, we modify 8 existing OpenMP ker-

### Table 4.3: OpenMP Kernels (Normalized Execution Times)

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CPU-4T -O2</th>
<th>GPU-OMP M2-</th>
<th>GPU-OMP M2S</th>
<th>GPU-CUDA S2*</th>
<th>Kernel</th>
<th>CPU-4T -O2</th>
<th>GPU-OMP M2-</th>
<th>GPU-OMP M2S</th>
<th>GPU-CUDA S2*</th>
</tr>
</thead>
<tbody>
<tr>
<td>TL-FG</td>
<td>1</td>
<td>D</td>
<td>3.41</td>
<td>N/A</td>
<td>BH-TB</td>
<td>1</td>
<td>D</td>
<td>0.28</td>
<td>9.66</td>
</tr>
<tr>
<td>TL-CG</td>
<td>1</td>
<td>D</td>
<td>0.01</td>
<td>N/A</td>
<td>BH-SM</td>
<td>1</td>
<td>3.78</td>
<td>1.72</td>
<td>3.54</td>
</tr>
<tr>
<td>AM-FG</td>
<td>1</td>
<td>D</td>
<td>32.58</td>
<td>N/A</td>
<td>BH-FC</td>
<td>1</td>
<td>0.70</td>
<td>0.73</td>
<td>1.25</td>
</tr>
<tr>
<td>AM-CG</td>
<td>1</td>
<td>D</td>
<td>17.87</td>
<td>N/A</td>
<td>BH-FC</td>
<td>1</td>
<td>0.44</td>
<td>0.44</td>
<td>9.00</td>
</tr>
<tr>
<td>BH-ST</td>
<td>1</td>
<td>D</td>
<td>2.97</td>
<td>3.17</td>
<td>BH-IN</td>
<td>1</td>
<td>1.21</td>
<td>1.21</td>
<td>7.53</td>
</tr>
</tbody>
</table>
nels to enable offloading parallel regions to Nvidia GPUs [22, 31, 94]. Six of these kernels are for an OpenMP BarnesHut implementation that uses an algorithm similar to the CUDA version except for some GPU specific optimizations. We also modify TL and AM kernels to emulate fine-grained and coarse-grained synchronization.

Table 4.3 shows the speed up of four different configurations run on a Tesla K20C GPU versus running the code with 4 threads on an Intel Core i7-4770K CPU. The base compilation to GPUs with current LLVM OpenMP 4.0 support encounters deadlocks in many cases (labeled ‘D’ in the table). However, with our detection and elimination passes all kernels run to termination maintaining portability between the CPU and the GPU. Numbers in bold shows instances where the GPU code achieved a speed up compared to the CPU without performance tuning. For other cases, the developer may choose either not to offload the kernel to a GPU or to perform tune the code for GPUs starting from a functionally correct (terminating) code that he can profile its execution and determine optimization opportunities. Due to false detections in BH-SM and BH-FC, OMP M2S performs differently from OMP M2- with a 2.2× slowdown for BH-SM. As expected, the GPU performs poorly compared to the CPU with high contention on locks (e.g., TL-CG where 3K threads are competing for the same lock) while performing much better with fine-grained synchronization (e.g., TL-FG). In AM, a non-blocking check, whether the current element is larger than the current maximum, happens before entering the critical section. This significantly reduces the contention over the critical section that updates the maximum value. Thus, the execution is highly parallel and achieves large speed up (17.87× for the coarse grained version and 32.58× for fine grain version as shown in Table 4.3) compared to the CPU. AM-FG finds multiple maximum values within smaller arrays reducing contention even further. Table 4.3 also compares the performance of the OpenMP version of BH with the CUDA version. In most cases, the CUDA version significantly outperforms the OpenMP version. Reducing this performance gap (either by improving OpenMP compiler support or performance tuning for the source code) is outside the scope of this paper.
4.5 Related Work

There are efforts to make GPU programming accessible through different well-established non-GPU programming languages. These efforts include source-to-source translation \[71, 101\] as well as developing non-GPU front-ends \[15\] for language and machine independent optimizing compilers such as LLVM \[69\]. However, these proposals do not handle SIMT deadlocks. MCUDA \[136\] is an opposite approach that translates CUDA kernels into conventional multicore CPU code. Auto vectorization is also a well studied area \[67\] to convert scalar code into codes that run on vector machines. However, these effort does not deal with inter-thread synchronization thus could lead to SIMT/SIMD deadlocks.

Recently, Google presented an open-source LLVM-based GPGPU Compiler for CUDA \[149\] that proposes GPGPU specific optimizations to generate high performance code. Similar prior work has also been proposed but using source to source CUDA transformation \[153\]. There are also numerous research papers that explore specific code optimizations for GPGPUs \[48, 49, 97\]. Such performance-driven compiler optimizations are complementary to the work presented in this chapter.

In \[78\], Li et al. propose a fine-grained inter-thread synchronization scheme that uses GPU on chip scratchpad memory. However, it is left to programmers to use their locking scheme carefully to avoid SIMT-induced deadlocks \[78\]. Recently, software lock stealing and virtualization techniques were proposed to avoid circular locking among threads in attempt to solve livelocks in nested locking scenarios, and to reduce the memory cost of fine-grain locks \[152\]. As acknowledged in \[152\], one of their limitations is that “locks are not allowed to be acquired in a loop way” due to deadlocks (i.e., SIMT deadlocks using our terminology). Our work assumes that the MIMD code is livelock free and that livelocks due to nested locking scenarios are taking care of by programmers as he would do on a MIMD machine (e.g., by forcing nested locks to be acquired in a certain order).

4.6 Summary, Conclusion and Future Directions

In this chapter, we presented a static SIMD-induced deadlock elimination algorithm. The algorithm is performed in two stages. The first stage identifies safe
postdominator points for problematic loops that are detected to potentially induce a SIMT deadlock. The second stage alters the CFG using the safe postdominator recommendations in a way that eliminates SIMT deadlocks.

Intellectually, this work is the first to propose MIMD-to-SIMD conversion for arbitrary code with inter-thread synchronization since Flynn’s taxonomy. This work is relevant to any architecture that relies on hardware-assisted implicit vectorization (e.g., Nvidia’s SASS [107] and AMD’s Southern Islands [5]) or that solely rely on explicit vectorization (e.g. Intels SSE and IBMs AltiVec). The later ones would need to integrate our SSDE algorithms with the regular auto-vectorization techniques (e.g., [67]).

This work also shows that SIMT implementations should not and do not have to dictate the level of abstraction used in their programming. In this chapter, we achieve this by using compiler methods that can selectively, through CFG transformations, decouple thread scheduling decisions from the application’s original code layout.

This work could be used to enable reliable library-based support for fine-grained synchronization on GPUs. Library based implementations for fine-grained synchronization primitives [78,152] are currently not possible because of SIMT-induced deadlocks. Another research direction could be to use the runtime JIT compilation techniques to filter out false detections within a specific kernels and then fine-tune the code transformation to avoid the large overheads in next kernel launches.

Further, one general insight from this work is that one can get represent all loops in a program with a single loop, and use forward branches to mimic the looping behaviour of merged loops. The forward branches can be further replaced by predication. This could be more efficient for architectures that can efficiently support only a limited number of loops (e.g., using hardware loops in digital signal processing chips).
Chapter 5

AWARE: Adaptive Warp Reconvergence

Chapter 4 presented SSDE—a compiler approach to enable MIMD-like execution on current stack-based SIMT implementations. However, as pointed out, the current version has limitations. In this section, we explore the potential for hardware modifications that can enable MIMD compatible execution to help avoid these limitations. The hardware mechanism we propose satisfies the following characteristics: (1) it maintains reconvergence at immediate post dominators when it does not interfere with inter-thread communication, (2) it guarantees that no thread would be indefinitely blocked due to scheduling constraints of divergent control flow paths, and (3) it enables delaying reconvergence when required for synchronization purposes. These characteristics maintain SIMD efficiency for applications without inter-thread synchronization while maintaining a SIMT deadlock free execution for applications with inter-thread synchronization. With such hardware, there is no need for the compiler to alter an applications’ CFG to avoid SIMT deadlocks which avoids many overheads and limitations.

This section describes AWARE—an Adaptive Warp Reconvergence mechanism that achieves the three characteristics mentioned above. AWARE avoids most of the limitations in our compiler-only approach. It decouples the tracking of diverged splits from their reconvergence points using two tables: a Splits Table (ST) and a Reconvergence Table (RT). ST and RT tables hold the same fields as the SIMT
stack with RT holding an extra field called the Pending Mask. The Pending Mask represents threads that have not yet reached the reconvergence point. AWARE uses these tables to avoid serializing divergent paths. AWARE replaces the depth-first traversal of divergent control-flow paths imposed by the reconvergence stack used in current implementations with a breadth-first traversal. It does this by selecting warp splits from the ST for greedy \textit{FIFO scheduling} with respect to their insertion to the ST either after a branch or after reaching a reconvergence point. In typical implementations, a depth-first traversal may lead to constant prioritization of a loop path over other parallel or following paths. This is avoided with a breadth-first traversal. This ensures fairness in scheduling different control paths. AWARE also enables \textit{delayed reconvergence} leveraging the SafePDOM points identified by the compiler analysis. For loops identified as a potential involved in SIMT deadlocks, the reconvergence point can be delayed to a SafePDOM using AWARE’s reconvergence table. The case where lock acquire and release occur in different functions can be supported in AWARE by employing \textit{timed-out reconvergence}. This mechanism works by allowing threads in a warp split that is stalled waiting for other warp splits to reconverge to proceed after a time exceeding a TimeOut value.

AWARE limits the architectural changes to the divergence unit. As we observe there tends to be a large gap between the maximum theoretical occupancy and the typical occupancy of the split and reconvergence tables we further propose to \textit{virtualize} them. We show this can be done by spilling entries that exceed the respective table’s physical capacity to a backing location in the memory system and filling them when needed. An advantage of AWARE is that it does not require changing the applications’ control flow graph. Thus, it does not suffer from the synchronization scope limitations and debuggability challenges associated with SSDE. It also transparently supports barriers in divergent code. Table 5.1 summarizes the above described advantages of AWARE over SSDE.

Note that at the time of writing this thesis, Nvidia announced a new GPU architecture called Volta. Volta changed the SIMT execution model to enable inter-thread synchronization. Volta’s execution model as described in [103] appears to be very similar to AWARE. The closest related Nvidia patent is a 2016 published patent that describes a notion of convergence barriers [34]. In the execution scheme described in this patent, convergence barriers are used to join divergent groups of
Table 5.1: AWARE vs SSDE

<table>
<thead>
<tr>
<th>Comparison Point</th>
<th>SSDE</th>
<th>AWARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-procedural synchronization</td>
<td>not supported</td>
<td>supported</td>
</tr>
<tr>
<td>CFG modifications</td>
<td>needed</td>
<td>not needed</td>
</tr>
<tr>
<td>Barrier in divergent Code</td>
<td>not supported</td>
<td>supported</td>
</tr>
<tr>
<td>SIMT-deadlocks detection</td>
<td>necessary and conservative</td>
<td>complimentary and relaxed</td>
</tr>
</tbody>
</table>

threads back together to maintain high SIMD efficiency while allowing for a flexible thread scheduling. In this proposal “the divergence management mechanism that relies on the convergence barriers is decoupled from the thread scheduling mechanism”. The compiler analyzes the program to identify the appropriate locations to insert convergence barriers. In hardware, “a multi-bit register may correspond to each convergence barrier name and a bit is assigned for each thread that may participate in the convergence barrier”. When all the threads reach a convergent barrier (i.e., the multi-bit register of the corresponding barrier is all zeros), the convergence barrier is cleared and all threads that participated at the convergence barrier are released (i.e, unblocked) and they resume execution in SIMD fashion. Similar to AWARE, the patent decides to decouple tracking divergence entries from reconvergence points. The multi-bit convergence barrier registers are simply another representation to the pending mask entry in the reconvergence table. The main difference is that our proposal as we describe in this chapter relies on the hardware implicitly adding reconvergence points to the reconvergence table, checking whether threads have reached reconvergence points or not, and deciding when to switch execution to other paths. On the contrary, in Nvidia patent, the compiler explicitly adds instructions to perform these actions when needed.

5.1 Decoupled SIMT Tables

The goal of AWARE is to impose fewer scheduling constraints than stack-based reconvergence while maintaining immediate post-dominator reconvergence when possible. To achieve this, AWARE decouples the tracking of diverged splits from their reconvergence points using the two tables shown in Figure 5.4. The warp Split Table (ST) records the state of warp splits executing in parallel basic blocks. The Reconvergence Table (RT) records reconvergence points for active warp splits.
The ST and RT tables hold the same fields as the SIMT stack with the RT holding an extra field called the Pending Mask. The Pending Mask represents threads that have not yet reached the reconvergence point. This decoupling enables AWARE to not serialize divergent paths up to the reconvergence point.

Figure 5.1a shows a simple divergent code example. Its corresponding control flow is shown in Figure 5.1b. This example assumes a single warp with four
threads traversing through the code. The bit mask in each basic block of the control flow graph denotes the threads that execute that block. Initially all threads execute basic block A. However, upon executing the divergent branch $BR_{B-C}^A$, warp $A_{1111}$ diverges into two warp splits $B_{1010}$ and $C_{0101}$. In the notation $BR_{B-C}^A$, branch is abbreviated as $BR$ and the superscript $A$ represents the basic block containing the branch while the subscript $B-C$ represents the successors. The state of a warp split is represented by a letter representing the basic block the warp split is executing and subscripts represent the active threads within the warp split. Figure 5.1c shows the operation of AWARE illustrating changes to the ST and RT tables. Labels are used to clarify which parts of the figure is being referred to in the text below.

At the top, Figure 5.1c shows the state of the ST and RT when the warp begins executing at block A. Since there is no divergence, there is only a single entry in the ST, and the RT is empty. The warp is scheduled repeatedly until it reaches the end of block A. After the warp executes branch $BR_{B-C}^A$, warp $A_{1111}$ diverges into two splits $B_{0101}$ and $C_{1010}$. Then, the $A_{1111}$ entry is moved from the ST to the RT with PC field set to the RPC of branch $BR_{B-C}^A$ (i.e., D). Similar to the SIMT stack, the RPC can be determined at compile time and either conveyed using an additional instruction before the branch or encoded as part of the branch itself (current GPUs typically include additional instructions to manipulate the stack of active masks). The Reconvergence Mask entry is set to the value of the active mask the warp split had before the branch. The Pending Mask field of the RT is used to keep track of which threads that have not yet reached the reconvergence point. Hence, it is also set equal to the active mask. Concurrently, two entries are inserted into the ST—one for each side of the branch. The active mask in each entry represents threads that execute the corresponding side of the branch.

At this point, both warp splits $B_{0101}$ and $C_{1010}$ are in the ST. In principle, AWARE does not restrict the order in which these splits are executed. As we will see in Chapter 7, the execution of both warp splits can be interleaved up to the reconvergence point (D). In Section 5.2 we discuss AWARE’s warp split scheduling policy in detail. For now, we assume we schedule warp split $B_{0101}$ first. Eventually, warp split $B_{0101}$ reaches the reconvergence point (D). When this happens its entry in the ST table is invalidated, and its active mask is subtracted from the pending
active mask of the corresponding entry in the RT table. Since only warp split C remains in the ST, execution switches to warp split C1010 which eventually reaches reconvergence point (D). When it does, its entry in the ST table is also invalidated, and its active mask is subtracted from the pending active mask of the corresponding entry in the RT table. Upon each update to the pending active mask in the RT table, the updated Pending Mask is checked. When the Pending Mask is all zeros the warp split entry is moved from the RT table to the ST table.

5.2 Warp Splits Scheduling

To limit architectural changes to the divergence unit, AWARE allows only one warp split to be eligible for scheduling at a time. AWARE switches execution to another warp split only at basic block boundaries. We found that AWARE does not require changes to the register scoreboard used in the baseline GPU [30] to perform well. Instead, AWARE can use the existing scoreboard which results in it conservatively respects register dependencies on a warp granularity. For example, in Figure 5.1b, warp split C0101 may stall for pending register write dependencies from warp split B1010 even though these two warp splits are guaranteed to access distinct physical register locations, potentially leading to unnecessary stalls. In Chapter 7, we explore the benefit of relaxing this constraint. Specifically, in Chapter [7] we propose and evaluate an efficient scoreboard mechanism that tracks register dependencies on a thread granularity to avoid unnecessary stalls due to false dependencies. In this chapter, however, our goal is to design a MIMD compatible reconvergence mechanism with minimal hardware changes.

The warp splits scheduling policy in AWARE is crucial to guarantee that no thread would be indefinitely blocked while scheduling divergent threads under synchronization and thus achieving the goal of MIMD compatible execution. For this purpose, AWARE replaces the depth first traversal imposed by the SIMT stack with a breadth first traversal using a FIFO queue. In Figure 7.4, the FIFO queue is included in the ST. We show more detailed microarchitecture in Section [5.5] A warp split is selected for greedy scheduling in FIFO order with respect to the ST. A warp split is pushed into the FIFO when it is first inserted into the ST as an outcome of a branch instruction or as a reconverged entry. It is scheduled when it reaches
the output of the FIFO. It is popped out of the FIFO when its entry in ST is in-validated after encountering a branch instruction or reaching a reconvergence point. This guarantees fairness in scheduling different control paths. AWARE switches from one warp split to another only after encountering divergent branches, reconver-
vergence points, or barriers. Such Greedy scheduling of the output entry ensures good memory performance [124].

5.3 Nested Divergence

If no synchronization is detected, AWARE is able to maintain IPDOM reconver-
gence, as defined in Section 2.3.1, even in nested divergence scenarios. Figure 5.2 illustrates an example. Figure 5.2c shows the state of the ST and RT tables after each step of executing the control flow graph in the left part of the figure.

Initially, a single entry in the ST exists for warp split A_{1111} (1). After branch BR_{B-C}, the branch control unit updates the ST and RT tables as in the prior ex-
ample from Section 7.2.1 (2). AWARE’s FIFO scheduler prioritizes the not-taken path in case of multi-path branches. We assume it prioritizes C_{1010} in this exam-
ple. Thus, subsequently, warp split C_{1010} diverges at BR_{D-E} (3). Hence, the entry corresponding to C_{1010} in the ST table is invalidated in the ST and inserted to the RT table with PC field set to the the reconvergence point of BR_{D-E}, which is F. Also, two new entries corresponding to both sides of BR_{D-E} are added to the ST table. The new entries are added to the first unallocated entries in the ST table. At this point the ST tracks three parallel control flow paths. However, following FIFO order, warp split B_{0101} executes first. Eventually, warp split B_{0101} reaches re-
convergence point G (4). The branch control unit updates the Pending Mask of the corresponding reconvergence table entry for G_{1111}. Concurrently, the B_{0101} entry in the ST table is invalidated (4). The ST stores indices to the reconvergence entries in the RT avoiding need for an associative search through the RT table.

The warp split scheduler then switches execution to the front of the FIFO queue. We assume it is warp split E_{1000} (i.e., it is the not-taken path for BR_{D-E}). Later warp split E_{1000} reaches its reconvergence point F (5). The Pending Mask of the reconvergence entry F_{1010} is updated accordingly, and the E_{1000} entry in the ST table is invalidated. Then, warp split D_{0010} reaches the same reconvergence point
// id = thread ID
if (id % 2 == 0) {
    // BB_B
} else {
    // BB_C
    if (id == 1) {
        // BB_D
    } else {
        // BB_E
    }
    // BB_F
} // BB_G

(a) Code

(b) CFG

(c) ST and RT tables (only valid entries shown)

Figure 5.2: Example of Multi-Path IPDOM execution with nested divergence

(F). The ST entry is removed and the reconvergence entry $F_{1010}$ is updated again to mark the arrival of warp split $E_{1000}$. 

71
Upon updating the Pending Mask of the reconvergence entry $F_{1010}$, the MP control unit detects that there are no more pending threads for this entry (the Pending Mask is all zeros). Hence, the MP control unit moves the reconvergence entry to the ST table, setting the active mask to the Reconvergence Mask $7$. Finally, warp split $F_{1010}$ reaches the reconvergence point $G$ and updates the reconvergence entry $G_{1111}$. The control unit detects that the Pending Mask of entry $G_{1111}$ is all zeros $8$ and moves the entry from the RT table to the ST table $9$.

This example does not cover three special cases that can be optimized. The first is when one side of the branch directly diverges to the reconvergence point of the branch (e.g., an if clause with no else). In this case, the Pending Mask of the corresponding reconvergence entry is updated to mark this side of branch as converged, and there is no need to register a new entry for it in the ST. The second is when there no actual divergence at run-time at a specific branch. In such case, there is no need to add a reconvergence entry in RT that corresponds to this branch. The third case is when a warp split encounters a branch whose reconvergence point is the same as the reconvergence point of the diverged warp split (e.g., backward branches that create loops or a forward branch that has its reconvergence point the same as a preceding branch). In such case, there is no need to add a new entry to the RT table, since the corresponding reconvergence entry already exists. If such entry were added, its PC and RPC fields would be identical and thus it reconverges automatically to the parent entry once it moves the ST table. Note that our proposed mechanism does not require to look up the RT to check if a RPC already exists with the insertion of reconvergence points. This may lead to multiple entries in the RT representing the same reconvergence points (but with different reconvergence masks). This is fine as long as each entry in the ST table is associated with a single entry in the RT table (e.g., using an RT index field in the ST table). We explain these implementation details in Section 5.5.

