Efficient Precise Dynamic Data Race Detection For Cpu And Gpu

Yuanfeng Peng
University of Pennsylvania, yuanfeng.jack.peng@gmail.com

Efficient Precise Dynamic Data Race Detection For Cpu And Gpu

Abstract

Data races are notorious bugs. They introduce non-determinism in programs behavior, complicate programs semantics, making it challenging to debug parallel programs. To make parallel programming easier, efficient data race detection has been a research topic in the last decades. However, existing data race detectors either sacrifice precision or incur high overhead, limiting their application to real-world applications and scenarios. This dissertation proposes approaches to improve the performance of dynamic data race detection without undermining precision, by identifying and removing metadata redundancy dynamically. This dissertation also explores ways to make it practical to detect data races dynamically for GPU programs, which has a disparate programming and execution model from CPU workloads. Further, this dissertation shows how the structured synchronization model in GPU programs can simplify the algorithm design of

data race detection for GPU, and how the unique patterns in GPU workloads enable an efficient implementation of the algorithm, yielding a high-performance dynamic data race detector for GPU programs.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Joseph Devietti

Keywords

Concurrency, Data Races, Debugging, GPU, Parallel Programming, Software Tools

Subject Categories

Computer Sciences

EFFICIENT PRECISE DYNAMIC DATA RACE DETECTION FOR CPU AND GPU

Yuanfeng Peng

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

To my wife Jing, for her love, support, and trust along this journey.

I would never have made it here without you.

ACKNOWLEDGEMENTS

Many people say surviving a Ph.D. program isn't an easy job. This is true, although I wouldn't use the word *survive* today, when I'm about to graduate. For me, the past 5 years have been an exciting adventure, made possible by many people I owe a debt of gratitude to.

I'll start by thanking my advisor Joe Devietti. Joe, thanks for being such an awesome mentor, collaborator, and friend. I can still remember the interview we had 6 years ago, the theme of which was on data race detection, a topic that became the research focus of my whole Ph.D. study. Thanks for admitting me as your first student, which is the start of the awesome journey. Your patience and trust meant a lot to me, especially in the first few years, when I, as an international student, was struggling to adapt to the new environment and overcome cultural and language barriers. I appreciate your optimism and constant encouragements, which are especially invaluable when things were not looking up. Thanks for being such a great teacher, I can still recall the 501 classes 5 years ago. Thanks for all the jokes and humor in our meetings and discussions, which spiced up the workdays and made the long journey more pleasant. Special thanks are also due to you for your understanding and openness for letting me finish up the program remotely in the past 3 years, which makes my life much easier. Most importantly, I want to thank you for your enthusiasm for research and hacking, which is contagious and I'm glad and proud that I still love research after all these years.

I'm lucky to have collaborated with a number of amazing people. I'd like to thank Ben Wood, for his insights, hard work, and jokes in our collaborations, and for his unparalleled ability to come up with creative acronyms. William Mansky is a magician of formal proofs, and I'm grateful for him teaching me how to do that. I'm grateful for the PL insights and perspectives from Steve Zdancewic, and for his helpful feedback on writing. I'm amazed at the incredible coding abilities of Arial Eizenberg, and I appreciate learning so much from his code. I also want to thank Christian DeLozier, for all the ideas and resources he kindly shared. I appreciate the energy and insights in research from Brandon Lucia. I owe my special thanks to Prof. Yuebin Bai, who gave me significant help and support for starting my adventure in research back in college.

I also got a tremendous amount of help from faculty members at Penn. I appreciate the constructive feedback from Andreas Haeberlen, and helpful advice from Steve Zdancewic, on my WPE II writing and presentation. I also want to thank Steve Zdancewic, Rajeev Alur, Mayur Naik, and Vinod Grover, for kindly agreeing to be my dissertation committee, and for their feedback on my thesis proposal.

In the summer of 2016, I had the opportunity to work with Vinod Grover at NVIDIA. I'm grateful for his insights and help in my intern project, and I'm happy that we had the opportunity to continue our collaboration after the internship. I also want to thank my colleagues at NVIDIA, especially to Yang Chen, Sean Lee, Mahesh Ravishankar, and Thibaut Lutz, for their help in hacking the NVCC compiler.

At Penn, my fellow students have made my life more rewarding. I would like to thank all past and current members of ACG, including Brooke Candelore, Christian DeLozier, Ariel Eizenberg, Omar Navarro Leija, Luo Liang, Laurel Emurian Mirarchi, Sameer Railkar, Kelly Shiptoski, Nimit Singhania, Akshitha Sriraman, Abhishek Udupa, for the great atmosphere of the group. I'm also grateful for making friends with many students in the department, including Yu Wang, Meng Xu, Xie Long, Sean Welleck, Sepehr Assadi, Yang Li, Teng Zhang, and Jizhou Yan: thank you all for making my time at Penn more memorable.

Finally, I would like to thank my family. I'm grateful for my parents, for their love to, support for and faith in me throughout my education. I want to thank my sister, for being a role model of mine when I was a kid and for taking care of my parents when I was abroad doing my Ph.D. Last but certainly not the least, a few words to my wife and soul mate, Jing. You have been a wonderful lover and partner in this journey, and I'm grateful for your patience and faith in me for all these years, especially during the difficult days, when we had to be physically apart and had no money. You made my life colorful, bringing smiles and laughs into our daily lives. Thanks for all the company and consolation, in all these days when we are not wealthy yet happy. I have a lifetime of words for you, but for now let me stop with one more thing: after all the difficulties you've encountered and sacrifices you've made, thank you for being such a warrior for your dream, and for being such a unique YOU that I love and am proud of, for just the way you are.

ABSTRACT

EFFICIENT PRECISE DYNAMIC DATA RACE DETECTION FOR CPU AND GPU

Yuanfeng Peng

Joseph Devietti

Data races are notorious bugs. They introduce non-determinism in programs behavior, complicate programs semantics, making it challenging to debug parallel programs. To make parallel programming easier, efficient data race detection has been a research topic in the last decades. However, existing data race detectors either sacrifice precision or incur high overhead, limiting their application to real-world applications and scenarios. This dissertation proposes approaches to improve the performance of dynamic data race detection without undermining precision, by identifying and removing metadata redundancy dynamically. This dissertation also explores ways to make it practical to detect data races dynamically for GPU programs, which has a disparate programming and execution model from CPU workloads. Further, this dissertation shows how the structured synchronization model in GPU programs can simplify the algorithm design of data race detection for GPU, and how the unique patterns in GPU workloads enable an efficient implementation of the algorithm, yielding a high-performance dynamic data race detector for GPU programs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

With the proliferation of multicore processors in everything from servers to wearables, parallel programming has become ever more critical to efficiently and effectively utilizing modern processors. However, how to write correct and efficient multithreaded programs remains a well-known challenge in parallel computing. In particular, the potential for programs to contain data races is an issue that programmers often have to address. Data races can introduce non-sequentially-consistent[69] and/or undefined behavior[13] into programs, making programs hard to understand and reason about. In fact, data races can cause disastrous consequences in the real world: they are the culprits in the Therac-25 disaster[62], the Northeastern electricity blackout of 2003[43], and the mismatched NASDAQ Facebook share prices of 2012[56, 10]. Unfortunately, even experienced programmers can write code that contains data races, especially as the complexity of a parallel program grows.

To prevent potential disastrous outcomes, programmers typically need to prevent the presence of data races in their code, a process that involves extensive testing and debugging efforts. A useful tool for debugging parallel programs is data race detectors; in fact, identifying general data races is a key part of several algorithms for checking or enforcing higher-level properties of multithreaded programs, such as atomicity[37, 42] or determinism[85, 24]. As a result, data race detection has been an attractive research topic for the past a few decades, and there exists extensive work on static, dynamic, or hybrid detection schemes[1, 79, 38, 31, 100, 25, 54, 117, 101, 113, 39, 91, 68].

Unfortunately, building data race detectors that are both efficient and precise is a challenging task[31, 38, 41]. Existing approaches remain limited by missing true data races, reporting false data races, or incurring prohibitively high run-time overheads. It remains an open question whether efficient, precise data race detection can be provided in production scenarios.

Another important aspect of data race detection research is to support parallel programs that run on hardware other than CPUs[119, 75, 118, 64] . In particular, with the increasing ubiquity of GPU hardware and applications, efficient data race detection schemes for GPU programs have become more and more necessary. However, data race detectors designed for the CPU execution model typically cannot scale to the large numbers of threads in a GPU program[81], thus cannot be applied. With more complex GPU applications emerging, the need for an efficient data race detector for GPU is increasingly pressing.

This work proposes approaches to improve the efficiency of dynamic data race detection without sacrificing precision for CPU, and algorithms to enable practical data race detection for GPU. A key insight in improving the efficiency of dynamic data race detectors is that redundancy abounds in the metadata maintained by the detectors, which promises potential performance gain if such redundancy can be reduced properly. Full precision can be guaranteed as long as the redundancy reduction is lossless, i.e., no useful information is lost in the reduction. With effective metadata redundancy reduction, both the temporal and spatial overheads of dynamic data race detection can be improved. The improvement is even more significant if the reduction is done in hardware, making practical a hardware-assisted race detection system that can be always-on.

Further, the general idea of identifying and reducing redundant metadata can be also useful in scaling classical data race detection algorithms designed for CPU applications to GPU programs, enabling the first fully precise dynamic data race detector for GPU. To build an even more efficient GPU data race detector, the next part of this work is to take into account the structured synchronization paradigm and common memory access patterns when designing an algorithm for GPU data race detection. This enables the construction of a new GPU data race detector that outperforms prior industry-level tools while providing stronger detection capability.

The subsequent chapters of this dissertation are organized as follows. Chapter 2 provides relevant background knowledge and related work about the data race detection problem. Chapter 3 explains the metadata redundancy in dynamic data race detection and introduces the SlimFast system, a pure-software scheme to reduce such redundancy and improve performance. Chapter 4 next discusses metadata redundancy shown at the hardware level and describes the design of Parsnip, a hardware-assisted data race detector that manages redundancy reduction in hardware. Chapter 5 switches focus to data race detection for GPU, by presenting Barracuda, a data race detector that scales classical CPU-oriented algorithm to GPU. Chapter 6 introduces a more efficient algorithm design for GPU race detection and describes Curd, an efficient implementation of the algorithm that provides higher performance and coverage than previous tools. Chapter 7 discusses related work to this dissertation. Chapter 8 concludes the dissertation.

This dissertation draws on multiple published works. The SlimFast [87] system, described in Chapter 3, was originally presented at IPDPS 2018. The Parsnip [89] design, described in Chapter 4, was originally published at MICRO 2017. The Barracuda [29] system, introduced in Chapter 5, was originally presented at PLDI 2017. The Curd [88] detector was presented at PLDI 2018, and Chapter 6 explains the design of Curd and includes evaluation results of it. The remaining content is original work.