5.4 Using AWARE to avoid SIMT Deadlock

At this point, we have shown that AWARE: (1) replaces the depth first traversal of the SIMT stack with alternatives such as a breadth first FIFO traversal which relaxes the serialization scheduling constraint described on Page 20 in Section 3.1.
(2) is able to maintain IPDOM reconvergence which maintains SIMD efficiency. These are two of the three characteristics we mentioned at the introduction of this chapter that a hardware mechanism needs to satisfy to efficiently avoid SIMT deadlocks. The third characteristic is the capability of delaying reconvergence beyond IPDOM when required for synchronization purposes. In this section, we explain refinements to AWARE to achieve this third characteristic. Further, we show how AWARE can handle barriers in divergent code.

5.4.1 Handling Divergent Barriers:

In AWARE, we support barriers in divergent code by adding a field in the FIFO queue to mark warp splits that reach the barrier as blocked (abbreviated BL in Figure 5.3). The PC of the blocked split is updated in both the ST and the FIFO to point to the instruction after the barrier. When a blocked entry is at the FIFO output, it is pushed to the back of the FIFO without being scheduled for fetch. Splits are released when all threads within a thread block or within a warp, depending on the barrier granularity, reach the barrier.

Figure 5.3 shows the operation of AWARE during the execution of the CFG shown in the top left of Figure 5.3. The top right portion of Figure 5.3 shows the state of both the ST and RT tables after the warp executes the branch at the end of BB\textsubscript{A}. It also shows the state of the FIFO queue. The FIFO field shows the FIFO order of warp splits and the BL field indicates if the warp split is blocked at a barrier. Initially, the FIFO queue shows that threads that diverged to BB\textsubscript{B} should execute first. Threads at BB\textsubscript{B} continue execution until reaching the barrier instruction. Then, the entry that corresponds to these threads in the FIFO queue is marked as blocked. Finally, the PC field in the ST table is updated to the instruction following the barrier (B') so that execution continues from this PC when the barrier is released. The next available warp split in the FIFO is BB\textsubscript{C}. It executes until the other barrier instruction is reached as well. In case of a thread block wide barrier, both warp splits will eventually be blocked until the barrier is released after all threads reach the barrier.
5.4.2 Delayed Reconvergence:

As discussed in Chapter 3, strict IPDOM reconvergence can lead to SIMT deadlocks. To address this, for any loop that Algorithm 1 reports as being the potential cause of a SIMT deadlock, AW ARE uses the safe reconvergence points computed using Algorithm 2. Since AW ARE does not have the serialization constraint imposed by stack-based SIMT implementations, Algorithm 2 can be simplified to consider only redefining writes that are reachable from the loop reconvergence points (removing lines 10–14).

It is necessary to recalculate reconvergence points of other branches to guarantee that the reconvergence point of any branch postdominates the reconvergence.
points of all branches on the path from the branch to its reconvergence point (Algorithm 5). In Algorithm 5, the initial SafePDOM point of a branch is either the SafePDOM identified by Algorithm 2 or the normal IPDOM point of the branch if it does not induce a SIMT deadlock. Note Algorithm 5 being applied to all branches, eliminates the need for the step of resolving SafePDOM conflicts among SIMT deadlock inducing loops. In SSDE, we needed to consider conflicts among loops only because the code is later transformed and the reconvergence points of the different branches are computed as the immediate post dominator points of the branches in the transformed code. However, in AWARE we apply algorithm 5 instead of 2.

Figure 5.4 illustrates the operation of AWARE with delayed reconvergence. The reconvergence point of the loop is modified from the IPDOM point (i.e., C) to a SafePDom point (specifically., C', the closest point following the unlock statement) 1. Once the loop branch instruction is encountered, two warp-split entries are added to the ST and the RPC of each entry is set to C' 2. The FIFO queue has two valid entries with priority given to the not-taken path 3. Hence, the thread that diverged to BB_C (i.e., exited the loop) executes first 4. It releases the lock and waits at the reconvergence point (C'). Eventually all threads exit the loop and reconverge at C' 5. Note that the choice between prioritizing the not-taken versus the taken path is arbitrary. In cases, where there are no branches between the lock and the unlock statement, choosing the take path may come with some benefit to performance in terms of reducing the amount of spinning. We discuss techniques to reduce warp spinning in the spin lock loop. In Chapter 6 we discuss more robust techniques to reduce warp spinning overheads.

5.4.3 Timed-out Reconvergence:

The compiler may fail to detect a SIMT deadlock if the synchronization is across function calls. Further, to guarantee SIMT deadlock free execution, it may take very conservative decisions in labeling SIMT deadlock inducing loops. To avoid these limitations. We extend AWARE with a timeout mechanism.

In this mechanism, a delay counter is associated with each entry in the RT. The counter starts counting when at least one thread reaches the associated reconver-
Algorithm 5 Resolve SafePDOM Conflicts - AWARE version

1: Inputs: initial SafePDOM(Br) for each branch
2: Outputs: final SafePDOM(Br) for each branch
3: do
4: converged = true
5: for each branch curBr ∈ BrSet do
6:   iSafePDOM(curBr) = SafePDOM(curBr)
7:   for each branch Br, where BB(Br) ⊂ PExits(curBr) → SafePDom(curBr) do
8:     SafePDom(curBr) = IPDom(SafePDom(curBr), SafePDom(Br))
9:   end for
10: if iSafePDOM(curL) ≠ SafePDOM(curL) then
11:   converged = false
12: end if
13: end for
14: while converged ≠ true

gence point (and updates the pending mask). The counter rests with the insertion of the entry or any update to its pending mask. Otherwise, it increments every T cycles. T represents the resolution of the counter. In our evaluation we used T=1 cycle. When the delay counter value exceeds a predetermined TIMEOUT value, this indicates a potential SIMT-deadlock due to the forced reconvergence constraint. Therefore, at this point, we enable threads that have reached the reconvergence point to proceed their execution by inserting them into a new entry in the ST table and updating the active mask of their RT table.

Figure 5.5 shows how the timed-out reconvergence mechanism operates on the CFG on the left. In this case, we assume that reconvergence points are set to the IPDOM points (i.e., delayed reconvergence is off). Initially, there is a single entry in the ST representing threads that are iterating through the loop attempting to acquire the lock whereas the thread that exits the loop keeps waiting at the reconvergence point. This state continues until the reconvergence timeout logic is triggered. Once the waiting time of threads at the reconvergence point exceeds the TimeOut value, a new entry is added to the ST with the same PC and RPC of the entry in the RT and an active mask that is the subtraction of the Active Mask and the Pending Mask from the RT entry 4. We also update the active mask of the RT entry to be the same as the Pending Mask to reflect the fact that the timed-out threads are no longer waiting at this entry.

The new entry is added to the FIFO queue 5. The new entry C1000 is guar-
Figure 5.5: Timed-Out Reconvergence

anted to be executed as entry $B_{0111}$ gets to the tail of the FIFO queue once the loop branch is executed $6$. In a nested control flow graph, threads that skip reconvergence at a nested reconvergence point can reconverge at the reconvergence point associated with a prior branch for which the immediate postdominator has not yet been encountered. The $TimeOut$ value could be determined in a number of ways. One way, is to determine the value empirically by profiling a large number of GPU kernels. However it is determined, a fixed value of $TimeOut$ should be large enough such that it does not impact the reconvergence behaviour of regular GPU applications (thus avoiding unnecessary performance penalties).

5.5 AWARE Implementation

This section describes a micro-architecture realization of AWARE. We start with describing a straightforward realization and the describe an optimization that reduces AWARE implementation cost.

5.5.1 AWARE Basic Implementation

Figure 5.6 shows a basic implementation of the ST and RT tables and its interaction with the branch resolution unit. Note that in AWARE, no changes are required to the baseline instruction buffer. The reason is that we only switch between warp splits at after a branch or after reaching a reconvergence point. In both cases, the
instructions that belong to a warp in the instruction buffer become invalid and new instructions are fetched from the new program counter of the split at the front of the FIFO. Note that this is identical to the interface of the stack with the instruction buffer except that in case of the stack, the instruction(s) in the instruction buffer that belong to a warp are for the warp split at the top of the stack, in AWARE, the instruction(s) are for the warp split the front of the FIFO.

Each warp has its own ST and RT tables. Upon divergence, the branch control unit invalidates the warp’s entry in the I-Buffer. Hence, it is no longer eligible for fetch or issue. The branch control unit transfers the content of the divergent entry from the ST to an unused (invalid) entry in the RT after modifying the PC field to the RPC of the branch. The index of this RT entry, R-Index, is stored, along with the other information required for each new warp splits resulting from the divergence, in the ST. The R-Index is used to access the reconvergence entry in the RT table when the warp split’s PC is equal to its RPC. The indices of the new entries in the ST are added to the back of the FIFO. Note that in this basic implementation, the FIFO does not need to be a separate structure from the ST table. However, as we explain in Section 5.5.2 we may need to spill the ST table entries to memory while keeping the FIFO within the SM. Therefore, we refer to them as separate logical structures.

The branch control unit also marks the entry of the divergent split in the ST table as invalid and the next split in the FIFO order becomes eligible for scheduling. The PC entry in ST and RT tables points to the first instruction in a basic block; thus it directs the fetch unit to the next instruction to be fetched from the corresponding warp.

Upon reconvergence, the branch control unit invalidates the reconverged entry in both the I-Buffer and the ST. It uses the R-Index field of the reconverging split to access the RT table and update the pending mask. Finally, the Pending Mask of the updated entry is checked; if it is all zeros, the entry is moved to the first unallocated entry in the ST table and inserts the ST entry index at the back of the FIFO.

With 32 threads per warp, the ST and RT tables have a maximum theoretical size of 32 entries per warp (max splits is 32 and RT entries are added only when a split diverges which implies a maximum of 32 RT entries). Thus, AWARE can be realized using RAMs by adding a 5-bit field to the FIFO, ST and RT tables. Upon
insertion of an entry into the ST, the 5-bit index of this ST entry is stored in the corresponding FIFO queue entry. Upon insertion of an entry in the RT table, the RT entry index is stored in a field in its ST entries. To avoid moving entries with the FIFO, two registers could be used to indicate the front and back of the FIFO in a circular manner. Look ups into ST and RT tables use these indices with no need to search the contents of the tables. We keep track of invalid entries in a free queue implemented with a small (32 x 5-bits) stack (not shown in the Figure). For AWARE basic implementation (described in this section), the free queue is only needed for RT table since the ST table is handled as a FIFO queue. To insert an ST or RT entry the next index from the associated free queue is used.

5.5.2 AWARE Virtualized Implementation

The basic implementation described above has high area overhead (about 1 KB storage requirement per warp). This is because the implementation sizes all tables for the worst case scenario (i.e., 32 ST and RT entries per warp). However, empirically we find the typical occupancy of ST and RT tables is much lower. Therefore, we study the impact of virtualizing the ST and RT tables by spilling entries that exceed their physical capacity to the memory system and filling them when they are to be scheduled (for ST entries) or updated (for RT entries).
Figure 5.7 illustrates our virtualized AWARE implementation. The virtualized AWARE implementation is composed of 10 parts organized around the ST and RT tables. The Physical ST Table potentially holds only a subset of active warp splits. Similarly, the RT table potentially tracks a subset of reconvergence points. When the total number of splits exceeds the capacity of the Physical ST table, the ST Spill Buffer is used to move some warp splits to the memory hierarchy as described below: The ST Fill Request and Response buffers are used to bring these warp split entries back into the Physical RT Table. The same applies to the RT buffers. The reconverged entry buffer and pending mask updates buffer store the data exchanged between the ST and RT tables.

A branch instruction of a warp is scheduled only if both the ST and RT Spill Request Buffers of this warp are empty. Also, instructions from a warp are eligible for scheduling only if the warp Reconverged Entry and Pending Mask Updates Buffers are empty. When a new entry is required to be inserted into a full ST or RT table, an existing entry is spilled to their respective Spill Request Buffers. We use a FIFO replacement policy for the ST and an LRU replacement policy for the RT. When an entry is spilled, its corresponding entry in the FIFO is labeled as virtual. When a virtual entry is at the FIFO output, a fill request for this entry is sent and the entry is labeled as transient. This is to avoid sending multiple fill requests for the same entry. Also, the entry is pushed to the back of the FIFO. When a pending mask update is required for a virtual RT entry, a fill request is sent for this entry and the pending mask update buffer remains occupied until a response to the fill request is received and the entry is inserted in the RT table. Further, a fill request is sent when an RT entry reconvergence is timed-out. An ST spill request is 12 bytes and an RT spill request is 16 bytes. Each global memory address space of $32 \times 32 = 1KB$ bytes per warp is reserved for virtual ST and RT Tables. The FIFO and free queues use virtual entry IDs (between 0 and warp-size-1) that are used along with the warp ID to decide the address of spill and fill requests. These virtual IDs are stored as new fields to the physical ST and RT entries. Age bits for RT entries are not virtualized. Each buffer in Figure 5.7 is sized to queue only one entry. As we discuss in Section 5.7, we found the physical sizes of ST and RT

---

1This is essentially to leverage the existing FIFO for the ST and the age bits used for Timeout calculation in the RT.
can be set to 4 and 2 entries respectively with limited impact on performance (see Figure 5.13).

This effectively reduces the storage required per warp by a factor of $5 \times$ compared to the basic implementation which makes the storage requirement comparable to the reconvergence stack. Table 5.2 compares the storage cost of AWARE with that of stack-based reconvergence. Upon a divergent branch, AMD GPUs select the path with fewer active threads first which limits the stack depth to $\log_2(\text{warp size})$ \[5\]. On the other hand, Nvidia’s GPUs push the not-taken entry then the taken entry to the stack, so that execution always starts with the taken entry first \[147\]. Thus, the stack depth can grow up to the warp size. Note that this behavior for Nvidia GPUs was highlighted in prior research \[147\] on old GPU architectures (GT200), and we confirmed using microbenchmarks that, the behaviour is the same on recent architectures (both Kepler and Pascal architectures).

\[2\]The warp size in AMD GPUs is 64 threads which requires 6 entries. However, for the sake of fair comparison, in Table 5.2, we assumed that the warp size is only 32 threads similar to Nvidia’s GPUs.
<table>
<thead>
<tr>
<th>Config</th>
<th>Cost (Bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack (32-entries - Nvidia)</td>
<td>3072</td>
</tr>
<tr>
<td>Stack (5-entries - AMD)</td>
<td>480</td>
</tr>
<tr>
<td>AWARE-basic</td>
<td>7680</td>
</tr>
<tr>
<td>AWARE-virtual (1 ST, 1 RT)</td>
<td>1088</td>
</tr>
<tr>
<td>AWARE-virtual (6 ST, 2 RT)</td>
<td>1696</td>
</tr>
</tbody>
</table>

Table 5.2: Storage Cost in Bits per Hardware Warp.

5.6 Methodology

This section describes our methodology for AWARE evaluation. We implement AWARE in GPGPU-Sim 3.2.2 [11, 138]. We use the TeslaC2050 configuration released with GPGPU-Sim (Table 5.3). However, we replaced the Greedy Then Oldest (GTO) scheduler with a Greedy then Loose Round Robin (GLRR) scheduler that forces loose fairness in warp scheduling as we observed that unfairness in GTO leads to livelocks due to inter-warp dependencies on locks.

Modified GPGPU-Sim code used for this evaluation can be found online [37]. We model insertion and lookup latency to non-virtualized ST and RT entries as 1 cycle. A warp split inserted into ST needs to wait for the next cycle to be eligible for fetch. We use the same benchmarks described in Section 3.4. We also use the same encoding in Table 4.1 in our discussion of the results.

5.7 Evaluation

We limit our evaluation of AWARE to CUDA and OpenCL applications because current OpenMP support relies on linking device code at runtime to a compiled OpenMP runtime library [16] which is not straightforward to emulate in GPGPU-Sim. We compare executing the M2D version on AWARE against executing S0- and M2S on the stack-based reconvergence baseline. Unless otherwise noted, we

3For kernels under study that do not suffer livelocks with GTO, the impact of GLRR on L1 miss rate is minimal compared to GTO with the exception of BH-ST (in Table 3.1) which suffers 7% increase in L1 cache misses [124]. The study of “fair” and efficient warp schedulers is left to future work.

4We only linked OpenMP library calls for synchronization at compile time. Enabling general link time optimizations for OpenMP runtime library requires an engineering effort that is beyond the scope of this paper.
# Compute Units | 14
---|---
warp Size | 32
warp Scheduler | Greedy Then Loose Round Robin
Splits Scheduler | FIFO
Number of Threads / Core | 1536
Number of Registers / Core | 32768
Shared Memory / Core | 48KB
Constant Cache Size / Core | 8KB
Texture Cache Size / Core | 12KB, 128B line, 24-way
Number of Memory Channels | 6
L1 Data Cache | 16KB, 128B line, 4-way LRU.
L2 Unified Cache | 128k/Memory Channel, 128B line, 16-way LRU
Instruction Cache | 4k, 128B line, 4-way LRU
Compute Core Clock | 575 MHz
Interconnect Clock | 575 MHz
Memory Clock | 750 MHz
Memory Controller | out of order (FR-FCFS)
GDDR3 Memory Timing | $t_{CL}=12$ $t_{RP}=12$ $t_{RC}=40$ $t_{RAS}=28$ $t_{RCD}=12$ $t_{RRD}=6$
Memory Channel BW | 4 (Bytes/Cycle)

### Table 5.3: GPGPU-Sim Configuration

**Figure 5.8:** Normalized Kernel Execution Time

configure AWARE to enable delayed reconvergence and with the TimeOut mechanism disabled. Figure 5.8 shows the normalized average execution time for individual kernels. On average, executing the MIMD version on AWARE is on par with executing the $M2S$ version on the stack-based reconvergence baseline (Fig-
Figure 5.9: Normalized Accumulated GPU Execution Time

However, for some kernels such as BH-TB, $M2D$ on AWARE has better performance versus the $M2S$ versions of these kernels run on stack-based reconvergence. This is mainly because AWARE does not require executing additional instructions. Figure 5.10 shows that executing the MIMD version of all kernel run on AWARE leads to a reduced number of instructions executed versus $M2S$ with the sole exception of BH-ST. The reason BH-ST differs is that it behaves similar to a spin lock; the AWARE FIFO-based scheduling mechanism allows a warp split that did not acquire a lock to attempt to acquire it again even before it is released by the other split. This execution pattern repeats as many times as the number of dynamic branch instructions encountered along the not-taken path before the lock is released. Manual transformation of the code to eliminate SIMT deadlock avoids this behaviour, and our compiler elimination algorithm also reduces this behaviour.

Figure 5.12 illustrates the sensitivity of AWARE to the value of TimeOut. Kernels that suffer SIMT deadlocks (e.g., BH-ST) favor smaller TimeOut values as this allows blocked threads to more readily make forward progress and release other threads attempting to enter a critical section. For kernels that do not suffer SIMT deadlocks (e.g., BH-SM), smaller TimeOut values reduce SIMD utilization and lower performance. On average, delayed reconvergence with TimeOut disabled (bars labeled “inf+DR”) achieves the best results. This suggests applying delayed reconvergence whenever a SIMT deadlock is detected and setting Time-

---

5 We simulate the PTX ISA on GPGPU-Sim because SASS is not fully supported. Note that PTX uses virtual registers rather than spilling to local memory diminishing some of the advantage of AWARE over SSDE.
Figure 5.10: Normalized Dynamic Instruction Count

Figure 5.11: Average SIMD Utilization

Figure 5.12: Sensitivity to the Timeout value (in cycles). "inf+DR" refers to a timeout that is infinity but with delayed reconvergence.

Out to a large value such that it is only triggered when there is a high likelihood of an undetected SIMT deadlock. This leaves room for improvement on the SIMT-induced deadlock compiler analysis to report the degree of confidence that a loop
may cause a SIMT-induced deadlock based on the code pattern.

Figure 5.13 illustrates the impact of AWARE virtualization on overall performance. We use AWARE with delayed reconvergence and TimeOut disabled in this experiment. We can see that in the worst case execution time increases by only 16% when we use 4 and 2 physical entries for ST and RT respectively instead of 32 entries; the average is only 5%. Our analysis suggests that the performance overhead is mainly due to the extra traffic caused by the fill and spill requests. For example, using a single entry for both ST and RT with CP-DS kernel increases memory requests by 21% and 15% of this extra traffic miss in L1 cache. Congestion on the miss queues increases MSHR (Miss Status Holding Register) reservation failure rate by a factor of $2.5 \times$. This leads to an increase in the stalls due to structural hazards by 51%. A potential solution that can further reduce performance overhead is a victim cache shared among warps and used to cache spilled entries. Using a central storage for all warps in an SM is motivated by our observation of a disparity in the ST and RT occupancy requirement across different warps at a given execution window. In depth study of this modification is left for future work.

5.8 Related Work

In [121], Ramamurthy describes a modification of the SIMT reconvergence stack when executing lock or unlock instructions to avoid possible deadlocks. However, the proposed solution does not address situations where the locking happens in diverged code [121]. The solution described is limited to mutexes and it applies only
in very restricted cases. In [155], hardware support for a blocking synchronization mechanism on GPGPU is described. The synchronization APIs proposed there, SIMT deadlocks is avoided in restricted cases (similar to [121]). We provide more details on both of these techniques in Chapter 8.

There are many research papers that recognize the performance implications of branch divergence in GPUs [33, 39, 41, 44, 91, 99, 122, 123]. However, less attention has been paid to the implications on functionality. Temporal-SIMT is a hardware proposal that enables more flexible placement of barriers [74] but the use of explicit reconvergence points in Temporal-SIMT can still cause SIMT deadlocks. Dynamic Warp Formation (DWF) [41] enables flexible thread scheduling that could be exploited to avoid SIMT-deadlocks, however, it has higher hardware complexity.

Very recently, Nvidia revealed some information about their Volta architecture [103, 109]. Volta replaces the old stack-based execution model with a new model that enables independent thread scheduling to enable inter-thread synchronization. Figures 12 and 13 in the Nvidia blog [103] suggests that the observable behaviour of Volta is very similar to AWARE with concurrent multi-path execution enabled (Multi-Path execution is discussed in Chapter 7). The closest related Nvidia patent is a 2016 patent that describes a notion of convergence barriers [34]. We provide detailed discussion about this patent in Chapter 8, Section 8.1.

5.9 Summary, Conclusion and Future Directions

In this chapter, we presented AWARE; a MIMD-Compatible reconvergence mechanism that avoids most of the limitations inherent in the SSDE compiler-only approach. The chapter discussed AWARE operation, implementation, and its interaction with synchronization.

Unlike the compiler only approach, discussed in Chapter 4, AWARE does not require changing the application CFG. Thus, it does not have the synchronization scope limitations and debuggability challenges associated with SSDE. It also supports barriers in divergent code. AWARE limits the architectural changes to the divergence unit. AWARE provides a realistic example that lays the foundation to consider MIMD-compatibility as a design goal for SIMT hardware implementa-
Future research could explore static and/or dynamic delayed and timed-out reconvergence mechanisms as these appear to have potential to further optimize performance by varying the selection of reconvergence points on the granularity of individual branches. Specifically, by adapting reconvergence point selection it may be possible to improve SIMD utilization [137] and/or caching behaviour [116] in divergent applications in general.
Chapter 6

BOWS: Back-Off Warp Spinning

In the previous chapters of this thesis, we tackled some of the limitations of the SIMT execution model that makes correct implementations of inter-thread synchronization on current SIMT machines challenging and unreliable. In this chapter, we focus on the performance side of the SIMT model implications on inter-thread synchronization.

Overheads of fine-grained synchronization have been well studied in the context of multi-core CPU architectures [36, 80, 143, 161]. However, the scale of multi-threading and the fundamental differences in the architecture in SIMT machines hinders the direct applicability of the previously proposed CPU solutions (more details in Section 7.7). In SIMT machines, barrier synchronization overheads have been recently studied [81, 82]. These studies proposed warp scheduling policy that accelerate warps that have not yet reached a barrier to enabled other warps blocked at the barrier to proceed. However, fine grained synchronization, with busy-wait synchronization, is a fundamentally different problem. In barrier synchronization, warps that reach a barrier are blocked and do not consume issue slots. However, with busy-wait synchronization, threads that fail to acquire a lock spin compete for issue slots and, in the absence of coherence L1 caches, memory bandwidth.

Yilmazer and Kaeli [155] quantified the overheads of spin-locks on GPUs and proposed a hardware-based blocking synchronization mechanism called hierarchal queue locking (HQL). HQL provides locks at a cache line granularity by adding
flags and pointer meta-data for to each L1 and L2 block, which can be in one of six states. Negative acknowledgments are used when queues are filled and in certain race conditions. An acquire_init primitive is added to the application to setup a queue. While HQL achieves impressive performance gains when an application uses a small number of locks relative to threads, it can experience a slowdown when using a large number of locks concurrently. Moreover, HQL adds significant area to the caches and requires a fairly complex cache protocol. While Yilmazer and Kaeli noted the potential for synchronization aware warp scheduling to help improve HQL, a detailed investigation was left to future work. By judiciously modifying warp scheduling, this paper shows how to effectively approximate the benefits of queue-based locking without the complexity and overhead of directly implementing queues.

Criticality-Aware Warp Acceleration (CAWA) [73] uses run-time information to predict critical warps. Critical warps are those that are slowest in a kernel and as they determine execution time CAWA prioritizes them. CAWA estimates warp criticality using a criticality metric that predicts which warp will take longer time to finish. CAWA outperforms greedy-than-oldest (GTO) warp scheduling across a range of traditional GPGPU workloads [73]. However, CAWA can reduce performance for busy-wait synchronization code as its criticality predictor tends to prioritize spinning warps.

We propose Back-Off Warp Spinning (BOWS), a scheduling policy that prevents spinning warps from competing for scheduler issue slots. BOWS approximates software back-off techniques used in multi-threaded CPU architectures [6], which incur limitations when directly applied to GPUs (Figure 6.2). Warp prioritization in stack-based SIMT architectures is complicated by the fact that some threads within a warp may hold a lock while others do not. In BOWS warps that are about to execute a busy-wait iteration are removed from competition for scheduler issue slots until no other warps are ready to be scheduled. On GPU kernels using busy-wait synchronizations BOWS achieves a speedup of 1.5× and energy savings of 1.6× versus CAWA.
6.1 Sensitivity to Warp Scheduling

In Section 1.2, we discussed the overheads of busy-wait synchronization on recent SIMT hardware. In this section, we consider the impact of warp scheduling policies. Greedy then Oldest (GTO) scheduling \[124\] selects the same warp for scheduling until it stalls then moves to the oldest ready warp. Older warps are those with lower thread IDs. GTO typically outperforms Loose Round Robin (LRR) \[124\]. In CAWA, warp criticality is estimated as: 
\[
n_{\text{Inst}} \times w.CPI_{\text{avg}} + n_{\text{Stall}}
\]
where \(n_{\text{Inst}}\) is an estimate of remaining dynamic instruction count (based on direction of branch outcomes), \(w.CPI_{\text{avg}}\) is per-warp CPI, and \(n_{\text{Stall}}\) is the stall cycles experienced by a warp. Critical warps are prioritized.

Figure 6.1 plots the distribution of lock acquire attempts in lock-based synchronization and the wait exit attempts in wait and signal based synchronization (benchmarks and methodology described in Section 7.4) using LRR, GTO, and CAWA scheduling policies. The figure also shows the distribution of whether the lock acquire failure is because the lock is held by a thread within the same warp (i.e., intra-warp lock fail) or in a different warp (i.e., inter-warp lock fail). Most lock failures are due to inter-warp synchronization. The figure shows that inter-warp conflicts are significantly influenced by the warp scheduling policy.

Figure 6.2 plots execution time of the hashtable insertion code in Figure 1.2a augmented with the software-only backoff delay code in Figure 6.2a running on
1. `clock_t start = clock();
2. clock_t now;
3. for(;;) {
4.   now = clock();
5.   clock_t cycles = now>start? now - start:
6.   now s[0xffffffff - start];
7.   if(cycles >= DELAY_FACTOR*blockIdx.x)
8.     break;
9. }

(a) Backoff Delay Code*

(b) Execution Time.

Figure 6.2: Software Backoff Delay Performance in GPUs. *omp_set_lock GPU implementation for OpenMP 4.0 [15].

GTX 1080 hardware. The results suggest that adding a backoff delay to a spin-lock degrades performance on recent GPUs. The reason is that, except at very high levels of contention, the benefits of reduced memory traffic appear insufficient to make up for wasted issue slots executing the delay code itself.

6.2 BOWS: Backoff Warp Spinning

To avoid wasted issue slots we propose Back-Off Warp Spinning (BOWS), a hardware scheduling mechanism that reduces the priority of spinning warps. We present BOWS assuming synchronization loops have been identified by programmer, compiler or DDOS (described in Section 6.3).

6.2.1 BOWS scheduling policy

The scheduling policies examined in Section 6.1 suffer from two limitations:

- The scheduler may prioritize spinning warps in the competition for issue slots over other eligible non-spinning ones. This slows down the progress of non-spinning warps. In cases when these non-spinning warps are holding locks, this decision also slows down the forward progress of spinning warps.

- The scheduler may return back to the same spinning warp too early even if it was at the bottom of the scheduling priority because other warps are stalling on data dependencies.

BOWS avoids these issues by modifying an existing warp scheduling policy as follows:
• It discourages warps from attempting another spin loop iteration by inserting the warp that is about to execute another iteration into the back of the warp scheduling priority queue. Warps in this state are called Backed-off Warps. Once a warp in the backed-off state issues its next instruction its priority reverses to normal and it leaves the backed-off state.

• It sets a minimum time interval between the start of any two consecutive iterations of a spin loop by the same warp. Warps that are about to start a new spin loop iteration prior to the end of their interval are not eligible for scheduling.

BOWS requires that Spin-Inducing Branches (SIBs) have been identified. SIBs are the backward branch of each spin loop. Once a warp executes a SIB, the scheduler control unit triggers BOWS’ logic.
BOWS Operation

BOWS works as follows: Once a warp exits its backed-off state, a pending back-off delay register is initialized to the back-off delay limit. The warp then continues execution normally with the pending back-off delay register decremented every cycle. If the warp executes a SIB it cannot issue its next instruction until its back-off delay is zero. The back-off delay value can be determined through profiling or tuned adaptively at run time.

Figure 6.3 shows an example of BOWS operation for warp W0 containing four threads for the code in Figure 6.6a. Backward branch 0x098: %p3 bra BB2; has been identified as a SIB. Scheduling priority is shown in the top of Figure 6.3. Initially, W0 has high priority 1. Once W0 encounters a spin-inducing branch 2, it is pushed to the back of the priority queue and marked as backed-off (shaded in Figure 6.3). W0 is scheduled when other warps are stalling (e.g., on memory accesses for line 6 in Figure 1.2a) and executes the lock-acquire atomic compare and swap instruction (Figure 6.6a PC=0x030). At this point 3 three actions are taken: First, the warp loses its backed-off state; second, the warp priority reverts to normal; and third a back-off delay value is stored in the warp pending back-off delay register 4. Two threads of W0 successfully acquire the lock and proceed to the critical section while the other two threads fail 5. Threads re-converge at the setp instruction and execute the spin-inducing branch. The two threads that executed the critical section exit the spin loop while the others proceed to another iteration. Once the spin-inducing branch is executed, the warp enters backed-off state and is pushed to the end of the priority queue 6. As the duration of the critical section is larger than that of the back-off delay limit W0’s back-off delay is already zero and so W0 is eligible for scheduling. After W0 is scheduled it executes the lock acquire and the two remaining threads in the spin loop again fail to acquire a lock 7. The two threads immediately proceed to another iteration of the spin-loop 8. However, once W0 enters the backed-off state, they cannot be scheduled until the pending back-off delay is zero 9. Once the pending back-off delay is zero, the W0 is eligible for scheduling 10.
for each Execution Window of T cycles:
  if (SIB Instructions > FRAC1 * Total Instructions)
      Delay Limit += Delay Step
  if ((Total Instructions) / (SIB Instructions)) <
      FRAC2 * (Prev. Total Instructions) / (Prev. SIB Instructions))
      Delay Limit -= 2 * Delay Step
  if (Delay Limit > Max Limit) Delay Limit = Max Limit
  if (Delay Limit < Min Limit) Delay Limit = Min Limit

Values used in Evaluation:
T=1000 cycles, FRAC1=0.05, FRAC2=0.8,
Delay Step=250 cycles, Max Limit = 10000 cycles,
Min Limit = 1000 cycles

Figure 6.4: Adaptive Back-off Delay Limit Estimation.

Adaptive Back-off Delay Limit

A small back-off delay may increase spinning overheads while a large back-off delay may throttle warps more than necessary. We adaptively set the delay by trying to maximize $(\text{Useful Instructions} / \text{Spinning Overheads})$ over a window of execution. We use

\[
\frac{\text{Total Inst.}}{\text{SIB Inst.}} = \frac{\text{Useful Inst.} + \text{SIB Inst.} \times \text{avg. Spin Overhead}}{\text{SIB Inst.}}
\]

as a rough estimate. As the average spin overhead is almost constant across the execution of the same kernel the ratio of the total number of instructions is proportional to $\frac{\text{Useful Instructions}}{\text{Spinning Overheads}}$.

The pseudo code in Figure 6.4 summarizes our adaptive back-off delay limit calculation. This algorithm is applied over successive time windows. During the current window the adaptive back-off delay estimation algorithm computes the back-off delay limit to use during the next window. Initially, the scheme attempts to increase the back-off delay limit by a fixed step as long as a non-negligible ratio of dynamic spin-inducing branches is executed. However, if the ratio of $\frac{\text{Total Instructions}}{\text{SIB Instructions}}$ in the current execution window is considerably smaller than the ratio in the previous window the back-off delay limit is decremented by a double step. Finally, lower and upper limits are applied to the back-off delay limit.

6.3 DDOS: Dynamic Detection of Spinning

It is possible to identify spin loops when explicit busy-wait synchronization APIs are used. The compiler can then translate a lock acquire API into a busy wait loop
(a) Two Nested Locks (ATM [43] and CP [20, 43]).

(b) Global Locking (TSP [119, 129]).

c) Wait and Signal (BH-ST [24]).

Figure 6.5: Examples of Inter-Thread Synchronization Patterns used in GPUs (See Section 7.4 for more details).

with the backward branch of the loop flagged as a spin inducing branch. However, such APIs are not available in current SIMT programming models. Therefore, in this section, we describe a mechanism for dynamically detecting SIBs.

Current GPU programmers write synchronization code tailored to their specific application scenario. For example, Figure 6.5a shows an implementation of two nested locks that avoid SIMT-induced deadlocks from ATM. Figure 6.5b shows an implementation of a global lock from TSP where the execution of the critical section is serialized across threads from the same warp. Figure 6.5c shows busy-wait synchronization from the ST kernel in BH that implements a wait and signal synchronization rather than a lock. A thread waits in a spin loop for a condition set by another thread.

The large variety of synchronization patterns makes it challenging to detect
busy-wait synchronization statically or to introduce primitives that support all use cases and avoid SIMT-induced deadlocks [38]. Thus, we propose a hardware mechanism, Dynamic Detection of Spinning (DDOS), to detect spinning warps. DDOS seeks to identify Spin-Inducing Branches (SIBs). We define a SIB as a backward branch that maintains the spinning behaviour. To identify a SIB, DDOS first makes a prediction regarding whether each warp is currently in a spinning state or not.

As noted by Ti et al. [80], a thread is spinning between two dynamic instances of an instruction if it executes the instruction and later executes the same instruction again (e.g., in another loop iteration) without causing an observable change to the
net system state (i.e., to its local registers or to memory). Ti et al. [80] proposed a thread spinning detection mechanism for multi-threaded CPUs that tracks changes in all registers. Directly applying such a technique to a GPU would be prohibitive given the large register files required to support thousands of hardware threads. Instead DDOS employs a speculative approach.

DDOS detects busy-wait loops in two steps. First, it detects the presence of a loop. DDOS does this by tracking the sequence of program counter values of a warp. Second, DDOS speculates whether a loop identified in the first step is a busy-wait loop or a normal loop. To distinguish these cases it leverages the observation that typically in normal loops found in GPU code an induction variable changes every iteration. Moreover, this induction variable typically contributes to the computation of the loop exit condition. In NVIDIA GPUs the loop exit condition and the divergence behaviour of a thread are typically determined using a set predicate instruction (available both in PTX and SASS). For each thread in a warp, the set predicate instruction compares two source registers and writes the result to a boolean destination register. The boolean values are typically used to predicate execution of both normal and branch instructions (e.g., instructions at address 0x090 and 0x098 in Figure 6.6a). In normal (none busy-wait) loops, the value of at least one source register of the set predicate (setp) instruction(s) that determines the loop exit condition changes each iteration. In a ‘for’ loop, one of these registers would be the loop counter. DDOS approximates the condition tested by Ti et al. [80] by tracking only the values of source registers of the set predicate instructions in determining whether a loop is a normal loop (i.e., setp source register values change) or a busy-wait spin loop (setp source register values do not change).

6.3.1 DDOS Operation

Conceptually, the spin loop detection step of DDOS works as follows: Each warp has two shift registers, a Path History Register and a Value History Register (Figure 6.6b). These registers track the execution history of the first active thread in the warp. We refer to this thread as the profiled thread. The Path History Register

---

1 AMD Southern Islands ISA has an equivalent vector compare instruction (v.comp) [5].
tracks program counter values of setp instructions. The Value History Register tracks the values of the source registers of setp instructions. To reduce storage overhead we hash program counter and source operand values before adding them to the Path History and Value History Registers. As elaborated upon in Section 6.3.3, the Value History Register is implemented in the execution stage. DDOS’ examines entries in Path and Value History Registers looking for repetition. If it finds sufficient repetition DDOS classifies the profiled thread as being in a spinning state.

Figure 6.6 illustrates operation of Path and Value History Registers on PTX assembly examples with (Figure 6.6a) or without (Figure 6.6c) busy wait code. Figure 6.6a is equivalent to Figure 1.2a. In Figure 6.6a assume the first active thread is executing the setp instruction at PC = 0x038. In the busy-wait example in Figure 6.6b, the program counter is first hashed using:

\[(PC - PC_{kernel\ start})/\text{Instruction\ Size}] \mod m\]

where \(PC_{kernel\ start} = 0x000\), \(m = 4\) and Instruction Size = 8. The result (0x7) is inserted into the Path History Register 1a. In parallel, the source operand values of the setp instruction are hashed and added to the Value History Register. We assume the profile thread fails to acquire the lock so that \%r15 is ‘1’. Only the least significant \(k\)-bits (here \(k = 4\)) are used 1b. To detect repetition DDOS keeps track of two other values, Match Pointer and Remaining Matches. The Match Pointer identifies which \(m\)-bit (\(k\)-bit) portion of the Path (Value) History Register to compare new insertions against. For each insertion into the path (value) history registers, the entry before the match pointer is compared with the new entry. If they are equal, a loop is detected. To enable better selectivity DDOS requires multiple consecutive loop detections before identifying a spin inducing loop. To facilitate this the remaining matches register tracks the number of remaining matches required.

Continuing the example in Figure 6.6b, eventually the warp executes the setp instruction at PC=0x90 in Figure 6.6a. The entries in both shift registers are (logically) shifted to the right and new values inserted to their left. No match is found between the new entry (0x2) and the entry before the match pointer (0x7) 2a. As the profiled thread fails to acquire the lock \%r21 remains ‘0’. Thus, the value his-

---

2PTX is Nvidia GPU virtual assembly [105].
3We discuss other hashing techniques in Section 6.3.2.
tory register is updated with two 4-bit zero values \( 2b \). When the warp reaches \( PC=0x038 \) again we assume the profiled thread again fails to acquire the lock leading to a match in both path and value histories \( 3 \). Once a match is detected, the match pointer is fixed and the remaining matches value is initialized to \((\text{matchpointer} - 1)\). Once the warp reaches the \texttt{setp} instruction at \( PC=0x090 \) again an additional match is found \( 4 \). Since the remaining matches value is now zero, the warp is identified as in a spinning state. After the profiled thread successfully acquires the lock the execution of the \texttt{setp} instruction at \( PC=0x040 \) leads to a mismatch in the value history and the warp loses its spinning state \( 5b \).

Next, we describe how DDOS identifies Spin-inducing Branches (SIBs). The key is that, if a backward branch is executed by a warp in a spinning state, it is likely spin-inducing (i.e., leads to a new iteration in the busy-wait loop).

To detect SIBs DDOS employs a spin-inducing branch prediction table (SIB-PT). The SIB-PT, shown in Figure 6.6b, is shared between warps executing on the same SM. The SIB-PT maintains a confidence value for each branch under consideration. When a warp is in a spinning state and it executes a backward branch that is not in the SIB-PT then it is added with a confidence value of 1. If the branch is in the SIB-PT, its confidence value is incremented. Once the confidence reaches a threshold the branch is identified as a spin-inducing branch. To guard against accumulated path and value hash aliasing errors that could happen over an extended period of execution, a nonzero confidence value for a branch that is not yet confirmed as spin-inducing is decremented every time the branch is taken by a warp that is currently classified as non-spinning. The threshold is a constant fixed for a given architecture (determined empirically).