CHAPTER 2

Background

Data races have been a topical research problem. This chapter provides a brief overview of the general data race detection problem.

2.1. Data races

A multithreaded program can be modelled as a single trace of operations, with operations from each thread interleaved. Operations consist of memory reads and writes, lock acquires and releases, and thread fork and join. The *happens-before* relation $\xrightarrow{hb}$ is a partial order over these trace events. Given events a and b, we say a happens before b, written $a \xrightarrow{hb} b$, if:

- a and b are from the same thread and a precedes b in program order; or
- a precedes b in synchronization order, e.g., a is a lock release $relt(m)$ and b the subsequent acquire $acqt(m)$; or
- (a, b) is in the transitive closure of program order and synchronization order.

If a *happens-before* b then we can equivalently say that b *happens-after* a. Two events not ordered by the *happens-before* relation are said to be concurrent. Two memory accesses to the same address form a *data race* if they are concurrent and at least one access is a write.

The impact of data races on a program's behaviour can be illustrated using a simple example in Figure 2.1. The program in this example has two threads, $Thread1$ and $Thread2$, and a shared variable x. $Thread1$ writes to x on line 1 and line 2, and $Thread2$ reads the value of x on line 1. As the writes by $Thread1$ and the read by $Thread2$ are not ordered by any synchronization operation, there are two instances of data races, namely (line 1 by $Thread1$, line 1 by $Thread2$) and (line 2 by $Thread1$, line 1 by $Thread2$). Because the value of x read by $Thread2$ can be either 1 or otherwise, the behavior of $Thread2$ becomes unpredictable: either $foo()$ or $bar()$

```
  Thread 1           Thread 2
   x = 0           if x == 1:
   x = 1               foo()
                   else:
                       bar()
```

Figure 2.1: Illustrative example of a program with data races.

can be called.

As data races can cause perplexing program behavior, detection of such conditions are often desirable. Both static and dynamic data race detection approaches exist. While static detectors can have the potential of detecting data races on all execution paths, due to the inherent conservatism of static analyses, static detection tends to report a large number of false races, limiting its application. On the other hand, dynamic detectors only focus on the observed execution paths of a program but can be fully precise (i.e. no false or missed races) in catching all data races that actually occurred in the execution. Since the work of this dissertation is focused on precise dynamic data race detection, discussion about static data race detection is elided. This chapter will next focus on precise dynamic data race detection.

2.2. Precise Dynamic Data Race Detection

An extensive body of prior work on precise dynamic data race detection exists. These detectors dynamically monitor or record relevant events in a program's execution, maintain a conceptual graph of the happens before relation, and use this graph to decide whether a given pair of events are concurrent. Typically, an access history is tracked per-location, to record the relevant memory events that need to be checked for potential races. As an unbounded amount of memory can be shared across multiple threads, the per-location access histories can incur high overhead.

The main limitation of full-precision dynamic data race detectors is their high overhead. To mitigate this, research efforts have been made along several different directions. Some detectors trade precision for performance, e.g. by using sampling to check only a subset of events, or by keeping only a bounded size of access histories, risking missing real data races. There is also work on optimizing the representation of the access histories and encoding of the happens beforerelation, but the slowdown and memory consumption of the detectors can still be prohibitive. Another approach is to design hardware-assisted race detection systems, but prior work along this line requires strong assumptions, such as type-safety and extra processor cores, to hold. These limitations of existing precise dynamic data race detectors motivate the work presented in this dissertation.

2.3. Vector Clock Algorithm For Dynamic Race Detection

A classic algorithm for precise dynamic race detection is the vector clock algorithm[36, 44], which lays the foundation of numerous subsequent data race detectors. To facilitate later discussions, this section provides a brief overview of the vector clock based data race detection algorithm.

Vector clocks are a data structure to represent and track the *happens-before* relation at runtime. A vector clock V records a timestamp for each thread t in a system, written as $V(t)$. The standard comparison ($\sqsubseteq$), join ($\sqcup$) and increment (inc_t) operations on vector clocks are defined as follows:

$$\begin{array}{rcl} V \sqsubseteq V' & \text{iff} & \forall t.\ V(t) \leqslant V'(t) \\ V \sqcup V' & = & \lambda t.\ \max(\ V(t), V'(t)\ ) \\ inc_t(V) & = & \lambda u.\ \text{ if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \end{array}$$

The *happens-before* relation can be implemented by the comparison ($\sqsubseteq$) operation of vector clocks. In a vector clock race detector, four kinds of state are kept:

- For each thread t , vector clock C_t represents the last event in each thread that happens before the current logical time of t.
- For each lock m, vector clock L_m represents the last event in each thread that happens before the last release of m.
- For each address x, vector clock M_x^R represents the time of each thread's last read of x.

If thread t has not read x, then $M_x^R(t) = 0$.

- For each address x , vector clock M_x^W represents the time of each thread's last write of x. If thread t has not written x, then $M_x^R(t) = 0$.

The race detection algorithm works as follows. Initially, set all vector clocks L_m, M_x^R and M_x^W to $v0$, where $\forall t, v0(t) = 0$. The initial vector clock for each thread is C_t, where each thread increments its own entry in its vector clock, *i.e.*, $C_t(t) = 1$ and $\forall u \neq t, C_t(u) = 0$. This represents the fact that threads are initially executing concurrently with respect to one another.

On a lock acquire $acq_t(m)$, set $C_t := C_t \sqcup L_m$. By acquiring lock m, thread t has synchronized with all events that happened before the last release of m, so t increases its logical clock to be well-ordered with these prior events. On a lock release $rel_t(m)$, set $L_m := C_t$, ordering t with events that happened before this release, then increment the entry for t in its own vector clock C_t to ensure that subsequent events in t are marked as concurrent with respect to other threads.

On a read $rd_t(x)$, first check if $M_x^W \sqsubseteq C_t$. Failure of this check indicates a data race, where the read of x by t is concurrent with some previous write to x. Otherwise, set the entry for t in M_x^R to the current logical clock of t, *i.e.*, $M_x^R(t) := C_t(t)$. On a write $wr_t(x)$, check that t is well-ordered with respect to the previous writes $M_x^W \sqsubseteq C_t$ and with respect to previous reads $M_x^R \sqsubseteq C_t$. Failure of either of these checks indicates a data race. If the write by t is race-free, set entry for t in M_x^W to the current logical clock of t *i.e.*, $M_x^W(t) := C_t(t)$.

Vector clocks provide an effective way to encode the *happens-before* relation, enabling checking for concurrent events via vector operations. Since its introduction, numerous dynamic race detectors based on vector clocks have been proposed, such as ThreadSanitizer [103], FastTrack [38], etc.. However, these tools incur high overheads. To show the efficiency problem of vector clocks, a simple calculation is sufficient. Let N, M, X be the number of threads, memory access events, and shared memory locations, respectively. The temporal complexity for checking all accesses to all shared memory locations by all threads is $O(N * M)$, and the spatial overhead for keeping the states (metadata) is $O(N * X + N^2)$. As a result, the overheads of vector clock based race detectors can make them impractical to apply, especially when a program has a large number of threads or a large amount of shared memory.

CHAPTER 3

Metadata Redundancy Reduction in Dynamic Race Detection

For precise dynamic data race detectors, one significant source of overhead is the huge amount of metadata that needs to be maintained. For example, as was shown in the previous chapter, the spatial complexity of vector-clocks based algorithms grows quadratically with respect to the number of threads, and linearly with respect to the number of shared memory locations.

Fortunately, it is possible to reduce the metadata overhead significantly. A key observation that enables the work of this dissertation is that redundancy abounds in the metadata maintained for precise dynamic data race detection. This chapter first describes metadata redundancy that exists in dynamic race detection, then presents SlimFast, a software-solution to reduce such redundancy.

3.1. Metadata Redundancy

Race detection algorithms rely on per-location metadata which tracks the most recent reads and writes to each location. Using this metadata, a race detector can identify conflicting concurrent accesses that indicate a data race.

Cross-location metadata redundancy arises if multiple memory locations follow the same access pattern during program execution. Figure 3.1 shows a trace of operations from a single thread and the metadata state for each variable after each operation in a vector clock based race detector. Initially, the metadata of the three variables, X, Y, Z, has the same value. When X is accessed, its metadata is updated according to the vector clock algorithm. When the accesses to Y and Z happen, the logical time of the accessing thread is the same with the time when X is accessed; therefore, the resulting states of metadata for X, Y, Z are identical. There exists redundancy in the metadata of X, Y, Z, as it is unnecessary, from a correctness standpoint, to maintain separate copies of metadata for multiple variables whose metadata is identical. Generally, cross-location metadata redundancy can arise via several access patterns, *e.g.*, whenever X and Y are written by the same thread without an intervening release synchronization operation, these two writes occur with the same logical timestamp. Thus, after the second write, the metadata for X and Y will have the same value.

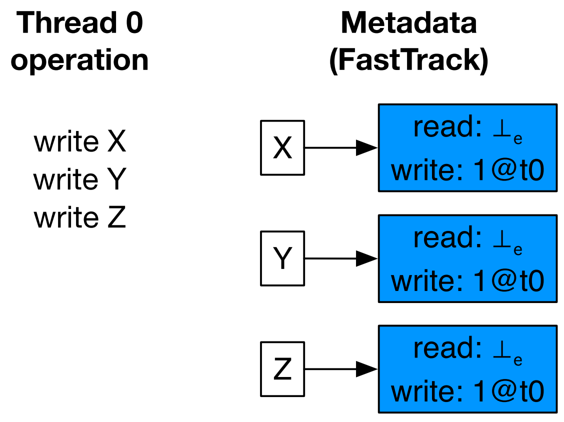


Figure 3.1: Example trace showing metadata redundancy between X, Y, Z. The metadata is in the format of FastTrack.

To measure the metadata redundancy in real data race detectors, an experiment is conducted. In this experiment, FastTrack [38], an optimized vector clock based data race detector, is modified such that a copy of all of its metadata is stored in a global set. If two metadata instances with the same value are inserted into this unique set, only one copy will be retained. The size of the unique set thus indicates the number of unique metadata objects that are generated during the course of a program's execution. The modified FastTrack implementation is run on a number of programs, to collect a metric called *redundancy ratio*. The definition of this metric is as follows:

$$redundancy\ ratio = \frac{number\ of\ all\ metadata\ objects}{number\ of\ unique\ metadata\ objects}$$

Intuitively, higher *redundancy ratio* suggests more redundancy in the metadata. Figure 3.2 shows the *redundancy ratio* collected in the execution of each benchmark.

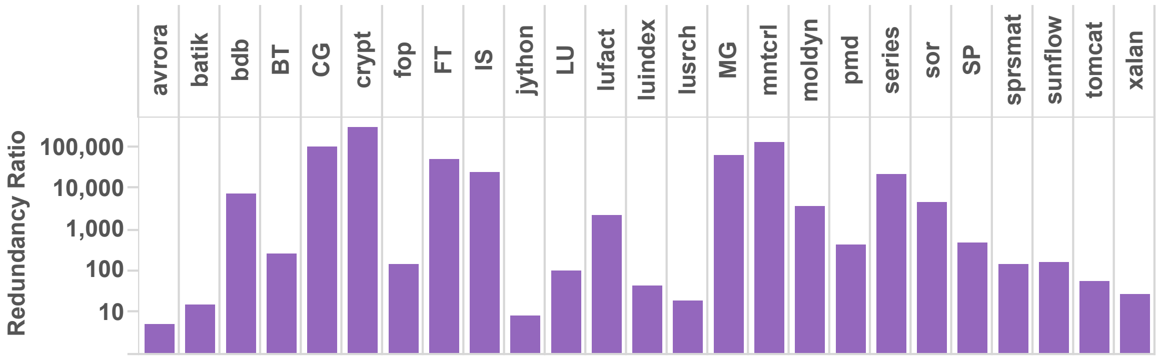


Figure 3.2: Redundancy ratio of programs from benchmark suites including DaCapo 2009, Java Grande, and NAS Parallel Benchmark 3.0.3.

As shown by the data in Figure 3.2, there typically exist several orders of magnitude of metadata redundancy in these programs. Removing this redundancy can yield substantial space and time savings for dynamic race detection without sacrificing precision.

3.2. The SlimFast System

The key idea of the SlimFast system is to share metadata states across multiple memory locations. This section first gives a brief introduction to the baseline data race detector, FastTrack, followed by a description of the core design and implementation of SlimFast.

3.2.1. Baseline system: FastTrack

FastTrack is a vector-clock based dynamic data race detector. Figure 3.3 shows the operational semantics of FastTrack.

Compared to the classic vector clock race detection algorithm, FastTrack makes an optimization to save space on the per-location access histories. In particular, instead of keeping a

$$\textsc{ReadSameEpoch}\;\dfrac{M^R_x = E(t)}{(C, L, M) \Rightarrow^{rd(t,x)} (C, L, M)}$$

$$\textsc{ReadSharedSameEpoch}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPlusVC} \\ M^W_x \preceq C_t \\ M^R_x[t] = C_t(t) \end{array}}{(C, L, M) \Rightarrow^{rd(t,x)} (C, L, M)}$$

$$\textsc{ReadShared}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPlusVC} \\ M^W_x \preceq C_t \\ M' = M[x := (M^R_x[t := C_t(t)], M^W_x)] \end{array}}{(C, L, M) \Rightarrow^{rd(t,x)} (C, L, M')}$$

$$\textsc{ReadInflate}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPair} \\ M^W_x \preceq C_t \\ M^R_x = c@u \\ V = \bot_V[t := C_t(t), u := c] \\ M' = M[x := (V, M^W_x)] \end{array}}{(C, L, M) \Rightarrow^{rd(t,x)} (C, L, M')}$$

$$\textsc{ReadExclusive}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPair} \\ M^R_x \preceq C_t \qquad M^W_x \preceq C_t \\ M' = M[x := (E(t), M^W_x)] \end{array}}{(C, L, M) \Rightarrow^{rd(t,x)} (C, L, M')}$$

$$\textsc{WriteSameEpoch}\;\dfrac{M^W_x = E(t)}{(C, L, M) \Rightarrow^{wr(t,x)} (C, L, M)}$$

$$\textsc{WriteExclusive}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPair} \\ M^R_x \preceq C_t \qquad M^W_x \preceq C_t \\ M' = M[x := (\bot_e, E(t))] \end{array}}{(C, L, M) \Rightarrow^{wr(t,x)} (C, L, M')}$$

$$\textsc{WriteShared}\;\dfrac{\begin{array}{c} M_x \in \mathit{EpochPlusVC} \\ M^R_x \sqsubseteq C_t \qquad M^W_x \preceq C_t \\ M' = M[x := (\bot_e, E(t))] \end{array}}{(C, L, M) \Rightarrow^{wr(t,x)} (C, L, M')}$$

$$\textsc{Fork}\;\dfrac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, M) \Rightarrow^{fork(t,u)} (C', L, M)}$$

$$\textsc{Join}\;\dfrac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, M) \Rightarrow^{join(t,u)} (C', L, M)}$$

$$\textsc{Acquire}\;\dfrac{C' = C[t := (C_t \sqcup L_m)]}{(C, L, M) \Rightarrow^{acq(t,m)} (C', L, M)}$$

$$\textsc{Release}\;\dfrac{\begin{array}{c} L' = L[m := C_t] \\ C' = C[t := inc_t(C_t)] \end{array}}{(C, L, M) \Rightarrow^{rel(t,m)} (C', L', M)}$$

Figure 3.3: FastTrack operational semantics. Shading indicates rules with different semantics in SlimFast.

full vector clock M^W_x to store the last writes by all threads, FastTrack uses a scalar value (called an *epoch* in FastTrack terminology) to store only the last write by the last thread. For the history of last reads, a full vector clock is used only when there are concurrent reads since the last write, otherwise, a scalar epoch is used to read the last read. It can be proved that this reduction still enables the race detector to precisely detect the first, if any, data race that occurs in an observed execution. With this modification on the classic vector clock algorithm, the FastTrack detector employs two metadata formats: a smaller format containing an *EpochPair* and a larger *EpochPlusVC* format containing a write epoch and a vector clock. Figure 3.4 illustrates the information encoded in each of these two formats of metadata.

3.2.2. SlimFast Operational Semantics

As described, metadata maintained by race detectors like FastTrack can be highly redundant across memory locations. To reduce such redundancy, SlimFast has a different metadata-management design from FastTrack, which enables the same metadata object to be shared across different memory locations. In particular, with SlimFast, race detection metadata in

```
class EpochPair {
  Epoch lastWrite;
  Epoch lastRead;
};

class EpochPlusVC {
  Epoch lastWrite;
  int[] lastReads;
};
```

Figure 3.4: Information encoded in EpochPair and EpochPlusVC in FastTrack.

EpochPair format is immutable so that it can be shared safely across threads.

Figure 3.5 shows the operational semantics of SlimFast, with shaded boxes marking rules that are different from FastTrack. These rules present several optimizations of SlimFast to reduce metadata redundancy.

Optimizing Writes

The first case SlimFast can reduce redundancy arises when writes happen. Observe that all of a thread t's writes within a given epoch update M_x^W with an identical value. Figure 3.6 gives an example where, within each epoch (*i.e.* logical time of a thread), the two writes to x and y result in $M_x^W = M_y^W$. Maintaining distinct metadata for each location is thus unnecessary.

To eliminate redundant metadata for such writes, SlimFast introduces a new piece of state $\mathbb{W} : Tid \rightarrow$ *EpochPair* to optimize write operations. Each thread t maintains an *EpochPair* $\mathbb{W}_t$ consisting of $(\bot_e, E(t))$, *i.e.*, an empty read epoch and t's current write epoch. $\mathbb{W}_t$ is updated whenever t's current epoch is updated, *i.e.*, on a Release (as in Figure 3.6) or Fork operation. Whenever t performs a write operation on a location x that updates x's metadata (FastTrack rules WriteExclusive or WriteShared), x's metadata can be set directly to $\mathbb{W}_t$. This eliminates redundancy from write operations in just $O(1)$ time.

$$\textsc{ReadSharedReuse}\ \frac{\begin{array}{c} M_x \in \textit{EpochPlusVC} \\ M_x^W \preceq C_t \\ M_x^{Next}(t)^R(t) = C_t(t) \\ M' = M[x := M_x^{Next}(t)] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{ReadSharedAlloc}\ \frac{\begin{array}{c} M_x \in \textit{EpochPlusVC} \\ M_x^W \preceq C_t \\ M_x^{Next}(t)^R(t) \neq C_t(t) \\ M_x'^R = M_x^R[t := C_t(t)] \\ N = M_x^{Next}[t := (M_x'^R, M_x^W, \bot_{Next})] \\ M_x^{Next} := N \\ M' = M[x := N(t)] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{ReadExclReuse}\ \frac{\begin{array}{c} M_x \in \textit{EpochPair} \\ M_x^R \preceq C_t \qquad M_x^W \preceq C_t \\ (E(t), M_x^W) \in \mathbb{S}_t \\ M' = M[x := \mathbb{S}_t[(E(t), M_x^W)]] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{ReadExclAlloc}\ \frac{\begin{array}{c} M_x \in \textit{EpochPair} \\ M_x^R \preceq C_t \qquad M_x^W \preceq C_t \\ (E(t), M_x^W) \notin \mathbb{S}_t \\ \mathbb{S}' = \mathbb{S}[t := \mathbb{S}_t \cup \{(E(t), M_x^W)\}] \\ M' = M[x := \mathbb{S}'_t[(E(t), M_x^W)]] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}', \mathbb{Q}, \mathbb{W})}$$

$$\textsc{Fork}\ \frac{\begin{array}{c} C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)] \\ \mathbb{S}' = \mathbb{S}[t := \varnothing] \qquad \mathbb{Q}' = \mathbb{Q}[t := \varnothing] \\ \mathbb{W}' = \mathbb{W}[t := (\bot_e, C'_t(t)@t)] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{fork(t,u)} (C', L, M, \mathbb{S}', \mathbb{Q}', \mathbb{W}')}$$

$$\textsc{ReadInflReuse}\ \frac{\begin{array}{c} M_x \in \textit{EpochPair} \\ M_x^W \preceq C_t \\ M_x^R = c@u \\ V = \bot_V[t := C_t(t), u := c] \\ (V, M_x^W, N) \in \mathbb{Q}_t \\ M' = M[x := \mathbb{Q}_t[(V, M_x^W, N)]] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{ReadInflAlloc}\ \frac{\begin{array}{c} M_x \in \textit{EpochPair} \\ M_x^W \preceq C_t \\ M_x^R = c@u \\ V = \bot_V[t := C_t(t), u := c] \\ (V, M_x^W, \bot_{Next}) \notin \mathbb{Q}_t \\ \mathbb{Q}' = \mathbb{Q}[t := \mathbb{Q}_t \cup \{(V, M_x^W, \bot_{Next})\}] \\ M' = M[x := \mathbb{Q}'_t[(V, M_x^W, \bot_{Next})]] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rd(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}', \mathbb{W})}$$

$$\textsc{WriteExcl}\ \frac{\begin{array}{c} M_x \in \textit{EpochPair} \\ M_x^R \preceq C_t \qquad M_x^W \preceq C_t \\ M' = M[x := \mathbb{W}_t] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{wr(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{WriteShared}\ \frac{\begin{array}{c} M_x \in \textit{EpochPlusVC} \\ M_x^R \sqsubseteq C_t \qquad M_x^W \preceq C_t \\ M' = M[x := \mathbb{W}_t] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{wr(t,x)} (C, L, M', \mathbb{S}, \mathbb{Q}, \mathbb{W})}$$

$$\textsc{Release}\ \frac{\begin{array}{c} L' = L[m := C_t] \\ C' = C[t := inc_t(C_t)] \\ \mathbb{S}' = \mathbb{S}[t := \varnothing] \qquad \mathbb{Q}' = \mathbb{Q}[t := \varnothing] \\ \mathbb{W}' = \mathbb{W}[t := (\bot_e, C'_t(t)@t)] \end{array}}{(C, L, M, \mathbb{S}, \mathbb{Q}, \mathbb{W}) \Rightarrow^{rel(t,m)} (C', L', M, \mathbb{S}', \mathbb{Q}', \mathbb{W}')}$$

Figure 3.5: SlimFast operational semantics. Shading indicates differences from corresponding FastTrack rule.