Returning to the example in Figure 6.6b, initially, the SIB-PT is empty. Once the warp executes the backward branch at \( PC=0x098 \) while in the spinning state (i.e., after \( 4 \) and before \( 5 \)) the branch is added to the SIB-PT with its confidence set to ‘1’.

Next, we briefly explain DDOS operation with a normal loop example. The
PTX code in Figure 6.6c is the assembly of a ‘for’ loop in k-means [26]. The backward branch is at 0x060 and its associated setp is at 0x058. The first source operand %r20 represents the ‘for’ loop induction variable that is incremented by one every iteration (at 0x050), while %r15 is a copy of the kernel input indicating the number of loop iterations. The PC of the setp instruction is hashed to (0x2) and inserted into the Path History Register every time the instruction is executed (6a, 7a, and 8a). In contrast to the busy-wait case, the contents of %r20 changes each iteration causing a mismatch with every insertion to the value history register (7b and 8b).

6.3.2 DDOS Design Trade-offs

DDOS as described in Section 6.3.1 has different design parameters to tune. These are the hashing function and width ($m$ and $k$), the confidence threshold ($t$), and the number of entries in the history shift register ($l$). We evaluate the impact of these parameters on the following metrics: (1) Average True Spin Detection Rate (TSDR), which is the percentage of spin-inducing branches accurately identified by DDOS; (2) Average False Spin Detection Rate (FSDR), which is the percentage of non-spin-inducing branches incorrectly classified as spin-inducing; and (3) Avg. Detection Phase Ratio (DPR), which is the average ratio of the detection phase duration of a branch to the cycles executed from the first encounter to the last encounter of the branch. The detection phase duration of a branch measures how many cycles were required to confirm a branch as a spin-inducing branch after its first encounter. For spin-inducing branches it is preferable to have a short detection phase. For each branch, these metrics are averaged over the different SMs that execute the branch, and over the different launches of the kernel that include the branch. For ground truth, we consider branches that are used to implement busy-wait synchronization as true spin-inducing branches.

Table 6.1 shows the sensitivity of these metrics to the different design parameters averaged over all our benchmarks (see Section 7.4 for details).

**Hashing Function:** The top sub-table in Table 6.1 studies the impact of XOR and MODULO hashing. In XOR hashing, the values inserted into the path register are hashed as follows (PC[m-1:0] xor PC[2m-1:m] xor PC[3m-1:2m] ... xor PC[31:32-
Sensitivity to the hashing function “h” at \( t=4 \) and \( l=8 \)

<table>
<thead>
<tr>
<th>h</th>
<th>Avg. TSDR</th>
<th>Avg. DPR</th>
<th>Avg. FSDR</th>
<th>Avg. DPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR, ( m=k=4 )</td>
<td>1</td>
<td>0.041</td>
<td>0.016</td>
<td>0.006</td>
</tr>
<tr>
<td>XOR, ( m=k=8 )</td>
<td>1</td>
<td>0.041</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>MODULO, ( m=k=4 )</td>
<td>1</td>
<td>0.041</td>
<td>0.17</td>
<td>0.014</td>
</tr>
<tr>
<td>MODULO, ( m=k=8 )</td>
<td>1</td>
<td>0.041</td>
<td>0.104</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Sensitivity to the Hashed Path/Value Width “m/k” at \( t=4, l=8 \), and \( h=XOR \)

<table>
<thead>
<tr>
<th>m/k</th>
<th>Avg. TSDR</th>
<th>Avg. DPR</th>
<th>Avg. FSDR</th>
<th>Avg. DPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0.042</td>
<td>0.078</td>
<td>0.062</td>
</tr>
<tr>
<td>3</td>
<td>0.983</td>
<td>0.074</td>
<td>0.012</td>
<td>0.008</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.041</td>
<td>0.016</td>
<td>0.006</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0.041</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sensitivity to Confidence Threshold “t” at \( m=k=4, l=8 \), and \( h=XOR \)

<table>
<thead>
<tr>
<th>t</th>
<th>Avg. TSDR</th>
<th>Avg. DPR</th>
<th>Avg. FSDR</th>
<th>Avg. DPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0.03</td>
<td>0.027</td>
<td>0.016</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.041</td>
<td>0.016</td>
<td>0.006</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0.075</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>12</td>
<td>0.992</td>
<td>0.105</td>
<td>0.002</td>
<td>0.003</td>
</tr>
</tbody>
</table>

Sensitivity to the History Registers Length “l” at \( t=4, m=k=8 \), and \( h=XOR \)

<table>
<thead>
<tr>
<th>l</th>
<th>Avg. TSDR</th>
<th>Avg. DPR</th>
<th>Avg. FSDR</th>
<th>Avg. DPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0.625</td>
<td>0.032</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0.041</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sensitivity to Time Sharing of History Registers “sh” at \( l=8, t=4, h=XOR \), and epoch=1000

<table>
<thead>
<tr>
<th>sh</th>
<th>Avg. TSDR</th>
<th>Avg. DPR</th>
<th>Avg. FSDR</th>
<th>Avg. DPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, ( m=k=4 )</td>
<td>1</td>
<td>0.041</td>
<td>0.016</td>
<td>0.006</td>
</tr>
<tr>
<td>0, ( m=k=8 )</td>
<td>1</td>
<td>0.041</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1, ( m=k=4 )</td>
<td>0.642</td>
<td>0.211</td>
<td>0.033</td>
<td>0.023</td>
</tr>
<tr>
<td>1, ( m=k=8 )</td>
<td>0.642</td>
<td>0.211</td>
<td>0.026</td>
<td>0.003</td>
</tr>
</tbody>
</table>

Table 6.1: Spin Detection Sensitivity to Design Parameters.

\( m \)), where PC is the program counter at the execution of a \texttt{setp} instruction. The value register XOR hashes are computed similarly but using the source registers in the \texttt{setp} instructions. In MODULO hashing, values are hashed by considering only the least significant \( m (k) \) bits of the value (as in Figure 6.6). XOR hashing considerably reduces false detections compared to MODULO hashing. With 8-bits hashing width, the XOR hashing has a zero false detection rate. False detections occur in Merge Sort and Heart Wall with MODULO hashing due to loops with power-of-two induction variable increments larger than \( 2^k \).
**Hashing Width:** The impact of the hashing width is quantified in the second sub-table in Table 6.1. A 2-bit path and value width leads to aliasing that leads to 7.8% false detection rate. With three the aliasing impact is smaller and eight bits are enough to eliminate false detections with XOR hashing.

**Confidence Threshold:** The third sub-table in Table 6.1 shows that as the confidence threshold \( t \) increases, the false detection rate decreases but the detection phase ratio increases for true detections. With \( t = 12 \) some SMs fail to confirm a spin-inducing branch (e.g., TB kernel of BH).

**Hashing Registers Length:** The fourth sub-table in Table 6.1 shows the sensitivity to the history length \( l \), which determines the number of `setp` instructions DDOS can track. A length of two instances fails to capture any repetitiveness in history. A history of four instances fails to capture the spin loops in one benchmark (NW: Needleman-Wunsch). This benchmark has two spin loops in two different kernels but each of them are launched several times. The loop involves four `setp` instructions, and thus DDOS needs at least five entries in its history registers to detect their spin behaviour.

**Time Sharing of History Registers:** As the goal of DDOS is to classify static instructions as either SIB or not tracking path and value histories for all warps seems unnecessary. The results of time-sharing a single set of path and value history registers among different warps in an SM is shown in the last sub-table in Table 6.1. Here a warp uses the history registers for a certain predetermined and fixed interval (1000 cycles), then another warp uses them. Time sharing reduces detection accuracy as the profiled warp may not complete a full spin twice within its time sharing interval and thus some SIBs may not be detected. With a single warp, time sharing leads to longer detection phase as the SIB-PT. It may be possible to find a “sweat spot” between tracking all warps and only one. We this tradeoff to future work.

In our evaluation, we use “h=XOR, t=4, m=k=8, l=8, and time sharing disabled”. The total storage per warp for both the path and value history registers is 192-bits. In our benchmarks, the maximum number of confirmed spin-inducing branches was three. However, the maximum number of concurrent entries in the SIB-PTX was 9 entries (the next maximum was only four). A conservative 16-entry SIB-PT requires 560 bits of storage per SM.
6.3.3 DDOS integration with BOWS

Figure 6.7 illustrates BOWS’ combined with DDOS. **Warp Scheduling:** BOWS modifies the warp scheduling and execution stages. We found that strict GTO scheduling (without BOWS) can lead to livelocks on two of our benchmarks (HT and ATM). To avoid this, we modify GTO to rotate the age priority periodically (every 50,000 cycles in our evaluation). Arbitration logic first checks whether the last issued warp is ready to issue its next instruction 1. If the last issued warp is not ready, the oldest ready warp that is not backed-off is selected 2. If no such warp is available the backed-off queue is checked. A warp is added to backed-off queue after executing a SIB. A warp in the backed-off queue can be scheduled only if it is both ready and its back-off delay is zero 3. If the arbitration selects such a warp it is removed from the backed-off queue. The “Backed-off” field for the warp is set to false and the “Pending Back-off Delay” is initialized to the back-off delay limit value when the warp exits the backed-off state 4.

**ALU Execution Stage:** Path and value history are updated during execution of `setp` instructions 5, 6. Current GPUs already supports instructions such
“shuffle” which allow threads within the same warp to access each other’s registers [111]. The underlying hardware can be used to select the source registers of the first active thread. If the warp executes a backward branch, then it looks up the SIB-PT 7. If the branch is predicted to be a spin-inducing branch the warp enters the backed-off state 9 and is pushed to the end of the backed-off queue.

6.4 Methodology

We implement BOWS in GPGPU-Sim 3.2.2 [11, 138]. We use GPGPU-Sim GTX480 for both GPGPU-Sim and GPUWattch for performance and energy evaluation. In Section 6.5.4, we report results for a Pascal GTX1080Ti configuration that has a correlation of about 0.85 for Rodinia to estimate the impact of BOWS on the performance of newer generations of GPUs. We evaluate the impact of BOWS on three scheduling policies; GTO, LRR, and CAWA. We use BOWS and DDOS design parameters detailed in Table 5.3.

For evaluation, we use Rodinia 1.0 [26] for synchronization free benchmarks (see Section 6.5.2). We use the kernels described below for kernels displaying different synchronization patterns.

BH: BarnesHut is an N-body simulation algorithm [24]. Its Tree Building (TB) kernel uses lock-based synchronization [24]. The kernel is optimized to reduce contention by limiting the number of CTAs and using barriers to throttle warps before attempting a lock acquire. Its sort kernel (ST) uses a wait and signal synchronization scheme. We run BarnesHut on 30,000 bodies.

CP: Clothes Physics perform cloth physics simulation for a T-shirt [20]. Its Distance Solver (DS) kernel lock-based implementation uses two nested locks to control updates to cloth particles.

HT: Chained HashTable uses the critical section shown in Figure 1.2a. We run 3.2M insertions by 40K threads on 1024 hashtable buckets.

ATM: An bank transfer between two accounts [43]. It uses two nested locks. We run 122K transactions with 24K threads on 1000 accounts.

NW: Needleman-Wunsch finds the best alignment between protein or nucleotide sequences following a wavefront propagation computational pattern. We implemented the lock-based algorithm in [78] which uses two kernels NW1 and NW2.
that perform similar computation while traversing a grid into opposite directions.

**TSP:** Travelling Salesman. We modified the CUDA implementation from [119] to use a global lock when updating the optimal solution. We run TSP on 76 cities with 3000 climbers.

### 6.5 Evaluation

Figure 6.14 shows normalized execution time and energy consumption on busy-wait synchronization kernels. Results are normalized to LRR. BOWS uses adaptive back-off delay. It uses DDOS for detecting spin loops. The design parameters are shown in Table 5.3).

Figure 6.14 shows that BOWS consistently improves performance over different baseline scheduling policies with a speedup of 2.2×, 1.4×, and 1.5× and energy savings of 2.3×, 1.7×, and 1.6× compared to LRR, GTO, and CAWA respectively.

BOWS has minimal impact on TB because TB’s code uses a barrier instruction to limit the number of concurrently executing warps between lock acquisition iterations. We note this barrier approach is fairly specific to TB. For example, it requires at least one thread from each warp to reach the barrier each iteration. Also,
the lack of adaptivity of this software-based barrier approach can be harmful even where it can be applied (would lead to a 28x slowdown if applied to HT, measured on hardware - Pascal GTX1080). ST shows 17.8% energy improvements with BOWS (Figure 6.14b) as it reduces dynamic instruction count but does not exhibit performance improvement because the performance is limited by memory latency. In TSP, the synchronization instructions consume <0.03% of the total number of instructions, thus synchronization code is not the dominant factor in execution time. Large back-off delay values may unnecessarily block execution leading to performance degradation (see TSP results in Figure 6.9).

For the NW kernels, the progress of younger warps is dependent on older warps finishing their execution. Therefore, NW prefers GTO scheduling over LRR as it gives priority to older warps. HT with the GTO scheduler runs into a pathological scheduling pattern where it prioritizes spinning warps which significantly reduce performance. BOWS eliminates such problems by deprioritizing spinning warps.

### 6.5.1 Sensitivity to Back-off Delay Limit Value

The following results use the GTX480 configuration with GTO as the baseline policy for BOWS. Figure 6.10 shows the average distribution of warps at the scheduler in terms of their status (backed-off or not). The first bar is GTO. The remaining bars
**Figure 6.10:** Distribution of Warps at the Scheduler. From left to right, GTO without BOWS, GTO with BOWS with delay limit in cycles 0, 500, 1000, 3000, 5000, Adaptive.

are for BOWS as the back-off delay limit value increases. The last bar to the right is BOWS with adaptive back-off delay limit. The figure shows how BOWS impacts warp scheduling. The back-off delay is not effective until reaching a threshold unique to each benchmark. The reason is that the back-off delay sets a minimum duration between two successive iterations of a spin loop. If warps already consume a time that is larger than the back-off delay limit before they attempt another iteration, then the back-off delay has no observable effect (recall the discussion of Figure 6.3). The effective back-off delay value depends upon how many instructions are along the failure path in the busy-wait code, how many warps are running and how much memory contention there is.

Figure 6.11 shows the distribution of Lock acquire and wait status. The behaviour aligns with the percentage of warps that are backed-off in Figure 6.10. This data elucidates performance gaps in some benchmarks – particularly, HT, ATM, and NW – between the different scheduling policies. For example, in HT BOWS reduces the lock failure rate by $10.8 \times$ compared to GTO.

Figure 6.12a shows the impact of BOWS on the dynamic instruction count. On average BOWS reduces dynamic instruction count by a factor of $2.1 \times$ compared to GTO. Figure 6.12b shows that BOWS also reduces the number of L1D memory transaction by $19\%$ compared to GTO. One of the side effects of BOWS is that it
Figure 6.11: Distribution of Warps at the Scheduler. From left to right, GTO without BOWS, GTO with BOWS with delay limit in cycles 0, 500, 1000, 3000, 5000, Adaptive.

Figure 6.12: BOWS Impact on Dynamic Overheads.

increases SIMD efficiency for some benchmarks. For example, BOWS improves HT and ATM SIMD efficiency by $3.4 \times$ and $1.85 \times$ respectively compared to GTO. In ST, the significant reduction of the number of spin iterations (see Figure 6.12a) biases the SIMD calculation results as the benchmark spends more time in executing the divergent code rather than spinning, and hence the reduction in SIMD efficiency.
6.5.2 Sensitivity to Detection Errors

Note that with the XOR hashing configuration we do not have any false detections. Thus, the results of Synchronization-Free benchmarks are identical to the baseline. Figure 6.13 reports the results of Synchronization-Free benchmarks under the MODULO hashing. For Synchronization-Free benchmarks, BOWS is expected to perform identically to the baseline under perfect spin detection. Only two applications from Rodinia have false detections with MODULO hashing, Merge Sort (MS) and Heart Wall (HL). In both of these applications, false detections were due to ‘for’ loops with a large power of two induction variable increment that is not reflected in the least significant 8-bits of `step` source registers. In this evaluation, we use an 8-bit hash width for the path and value registers. On average, over Rodinia’s 14 benchmarks, BOWS with a 5000 cycles back-off delay and MODULO hashing downgrades GTO performance by only 2.1% on these synchronization free applications. However, for MS, BOWS with MODULO has and a large backoff delay downgrades performance versus GTO significantly.

Note that it is possible to come up with a code example that may result in false detection regardless of the hashing decision. For example, consider the case of a loop that exits at the 10th iteration. However, the exist condition does not compare the induction variable with 10 directly, and instead, an intermediate value is
computed (e.g., \(i \% 10\)) and compared with zero. This effectively hash the induction variable of the loop and tricks DDOS into detecting such loop as a spin loop. We did not find such code examples in our evaluated benchmarks.

### 6.5.3 Sensitivity to Contention

Figure 6.15 uses the hashtable benchmark to study BOWS sensitivity to contention. A small number of hashtable buckets indicate higher contention. The figure shows that BOWS provides a speedup of up to \(5 \times\) at high contention and down to \(1.2 \times\) at low contention. Similarly, the dynamic instruction count savings ranges from \(3.7 \times\) to \(1.3 \times\). Figure 6.15b also includes data, “Ideal Block Inst. Count”, that serves as a proxy for how HQL \([155]\) might perform on this workload. This curve shows the instruction count assuming locks do not require multiple iterations to acquire. The difference between the two curves thus represents the overhead introduced by BOWS versus an ideal queuing lock system. As we can see, the benefits of an (idealized version of) HQL appear to diminish as the number of hash buckets increases.
6.5.4 Pascal GTX1080Ti Evaluation

To evaluate the impact of BOWS on recent architectures, we configured GPGPU-sim and GPUWattch to model GTX1080Ti (the configurations are currently available in GPGPU-sim GitHub repository). We evaluate the same benchmarks with the same inputs used for GTX480. BOWS consistently improves performance over different baseline scheduling policies with a speedup of $1.9 \times$ on LRR, $1.7 \times$ on GTO, and $1.5 \times$ on CAWA.

One observation is that on Pascal, except for DS, the behavior is flat across the different baseline scheduling policies. The reason is that most of the input data sets for the workloads we run are set to fully utilize (without oversubscribing) the Fermi GPU but they under-utilize Pascal. Pascal has almost double the number of cores compared to Fermi (Table 5.3). Thus, on Fermi, each core has many warps to choose from and the scheduling policy is of a great impact. However, in Pascal, each core will have about half the number of warps distributed on four warp schedulers instead of two. Thus, the number of warps available at each scheduler in Pascal is one fourth that in Fermi making the baseline scheduling policy less important (e.g., unlike the case with Fermi on NW and HT benchmarks). DS, on the other hand, is oversubscribed in the Fermi configuration and the number of concurrently running CTAs is limited to four due to the number of available registers per core. This helps to limit contention. However, in Pascal, each core runs up to 8 CTAs/Core and Pascal has more cores. This significantly increases the number of concurrent warps and thus lock contention. Therefore, DS performs worse with Pascal baseline than Fermi. BOWS significantly improves performance as it combines deprioritizing spinning warps (which helps when there are many warps to schedule from) and throttling spinning warps by forcing them to wait by the back-off delay limit (which helps when there are few warps to schedule from).

6.5.5 Implementation Cost

Table 6.2 identifies the basic components in both DDOS and BOWS and estimates their costs per SM. The main cost of DDOS is the history registers, but using time-sharing (Section 6.3.2) it may be possible to reduce this cost. Comparison and hashing logic can be shared across warps in the same SM. To enable back-off
delay up to 10,000 cycles requires 14-bits per Pending Delay counter. Adaptive estimation requires division. This can be done using reduced precision computation or by using existing arithmetic hardware when not in use.

### 6.6 Related Work

Numerous research papers have proposed different warp scheduling policies with different goals (e.g., improving latency hiding [99], improving locality [124], reducing barrier synchronization overheads [81, 82], reducing load imbalance overhead across warps from the same CTA [72]). However, none of these scheduling policies have considered the challenge of warp scheduling under inter-thread synchronization.