Optimizing Reads Involving EpochPair Metadata

When a read happens and the metadata format is in *EpochPair* after the read, redundancy can be potentially removed. The *EpochPair* format is of constant size and is highly amenable to compression via SlimFast. To optimize *EpochPair* reads, a set of *EpochPair*s for each thread $\mathbb{S} : Tid \rightarrow \{$*EpochPair*$\}$ is introduced. This set contains all *EpochPair* values that were used to update metadata due to reads. Whenever an update is about to occur, a thread checks its set $\mathbb{S}_t$ to see if the value already exists there, reusing the existing metadata if possible.

Consider the case of t performing a read that triggers an update of *EpochPair* metadata (the FastTrack ReadExclusive rule). In SlimFast, before t performs its metadata update, it consults $\mathbb{S}_t$ to determine whether the new metadata value already exists and can be reused (ReadExclReuse) or whether new metadata needs to be allocated (ReadExclAlloc).

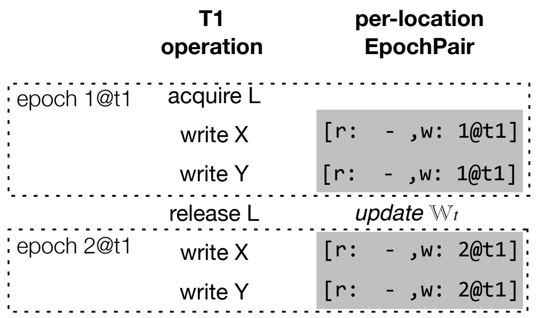


Figure 3.6: The metadata used by a given thread for write operations is invariant within an epoch (shaded boxes).

Optimizing EpochPair to EpochPlusVC Inflations

Removing redundancy from *EpochPlusVC* metadata is challenging because of their size: redundancy checks require $O(n)$ time where n is the number of threads. Fortunately, the structure of *EpochPlusVC* updates can be exploited to eliminate most redundancy in just $O(1)$ time.

The state $\mathbb{Q} : Tid \rightarrow \{$*EpochPlusVC*$\}$ is used to remove redundancy arising from *EpochPair* to *EpochPlusVC* inflations triggered by concurrent reads (SlimFast's ReadInfl* rules). $\mathbb{Q}_t$ describes a thread t's set of *EpochPlusVC* metadata. The $\mathbb{Q}_t$ set is used similarly to the $\mathbb{S}_t$ set, except that $\mathbb{Q}_t$ holds *EpochPlusVC*s instead of *EpochPair*s.

Consider a concurrent read by a thread t to a location with *EpochPair* metadata. With SlimFast, t checks $\mathbb{Q}_t$ to determine whether existing metadata can be reused (ReadInflReuse) or needs to be allocated and added to $\mathbb{Q}_t$ (ReadInflAlloc). The *EpochPlusVC* metadata in $\mathbb{Q}_t$ uphold several invariants. There are only ever two non-empty entries in each *EpochPlusVC*'s vector clock (as can be seen from ReadInflAlloc). One of the non-empty entries is the remote read from the previous *EpochPair* metadata. The other non-empty entry represents t's read and is equal to t's current clock because $\mathbb{Q}_t$ is cleared on every Release and Fork. Thus, $\mathbb{Q}_t$ supports O(1) lookups using just the remote read M_x^R and the last write M_x^W.

```
class EpochPlusVC {
  Epoch W; // immutable
  int[] R; // immutable
  Map <Tid,EpochPlusVC> Next; // mutable
}
```

Figure 3.7: The EpochPlusVC format in SlimFast.

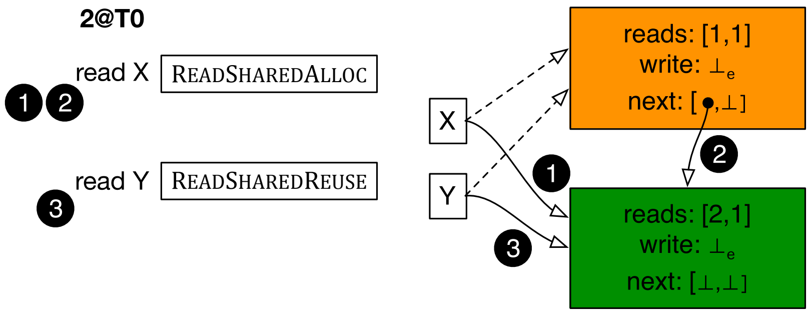


Figure 3.8: The accesses on the left illustrate how the $Next$ map allows metadata sharing across locations x and y.

Optimizing EpochPlusVC Updates

In SLIMFAST, *EpochPlusVC* metadata M_x is extended with a map $M_x^{Next} : Tid \rightarrow$ *EpochPlusVC* (Figure 3.7). The $Next$ map eliminates most *EpochPlusVC* redundancy in O(1) time. The M_x^{Next} map is *mutable*, unlike other SLIMFAST metadata. For ease of discussion, the notation $M_x^{Next}(t)$ is used to extract the *EpochPlusVC* for thread t, and $\perp_{Next}$ for the empty map.

The core insight that enables optimizing EpochPlusVC updates is as follows. For a read of *EpochPlusVC* metadata that triggers an update (FASTTRACK's READSHARED rule), a thread t only updates its own entry of the vector clock. Figure 3.8 shows how the $Next$ map leverages this structure. Suppose that locations x and y initially share *EpochPlusVC* metadata (dashed arrows) due to previous use of the READINFL* rules. When $T0$ reads x during epoch 2, it needs to update M_x^R from the top *EpochPlusVC* to the bottom one (❶). A link between these *EpochPlusVC*s is also created (❷), by updating the (mutable) $Next$ map in the *EpochPlusVC* previously used for M_x.

The $Next$ map serves as a fast way to find a new *EpochPlusVC* that differs from the old

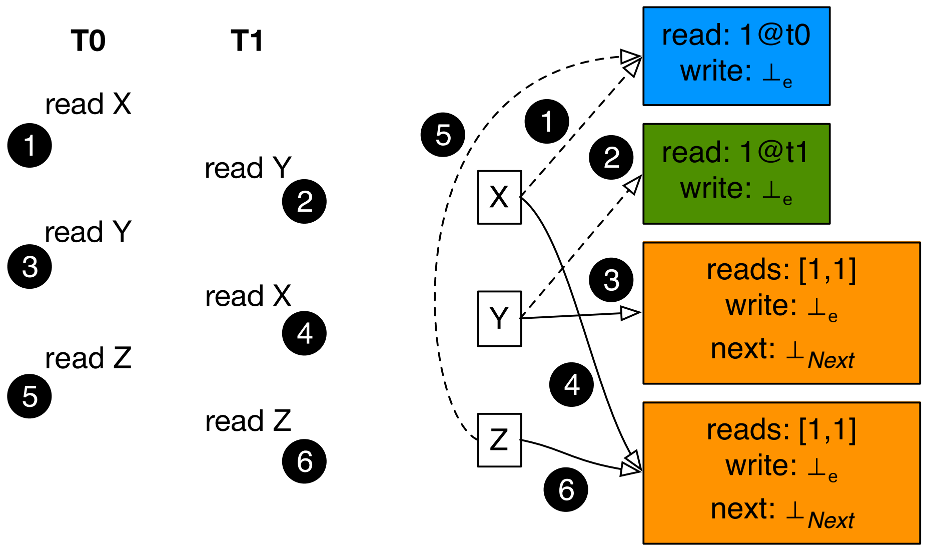


Figure 3.9: On the left, a series of accesses by two threads resulting in EpochPlusVC redundancy. The right shows the metadata for each location just after each access.

EpochPlusVC only in the value of a particular thread's vector clock entry. When $T0$ reads location y (which initially uses the top *EpochPlusVC*), $M_y^{Next}(t)$ can be checked in O(1) time to see if the desired metadata already exists. In this case it does, so the bottom *EpochPlusVC* can be reused for location y (❸). In Figure 3.5, we write $M_x^{Next}(t)^R(t)$ to represent accessing t's entry of the *EpochPlusVC* M_x^{Next}, and then accessing t's entry of the read vector clock of that *EpochPlusVC* to check whether the metadata has the desired value.

The price of O(1)-time redundancy checking is occasional redundancy, as Figure 3.9 shows. The initial reads of x and y result in *EpochPair* metadata. Each thread's second access triggers inflation, but each thread's $\mathbb{Q}$ set is empty, so two pieces of *EpochPlusVC* metadata are allocated with identical values (ReadInflAlloc). Detecting such redundancy would require some kind of global, synchronized map where the costs of access would quickly outweigh the benefits of redundancy elimination. $T0$'s read of z reuses the *EpochPair* metadata (ReadExclReuse). $T1$'s read of z is able to reuse a previously-allocated *EpochPlusVC* from $\mathbb{Q}_{T1}$ (ReadInflReuse). At the end of this program, some but not all *EpochPlusVC* redundancy has been eliminated: all three locations have identical metadata but only two *EpochPlusVC*s are needed to represent this.

3.3. Implementation of SlimFast

SlimFast is implemented using the RoadRunner framework [40], on which the baseline system, FastTrack, is also implemented. The RoadRunner framework provides the shadow-memory implementation that SlimFast uses to store its metadata.

3.3.1. EpochPair and EpochPlusVC

SlimFast uses the *EpochPair* and *EpochPlusVC* classes to store its metadata. In our implementation, *EpochPlusVC* extends *EpochPair* with a vector clock and an array of references to *EpochPlusVC*s (Figure 3.7). On each variable access, SlimFast uses dynamic dispatch to call the correct method to check and update the metadata for that variable. This implementation wastes some space in *EpochPlusVC* due to an unnecessary read epoch field that is inherited from *EpochPair*. However, this implementation was slightly faster in practice than the alternative, space-efficient implementation in which *EpochPair* and *EpochPlusVC* inherit from a common base class.

3.3.2. Storing and Retrieving Immutable Metadata

As described in the previous section, SlimFast maintains per-thread sets $\mathbb{S}_t$ of immutable *EpochPair*s and $\mathbb{Q}_t$ of *EpochPlusVC*s. In practice, as the sizes of these sets are small (typically less than 10, see Section 3.4.4 for detailed data), implementing them using a fixed-size array is an efficient design choice. With a fixed-size array, overflow is possible. In such cases, wrap-around happens and the initial array entries are overwritten. This overwriting does not affect correctness, but it can lose some opportunity for redundancy reduction, because redundancy cannot be detected for an overwritten *EpochPair*. In practice, overflow never occurred in any of the 25 benchmarks used in the evaluation of SlimFast.

3.3.3. Reducing the Size of $\mathbb{S}_t$

As described, a $\mathbb{S}_t$ set grows without bound. To counter this, we need to be able to remove elements. We observe that in the ReadExcl* rules that access $\mathbb{S}_t$, $\mathbb{S}_t$ lookups only ever use the read epoch equal to t's current epoch $E(t)$. Thus, any *EpochPair* $(r, w) \in \mathbb{S}_t : r \neq E(t)$ can never be returned by any lookup. Figure 3.10 illustrates this invariant: no metadata with read epoch `2@t1` will be used for metadata updates once epoch `3@t1` begins. After each read of a location x during `3@t1`, M_x^R must instead be set to `3@t1`.

We can thus clear $\mathbb{S}_t$ whenever a new epoch begins, *i.e.*, on RELEASE and FORK, which keeps the size of $\mathbb{S}_t$ small. Note that clearing $\mathbb{S}_t$ does not affect the metadata objects referenced by the elements of $\mathbb{S}_t$. Such objects may be used in future checks, though never in future updates.

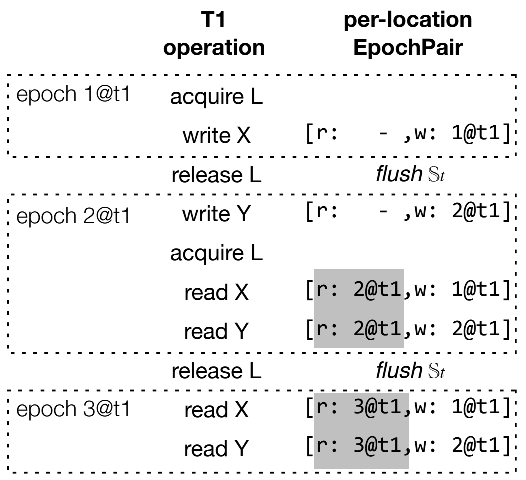


Figure 3.10: SlimFast exploits the invariant that EpochPair metadata updates only ever involve the current read epoch to reduce the size of $\mathbb{S}_t$ and make $\mathbb{S}_t$ lookups cheaper.

3.3.4. Optimizing $\mathbb{S}_t$ Lookups

When checking whether a given *EpochPair* is present in $\mathbb{S}_t$, we can optimize this lookup by observing that all the $\mathbb{S}_t$ lookups a thread t performs within a given epoch (the READEXCLREUSE and READEXCLALLOC rules) use the same value for the read epoch, namely $E(t)$. Combining this invariant with the fact that $\mathbb{S}_t$ only contains *EpochPair*s inserted during the current epoch (Section 3.3.3), $\mathbb{S}_t$ will only ever contain *EpochPair*s with a read epoch of $E(t)$. Thus, we can treat $\mathbb{S}_t$ as a map from $\textit{write epoch} \rightarrow$ *EpochPair*, avoiding comparisons that involve the read epoch.

Figure 3.10 illustrates these invariants: the shaded boxes show that the read epochs used in metadata updates are constant within an epoch. The first access in Epoch 2 does not have a shaded read epoch because writes do not access $\mathbb{S}_t$ and instead are set to $\mathbb{W}_t$. Removing the read epoch from $\mathbb{S}_t$ lookups accelerates the $\mathbb{S}_t$ hash function. For clarity we elide this optimization in the SlimFast semantics (Figure 3.5) but it is used in the implementation.

3.3.5. Enforcing Rule Atomicity

The semantics presented in Figure 3.5 rely on the fact that each rule executes atomically. The SlimFast implementation performs updates on local state and then makes them visible via an atomic operation.

Figure 3.11 shows the pseudocode for the SlimFast metadata update algorithm. We discuss the correctness of the cases below.

We observe that the mutable elements of M_x^{Next} are thread-private, as $M_x^{Next}[t]$ is only ever accessed by thread t (in the ReadShared* rules). Thus, operations on M_x^{Next} are trivially atomic.

```
1 START: Metadata e = M_x
2 if *SameEpoch: // Case 1
3   return
4 else: // Case 2
5   Metadata e' = updateMetadata(e)
6   if !CAS(&M_x, e => e'): goto START
7   return
```

Figure 3.11: SlimFast's metadata update algorithm.

Case 1: No Update The *SameEpoch rules do not need synchronization because metadata is immutable.

Case 2: Update For the remaining rules, the updateMetadata function encapsulates the logic of the corresponding rule. After the updated metadata is calculated, the CAS operation on line 12 is used to update the metadata for x via the pointer &M_x. The CAS ensures that the rule executes atomically by verifying that the metadata has not changed during the rule's execution. If the CAS operation fails, then there may have been concurrent updates that have changed the metadata state and so the update restarts from line 1.

We must take care to avoid ABA issues [74]. There are two classes of ABA issues to consider: memory reuse and direct metadata reuse. Memory reuse does not arise in SlimFast as it is implemented in Java, a garbage-collected language. Direct metadata reuse is subtler, but is avoided due to the *monotonicity* of metadata updates.

Monotonicity of Metadata Updates Once a metadatum is modified it never returns to its original value. We observe that a location's read or write epoch is updated, then the original write epoch can never be restored because an epoch $c@t$ will only ever be written by thread t, and t's epoch for itself never decreases. Similar reasoning shows that for *EpochPlusVC* metadata, $M_x^R(t)$ can never decrease.

3.4. Evaluation of SlimFast

The main goal of our evaluation is to measure how well SlimFast achieves its goal of reducing the space and runtime overheads of dynamic data race detection.

3.4.1. Experimental Setup

We use DaCapo 2009 [12], Java Grande [22, 72], NAS Parallel Benchmark 3.0.3 [45], and Oracle's BerkeleyDB (bdb) Java Edition 6.4.25 [86] to evaluate SlimFast. RoadRunner is incompatible with the h2, eclipse, daytrader, tradebeans in the DaCapo suite so we exclude them from our evaluation. We use the largest provided inputs for the DaCapo and Grande programs, and the A inputs for NPB_CG, NPB_MG, NPB_FT, NPB_IS, and the W inputs for NPB_BT, NPB_LU, NPB_SP, from the NAS benchmarks. bdb is a Java library with 330K LoC across 919 classes. We wrote a simple driver program that performs 5,000 sequential reads from a database of 1,000 integer keys with 64KB values.

We used RoadRunner version v0.3 with the fast-path optimization enabled. We use the lock-based version of FastTrack from RoadRunner v0.3. While a CAS-based implementation of FastTrack exists, it supports only 24-bit clock values which suffer from rollover on some

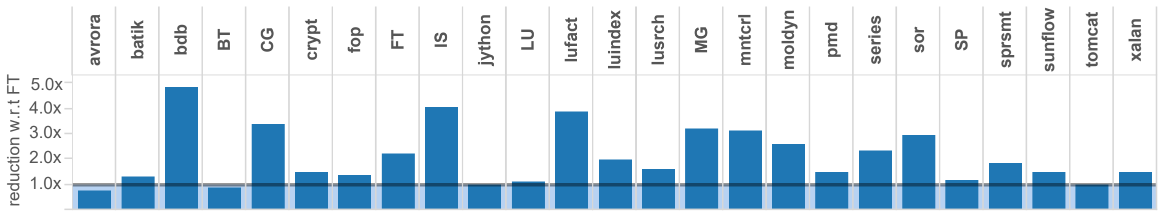


Figure 3.12: Average space reduction over FastTrack of the entire Java heap. Higher is better.

benchmarks [11]. The CAS-based FastTrack has equivalent performance to the lock-based version on average on our benchmarks, and is sometimes slower because the lock-based FastTrack can avoid CAS operations with the JVM's biased locking [26]. SlimFast uses 32-bit clocks to avoid rollover (and 8-bit thread IDs) and we modify FastTrack likewise. The default size of vector clocks used in both FastTrack and SlimFast is 4.

We found a data race in the FastTrack ReadShared* fast-path code where the read vector clock is read without holding any locks. To avoid the complicated semantics of races in the Java Memory Model, and to provide a fair comparison with the SlimFast implementation which we believe to be race-free, we have fixed this race. We found no meaningful difference in memory consumption from fixing the FastTrack race, though it does reduce performance. SlimFast improves performance by just 1.04x on average over the racey version of FastTrack, though it still offers substantial speedup on some individual benchmarks such as a 2.5x speedup on bdb. All subsequent performance comparisons use the fixed version of FastTrack.

All experiments were performed on a quad-socket machine consisting of four 2.0 GHz Intel Xeon E7-4820 (Westmere) chips (8 cores/16 threads each) with 128GB of RAM, running 64-bit Linux 3.11.10. All code was compiled with JDK 1.7.0_25 and run on HotSpot 64-Bit Server VM 23.25-b01, with a heap size of 64GB and the default Parallel Scavenger collector. Our results are for running each benchmark with 16 threads, unless specified otherwise or the benchmark only supports a fixed number of threads as with avrora (7 threads), bdb (6), batik (7), fop (1), jython (2), luindex (2), pmd (64), and tomcat (80).