Overheads of fine-grained synchronization have been well studied in the con-
<table>
<thead>
<tr>
<th></th>
<th>DDOS</th>
<th>BOWS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIB-PT</td>
<td>Pending Delay Counters</td>
</tr>
<tr>
<td>16-entry</td>
<td>35 bits each (560 bits)</td>
<td>48* 14 (bits) = 672 bits</td>
</tr>
<tr>
<td>History Registers</td>
<td>48 warps * 192 bits,(9216 bits)</td>
<td>Backed-off Queue</td>
</tr>
<tr>
<td>Comparison</td>
<td>8-bit comparator + 8:1 8-bitMux</td>
<td>Arbitration Logic Changes</td>
</tr>
<tr>
<td>Hashing (XOR)</td>
<td>8 4-bit XORs</td>
<td>Delay Limit Estimation Logic</td>
</tr>
<tr>
<td>FSM</td>
<td>48* 4-state FSM states</td>
<td>(can use functional units when available)</td>
</tr>
</tbody>
</table>

Table 6.2: DDOS and BOWS Implementation Costs

Text of multi-core CPU architectures [36, 80, 142, 143, 161]. Ti et al. [80] proposed a thread spinning detection mechanism for multi-threaded CPUs that tracks changes in all registers. Directly applying such a technique to a GPU would be prohibitive given the large register files required to support thousands of hardware threads. Instead, DDOS employs a speculative approach. In [161], the authors propose a synchronization state buffer that is attached to the memory controller of each memory bank to cache the state of in-flight locks. This reduces the traffic propagated to the main memory and the latency of synchronization operations. However, when the buffer is full the mechanism falls back to software synchronization mechanisms. This work builds on the following observation “at any instance during the parallel execution only a small fraction of memory locations are actively participating in synchronization” to maintain a reasonably sized buffer [161]. Although this observation holds true for modestly multi-threaded CPUs, it does not apply to massively multi-threaded SIMT architectures with tens of thousands of threads running in parallel. A similar technique that requires an entry per hardware thread to track locks acquired by each thread is used in [142].

In [155], the authors propose hierarchal queuing at each block in L1 and L2 data caches with the use of explicit acquire/release primitives. Their goal is to implement a blocking synchronization mechanism on GPGPU. In that work, locks can be acquired only on a cache line granularity. Locked cache lines are not re-
placeable until released. If a cache set is full with locked lines, the mechanism reverts back to spinning for newer locks mapped to the same line. Thus, the efficiency of this mechanism drops as the number of locks increase and starts to perform worse than the baseline [155]. For example, in the hashtable benchmark, the proposal in [155] performs worse than the baseline starting from 512 buckets (in contrast to our proposal, see Section 6.5.3 which consistently outperform the baseline). Further, unlike [155], our work does not assume explicit synchronization primitives which require non-trivial compiler support and/or significant hardware modifications [38] to run correctly on SIMT architectures.

Transactional memory and lock-free synchronization are other approaches to implement inter-thread synchronization [43, 95, 151]. However, both techniques rely on retries upon failure which lead to overheads and contention that is similar to busy-wait synchronization. GPU transactional memory proposals to date achieve lower performance versus fine-grained synchronization [43, 151]. Similar results have been also reported for lock-free synchronization [96]. Tuning DDOS and BOWS to reduce commit failures in lock-free synchronization is left for future work.

## 6.7 Summary, Conclusion and Future Directions

This chapter proposes DDOS, a low cost dynamic detection mechanism for busy-wait synchronization loops on SIMT architecture. DDOS is used to drive BOWS a warp scheduling policy that throttles spinning warps to reduce competition for issue slots allowing more performance critical warps to make forward progress. On a set of kernels that involve busy-wait synchronization, BOWS reduces dynamic instruction count by a factor of $2.1 \times$ and reduces memory system accesses by 19% compared to GTO. This leads to an average speedup of $1.4 \times$ and dynamic energy reduction by a factor of $1.7 \times$ on a set of GPU application kernels employing busy-wait synchronization.

The low cost required to implement DDOS and BOWS makes them a viable extension to any baseline scheduling policy. The low cost is essential since although the targeted application scope (i.e., applications with inter-thread synchronization) for these optimizations is emergent, it is not the common case. Therefore, from
a market perspective, only low cost solutions for the inter-thread synchronization problem can make their way into production.

One direction for future work is to explore the possibility of tuning DDOS and BOWS to reduce commit failures in lock-free synchronization. Most lock-free synchronization algorithms are not wait free and thus exhibit a behaviour that similar to busy-wait synchronization. One challenge would be to modify DDOS to detect lock-free synchronization loops. Unlike busy-wait synchronization, in lock-free synchronization, the loop induction variables may change over iterations even when the commit fails.

Another research direction is to explore how DDOS and BOWS can be efficiently integrated with AWARE. The performance of AWARE also suffers from warp split spinning (e.g., see comments about BH-ST performance in Figure 5.10). One challenge would be to scale DDOS to work on warp split granularity. This challenge is probably solvable with a version of the history registers time sharing proposed for DDOS. Another challenge would be to integrate BOWS with the FIFO scheduling policy of warp splits proposed in AWARE while still maintaining the scheduling fairness property of AWARE.
Chapter 7

MP: Multi-Path Concurrent Execution

Chapter 5 presented AWARE, an adaptive warp reconvergence mechanism that enables MIMD compatible execution on SIMT hardware. The main building block of aware was the Split and Reconvergence SIMT tables that decouples tracking of divergent splits from their reconvergence points. These SIMT tables enabled AWARE to work around the SIMT scheduling limitations of the conventional stack-based reconvergence mechanism, avoiding SIMT specific deadlock scenarios for parallel kernels with inter-thread synchronization. In this chapter, we explore an orthogonal application for these SIMT tables. In particular, along with additional microarchitectural modifications, the SIMT tables can be used to enable concurrent multi-path execution on GPUs as a performance optimization for divergent applications. We refer to this mechanism as the Multi-Path (MP) execution model.

7.1 Stack-Based Reconvergence Performance Limitations:

Section 3.1 discussed the scheduling constraints imposed by the stack-based reconvergence implementations in SIMT architectures. However, the focus was on the functional implications of such constraints. In this section, we revisit these constraints focusing on the performance implications of such constraints.
1. // id = thread ID
2. // BB_A Basic Block "A"
3. if (id % 2 == 0) {
4.   // BB_B
5. } else {
6.   // BB_C
7. }
8. // BB_D

Figure 7.1: Divergent code example

Figure 7.2: Execution with the Stack-Based Reconvergence Model. The figure refers to the stack as a Single Path Stack to distinguish it from latter proposals that support dual path execution [123].

The stack-based execution model allows only a single control flow path to execute at a time, which reduces the number of running threads. Active threads on alternate paths that are not on the top of stack may be either waiting at a reconvergence point or ready to execute a parallel control flow path. Thus, the stack-based execution model captures only a fraction of the actual thread level parallelism (TLP). The remaining TLP is essentially masked by a structural hazard implicit in the use of a stack for implementing IPDOM reconvergence.

For example, consider executing the divergent code in Figure 7.1 on the stack-based execution model 7.2. Initially, the stack has a single entry during the execution of basic block A. Once branch BR_A-B-C is executed, the PC of the diverged
entry is set to the reconvergence PC (RPC) of the branch (D). Also, resulting warp
splits, C_{1010} and B_{0101}, are pushed onto the stack. The RPC of the new entries
is set to the RPC of the executed branch (D). At this point, only warp split B_{0101}
is eligible for scheduling, as it resides at the top of the stack (TOS entry). Warp
split C_{1010} is not at the top of the stack, hence, it cannot be scheduled. As a result,
on cycle 4, there are no instructions to hide the latency of the first instruction in
basic block B. Once warp split B_{0101} reaches its reconvergence point (D), its corre-
sponding entry is popped from the stack. Then, warp split C_{1010} executes until
it reaches its reconvergence point (D) after which it is popped from the stack.
Finally, the diverged threads reconverge at D_{1111}. The above execution results in
two idle cycles.

Figure 7.3 quantifies this by showing the amount of TLP available to the sched-
uler as we increase the maximum number of concurrently executable warp splits
supported by hardware while IPDOM reconvergence is maintained. The graph
plots the average portion among all the scalar threads that can be scheduled be-
cause they are active in the top entry of the stack. The stack-based execution
model corresponds to enabling a single warp split. Section 7.4 gives more de-
tails about the benchmarks and the methodology. For this set of benchmarks, the
stack-based execution model captures from 15% of overall TLP in the Monte Carlo
(MC) benchmark up to around 65% in the Memcached (MEMC) benchmark. Fig-
ure 7.3 suggests that up to 35% more TLP is available when moving from the
stack-based reconvergence to a mechanism allowing an arbitrary number of warp
splits to be concurrently scheduled while maintaining IPDOM reconvergence. TLP
does not go to 100% with unlimited warp splits because some threads need to wait
at reconvergence points.

7.2 Multi-Path IPDOM (MP IPDOM)

During cycles when the pipeline is idle due to long latency events these alternate
control paths could make progress, as observed by Meng et al. [91]. This section
builds on this observation and presents a hardware mechanism that allows concur-
rent scheduling of any number of warp splits while still maintaining IPDOM re-
convergence (unlike [91] where concurrent execution trades-off SIMD utilization).
7.2.1 Warp Split Scheduling

Unlike AWARE, where execution switches from one path to the other only after encountering a branch, barriers and/or reconvergence points, MP-IPDOM allows execution to switch to other paths to fill an idle issue slot in the scheduler. To illustrate this, we use the same simple control flow graph in Figure 7.1 to explain the operation of the Multi-Path IPDOM. Figure 7.4 shows the operation of the MP IPDOM illustrating changes to the ST and RT tables (top) along with the resulting pipeline issue slots (bottom).

The warp begins executing at block A. Since there is no divergence, there is only a single entry in the ST, and the RT is empty. The warp is scheduled on the pipeline until it reaches the end of block A. After the warp executes branch BR_{B-C} on cycle 2, warp A_{1111} diverges into two splits B_{0101} and C_{1010}. Then, the A_{1111} entry is moved from the ST to the RT with PC field set to the RPC.
of branch BR\textsubscript{A}B\textsubscript{C} (i.e., D). The RPC can be determined at compile time and either conveyed using an additional instruction before the branch or encoded as part of the branch itself (current GPUs typically include additional instructions to manipulate the stack of active masks). The Reconvergence Mask entry is set to the same value of the active mask of the diverged warp split before the branch. The Pending Mask entry is used to represent threads that have not yet reached the reconvergence point. Hence, it is also initially set to the same value as the active mask. At the same time, two new entries are inserted into the ST; one for each side of the branch \(2b\). The active mask in each entry represents threads that execute the corresponding side of the branch.

On the clock cycle 3, warp splits B\textsubscript{0101} and C\textsubscript{1010} are eligible to be scheduled on the pipeline independently. We assume that the scheduler interleaves the available warp splits. Warp splits B\textsubscript{0101} and C\textsubscript{1010} hide each others’ latency leaving no idle cycles (cycles 3-5). On cycle 6, warp split B\textsubscript{0101} reaches the reconvergence point (D) first. Therefore, its entry in the ST table is invalidated \(3a\), and its active mask is subtracted from the pending active mask of the corresponding entry in
the RT table. Later, on cycle 7, warp split \( \text{C}_{1010} \) reaches reconvergence point (D). Thus, its entry in the ST table is also invalidated, and its active mask is subtracted from the pending active mask of the corresponding entry in the RT table. Upon each update to the pending active mask in the RT table, the Pending Mask is checked if it is all zeros, which is true in this case. The entry is then moved from the RT table to the ST table. Finally, the reconverged warp \( \text{D}_{1111} \) executes basic block D on cycles 7 and 8.

### 7.2.2 Scoreboard Logic

Current GPUs use a per-warp scoreboard to track data dependencies [30]. A form of set-associative look-up table (Figure 7.5a) is employed, where sets are indexed using warp ids and entries within each set contain a destination register ID of an instruction in flight for a given warp. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries of its warp. A dependency mask that represents registers that are causing data dependency hazards is produced from these comparisons and stored in the I-Buffer with the decoded instruction. The dependency mask is used by the scheduler to decide the eligible instructions at each issue slot. An instruction is eligible only if its dependency mask is all zeros. After writeback, both the scoreboard and the I-Buffer entries are updated to mark the dependency as cleared. In particular, the entries in the scoreboard look-up table that correspond to the destination registers of the written-back instruction are invalidated, and the bits that correspond to these destination registers in the dependency mask are cleared for all decoded instructions from this warp.

The Multi-Path IPDOM supports multiple number of concurrent warp splits running through parallel control flow paths. Hence, it is essential for the scoreboard logic to correctly handle dependencies for all warp splits and across divergence and reconvergence points. It is also desirable for the scoreboard to avoid declaring a dependency exists when a read in one warp split follows a write to the same register but from another warp split.

Therefore, we modify the scoreboard design by adding a reserved mask (R-mask) field to each entry in the scoreboard look-up table as shown in Figure 7.5b.
When an instruction is scheduled it bitwise-ORs the R-mask of its destination register with its active mask. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries of its warp. If the R-mask bit of a register operand of the instruction is set for any of the instruction’s active threads then a bit is set in the dependency mask for the I-Buffer entry associated with the instruction. This means that there is a pending write to this register by at least one active thread.

Upon writeback, the destination register’s dependencies are cleared from the scoreboard by clearing the bits in the R-mask that correspond to the active threads in the written-back instruction. The dependency bit masks in the I-Buffer are also updated. To do so, the active mask of each instruction that belongs to the written-back warp is compared with the R-mask of the destination registers of the written-back instruction. The respective destination register bit in the dependency mask is cleared if the instruction active mask and the R-mask do not have any common active bits. An instruction in the I-Buffer is available to be issued if all the bits in the dependency mask are zero.

To illustrate the operation of the modified scoreboard and its interactions with the I-Buffer, we use the example code snippet in Figure 7.6. Initially, the scoreboard is empty and I₀ has no pending dependencies with any registers. At Step 1, I₀ is issued and it reserves a scoreboard entry for its destination register R0 with R-mask=1111. Also, the branch instruction splits the warp into two splits that fetch and decode instructions from the two sides of the branch (i.e., I₁ and I₂). At the decode stage, all the source and destination registers of the decoded instruction are checked against the reserved registers in the scoreboard unit to see if there are any
pending dependencies, and, accordingly, the dependency mask of the instruction is generated. In this example, I_1 is dependent on R0 while I_2 has no pending dependencies. Hence, I_2 is eligible to be issued, and once it is issued, at 2, it reserves R1 with an R-mask=1010. At 3, I_3 is fetched and decoded. I_3 has pending dependencies on both R0 and R1, hence its Dep-mask=11. At 4, I_0 writes the load value to R0 and hence it releases R0 and clears the R-mask of the R0 entry from the scoreboard unit (the R0 entry becomes invalid since its R-mask is all zeros). Also, it clears the dependency mask of both I_1 and I_3 since the R-mask of R0 is now zeros for all active threads of both instructions. Since the dependency mask of I_1 is all zeros, it becomes eligible to be issued. Hence, at 5, it reserves its des-
tination register R1 with an R-mask=1010; such that R1 becomes reserved by all lanes—odd lanes due to pending writes of I1 and even lanes due to pending writes of I2. At 6, I2 writes the load value to R1 and hence it clears the even bits in the R-mask of register R1 in the scoreboard unit. Currently, R1 has an R-mask=0101 and it does not have any common active threads with I3 which has an I-mask=1010. Hence, the dependency bit that corresponds to R1 in the dependency mask of I3 is cleared and I3 becomes eligible for scheduling.

7.3 Opportunistic Early Reconvergence

MP-IPDOM maintains reconvergence at immediate postdominator (IPDOM) reconvergence points. The IPDOM reconvergence point is the earliest guaranteed reconvergence point. However, in certain situations, there are opportunities to reconverge at earlier points than the IPDOM point. Such situations depend on the outcomes of a sequence of branch instructions and the scheduling order of warp splits. Hence, early reconvergence is not guaranteed for all executions, but rather occurs dynamically when certain control paths are followed. Early reconvergence opportunities are common in applications with unstructured control flow [33, 44].

Figure 7.7 shows a code snippet that has unstructured control flow. We will use this code snippet to illustrate the modified operation of MP with support for opportunistic early reconvergence. Figure 7.8 shows instantaneous snapshots for a warp with four threads traversing through the control flow graph corresponding to the code in Figure 7.7. Active masks within basic blocks represent threads that are executing basic blocks at a specific time. Basic blocks with no active masks are not executed by any threads at that time. Figure 7.8a shows a snapshot of the execution where there are two warp splits, A_{0101} and B_{1010}, executing basic blocks A and B respectively. Both diverged warp splits have their IPDOM reconvergence point at D. This initial state results if a divergence at BR_{B−C} results in two splits B_{1010} and C_{0101}, and split C_{0101} reaches BR_{A−D} before split B_{1010} finishes executing basic block B.

Next, warp split A_{0101} branches to basic block B after executing BR_{B−C}. This scenario creates an early reconvergence opportunity, where there are two splits (B_{0101} and B_{1010}) of the same warp executing the same basic block B. To detect
an early reconvergence opportunity, an associative search within the ST using the first PC of the basic block is performed upon the insertion of any new entry. If there is an existing entry that matches the new entry in both the PC and RPC entries then an early reconvergence opportunity is detected. The early reconvergence point is the program counter of the next instruction of the leading warp split. In this example, $B_{1010}$ is the leading split, and the early reconvergence point is labeled $B_R$. A new entry is added to the RT to represent the early reconvergence point. The RPC of the new entry ($B_{0101}$) is set to the early reconvergence point ($B_R$). Finally, warp split $B_{1010}$ in the ST is invalidated, and accordingly the pending mask of the early reconvergence entry is updated (warp split $B_{1010}$ is already at the early reconvergence point).

The execution continues as explained in Section 7.5. Warp split $B_{0101}$ reaches the early reconvergence point $B_R$. Its entry in the ST is invalidated, the pending mask of the reconvergence entry $B_{R1111}$ is updated, and the reconvergence entry $B_{R1111}$ moves from the RT to the ST. Similar cases to this example occur with more complex code in several GPU applications [33]. We evaluate the benefits of opportunistic reconvergence in Section 7.5.

### 7.4 Methodology

We model MP IPDOM as described in Section 7.2 in GPGPU-Sim (version 3.1.0) [11]. Our modified GPGPU-Sim is configured to model a GeForce GTX 480 (Fermi) GPU with the configuration parameters distributed with GPGPU-Sim 3.1.0 (7.1).
Figure 7.8: Operation of the Multi-Path with the Opportunistic Reconvergence (OREC) enabled

but with 16KB instruction cache per core (see Section 7.5.4 for details). MP IP-DOM does not restrict the scheduling order, so we can use any scheduler. For both the baseline (i.e., SPS model) and our MP variations, we use a greedy-then-oldest (GTO). GTO runs a single warp until it stalls then picks the oldest ready warp [124]. On the baseline, the GTO scheduler outperforms the Two-Level and the Loose Round Robin schedulers. For splits within the same warp, we use a simple Round Robin scheme to alternate between them. We model, the opportunistic reconvergence optimization described in Section 7.3 as a separate configuration.
In Section 7.5, we present results for MP IPDOM when it limits the number of concurrently supported warp splits. This is modeled by setting a maximum constraint on the active number of entries in the ST. If, upon a branch, a new entry is required to be inserted to the ST, and the table is already at its maximum capacity, the new warp split is not considered for scheduling until an ST entry is invalidated. The configuration with two entries models the effect of the Dual-Path Stack (DPS) with the Path-Forwarding optimization [123] (more details are provided in Section 7.7).

We study benchmarks from Rodinia [26] and those distributed with GPGPU-Sim [11]. We also use some benchmarks with multi-path divergence from other sources:

**MEMC:** Memcached is a key-value store and retrieval system. It is described in detail by Hetherington et al. [53].

**REND:** Renderer performs 3D real-time raytracing of triangle meshes. For

### Table 7.1: GPGPUSim Configuration

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># Compute Units</td>
<td>15</td>
</tr>
<tr>
<td>warp Size</td>
<td>32</td>
</tr>
<tr>
<td>warp Scheduler</td>
<td>Greedy Then Oldest</td>
</tr>
<tr>
<td>Splits Scheduler</td>
<td>Loose Round Robin (LRR)</td>
</tr>
<tr>
<td>Number of Threads / Core</td>
<td>1536</td>
</tr>
<tr>
<td>Number of Registers / Core</td>
<td>32768</td>
</tr>
<tr>
<td>Shared Memory / Core</td>
<td>16KB</td>
</tr>
<tr>
<td>Constant Cache Size / Core</td>
<td>8KB</td>
</tr>
<tr>
<td>Texture Cache Size / Core</td>
<td>12KB, 128B line, 24-way</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
<td>6</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32KB, 128B line, 8-way LRU.</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>128k/Memory Channel, 128B line, 16-way LRU</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>16k, 128B line, 4-way LRU</td>
</tr>
<tr>
<td>Compute Core Clock</td>
<td>1400 MHz</td>
</tr>
<tr>
<td>Interconnect Clock</td>
<td>700 MHz</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>924 MHz</td>
</tr>
<tr>
<td>Memory Controller</td>
<td>out of order (FR-FCFS)</td>
</tr>
<tr>
<td>GDDR3 Memory Timing</td>
<td>$t_{CL}=12 \ t_{RP}=12 \ t_{RC}=40 \ t_{RAS}=28 \ t_{RCD}=12 \ t_{RRD}=6$</td>
</tr>
<tr>
<td>Memory Channel BW</td>
<td>4 (Bytes/Cycle)</td>
</tr>
<tr>
<td>Name</td>
<td>Abbr.</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>3-D Renderer</td>
<td>REND</td>
</tr>
<tr>
<td>Laplace Solver</td>
<td>LPS</td>
</tr>
<tr>
<td>LU Decomp.</td>
<td>LUD[11]</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>MAND[106]</td>
</tr>
<tr>
<td>Memcached</td>
<td>MEMC [53]</td>
</tr>
<tr>
<td>MUMMER++</td>
<td>MUMpp[46]</td>
</tr>
<tr>
<td>MUMMER</td>
<td>MUM[11]</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>MC[10]</td>
</tr>
<tr>
<td>Ray Tracing</td>
<td>RAY[11]</td>
</tr>
</tbody>
</table>

**Table 7.2: Studied Benchmarks**

benchmarking, we use pre-recorded frames provided with the benchmark [141].

**MC:** MC-GPU is a GPU-accelerated X-ray transport simulation code that can generate clinically-realistic radiographic projection images and computed tomography scans of the human anatomy [10].

**MAND:** Mandelbrot computes a visualization of mandelbrot set in a complex cartesian space. It partitions the space into pixels and assigns several pixels to each thread [106].

Note that out of 32 benchmarks, we only report the detailed results for the 9 benchmarks shown in Table[7.2] because the other benchmarks execute identically over SPS, DPS and MP variations. They perform identically because they are either non-divergent or they are divergent but all branches are one-sided branch hammocks such that the branch target is the reconvergence point. Under IPDOM reconvergence constraints, these applications do not exhibit parallelism between their basic blocks (i.e., non-interleavable [123]).

### 7.5 Experimental Results

This section presents our evaluation for the MP model.
7.5.1 SIMD Unit Utilization

Figure 7.9 shows the SIMD unit utilization ratio for benchmarks in Table 7.2. As expected, the SPS, DPS and basic MP have the same SIMD unit utilization because they all reconverge at the IPDOM reconvergence points. However, the opportunistic reconvergence optimization provides an average of 48% and up to 182% improvement in SIMD unit utilization. Benchmarks that exhibit unstructured control flow benefit from the opportunistic reconvergence.

7.5.2 Thread Level Parallelism

Figure 7.10 shows the average ratio of warp splits to warps. A value greater than one means an increase in the schedulable entities at the scheduler. Hence, it is more likely for the scheduler to find an eligible split to schedule. The SPS exposes only one split at a time to the scheduler. Hence, its average ratio of warp splits to warps is always one.

As shown in the figure, benchmarks such as REND, MAND, MC, MUM and MEMC show a considerable increase in the average ratio of the warp splits as the maximum number of supported warp splits increase. The MP IPDOM achieves ~50%-690% increase in the warp splits compared to SPS, and ~11%-400% compared to the DPS. This is mainly because some of these benchmarks have highly unstructured control flow (e.g. REND, MAND and MC) and they also have switch and else if statements that increase the number of parallel control flow paths (e.g. MEMC, MUM and MC).

The MUMpp, LPS, LUD and RAY benchmarks have a limited increase in the average number of warp splits (~5%). This is mainly for two reasons. The LPS, LUD and RAY benchmarks have SIMD utilization (>75%), hence, for a large portion of time there is no divergence at all. This biases the average towards a single split per warp. Also, the four benchmarks have a maximum of two parallel control flow paths at a time, otherwise, they are dominated by single-sided branches (i.e., one of their two outcomes is the branch reconvergence point). Therefore, for these applications MP acts identically to DPS.

The data in Figure 7.10 shows that the opportunistic early reconvergence bounds the increase in the number of warp splits. This is expected because it forces the
splits to wait for each other at the early reconvergence points, and it tends to combine multiple splits into a single one.

Figure 7.11 shows the average breakdown of the threads’ state at the scheduler. The threads’ state means whether the thread can issue its next instruction (i.e., eligible) or not. Also, it breaks down the different possible reasons that stalls a thread. Since a single thread can be stalled due to more than one reason (e.g., data hazard and structural hazard at the same time), the breakdown assumes priorities for the different possible stalling reasons. The priority order is the order in Figure 7.11 from bottom to top.

“Suspended” threads are those stalled due to divergence (i.e., they are either waiting at reconvergence points or they are waiting at a parallel control flow path). There is a gradual decrease in the number of “Suspended” threads for mechanisms that support a larger number of warp splits. For example, on MAND, MP has an average number of suspended threads that is \( \sim 35\% \) less than SPS.

However, the decrease in the average number of suspended threads does not directly translate to an increase in eligible threads. In particular, the non-suspended threads can be stalled due to data or structural hazard. This effect is pronounced in the MUM and MEMC benchmarks, where the gradual decrease in the average
suspended threads is followed by a gradual increase in the average threads stalled due to structural hazard.

The REND benchmarks suffers from load imbalance, where some warps exit the kernel early while others continue execution under divergence. The splits of the diverged warps are serialized in SPS. This biases the average result to have a large portion of "Finished" threads on the SPS model. As we increase the number of allowed warp splits, the scheduler interleaves the execution of diverged splits which in turn speeds up their execution. Hence, on average, we have more "FINISHED" threads.

7.5.3 Idle Cycles

Figure 7.12 shows the idle cycles accumulated for all cores for our benchmarks. The increase in the average number of eligible threads in Figure 7.11 translates into a decrease in the Idle cycles. Only the MEMC benchmark shows ~7% increase in the idle cycles when we adapt MP with opportunistic reconvergence compared to SPS model. We discuss this in detail in Section 7.5.4.

7.5.4 Impact on Memory System

Instruction Locality: MP has a direct impact on instruction cache locality. Unlike the SPS model, MP may require frequent fetching of instructions from distant blocks in the code. While this is not a problem for most GPU kernels which tend to
have small static code size, it can considerably affect the instruction cache misses in a large kernel. We find that the effect of instruction misses on the overall performance is negligible with a 16 KB cache. Figure 7.13 shows the instruction cache misses normalized to the SPS model. There is a considerable increase in instruction cache misses for the REND benchmark but it has limited effect on the overall performance (only an average of up to 2.5% of threads are stalled due to empty instruction buffers; see Figure 7.11). Since instruction fetch is done at warp splits granularity, MP with opportunistic reconvergence tends to have less instruction cache accesses and misses.

Data Locality: The effect of MP execution on data cache locality depends on the application and whether parallel control flow paths access contiguous memory locations or not. Figure 7.14 shows the L1 data cache misses normalized to the SPS
execution model. The MUM benchmark has reduced L1 data cache misses in MP compared to the SPS model, but it does not have a big impact on its performance because it already has low data cache misses (<0.3 MPKI “misses per thousand instructions”). However, the MEMC benchmark suffers from a significant increase in its misses (>80%). In particular, the total misses jumps from 30 MPKI to 82 MPKI. In depth analysis suggests that the MEMC benchmark loses its intra-warp locality. That is, warp splits evict each others’ data from the data cache before they get accessed again by the same warp split. These observations are consistent with prior work [124].

7.5.5 Overall Performance

Figure 7.15 shows the speedup over SPS. The speedup comes mainly due to the reduced idle cycles (i.e., more warp split instructions per cycle) and the improved SIMD units’ utilization (i.e., more throughput per warp split instruction). MP with opportunistic reconvergece has 32% harmonic mean speedup over the SPS model, compared to 18.6% and 12.5% for the basic MP and DPS models.

7.5.6 Implementation Complexity

Implementing MP requires modifications to the branch unit and the scoreboard logic. The modified scoreboard logic adds 1.5KB storage requirement for 48 warps per SM each with 8 register entries (GTX480 configuration). We synthesized both the basic scoreboard and the modified scoreboard on NCSU FreePDK 45nm [135]. We model the scoreboard as a small set associative SRAM. The synthesis results estimates an area 175,432 $\mu m^2$ and a total power of $\sim$4.4$mW$ total power at 50% activity factor; compared to 91,365 $\mu m^2$ and 1$mW$ power for the original score-
board. The SRAM used is based on NCSU’s FabScalar memory compiler, FabMem [130], with 6 read ports and 3 write ports. The cost of the ST and RT table are similar to those discussed in Section 5.5.

We also use GPU-Wattch [75] to estimate the increase in the dynamic power due to the associated with overall increased performance. For all our benchmarks, we find that the maximum observed increase in the average dynamic power is (37.5%) for the REND benchmark. However, the $7 \times$ speedup justifies such increase in power.

7.6 Related Work

This section discusses the closest related work to MP. A wider scope of related work is discussed in 8. Dual Path Stack (DPS) extends the SPS stack to support two concurrent paths of execution [123] while maintaining reconvergence at immediate postdominators. Instead of stacking the taken and not-taken paths one after the other, the two paths are maintained in parallel. DPS maintains separate scoreboard units for each path to avoid false dependencies between independent splits. However, it is necessary to check both units to make sure there are no pending dependencies across divergence and reconvergence points. As shown in Section 7.5, DPS has limited benefits on benchmarks that have multi-path divergence or benchmarks that have unstructured control flow behavior. Similar to DPS, Simultaneous Branch Interweaving (SBI) allows a maximum of two warp splits to be interleaved [21]. However, SBI targets improving SIMD utilization by spatially interleaving the diverged warp splits on the SIMD lanes. The reconvergence tracking mechanism proposed with the SBI requires constraints on both the code layout and the warp splits’ scheduling priorities to adhere to thread-frontier based reconvergence [33].

Dynamic Warp Subdivision (DWS) adds a warp splits table to the conventional stack [91]. Upon a divergent branch, it uses heuristics to decide which branches start subdividing a warp into splits and which do not. If a branch subdivides a warp, DWS ignores IPDOMs nested in that branch. This often degrades DWS performance compared to the SPS model [123]. Unlike DWS, MP IPDOM manages to maximize TLP under the IPDOM reconvergence constraints. Dynamic Warp
Formation (DWF) is not restricted to IPDOM reconvergence [41]. Instead, it opportunistically group threads that arrive at the same PC, even though they belong to different warps. DWF performance is highly dependent on the scheduling to increase the opportunity of forming denser warps, and sometimes leads to starvation eddies.

Thread Block Compaction (TBC) and TBC-like techniques allow a group of warps to share the same SIMT stack [44, 99]. Hence, at a divergent branch, threads from grouped warps are compacted into new more dense warps. Since TBC employs a thread block wide stack, it suffers more from the reduced thread level parallelism [122]. This makes MP IPDOM a good candidate to integrate with TBC to mitigate such deficiencies. For this purpose, the warp-wide divergence and reconvergence tables would need to be replaced with thread block wide tables.

7.7 Summary, Conclusion and Future Directions

This chapter presented a novel mechanism which enables efficient multi-path execution in GPUs. This mechanism enables tracking IPDOM reconvergence points of diverged warp splits while interleaving their execution. This is achieved by replacing the stack-based structure that handles both the divergence and reconvergence in the current GPUs with two tables. One table tracks the concurrent executable paths upon every branch, while the other tracks the reconvergence points of these branches. Furthermore, we illustrate that our multi-path model can be modified to enable opportunistic early reconvergence at run-time to improve SIMD units utilization for applications with unstructured control flow behavior. Evaluated on a set of benchmarks with multi-path divergent control flow, our proposal achieves 32% speedup over conventional single-path SIMT execution.

In the context of this thesis, this chapter shows that a hardware used to support Compatible MIMD Execution on SIMT architecture can be further leveraged to achieve performance improvements.

Combining MP-IPDOM with proposals such as Simultaneous Branch Interweaving [21] would make an interesting future work as it would extend the benefit of such work to throughput-bound divergent applications and not just latency-bound ones.
Chapter 8

Related Work

This chapter expands on the related work subsections in previous chapters. It gives more details about related work and surveys related research areas that are complementary to the work done in this thesis. We discuss work related with thread synchronization in GPGPUs in Section 8.1. In Section 8.2, we survey the different warp scheduling policies proposed for GPGPUs. In Section 8.3, we discuss survey the different techniques proposed to handle divergence in GPGPUs. In Section 8.4, we survey the verification tools proposed to verify the conformance of parallel kernels to correctness criteria on SIMT architectures.

8.1 Enabling Thread Synchronization in GPGPUs

Thread synchronization in GPGPUs face both reliability and performance challenges that are induced by the limitations of GPGPUs execution and memory models. Numerous hardware and software research efforts have tackled these challenges. The presence of SIMT deadlocks have been highlighted initially in different developers forums [114, 115]. The first research work to pay attention to this problem and attempt to find a solution is Ramamurthy’s MSc thesis [121]. The thesis proposed adding two instructions to the GPU ISA for lock and unlock operations. The instructions send out an atomic memory request which performs the same function as the atomicCAS besides manipulating the stack. The lock instruction changes the PC of executing entry to point to the program counter of the
instruction after the unlock statement. Then it pushes two entries on the stack; one for threads that failed to acquire the lock and the other for the ones that succeeded. Once an unlock instruction is encountered, the entry that represent threads that acquired the lock is popped out. Thus, threads that have failed to acquire the lock can proceed from where they were left. When threads within this entry succeed in acquiring the lock, their entry is popped out of the stack and execution continues with all threads converged. The thesis then adds further complexity to this fairly simple mechanism to handle cases where there are multiple unlock statement associated with every lock and cases of nested locks. However, as highlighted in the thesis, it still fails to address cases where there are multiple locks in a divergent code associated with a single unlock and when nested locks at different divergence levels.

The problem of SIMT deadlocks have been also highlighted in [47]. In this work, the authors provide formal semantics for NVIDIA’s stack-based reconvergence mechanism. They formally prove that the observable behaviour of executing a program on a SIMT execution model matches at least one possible behaviour from executing a program on traditional simultaneous multi-threaded machine (MIMD machines). However, they exclude the termination property which means that a program may not terminate on a SIMT machine even though it would always terminate on a MIMD machine. They provide a formal definition for the scheduling unfairness problem in the stack-based execution model. However, this work does not attempt to provide ways to detect or to prevent this problem.

In [155], the authors propose a blocking synchronization mechanism in GPGPUs. They propose explicit synchronization lock initialization, acquire, and release APIs. Their paper along with the related thesis [154] provides limited details regarding how the proposed APIs deal with the SIMT deadlock problem. They mentioned that they “use control flow instructions (i.e., jumps) for looping and enabling/disabling the threads conditionally using the active mask, active mask stack, and execute instructions for managing the stack” and that they “push or pop the active mask stack as required for the re-execution of lock init/acquire instructions”. This suggests that they replace the suggested APIs with fixed sequence of instructions to manipulate the stack. This can only work for the simple case where there is a single lock and unlock pair where the unlock instruction postdominates the
lock. More complex usage of these APIs will need some analysis and code transformations to work correctly which is not provided in this work. The authors in this paper, however, focus on the performance implications of the SIMT model on synchronization. They propose hierarchal queuing at each block in L1 and L2 data caches. For this to be feasible, locks can be acquired only on a cache line granularity and Locked cache lines are not replaceable until released. If a cache set is full with locked lines, the mechanism reverts back to spinning for newer locks mapped to the same line. The main limitation of this work as acknowledged by the authors is that the efficiency of this mechanism drops as the number of locks increase and starts to perform worse than the baseline [155].

In [78], the authors make the observation that shared scratch-pad memory atomic instructions is translated in Fermi SASS ISA as a series of instructions that implements a busy-wait loop. The busy wait loop checks whether a hardware lock bit associated with a memory location is clear to acquire before reading the value of the memory location. After the lock bit is acquired, the memory location value is then updated and the lock bit is cleared. The hardware only supports 1024 lock bits and all shared memory locations are mapped to these locks. Therefore, a typical lock implementation using a busy-wait loop of an atomic compare and swap instruction will essentially includes two nested loops. The low level busy-wait loop to acquire the lock bit and the outer loop that checks whether the mutex has been acquired or not. This introduces some redundancy that the paper explored how to eliminate. They propose low level implementation of lock and unlock primitives that uses the lock bit directly. This proposal reduces SASS instructions required to implement these primitives and the storage requirement to hold the mutex variables. However, the paper acknowledges two main limitations, SIMD (SIMT) and alias deadlocks. They do not provide a solution for SIMT deadlocks and only deals with it using workarounds on the programming level. Alias deadlocks emit from the fact that different shared memory locations can map to the same lock bit. Therefore, deadlocks occurs if the same thread attempts to acquire two locks at different memory locations that map to the same lock bit.

In [152], a software implementation for lock and unlock APIs that enables lock stealing between threads of the same warp is proposed in an attempt to solve livelocks in nested locking scenarios. In their lock implementation, a thread can
steal a lock from another thread in the same warp if the later ID is larger. A rollback function may be necessary to be implemented by the programmer to undo updates to the shared data made by a thread before its lock is stolen. The APIs are still prune to SIMT deadlocks and ”locks are not allowed to be acquired in a loop way”. In SSDE and AWARE, we assume a livelock free MIMD programs under arbitrary fair scheduling.

Recently (May 2017), NVIDIA revealed that Volta, their newest GPU architecture, incorporates changes specifically to enable easier programming with fine-grained synchronization (see Figures 12 through 14 in [103]). The description in [103] suggests that the observable behaviour of Volta’s execution model is similar to the Multi-Path execution model with delayed reconvergence enabled when it is needed. Volta also allows programmers to use a \texttt{syncwarp()} primitive to explicitly force threads within a warp to reconverge. This enables programmer to explicitly avoid conservative safe reconvergence point estimate due to false detection. The specific implementation of Volta’s independent thread scheduling execution model is not released by Nvidia.

The closest related Nvidia patent is a 2016 patent that describes a notion of convergence barriers [34]. In the execution scheme described in this patent, convergence barriers are used to join divergent groups of threads back together to maintain high SIMD efficiency while allowing for a flexible thread scheduling that used to be restricted by stack-based reconvergence. In this proposal “the divergence management mechanism that relies on the convergence barriers is decoupled from the thread scheduling mechanism”. The compiler analyzes the program to identify the appropriate locations to insert convergence barriers. In hardware, “a multi-bit register may correspond to each convergence barrier name and a bit is assigned for each thread that may participate in the convergence barrier”. A compiler may insert an ADD instruction that specify a convergence barrier name. Bits in corresponding to threads executing the ADD instruction will be set in the associated convergence barrier register. A compiler inserts a WAIT instructions at the required convergence barrier location. The WAIT instruction also specifies the name used by the ADD. When a thread reaches the WAIT instruction, it is blocked from execution. It also resets its corresponding bit in the convergence barrier multi-bit register. When all the threads reach a convergent barrier (i.e., the multi-bit register of the corre-
sponding barrier is all zeros), the convergence barrier is cleared and all threads that participated at the convergence barrier are released (i.e., unblocked) and they resume execution in SIMD fashion. Finally, the patent also introduces a YIELD instruction that may be inserted by the compiler to guide the thread scheduler to switch execution to other paths.

Interestingly, on a high level, the proposal in the patent is very similar to the Multi-Path and AWARE execution models. Similar to the MP and AWARE, the patent decides to decouple tracking divergence entries from reconvergence points. The multi-bit convergence barrier registers are simply another representation to the pending mask entry in the reconvergence table. The main difference is that our proposal relied on the hardware implicitly adding reconvergence points to the reconvergence table, checking whether threads have reached reconvergence points or not, and deciding when to switch execution to other paths. On the contrary, in Nvidia patent, the compiler explicitly adds an ADD instruction to add a new reconvergence point, a WAIT instruction to explicitly indicate that threads have reached a reconvergence point, a YIELD instruction that can be used to instruct the hardware scheduler to switch execution to other paths.

Transactional memory is another approach to deal with synchronization. Hardware transactional memory for GPGPUs was initially explored in [43]. The paper propose a word-level, value-based conflict detection mechanism where each transaction compares the saved value of its read-set against the value in memory upon completion where a change in value indicates conflict and the transaction is retried. The paper proposes a SIMT stack extension to deal with divergence after the commit stage where some threads pass the conflict detection stage and others fail. The extension is similar to the simple version of the stack extension proposed in [121]. This is enabled by the fact that in [43], nested transactions are flattened into a single transaction. Therefore, synchronization always maps to the simple case where there is a single transaction begin and commit pair and the commit postdominates the transaction begin. Support of transactional memory also requires significant hardware changes to log write history and manage conflict detection. The evaluation shows that this hardware transactional memory proposal can capture only 59% of fine-grained locking performance. Follow-up hardware transactional memory proposals have been explored to reduce this performance gap [27, 42]. According
to their evaluation, these cumulative efforts still leaves a performance gap of 7% compared to fine-grained synchronization on baseline GPUs.

Software transactional memory support in GPUs have been also proposed. In [56, 151], the authors propose a software implementation of transaction begin, commit, read, and write APIs. The programmer should surround the transaction begin and commit APIs with a loop to enable retries at commit failures [151] or label the transaction region so that the compiler can insert the retry loop [56]. This eliminates the need for changing the stack behaviour and puts the burden on the programmer to manage retries. The different APIs maintains the write logs and perform conflict detection. Software transactional memory has a large storage overhead to store transactions’ meta-data. Therefore, software transactional memory proposals to date achieve significantly lower performance versus fine-grained synchronization [56].