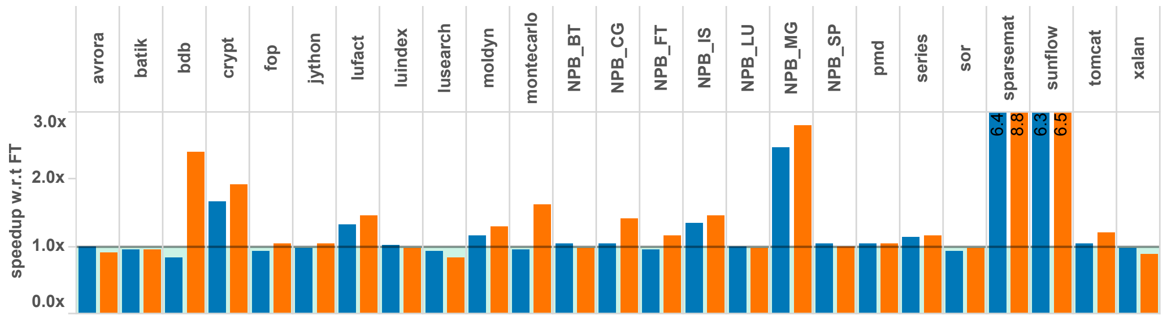


Figure 3.13: Average speedup of FastTrack-reference (dark bars) and SlimFast (light bars) over FastTrack. Higher is better.

3.4.2. Memory Savings

We measured the total average memory usage of SLIMFAST, FASTTRACK, and ROADRUNNER by forcing a garbage collection every 100ms, recording the heap usage after each collection, and averaging together these measurements. This measures the entire heap, including both application data and race detector state. Figure 3.12 shows the average space reduction of SLIMFAST over FASTTRACK. Among all 25 benchmarks we ran, SLIMFAST consumes 1.83x less memory on average than FASTTRACK, with up to 4.90x savings on bdb. Overall, SLIMFAST's space reduction is highly correlated with the metadata redundancy ratio of each benchmark (Figure 3.2).

SLIMFAST consumes more space than FASTTRACK for avrora and NPB_BT because these benchmarks have relatively low redundancy ratios: avrora has the lowest among all our benchmarks. For avrora and NPB_BT, SLIMFAST's *EpochPlusVC* space optimizations (principally the M_x^{Next} array inside each *EpochPlusVC* object) backfire and increase space usage mildly. When using a version of SLIMFAST with mutable *EpochPlusVC*s, SLIMFAST has less heap usage than FASTTRACK on all benchmarks.

3.4.3. Performance

To compare the performance of SLIMFAST and FASTTRACK, we ran each configuration 15 times and recorded the average runtime. We use a 64GB heap to minimize the impact of garbage collection, excluding the collection that ROADRUNNER always triggers upon start-up.

For ease of discussion, we refer to the original implementation of FastTrack that has the race as FastTrack-racy, and the version that has the race fixed as FastTrack-fixed.

The light bars in Figure 3.13 illustrate the average speedup of SlimFast over FastTrack-fixed. For 15 programs, SlimFast outperforms FastTrack-fixed, with speedups of up to 8.8x (sparsemat); for 4 programs, SlimFast runs slower than FastTrack-fixed, with the overhead ranging from 1.7% (batik) to 11.5% (xalan); for the rest of the programs, the difference in runtime is negligible between SlimFast and FastTrack-fixed. Overall, the geometric mean of speedups of SlimFast over FastTrack-fixed is 1.40x.

The dark bars in Figure 3.13 show the average speedup of FastTrack-racy over FastTrack-fixed. Fixing the implementation issues has a significant performance impact: FastTrack-racy is on average 1.24x faster than the baseline FastTrack-fixed. Note that SlimFast is still 1.12x faster than FastTrack-racy.

According to additional experiments we have done, the speedups SlimFast gains can mainly be attributed to three reasons. First, SlimFast's space reduction shrinks the working set of the instrumented programs, bringing better cache behavior. We validated this hypothesis using the Linux perf tool to see the miss rate of L1 data caches when running the programs with FastTrack and SlimFast: *e.g.*, the cache miss rate when running sparsemat with SlimFast is only 47% of that of FastTrack. Second, SlimFast's immutable *EpochPlusVC*s reduce contention over metadata, especially on sunflow and sparsemat which have many *EpochPlusVC* accesses. Third, as is the case with bdb, we found that FastTrack's additional space consumption and the application's naturally high rate of allocation trigger frequent garbage collections. SlimFast's reduction in heap size and lower metadata allocation rate avoid this behavior. Given that many of our workloads have small heap sizes, we expect SlimFast's performance advantage to increase as input sizes grow.

Why SlimFast is sometimes slower than FastTrack is also an interesting question. For avrora, lusearch, and xalan, the costs of immutable *EpochPlusVC*s sometimes outweigh their

	2			3	4	5	6	7	8	9	10	11
program	RR/FT/SF slowdown (x)			EVC accesses	EVC metadata	$\overline{\lvert\mathbb{S}_t\rvert}$	$\overline{\lvert\mathbb{Q}_t\rvert}$	EVC reuse	EP reuse	max heap (MB)	mem. w.r.t. RR (x)	FT:SF metadata allocations
avrora	1.4	1.6	1.7	3.80%	22.60%	6.6	-	2.00%	86.50%	82	1.45	0.1
batik	3.8	4.4	4.7	-	-	2.7	-	-	95.00%	327	1.06	12.2
bdb	12.7	103.3	38.7	-	-	6.20	-	-	100%	585	1.02	1,108.1
crypt	8.4	53.4	27.7	68.40%	29.40%	1.6	-	94.30%	100%	3,188	1.01	474,684.4
fop	5.3	6.2	5.7	-	-	0.9	-	-	99.90%	243	1.05	1,130.5
jython	6.2	7.1	6.8	-	-	0.5	-	-	91.10%	634	1.24	11.3
lufact	2.2	4.8	3.3	33.70%	62.30%	1.6	0.1	99.40%	100%	516	1.11	7.0
luindex	4.6	5.8	6.0	-	-	6.2	-	-	82.80%	131	1.26	0.4
lusearch	10.1	12.1	14.3	5.90%	22.40%	10.8	-	1.10%	89.80%	3,092	1.02	2.8
moldyn	5.4	9.3	7.0	66.10%	90.50%	3	0.1	99.70%	100%	126	1.08	12.8
montecarlo	2.7	13.3	8.2	0.70%	59.40%	1.9	0.2	43.80%	100%	3,909	1.12	499,458.2
NPB_BT	11.7	13.2	13.3	59.60%	51.80%	3.7	-	2.40%	99.80%	121	1.50	0.5
NPB_CG	8.1	29.1	20.5	32.70%	99.60%	3.3	1.4	5.90%	100%	553	1.10	0.1
NPB_FT	10.4	19.3	16.8	0.50%	85.10%	2.8	-	79.80%	100%	1,265	1.00	972.2
NPB_IS	30.1	33.2	22.8	63.20%	59.20%	2.9	0.2	13.90%	100%	871	1.11	975.6
NPB_LU	5.3	5.7	5.9	23.70%	43.00%	4.7	0.9	98.70%	99.80%	182	2.24	0.9
NPB_MG	22.5	79.1	28.2	48.20%	44.70%	2.4	0.6	100%	100%	3,690	1.02	1,289.5
NPB_SP	4.8	5.4	5.5	45.50%	48.90%	3.1	-	3.60%	99.90%	111	1.37	1.0
pmd	5.2	8.1	7.9	19.30%	55.80%	5.1	0.1	50.50%	99.80%	389	1.00	144.4
series	1.3	1.2	1.0	66.70%	46.60%	1.2	-	25.80%	100%	117	1.00	13,514.0
sor	3.5	6.2	6.4	1.50%	59.60%	4.4	0.9	99.80%	100%	233	1.22	112.7
sparsemat	9.7	151.2	17.2	98.80%	100%	1.5	3.4	77.40%	100%	516	1.79	22.8
sunflow	6.4	70.0	10.7	92.80%	99.70%	3	1.4	80.20%	100%	125	1.09	895.0
tomcat	3.3	4.1	3.4	25.40%	58.30%	2.7	0.1	38.20%	95.40%	451	1.08	8.1
xalan	1.4	1.9	2.2	15.50%	38.90%	7.2	-	29.70%	82.90%	628	1.00	0.6

Table 3.1: SlimFast characterization data.

benefits, due to the relatively low redundancy ratios (Figure 3.2) and low percentages of reuse in *EpochPlusVC*s objects (Table 3.1) in these benchmarks. For tomcat, because its worker threads are highly independent of each other with little sharing, SlimFast's reduction of synchronization contention and working set size does not translate into speedup.

3.4.4. Additional Characterization

To better understand SlimFast's performance, we performed a series of characterization experiments; Table 3.1 shows the results. Column 2 shows the slowdown of RoadRunner, FastTrack, and SlimFast, normalized to native JVM execution. The slowdown of RoadRunner ranges from 1.3x to 30.1x, which accounts for a large part of the slowdown of FastTrack and SlimFast. A more efficient framework for dynamic analysis would likely reduce this overhead substantially. Columns 3 & 4 show the percentage of accesses to *EpochPlusVC*s, among all accesses, and the percentage of *EpochPlusVC*s allocations, among all metadata allocations. These two indicators are highly correlated with the speedup of SlimFast over FastTrack: for instance, fop and jython have no *EpochPlusVC*s (they're mostly single-threaded), and the speedup of SlimFast on them is small. In contrast, most of the metadata objects accessed in sunflow and sparsemat are *EpochPlusVC*s where SlimFast shows significant speedups.

Columns 5 & 6 show the average occupancy of the $\mathbb{S}_t$ and $\mathbb{Q}_t$ sets: for all the benchmarks, these sets have $\leqslant 11$ elements on average. This motivates our implementation of $\mathbb{S}_t$ and $\mathbb{Q}_t$ with fixed-size arrays. Columns 7 & 8 give the percentages of reuse of *EpochPlusVC*s and *EpochPair*s among the total lookups to $\mathbb{S}_t$ and $\mathbb{Q}_t$, respectively. Higher percentages mean that fewer *EpochPair*s and *EpochPlusVC*s need to be allocated, saving space. All 25 benchmarks have high percentages of *EpochPair* reuse (the lowest, luindex, is 82.8%), showing that *EpochPair*s are highly redundant. In contrast, the percentage of *EpochPlusVC* reuse is low on some benchmarks, showing that the redundancy among vector clocks is less significant.

Column 9 in Table 3.1 shows the maximum heap usage with SlimFast, in megabytes. Note that SlimFast both saves space and reduces runtime more effectively on programs with larger heap sizes. Column 10 in Table 3.1 shows the memory overhead of SlimFast with respect to RoadRunner: most programs incur less than 51% overhead, except NPB_LU (124.03%) and sparsemat (79.25%). The relatively small spatial overhead of SlimFast may make precise dynamic data race detection more feasible in memory-constrained systems, such as mobile devices.

The final column in Table 3.1 shows ratio of the cumulative numbers of metadata allocations in FastTrack and SlimFast. On 18 out of the 25 programs, SlimFast allocates fewer metadata objects than FastTrack. For 6 programs, SlimFast has more metadata allocations than FastTrack, which is largely due to their relatively low redundancy among the metadata (as shown in Figure 3.2). In fact, Table 3.1 and Figure 3.2 together show a correlation between SlimFast's reduction of metadata allocations and the redundancy ratio in FastTrack.

To evaluate how effective SlimFast is in reducing metadata redundancy, we measured the redundancy ratio on *EpochPlusVC*s in SlimFast, and the results are shown in Figure 3.14. On most benchmarks, essentially *no* redundancy exists on *EpochPlusVC*s, which means SlimFast does in fact gain near-optimal space reduction on these programs. On lufact and montecarlo,

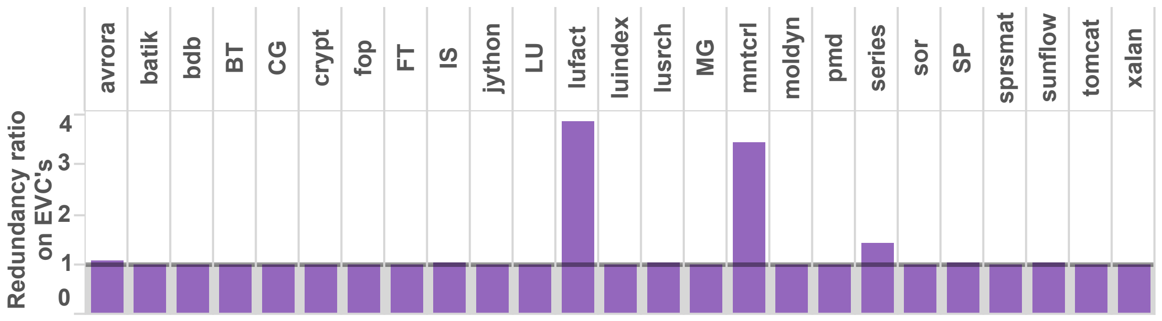


Figure 3.14: Redundancy ratio of EpochPlusVC's in SlimFast. Lower is better.

about $\frac{2}{3}$ of *EpochPlusVC*s are redundant. This is a relatively low ratio, and *EpochPlusVC*s represent only about 60% of the allocated metadata for these workloads, further shrinking the opportunity for any meaningful additional space reduction. Set against the O(n) cost of doing optimal redundancy elimination, SLIMFAST strikes a good balance of finding most of the redundancy in just O(1) time.

3.5. Conclusion

Existing dynamic race detectors store many redundant copies of metadata. This chapter discusses such redundancy, then describes the SLIMFAST race detector that eliminates the vast majority of this redundancy via shared references to a single, immutable copy of each metadata value, leveraging novel invariants of dynamic race detection to reduce metadata redundancy, memory usage and runtime. Evaluation results show that SLIMFAST is able to both reduce memory consumption and improve runtime performance of existing dynamic race detectors.

CHAPTER 4

Performant Architecture for Race Safety with No Impact on Precision

As presented in Chapter 3, the SlimFast system improves the efficiency of precise dynamic data race detection. However, being a software-only solution, the overheads of SlimFast can still be too high, limiting its application.

One approach to boost the performance of dynamic race detection is to exploit hardware support. This chapter first describes the metadata redundancy visible at the hardware level, then presents the Parsnip system, a hardware-assisted data race detector that builds upon the idea of metadata de-duplication in hardware. An evaluation of the Parsnip design is provided at the end of this chapter.

4.1. Metadata Redundancy at the Hardware Level

The work of a precise dynamic data race detector comprises two components. The detector must: update information about cross-thread ordering on each synchronization event; and check and update per-location access history on each memory access event. Memory accesses typically occur much more frequently than synchronization, so optimizing the analysis, update, and representation of per-location access histories is crucial to data race detector performance.

As described in Chapter 3, metadata (more precisely, the per-location access histories) used in precise dynamic race detectors can be highly redundant across variables. How does the metadata redundancy manifest at the hardware level? To answer this question, a characterization experiment is conducted. We wrote a cache simulator and drove the simulator on the PARSEC 3.0[9] workloads via the Intel PIN[66] binary instrumentation framework. The simulator counted how many unique access histories appear within each cache line. Figure 4.1 shows the distribution of the number of distinct access histories per 64B cache line.

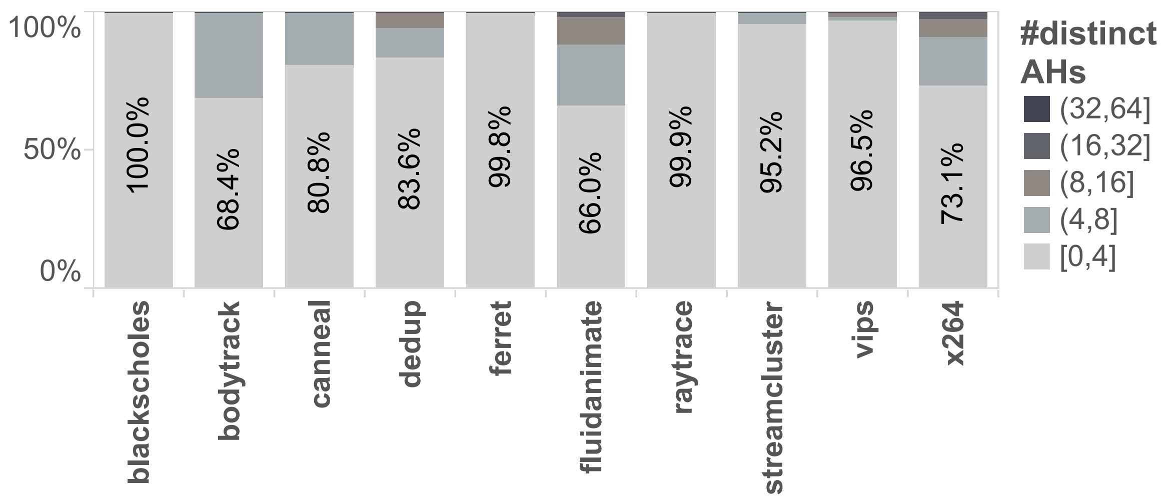


Figure 4.1: Distribution of the number of distinct access histories per 64B cache line.

While the 64B cache lines modeled by the simulator can theoretically require a distinct access history for each byte, on every program (except fluidanimate and x264) the number of distinct access histories per line is never more than 16. Even for fluidanimate and x264, fewer than 2% of lines have more than 16 distinct access histories. This result is unsurprising given that programs typically access data at multi-byte granularity, and exhibit spatial locality. These properties make it highly likely that nearby locations will have the same access history.

4.2. The Parsnip System

The guiding principle of the PARSNIP system is to handle the common cases of memory access checks and access history metadata storage in hardware and fall back on a flexible software layer for uncommon cases that require additional access history to maintain soundness and completeness. PARSNIP leverages the following observations about the empirical behavior of dynamic race detection. First, access history metadata for adjacent data locations are likely to be similar or identical, so PARSNIP tries to reduce access history metadata redundancy across memory locations at various levels. Second, most runtime checks need information about only the most recent access, so PARSNIP organizes access history metadata to keep last-access information in a hardware-managed format on chip or in cache, even when full access history for a memory location may be available only in a software-managed format in memory.

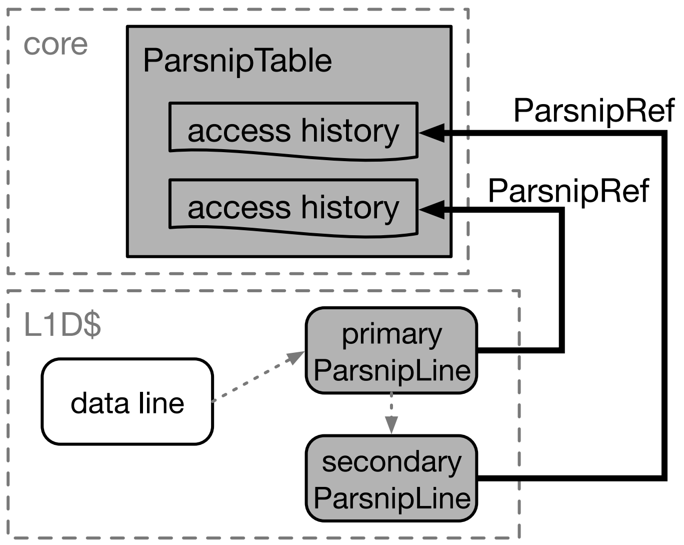


Figure 4.2: An overview of Parsnip's key pieces of state: ParsnipLines, ParsnipRefs and the per-core ParsnipTable.

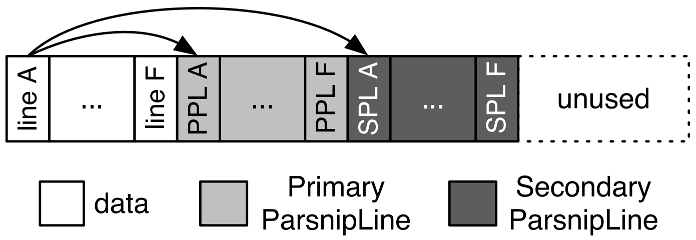


Figure 4.3: Parsnip's physical address space layout.

4.2.1. Core Design

The first task a race detector faces when needing to check an access to location x is to find the associated metadata for x . Figure 4.2 gives an overview of PARSNIP's key states. PARSNIP manages metadata at cache line granularity, so for each cache line of program data, there is a corresponding ParsnipLine given by this mapping. These ParsnipLines reside in the data cache hierarchy and compete for space with regular program data (as shown in Figure 4.2; Primary versus Secondary ParsnipLines are explained later). PARSNIP steals 2 bits from the physical address space to afford a simple data:metadata mapping based on physical addresses (Figure 4.3), constructed to map x and its metadata to different cache sets to reduce conflict misses.

Access History Type	Read Check requires:	Write Check requires:
last access	last access	last access
RAW	last read + last write, or last access	last access
RsAW	last read + last write, or last access	all reads
Rs	last access	all reads

Table 4.1: Components required for different types of access histories and memory access checks.

While ParsnipLines can record an access history directly, doing so is not space efficient. According to the characterization data shown in Figure 4.1, access histories for bytes within the same cache line are highly redundant. To take advantage of the redundancy among access histories, a ParsnipLine contains a collection of references (called ParsnipRefs) which point to entries in a per-core ParsnipTable. The ParsnipTable entries themselves contain the access history information. By adding a level of indirection, PARSNIP is able to share ParsnipTable entries across a large number of program locations. Because only a small number of ParsnipTable entries are typically needed, each ParsnipRef can itself be small, helping to reduce PARSNIP's footprint in the data cache.