There is also a numerous research body that deals with the interaction of interthread synchronization and the memory model in GPGPUs. In [2], the authors reveal some of the GPU concurrency bugs that result from invalid programming assumptions about the GPU memory model and its interaction with related instructions (e.g., atomics and memory fences). Different memory consistency models and cache coherence protocols have been proposed to provide a more intuitive interface to the GPU memory model while minimizing synchronization overheads [3, 45, 51, 118, 132, 133, 150].

8.2 Warp Scheduling Policies in GPGPUs

There are a large number of research papers that explore warp scheduling policies in GPGPUs. The first paper to propose an alternate warp scheduling policy to loose round-robin was the two level scheduling paper [100]. The paper proposes a two-level warp scheduling where warps are divided into fixed size groups. Warps within each group is scheduled in a round-robin fashion, while different groups are scheduled in a greedy then oldest fashion. This scheduling policy aims to get the benefits of the round-robin policy in catching inter-warp locality and the greedy scheduling in forcing different different groups to progress at different rates such that not all warps arrive at long latency operations at the same time.
Later, the cache conscious warp scheduling paper [124] has opened the door for a series of adaptive warp scheduling policies. In [124], the number of actively schedulable warps is adjusted according to the intra-warp lost data locality. A small victim cache is used to estimate the lost data locality metric. A follow-up work [125] makes proactive warp scheduling decision based on predicted cache usage. The paper makes the observation that intra-warp data locality is between instructions in consecutive iterations of a loop. Thus, it is possible to predict a warp cache footprint from the number of load instructions executed in a loop iteration and the divergence pattern of threads within the loop. The scheduling policy uses these predictions to schedule warps with aggregate predicted cache footprint that is less than the effective cache size.

In [72], the authors observe a large execution time disparity between warps within the same thread block. This leads to the underutilization of the GPGPU resources since the allocation granularity of resources inside a GPU shader is a thread block. The paper then proposes a set of heuristics to prioritize the scheduling of critical warps that prevent a thread block from terminating. In [81], the authors tackle a similar problem. The main observation of this paper is that warps in the same thread block may arrive to a thread-block wide barrier at different times leading to excessive stall cycles. They show that the distribution of warps in the same thread block over different physical warp scheduler complicates the problem. Thus, propose a dynamic warp scheduling policy where different warp schedulers coordinate to prioritize warps in threads blocks where some warps are already waiting at a barrier. The same problem is also addressed in a concurrent work [83]. The main distinction is that in [83], the thread block with largest number of warps waiting at the barrier is prioritized while in [81], the thread block that first hit the barrier is prioritized.

Various other warp scheduling policies have been proposed with different heuristics. For example, in [156], the authors propose a two-level warp scheduling policy that dynamically adjust the warp groups size and moves warps from the active group to the the pending group according to their pipeline stall pattern. In [127], the authors combine two techniques that attempt to balance the preservation of inter and intra thread locality. In [9], a compiler analysis is used to detect which of a two-level warp scheduler or a GTO warp scheduler should be used for each
phase of a kernel execution. A similar approach is used in [70] except that the
switching between the two scheduling policies is detected at runtime according the
instruction-issue pattern. In [144], the MSHR consumption is used as a heuristic
to adjust the amount of thread level parallelism allowed by a warp scheduler.

8.3 Alternate GPGPU Execution Models

Dynamic Warp Formation (DWF) [41] has been proposed as alternative to the
stack-based reconvergence. Instead of restricting reconvergence to the IPDOM
point, DWF opportunistically group threads that arrive at the same PC, even though
they initially belong to different warps. DWF proposes different warp scheduling
policies that attempt to increase the chance of forming denser warps. The per-
formance of DWF is highly dependent on the scheduling policy. Although none
of the scheduling policy proposed in [41] was a fair scheduling policy, DWF can
avoid SIMT deadlocks if it adopted a fair scheduling policy across threads of the
same warp. However, this would undermine its ability to condense warps. Further,
the lack of guaranteed reconvergence points in DWF compromise programmers’
ability to predict the performance of their applications.

Dynamic Warp Subdivision (DWS) [41] adopted an alternative approach. In
this work, the authors attempt to maximize thread level parallelism to be able to
better hide long latency operations. DWS adds a warp splits table to the conven-
tional stack [91]. Upon a divergent branch, it uses static heuristics to decide which
branches start subdividing a warp into splits and which do not. If a branch subdi-
vides a warp, DWS ignores IPDOMs nested in that branch. This often degrades
DWS performance compared to the SPS model [123]. Unlike DWS, MP IPDOM
manages to maximize TLP under the IPDOM reconvergence constraints. DWS is
also prune to SIMT deadlocks as it still enforce IPDOM reconvergence to some
branches.

Follow up proposals can be generally categorized into one or a combination
of these two categories: maximizing SIMD efficiency or maximizing thread level
parallelism. Dual Path Stack (DPS) extends the single path stack to support two
concurrent paths of execution [123] while maintaining reconvergence at immediate
postdominators. Instead of stacking the taken and not-taken paths one after the
other, the two paths are maintained in parallel. DPS maintains separate left and right scoreboards to allow the two independent paths to operate in parallel with no false dependencies. However, they add extra bit (for each register) to store pending dependencies before divergence and reconvergence point. For each instruction, it is necessary to check both scoreboard units (which is done in parallel) to make sure there is no pending dependencies. So it is designed to work for two paths only.

Simultaneous Branch Interweaving (SBI) allows a maximum of two warp splits to be interleaved [21]. However, SBI targets improving SIMD utilization by spatially interleaving the diverged warp splits on the SIMD lanes. The reconvergence tracking mechanism in SBI uses thread frontiers [33] which sets constraints on both the code layout and the warp splits’ scheduling priorities.

Temporal SIMT (T-SIMT) maps each warp to a single lane, and the threads within a warp dispatch an instruction one after the other over successive cycles [74]. Upon divergence, threads progress independently; and hence divergence does not reduce the SIMD units utilization. However, reconvergence is still favourable to perform memory address coalescing and scalar operations [74]. The T-SIMT microarchitecture lacks a hardware mechanism to track reconvergence of diverged warp splits, therefore, they insert (syncwarp) instructions at the immediate postdominator of the top-level divergent branches [74]. Our MP microarchitecture provides a hardware mechanism to track nested reconvergence points. Hence, it can be integrated with T-SIMT.

Multiple SIMD Multiple Data (MSMD) [145] proposes quite large changes to the baseline architecture to support flexible SIMD data paths that can be repartitioned among multiple control flow paths. Similar to T-SIMT, MSMD proposes to use a special synchronization instruction to reconverge at postdominators, however, the paper does not specify an algorithm that determines where to place these synchronization instructions and how to determine which specific threads to synchronize at each instruction.

Thread Block Compaction (TBC) and TBC-like techniques allow a group of warps to share the same SIMT stack [44, 99]. Hence, at a divergent branch, threads from grouped warps are compacted into new more dense warps. Since TBC employs a thread block wide stack, it suffers more from the reduced thread level parallelism [122]. This makes MP IPDOM a good candidate to integrate with TBC to
mitigate such deficiencies. For this purpose, the warp-wide divergence and reconvergence tables would need to be replaced with thread block wide tables.

Variable Warp Sizing (VWS) [126] makes the observation that smaller warp sizes are convenient for control flow and memory divergent applications while larger warps are convenient for convergent applications. Therefore, VWS starts with a small warp size and then groups these smaller warps into larger ones in the absence of control and memory divergence.

8.4 Verification Tools for SIMT architectures

There is a body of research on the verification of GPU kernels that focuses on detecting data-races and/or barrier divergence freedom in GPU kernels. In [17], a tool is proposed to verify the freedom of GPU kernels from data races in shared memory and divergent barriers. GPUVerify transforms the kernel into two-threaded predicated form that is suitable for verifications. To detect data races, the kernel is instrumented to log accesses to shared arrays. To declare race-freedom, for each log, GPUVerify verifies that prior reads and write sets for the same array from different threads do not conflict. Barriers resets the read and write logs. Barrier divergence-freedom is verified if the aggregate predicate of the two threads is always the same at barrier points.

In GRace [160], a static analysis is combined with run-time checker to detect data-races in GPUs. If the memory address of an instruction can be statically determined, a linear constraint solver is used to determine whether two statically determinable write and write or read and write pairs are aliasing and thus induce a data race. Otherwise the pair is marked to be monitored at run-time. This reduces the amount of information needs to be logged at runtime. GLKEE [79] proposes a framework that analyze GPU kernels to detect both functional and performance bugs such as non-coalesced memory accesses, bank conflicts and divergent warps. GLKEE performs the analysis symbolically. GLKEE can also automatically generate test cases.

Although none of this work has addressed SIMT deadlocks, the analysis techniques used for data-race detections can be leveraged to improve the accuracy of our SIMT deadlock static analysis.
Chapter 9

Conclusions and Future Work

This chapter concludes the thesis, reflects on potential areas of impact by the work done in this thesis, and proposes few examples of future work that expands and improve on this work.

9.1 Conclusion

The convenience of the SIMT programming model has encouraged programmers to use it in accelerating irregular data parallel computations achieving in many cases significant speedups and energy savings over CPU multi-threaded implementations. Compared to other energy efficient alternatives such as ASIC and FPGAs, SIMT architectures have a programmability advantage that enables workload consolidation. However, current SIMT implementations lack reliable and efficient support for inter-thread synchronization that is essential for efficient implementations of many irregular applications.

This challenge has been facing both general purpose programmers (e.g., in CUDA and OpenCL [65]) and graphics programmers (e.g., in GLSL and HLSL [113]). That being said, existing GPU applications [25] that worked around current SIMT limitations to implement algorithms with fine-grained synchronization have achieved significant improvement over CPU implementations. However, as we showed in this thesis, they are vulnerable to portability and compatibility issues across compilers and/or GPU architectures. Further, such individual workarounds do not pro-
vide general rules that could ease the adoption of other algorithms with different inter-thread synchronization patterns. Their positive performance results, however, encouraged us to explore reliable and more efficient support of fine-grained synchronization in the SIMT execution model.

Another motivation is the wide interest in high level programming languages for accelerators such as OpenMP 4.0. The abstraction and portability of the OpenMP programming model will help SIMT accelerators reach a broader range of developers. However, support of fine-grained synchronization in OpenMP relies on runtime library calls that is challenging to properly implement on current SIMT implementations. This would equally apply to any future CUDA or OpenCL API extensions that could be proposed to abstract fine-grained synchronization.

In this thesis, we try to answer these questions:

- Can a compiler workaround the current limitations of the SIMT hardware to enable a true MIMD abstraction for synchronization on the SIMT hardware? What are the limitations?

- How could we enable the MIMD abstraction with minimal SIMT hardware changes without losing the efficiency of the SIMT model?

- How can we improve the efficiency of fine-grain synchronization on SIMT architectures with a low cost mechanism?

We found that it is possible to the compiler to enable MIMD abstractions through control flow graph transformations of the input code that contains inter-thread synchronization. However, this technique is limited to synchronization within a local function scope. Further, these control flow graph changes impose limitations on the code debuggability. False detection of synchronization can also lead to performance degradation.

Therefore, we proposed an adaptive warp reconvergence mechanism that avoids the thread scheduling constraints imposed by the current SIMT implementations. This mechanism requires limited hardware changes. It is capable to maintain traditional SIMT execution in the absence of synchronization. The hardware mechanism mitigates the key compiler limitations as it does not require code transformations to operate. We also showed that with further hardware extensions, this
divergence mechanism can be used to improve thread level parallelism in heavily
diverged kernels.

We found that the main source of inefficiency in inter-thread synchronization
on SIMT hardware is the warp scheduling policy that is oblivious to such synchro-
nization. Therefore, we developed a low cost mechanism to dynamically detect the
presence of synchronization and accordingly tune the warp scheduling policy. We
showed that this mechanism, though simple, provide significant performance and
energy improvements for applications with inter-thread synchronization.

9.2 Potential Areas of Impact

This section provides a brief description of areas on which we foresee a non-trivial
impact from this work:

SIMT Verification Tools: The increasing complexity of algorithms mapped to
SIMT accelerators mandates the development of verification tools. None of the
current verification tools considers the problem of SIMT deadlocks due to con-
ditional loops. Our paper shows that a simple static analysis can detect SIMT
deadlocks with low false detection rate. Future work can improve this rate by
leveraging runtime information and elaborate static analysis to perform less con-
servative reachability and dependence analysis.

Reliable SIMT Synchronization Primitives: Recent work [78] has proposed
promising efficient fine-grained synchronization primitives on GPUs. However,
a main limitation this work cites is SIMT deadlocks. This limitation rules out li-
brary based implementations for such primitives that could boost efficiency and
ease programmability. Our proposed techniques enable practical adoption of such
proposals. More about this in Section 9.3.

SIMT Compilers: This work makes the observation that SIMT compilers need to
be aware of the SIMT scheduling constraints to avoid generating SIMT deadlocks.
This affects all optimizations that alter the CFG or that move instructions across
CFG basicblocks. Our work provides a way to detect whether a certain application
is prone to SIMT deadlocks given a certain transformation (using variations of our
detection algorithm). Alternatively, our SSDE algorithms can be used to resolve
any SIMT deadlocks caused by other transformations.
**SIMT Programming Model:** The paper shows that the choice of conventional high level MIMD semantics versus low level SIMD semantics can be made by the programmer, rather than dictated by the underlying implementation. Our SSDE and AWARE techniques maintain default SIMT behaviour if disabled when a programmer is interested in low level optimizations that are dependent on predefined reconvergence locations.

**MIMD-compatible SIMT Architectures:** AWARE, as a realistic example, lays the foundation to consider MIMD-compatibility as a design goal for SIMT hardware implementations. Delayed and timed-out reconvergence mechanisms have a potential for being static and/or dynamic mechanisms the optimize the selection of reconvergence points on the granularity of individual branches. This could help improve SIMD utilization [137] and/or caching behaviour [116].

**Warp Scheduling Policy:** BOWS shows that a low cost and simple extension to the warp scheduling policy can significantly improve the performance of applications with inter-thread synchronization. Thus, it could be integrated with current warp scheduling policies to accelerate the ongoing improvement in thread synchronization in GPGPUs.

### 9.3 Directions of Future Work

We have already hinted to some of the ideas that can be explored as an extension or improvement for this work at the end of each chapter. In this chapter, we discuss some of these ideas in more details.

#### 9.3.1 SIMT Synchronization APIs

The support of high level synchronization APIs on SIMT architectures seems inevitable with the current interest in high level programming models. There have been different proposals that assumes such APIs [121, 155]. However, as detailed in Chapter [8], these proposals overlooked how such APIs can be used arbitrarily without introducing SIMT deadlocks. In this thesis, we showed how the compiler or the hardware can eliminate SIMT deadlocks in general scenarios. However, one of the main challenges we faced is false detections. Both SSDE (Chapter [3]) and AWARE (Chapter [5]) suffered performance degradation due to excessive false de-
tections. The introduction of explicit synchronization APIs can alleviate some of the burden required to detect synchronization. Further, BOWS (Chapter 6) currently reacts to the run-time detection mechanism of spinning loops. With explicit synchronization APIs, the compiler can add annotations that helps BOWS to proactively adjust the warp scheduling policy.

However, false detection can still be present with explicit synchronization APIs. For example, for lock-based synchronization, avoiding SIMT induced deadlocks while still maintaining reconvergence as soon as possible requires the detection of lock and unlock pairs. If the code contains multiple critical sections then conservative decisions should be made regarding this pairing. Note that this is necessary regardless of how the SIMT architecture implements the lock and unlock APIs. If it is implemented by translating the lock API to a single ISA instruction, then the divergence mechanism should set the reconvergence point of the implicit divergent at the end of the lock statement to beyond its unlock pair(s). Similarly, if the lock API is translated to a busy-wait loop then the code should be transformed to include the release statements from the unlock APIs inside the loop.

Therefore, we propose to use a notion of named locks that is analogous to the notion of named barriers used in CUDA. In named locks, a programmer adds a name parameter to the lock and its corresponding unlock statement(s). For example, Figure 9.1 illustrates how named locks can be used. In this contrived example, the compiler with the help of the names can clearly identify the bounds of a critical section. Note that the names does not need to have any representation in the hardware. The compiler uses the name hint to set the synchronization primitives

Figure 9.1: Illustration to the use of named locks.
correctly. If names are not provided, then it fails back to the conservative decisions.

Named locks can be part of programming language extensions that provide a solution for the dilemma of supporting robust and efficient synchronization on SIMT programming model. Explicit warp level barriers introduced in CUDA 8.0 can be also used by programmers to limit false detections by the compiler.

9.3.2 Runtime Livelock Detection

Our proposal for Dynamic Detection of Spinning (DDOS) can be further extended to detect the presence of livelocks in parallel kernels. It can be used as an initial light weight mechanism that monitors the overall progress of warps in the system. If our mechanism indicates that all warps have been spinning for a long period of time, a heavy weight mechanism that checks the full system state can be triggered to confirm.

One potential challenge is that DDOS is designed to detect spinning in short loops as is the case with busy-wait loops. This helped to reduce its value and path history register length. However, livelock can appear due to complex scenarios that involve larger, consecutive and/or nested loops. To address longer loops, longer history registers can be used with time sharing enabled to reduce costs. To address consecutive and nested loops, different history registers need to be allocated for each loop.

9.3.3 Reconvergence adequacy prediction

The flexibility of AWARE enables an opportunity to optimize the location of reconvergence points. The immediate postdominator point of a branch could be a safe choice to ensure adequate SIMD utilization but it is not necessarily the optimal choice. In some cases, it may be beneficial for some threads to run ahead beyond the reconvergence point to bring data into the cache for lagging ones. Further, in nested loops when the opportunistic early reconvergence is enabled, it may be beneficial for threads waiting at the reconvergence point of an inner loop to skip the reconverge and start another iteration of the outer loop to have a chance to merge with threads executing the inner loop.

The optimal reconvergence points can be taken by the programmer, imple-
mented as profiling-based optimization, or as a compiler optimization. In such a case, the Instruction Set Architecture should support a special instruction to set reconvergence points (such instruction already exists in Nvidia GPUs). The reconvergence adequacy can be also dynamically predicted. For examples metrics such as, the number of available warps, frequency of long latency operations, the inter-thread locality within the same warp and the loop nesting depth can be considered to tune reconvergence decisions.
Bibliography


Programming Languages and Systems, pages 270–289. Springer, 2013. → pages[170,177]


159


[61] IBM. Vector technology. . URL

[62] IBM. Auto-vectorization limitations. . URL


[64] Intel Corporation. The ISPC Parallel Execution Model. URL
https://ispc.github.io/ispc.html#the-ispc-parallel-execution-model. Accessed on 2017-09-06. → pages 1, 2, 3, 5, 17, 18

[65] Intel Developer Zone. Weird behaviour of atomic functions. URL 2012
Accessed on 2017-09-06. → pages 147

[66] Jeff Larkin, NVIDIA. OpenMP and NVIDIA. URL


[78] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar. Fine-grained Synchronizations and Dataflow Programming on GPUs. In *Proc. ACM Conf. on Supercomputing (ICS)*, 2015. → pages\[4, 33, 62, 63, 105, 139, 149\]


[86] LLVM Compiler. LLVM 3.6 Release Information. URL http://llvm.org/releases/3.6.0/ Accessed on 2017-09-06. → pages 30, 39, 55


162


[98] S. Naffziger, J. Warnock, and H. Knapp. When processors hit the power wall (or “when the cpu hits the fan”). In Proc. IEEE Int’l Conf. on Solid-State Circuits Conference (ISSCC), pages 16–17, 2005. → pages 1


163


[106] NVIDIA. CUDA SDK 3.2, September 2013. → pages 39, 55, 129


[108] Nvidia. NVIDIA Tesla P100. 2016. → pages 13

[109] Nvidia. NVIDIA TESLA V100 GPU ARCHITECTURE. 2017. → pages 87


[116] O. Mutlu et al. Runahead execution: An effective alternative to large instruction windows. In HPCA ’03. → pages 88, 150


Appendix A

SSDE Correctness Discussion

This section presents a semi-formal proof to address the correctness of the Static SIMT Deadlock Elimination algorithm.

A.1 Proof Outline

This section presents the outlines of a proof that the transformation $T$ provided by Algorithm 4 maintains the following theorem.

**Theorem 1** Let $P_i$ be a program that is an input to transformation $T$ and $P_o = T(P_i)$ be a program that results from applying $T$ on $P_i$, then any observable behaviour of $P_o$ on a SIMT machine is also an observable behaviour of $P_i$ on a MIMD machine (i.e., a machine that guarantees loose fairness in thread scheduling).

This definition of correctness has been used as a correctness criteria for both program transformations [55, 76, 88, 89] and execution models [28, 47]. We define observable behavior as the shared memory state, the return value of the program -if any- and the termination properties. We divide the proof of Theorem 1 into two steps that need to be proven:

A. Any observable behaviour of $P_o$ is also an observable behaviour of $P_i$ when both are executed on a MIMD machine.

B. Any observable behaviour of $P_o$ on a SIMT machine is also an observable be-

---

1 We assume $P_i$ maintains the assumptions outlined at the beginning in 3.3.1. We also assume that in both SIMT and MIMD machines, the number of launched threads do not exceed the number of threads that can reside simultaneously on the hardware.
haviour of $P_o$ on a MIMD machine.

A.2 Proof Details

A. Preservation of Semantics across transformations: To prove this part we use the same methodology used in TRANS [68, 89].
Let a program $P$ has form:

$$ P : \text{Entry; } I_1; I_2; \cdots; I_{m-1}; \text{Exit} $$

- Inst $\Rightarrow I ::= \text{nop} \mid X := E \mid M | \text{if } E \text{ goto } n$
- M. Inst $\Rightarrow M ::= X := m(E) \mid m(E) := X$
- Mem. $\Rightarrow M ::= \text{shared memory}$
- Expr $\Rightarrow E ::= X \mid O(E) \mid C$
- Op $\Rightarrow O ::= \text{various unspecified operators}$
- Var $\Rightarrow X ::= x \mid y \mid z \mid \ldots$
- Const $\Rightarrow C ::= \text{bool} \mid \text{integer} \mid \text{float} \mid \text{value}$
- Label $\Rightarrow n ::= 0 \mid 1 \mid \ldots \mid m$

Program $P$ can be represented as a labeled directed flow graph $G_P$ formed from the program nodes ($N = \{0, 1, \ldots, m\}$). Each node $n$ has at least one sequential edge to another node $\text{seq}(n)$; and possibly another branch edge to a node $\text{brn}(n)$. Thus, an edge in the flow graph is defined by two nodes and an edge type. $I_n$ refers to the instruction labeled by node $n$. Hence, $G_P$ can be represented by the tuple $\langle N,E \subseteq N \times N \times \text{EdgeType},I : N \rightarrow \text{Instr} \rangle$. Finally, a valuation function $\sigma$ is used to map a pattern of meta variables to a (sub)object in $G$ (i.e., node, edge, instr, expr, ..).

In concurrent MIMD execution, a set of thread traverse through $G_P$ such that at any time the execution state of $P$ can be represented by a tuple $(\text{nodes}, \text{states}, m)$; where $\text{nodes}$ is a vector of the labels representing the current location of each thread in $G_P$, $\text{states}$ is a vector of the local state of each thread, and $m$ is the current state of the externally observable shared memory. Thus, the execution trace of a program can be modeled by state transitions: $st_i \rightarrow st_{i+1}$, where $st_i = (\text{nodes}, \text{states}, m)_i$. The state of an individual thread is represented as $st(t) = (n,s,m) = (\text{nodes}_t, \text{states}_t, m)$. A state transition happens when a thread executes the instruction labeled by its current node. Next, we present the state transition relations for an instructions $I_n$ at node $n$: 172
Next, we describe the transformation in Algorithm 4 using the TRANS transformation language. For clarity, the transformation is simplified to handle transformation of a single structured loop ($L_0$). However, both the algorithm description and the proof can be extended in a straightforward manner to handle the transformation of multiple (un)structured loops. In TRANS, a transformation is specified as a set of actions performed on a flow graph $G$ under certain conditions. We describe our transformation $T$ as follows:

\[
\begin{align*}
(s,n,m) & \rightarrow (s,\text{seq}(n),m); \ I_n = \text{nop} \\
(s,n,m) & \rightarrow (s + x \mapsto \text{eval}(e), \text{seq}(n),m); \ I_n = x := e \\
(s,n,m) & \rightarrow (s + x \mapsto m(\text{eval}(e)), \text{seq}(n),m); \ I_n = x := m(e) \\
(s,n,m) & \rightarrow (s,\text{seq}(n),m + m(\text{eval}(e))); \ I_n = m(e) := x \\
(s,n,m) & \rightarrow (s,\text{seq}(n),m); \ I_n = \text{if } e \text{ goto } brn(n), \text{eval}(e) = F \\
(s,n,m) & \rightarrow (s,brn(n),m); \ I_n = \text{if } e \text{ goto } brn(n), \text{eval}(e) = T
\end{align*}
\]

Next, we prove the correctness of transformation $T$ when applies to $L_0$ assuming a MIMD execution model.

\[
\begin{align*}
\boxed{\text{replace } r \text{ with } r \mapsto \sigma(\text{nop}); \ rn \mapsto I_r} & (\sigma,G) \\
\boxed{\text{move edge}(t,h,r)} & (\sigma,G) \\
\boxed{\text{add cond edge}(r,h,t)} & (\sigma,G) = \\
\boxed{\text{replace } d \text{ with } d \mapsto \sigma(c := 0); \ dn \mapsto I_d} & (\sigma,G) \\
\text{if } d \models EX(\text{node}(r)) \land d \neq t \land A \neg E(\text{true } U \text{ use}(c)) & \\
\boxed{\text{replace } t \text{ with } t \mapsto \sigma(c := 1); \ tn \mapsto I_t} & (\sigma,G) \\
\text{if } A \neg E(\text{true } U \text{ use}(c)) & \\
\boxed{\text{replace } s \text{ with } s \mapsto \sigma(\text{if}(c) \text{ goto } h)} & (\sigma,G) \\
\text{if} & \\
\text{loop}(p,h,b,t) & \equiv L_0 \\
\text{r } \models \text{SafePDOM}(L_0)
\end{align*}
\]
**Single-Thread Equivalence:** We prove that there exists a relation $R$ that relates the execution states of the original program with that of the transformed program for a thread $t$ with the assumption that other threads $u \neq t$ remain at the same initial execution state. We then prove that if this relation $R$ holds then it implies that the two programs have the same observable behavior as defined earlier.

The set of actions are performed on a loop $L_0$ with preheader $p$, header $h$, break (exit) $b$ and tail $t$. Node $r$ is the $SafePDOM(L_0)$. Action $a_1$ adds a nop instruction at node $r$. The backward edge from the loop tail $t$ to the loop header $h$ is moved from $h$ to $r$ by Action $a_2$. Finally, Action $a_3$ adds an edge from $r$ to $h$ that would be only taken if node $r$ is entered from $t$. This replaces the nop instruction in $r$ with a conditional branch instruction that diverges to $h$ if a variable $c$ had a value of '0' (i.e., if the thread is reaching $r$ from any predecessor $d$ other than $t$) and to $h$ if the $c$ had a value of '1' (i.e., if the thread is reaching $r$ from $t$). Thus, node $r$ acts as a switch that redirects the flow of the execution to an output node according to the input node. Conditions on actions $a_3.1$ and $a_3.2$ assures that $c$ is not used by any node that follows $t$ or $d$ other than the new added branch instruction at $r$.

According to these actions the following properties are satisfied:

$$
\begin{align*}
  I_{n})_{P,} &= I_{n})_{P}, \quad seq(n)_{P,} = seq(n)_{P,} \text{ for } n \notin \{d, r, t\} \quad \text{(p1)} \\
  I_{d})_{P,} &= \sigma(c := 0), \quad seq(d)_{P,} = dn \quad \text{(p2)} \\
  I_{dn})_{P,} &= I_{d})_{P,}, \quad seq(dn)_{P,} = seq(d)_{P} \quad \text{(p3)} \\
  I_{r})_{P,} &= \sigma(if(c \text{ goto } h), \quad seq(r)_{P,} = rn \quad \text{(p4)} \\
  I_{m})_{P,} &= I_{r})_{P,}, \quad seq(rn)_{P,} = seq(r)_{P} \quad \text{(p5)} \\
  I_{t})_{P,} &= \sigma(c := 1), \quad seq(t)_{P,} = tn, \quad I_{m})_{P,} = I_{r})_{P,} \quad \text{(p6)} \\
  seq(r)_{P,} &= h, \quad seq(tn)_{P,} = r \quad \text{(p7)}
\end{align*}
$$

Assume the following state transitions for a thread $t$:

$$
\begin{align*}
  st_0(t) \rightarrow st_1(t) \rightarrow ... \rightarrow st_i(t) = (n, s, m) \rightarrow st_k(t) \text{ from } P_i \text{ and} \\
  st'_0(t) \rightarrow st'_1(t) \rightarrow ... \rightarrow st'_i(t) = (n', s', m') \rightarrow st'_k(t) \text{ from } P_o
\end{align*}
$$

also assume that for all other threads $u \neq t \bar{st}_0(u) = st'_0(u)$ Then, the following relation $R$
holds

\[ R1. \quad s'_{\text{except } \sigma(c)} = s_k \quad R2. \quad m'_l = m_k \quad R3. \quad l'_l = l_k \]

\[
R4. \quad \text{node}(n'_l) = \begin{cases} 
\text{node}(n_k) & \text{node}(n_k) \notin \{d, r, h\} \\
\text{dn} & \text{node}(n_k) = d \\
\text{rn} & \text{node}(n_k) = r \\
\text{tn} & \text{node}(n_k) = t 
\end{cases}
\]

for \( l = k + u + v + w + y \) where:

- \( u \) = no. of nodes \( n_i \) where \( i \leq k \) such that \( \text{node}(n_i) = d \),
- \( v \) = no. of nodes \( n_i \) where \( i \leq k \) such that \( \text{node}(n_i) = r \),
- \( w \) = no. of nodes \( n_i \) where \( i \leq k \) such that \( \text{node}(n_i) = t \),
- \( y \) = no. of transitions \( \text{node}(n_i) \rightarrow \text{node}(n_{i+1}) = t \rightarrow h \) where \( i < k \);

\( R \) simply states that \( P_o \) simulates the behavior of \( P_i \) but in potentially more execution state transitions according to the specific execution path that was taken\(^2\).

These extra transitions account for the execution of the added nodes during the transformation \( T \). The local state of a thread executing \( P_o \) at \( l \) may only be different by the new added variable \( c \) that control the branch at \( r \). We prove \( R \) by induction over \( k \).

**Proof Logic:** assuming that the relation holds for \( k \), then we consider one transition from \( k \) to \( k + 1 \). We find that according to \( R \), \( P_o \) should simulate the behaviour of \( P_i \) in either one or two transitions from \( l \). This is determined according to \( \text{node}(n_{k_n}) \) and \( \text{node}(n_{k_{n-1}}) \). Then, we proceed by proving that this indeed holds for all possible transitions from \( k \) to \( k + 1 \). Given that relation \( R \) holds, it is trivial to prove program equivalence. First, if an execution state transition terminates for \( P_i \) at \( k \) then it will terminate for \( P_o \) at \( l = k + u + v + w + y \). Further, \( I_k = I_l = \text{ret}(e) \), \( s'_{\text{except } \sigma(c)} = s \) and \( e \) does not use \( \sigma(c) \), then return values will be the same.

**Base Case** is trivially . For the same input and same initial state \( R1,R2 \) and \( R3.2 \) holds.

**Step Case** Assume that \( R \) is true for \( k \). Prove \( R \) is true for \( k_n = k + 1 \).

**Case-1:** \( \sigma(n_{k_n}) \notin \{d,r,t\} \) and \( \sigma(n_k) \rightarrow \sigma(n_{k+1}) \neq t \rightarrow h \) Given the condition on Case-1, \( u_n = u, v_n = v, w_n = y \), and \( l_n = l + 1 \) (i.e., execution of \( P_o \) should simulate

\(^2R\) in this case is called a *simulation* relation.
the execution of \( P_i \) at \( k_n \) in a single step from \( l \). From R1 and R3 at \( k \) and the state transition relations, it is trivial to prove that that \( s'_{n_u}/\sigma(c) = s'/\sigma(c) \oplus \sigma(l'_i) = s_t \oplus \sigma(l_i)=s_{k_n} \); i.e., R1 holds for \( k_n \). Similarly, \( m'_{l_n} = m'_i \oplus \sigma(l'_i) = m_{k_n} \); i.e., R2 holds for \( k_n \). From conditions of Case-1, \( n_k \notin \{d,t\} \). In case \( \sigma(n_k) \neq r \), then \( \sigma(n'_i)=\sigma(n_k) \). Thus, \( \sigma(n'_i)\oplus l'_i)=\sigma(n_k \oplus l_i)=\sigma(n'_i)=\sigma(n_{k_n}) \). Also for \( \sigma(n_k) = r \), \( \sigma(n'_i) = rn \) and \( seq(r)_{P_1} = seq(rn)_{P_2} \). Thus, \( \sigma(n'_i) = \sigma(n_{k_n}) \) for \( \sigma(n_k) = r \). Therefore, R4 applies for \( k_n \) in all possible cases. Finally, since R4 holds and that both executions are at the same label then from property \( p1 \) we can conclude that R3 holds at \( k_n \).

**Case-2:** \( \sigma(n_{k_n}) = d \). Then, \( u_n = u_1 + 1 \) and \( l_n = k_n + 1 = k_2 \). This means that execution of \( P_o \) should simulate the execution of \( P_i \) at \( k_n \) in two steps from \( l \). Since R holds for \( k \), we can prove that \( \sigma(n'_{l_n}) = \sigma(n_{k_n}) = d \). However, according to property \( p2 \), \( I_d \neq I_d \) (i.e., R3 does not hold from a single step from \( l \)). Instead, using properties \( p2 \) and \( p3 \), we find that R3 is satisfied after two steps. Similar to Case-1, we can prove all other properties at \( l_n = l + 2 \). We can similarly prove

**Case-3:** \( \sigma(n_{k_n}) = t \).

**Case-4:** \( \sigma(n_k) \rightarrow \sigma(n_{k+1}) = t \rightarrow h \). Then, \( y = 1 \) and \( l_n = l + 2 \). Since R4 holds at \( k \), we know that \( \sigma(n'_l) = t_n \). According to property \( p7 \), the next node in \( P_o \) is \( h \) and in \( P_i \) is \( r \) (i.e., R3 and R4 does not hold from a single step from \( l \)). We also know that by coming from \( tn \) that \( \sigma(c) = 1 \) and that \( I_t \) evaluates to taken branching to \( h \). Thus, we can simply find that R holds for \( k_n \) at \( l_n = l + 2 \). We can similarly prove

**Case-5:** \( \sigma(n_{k_n}) = r \).

**Lifting simulation relation to parallel execution:** We need to prove that steps by threads other than \( t \) preserve the simulation relation \([88, 89]\). \( R \) implies that \( (m' = m) \) then it is guaranteed that both the original and the transformed program maintains exactly the same view of shared memory for a thread \( u \neq t \). Thus, a memory read from an arbitrary location \( loc \) by a thread \( u \) from \( m' \) and \( m \) will yield the same values. This conclusion is intuitive as the transformation does not reorder memory operations (read, writes or memory barriers). Further, we need to prove that memory updates by a thread \( u \) can not change the memory such that the simulation relation \( R \) no longer holds. This is evident by the proof of \( R \) that is independent from the shared memory state. Relation \( R \) is built on subtle changes
in the CFG that is only dependent on the new added local variable $c$. Note that we did not depend on a specific memory model to prove that the simulation relation holds for the case of parallel execution.

**B. Preservation of Semantics across execution models:** In this part we rely on prior work [28, 47] that proves that the execution of an arbitrary program $P$ on a SIMT machine can be simulated by some schedule of the traditional interleaved thread execution (i.e., MIMD execution) of the same program. Thus, terminating kernels on a SIMT machine produce a valid observable behaviour compared with MIMD execution. However, it is still possible that a program that always (i.e., under any loosely fair scheduling) terminate on MIMD to not terminate on SIMT. Therefore, it is sufficient for us to prove that $P_o$, an output of $T$, always terminate on a SIMT machine if $P_i$ (or equivalently $P$) always terminate on a MIMD machine (i.e., with any arbitrary loosely fair schedule).

**Proof Logic** Termination is trivially proven if we can prove that all threads executing any arbitrary branch in $P_i$ eventually reach to the branch reconvergence point. To construct such a proof, we rely on two main claims:

**Claim-1:** $P_o$ terminates on a MIMD machine under any arbitrary loosely fair scheduling.

**Proof:** We assume that $P_i$ terminates on a MIMD machine under any arbitrary loosely fair scheduling. However, according to the proof presented earlier the transformation $T$ preserves the program semantics on a MIMD machine and that $P_o$ simulates the behaviour of $P_i$ on a MIMD machine including the termination properties. Thus, we conclude that $P_o$ terminates on a MIMD machine under any arbitrary fair scheduling.

**Claim-2:** The valuation of the exit condition in any loop in $P_o$ is independent of the valuation of paths parallel to or reachable from the loop. The definitions of parallel and reachable paths is listed in Listing 1.

**Proof:** This is a forced property by transformation $T$ presented in Algorithm 4. As shown by Algorithm 4 and as explained in Section 4.2.1 any loop that has its exit dependent on the valuation of paths parallel to or reachable from the loop is transformed such that the backward edge of the loop is converted into a forward
edge to SafePDom and a backward edge to the original loop header. SafePDOM postdominates the original loop exits, the redefining writes, and all control flow paths that could lead to redefining writes that are either reachable from the loop.

Now we proceed to prove that all threads executing any arbitrary branch in $P_o$ eventually reach to the branch reconvergence point. We prove this by induction over the nesting depth of the control flow graph. For this purpose let’s consider an arbitrary branch $I_{BR,T,NT\rightarrow R}^k$ which is a branch with a nesting depth of $k$. We define the nesting depth as the maximum number of static branch instructions encountered in the control flow path connecting the branch instruction with its reconvergence point $R$.

**Base Case-1:** $I_{BR}^0$. No static branches between the branch instruction and its reconvergence point. Since there is no barriers placed in divergent code, threads diverged to either sides of the branch are guaranteed to reach its reconvergence point. The follows from two facts: 1) in the absence of barriers and loops (i.e., branches), nothing prevents the forward progress of threads, 2) according to constraint-2, once threads executing one side of the branch reach its reconvergence point, execution switches to threads diverged to the other side.

**Base Case-2a:** $I_{BR}^1$ and $I_{BR}^1 \in P_{T\rightarrow R} \lor I_{BR}^1 \in P_{NT\rightarrow R}$. This means that the branch itself is encountered again before reaching $R$ (i.e., it is a loop exist). Threads may never reach $R$ if the valuation of $I_{BR}^1$ does not lead to exit the loop. This could happen under only two hypothesis.

1. The valuation of $I_{BR}^1$ is independent of thread scheduling (i.e., it is independent of the execution of other paths parallel to or reachable paths to the loop whose exit is ($I_{BR}^1$)). However, it never evaluates to decision that leads to exit the loop. This contradicts with Claim 1 as it implies that $P_o$ does not terminate under fair scheduling. Thus, we exclude this hypothesis.

2. The valuation of $I_{BR}^1$ is dependent on scheduling threads at the bottom of the stack (i.e., it is dependent on the execution of other paths parallel to or reachable paths to the loop whose exit is ($I_{BR}^1$)). However, for these threads to get scheduled, the looping threads need to exit and reach their reconvergence point to be popped out of the stack allowing for threads at bottom stack entries to get schedule. However, this hypothesis contradicts with Claim 2 since the operation of $T$ forces the valuation of the loop exit condition to be independent of parallel to or reachable
from the loop. Thus, we reject this hypothesis.

From (1) and (2), we conclude that threads divergent at $I_{BR}^1$ reach their reconvergence point.

**Base Case-2b:** $I_{BR}^1$ and $I_{BR}^1 \notin P_{T\rightarrow R} \land I_{BR}^1 \notin P_{NT\rightarrow R}$. This means that the branch is not encountered again before reaching $R$, however another branch is encountered. This other branch could follow the pattern of Base Case 1 or Base Case 2a.

(i.e., it is a loop exist). Threads may never reach $R$ if the valuation of $I_{BR}^1$ does not lead to exit the loop. This could happen under only two hypothesis.

(1) The valuation of $I_{BR}^1$ is independent of thread scheduling (i.e., it is independent of the execution of other paths parallel to or reachable paths to the loop whose exit is $(I_{BR}^1)$). However, it never evaluates to decision that leads to exit the loop. This contradicts with Claim 1 as it implies that $P_o$ does not terminate under fair scheduling. Thus, we exclude this hypothesis.

(2) The valuation of $I_{BR}^1$ is dependent on scheduling threads at the bottom of the stack (i.e., it is dependent on the execution of other paths parallel to or reachable paths to the loop whose exit is $(I_{BR}^1)$). However, for these threads to get scheduled, the looping threads need to exit and reach their reconvergence point to be popped out of the stack allowing for threads at bottom stack entries to get schedule. However, this hypothesis contradicts with Claim 2 since the operation of $T$ forces the valuation of the loop exit condition to be independent of parallel to or reachable from the loop. Thus, we reject this hypothesis. From (1) and (2), we conclude that threads divergent at $I_{BR}^1$ reach their reconvergence point.

Finally, from A. and B., we conclude that Theorem 1 is correct.