However, several challenges remain. Access histories are of variable size, and at their largest can require tracking a clock value for each thread in the program–much larger than can fit into a fixed-size hardware table. The ParsnipLine encoding must similarly be able to exploit the common case while supporting precise tracking even when there are 64 unique histories in a single line.

4.2.2. Access History Organization

To encode access histories of memory locations efficiently, PARSNIP exploits the observation that most data race checks need only a small part of the full access history.

Access histories of memory locations can be classified into the following types: 1) *last access*, which contains only the most recent write/read; 2) *RAW*, which consists of the last write and the last read that happens-after the write; 3) *RsAW*, which keeps the last write and all concurrent reads that happen-after the write; and 4) *Rs*, which tracks all concurrent reads with no prior write.[1]

Table 4.1 shows what piece(s) of the access history a runtime read or write check needs. In most cases a check can be done with only the last access, or both the last read and the last write; only when a write occurs after some concurrent reads (where the access history is in RsAW/Rs form) does the check require all concurrent reads since the last write. In fact, such cases are rare in practice, which implies that an efficient race detector can complete most checks with a history of at most two accesses, without losing soundness.

Based on this observation, PARSNIP organizes the access history in a way such that the last read and last write information is in hardware in most cases, with the rest of the access history stored in software. As discussed in Section 4.2.1, each data location x has a corresponding Primary ParsnipRef that points to its access history in the ParsnipTable. To maintain a fixed size, each ParsnipTable entry contains a single clock value (and a reference count, explained later in Section 4.2.5). Thus, tracking the last read *and* last write for x requires two Parsnip-Refs: the **Primary** ParsnipRef and the **Secondary** ParsnipRef. These primary and secondary references are stored in Primary and Secondary ParsnipLines, accordingly, and their locations are computed from the corresponding data address (Figure 4.3).

The Primary ParsnipRef refers to the last access, which is a read for the RAW and RsAW access history types. A Primary ParsnipRef contains several fields (detailed in Section 4.2.4), including a *hasNext* bit indicating whether x has a Secondary ParsnipRef. The Secondary ParsnipRef contains the last write when the access history is of RAW or RsAW type. The Secondary ParsnipRef has the same structure as the Primary ParsnipRef, but its *hasNext* bit indicates whether additional access history information, *i.e.*, additional concurrent readers, is stored in software.

[1] Well-defined C/C++ programs should precede any read to a memory location by an initializing write, but some programs (including our benchmarks) read uninitialized data in practice. Another potential source of reads before writes is the use of facilities such as demand-zeroed pages that initialize memory contents with other mechanisms.

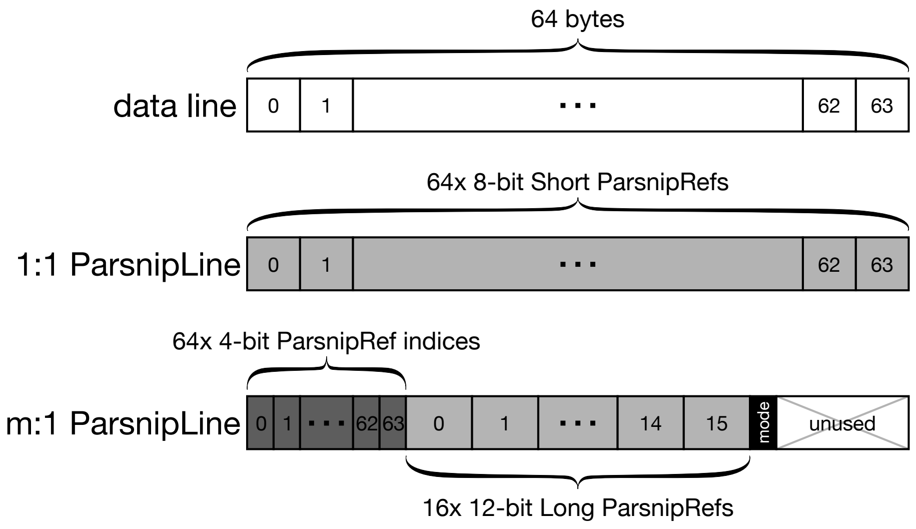

64 bytes
data line
0
1
62
63
64x 8-bit Short ParsnipRefs
1:1 ParsnipLine
0
1
62
63
64x 4-bit ParsnipRef indices
m:1 ParsnipLine
0
1
62
63
0
1
14
15
mode
unused
16x 12-bit Long ParsnipRefs

Figure 4.4: A ParsnipLine in 1:1 mode (middle) and m:1 mode (bottom)

The access history of a data location x is thus stored across up to 3 parts: Primary ParsnipRef, Secondary ParsnipRef, and remaining information in software. This organization helps minimize cache pollution, as most race checks can be discharged with the Primary ParsnipRef alone.

4.2.3. ParsnipLine Format

The results shown in Figure 4.1 indicate an opportunity for PARSNIP to have ParsnipRefs that are larger than a byte while using a single ParsnipLine to track all the ParsnipRefs for a cache line of data. Specifically, ParsnipLines in PARSNIP have two modes: 1:1 mode and m:1 mode. Figure 4.4 shows the two ParsnipLine formats. In 1:1 mode, a ParsnipLine is filled with 8-bit ParsnipRefs, each associated in a 1:1 fashion with the bytes in the corresponding data line. In m:1 mode, each ParsnipRef is 12 bits in size, and we introduce another layer of indirection to map data line bytes through a 64-entry bitmap to one of the 16 ParsnipRefs. For the i^{th} byte of a data line, its ParsnipRef can be found by first looking up the bitmap to get an index $bitmap[i]$, then reading the $bitmap[i]^{th}$ ParsnipRef. The bitmap occupies 32B, and the ParsnipRefs 24B, with 8B left over. The byte following the ParsnipRefs is used to indicate whether a ParsnipLine is in m:1 or 1:1 mode by reserving a special bit pattern. (detailed in Section 4.2.4). The other 7B are unused.

Having smaller ParsnipRefs in 1:1 mode means that for each byte less information of the access history can be encoded than the m:1 mode, but in practice even these 8-bit ParsnipRefs are sufficient for many access checks(see Figure 4.10 in Section 4.4). The number of distinct ParsnipRefs per ParsnipLine is computed by dedicated hardware (see Figure 4.6). Whenever a ParsnipRef in m:1 mode needs to switch to 1:1 mode or vice versa, the ParsnipLine logic triggers a rewrite of the ParsnipLine. In practice, 1:1 mode ParsnipLines are rarely needed and mode switches are rare.

4.2.4. ParsnipRef Format

As described in Section 4.2.3, PARSNIP can have two modes of ParsnipLines. In m:1 Parsnip-Lines, ParsnipRefs occupy 12 bits, whereas in 1:1 ParsnipLines, ParsnipRefs occupy 8 bits. For brevity, we refer to the 12-bit ParsnipRefs as **Long** ParsnipRefs, and the 8-bit ParsnipRefs as **Short** ParsnipRefs. Figure 4.5 shows the format of both Long and Short ParsnipRefs. Each ParsnipRef starts with a 6-bit *tid* that stores the thread that performed the access, followed by a R/W bit indicating whether the ParsnipRef points to a read or write access, and a *hasNext* bit (introduced in Section 4.2.2) indicating whether additional access history information is available in a Secondary ParsnipRef or in software.

For brevity, we define the pair $(tid, r/w)$ to be an **access capability**; therefore, a Short Parsnip-Ref encodes an access capability. In a Long ParsnipRef, another 4 bits are used to hold an index into the 16-entry ParsnipTable, whose entries contain clock values. Together, the *tid* field and ParsnipTable entry constitute an epoch which can be used to perform race detection checks.

As mentioned in Section 4.2.3, PARSNIP reserves a special bit pattern to distinguish m:1 mode ParsnipLines from 1:1 lines. Specifically, the byte value 0xFF is reserved to mean m:1 mode, whereas all other values of the *mode* byte are interpreted as a Short ParsnipRef, indicating that the current ParsnipLine is in 1:1 mode. One implication is that any Short ParsnipLine must not have the *tid* field set to be 0b111111 to avoid collision with the special mode byte pattern.

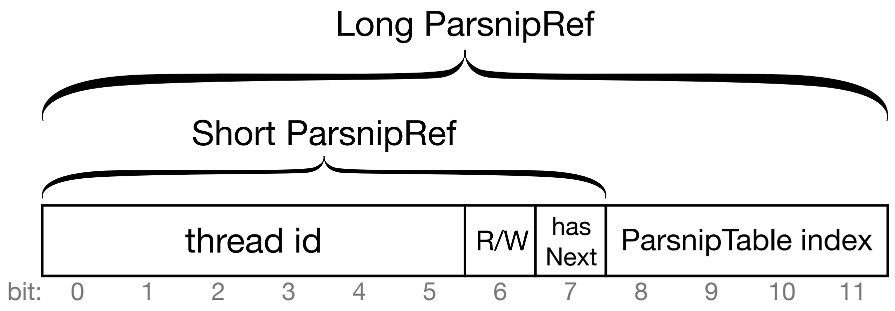


Figure 4.5: Format of Short and Long ParsnipRefs.

4.2.5. ParsnipTables

ParsnipTables are small per-core tables that hold logical times. Each ParsnipTable entry consists of a 64-bit clock and a 24-bit reference count to record ParsnipRefs that refer to this entry. PARSNIP recycles an entry once its reference count reaches 0. The value i of the 4-bit *index* field in a Long ParsnipRef points to the i^{th} entry in the ParsnipTable of thread t, where t is the value of the ParsnipRef's tid field. PARSNIP reserves the value 0xF to indicate an "invalid" ParsnipTable index, which is used to represent clock values that are not present in the ParsnipTable (and must be retrieved from software instead). Given the 4-bit indices in a Long ParsnipRef, each ParsnipTable can hold up to 15 distinct clock values. A 15-entry ParsnipTable occupies just 165B of space.

Although quite small in size, ParsnipTables allow a large proportion of the runtime checks to be completed in hardware. Whenever an epoch $c@t$ of an access to a memory location x needs to be recorded, PARSNIP first traverses the ParsnipTable of thread t looking for an entry that already holds the clock value c. If such an entry is found, the reference count of the entry is incremented and the index of the entry is written in the corresponding ParsnipRef for x. Otherwise, a new ParsnipTable entry needs to be allocated to hold c. If there is free space in the ParsnipTable, then the index of the newly allocated entry is written in the ParsnipRef of x. Otherwise, the ParsnipTable is full and a ParsnipTable *rejection* occurs: the invalid index 0xF is written in the ParsnipRef, and the epoch value is persisted into software.

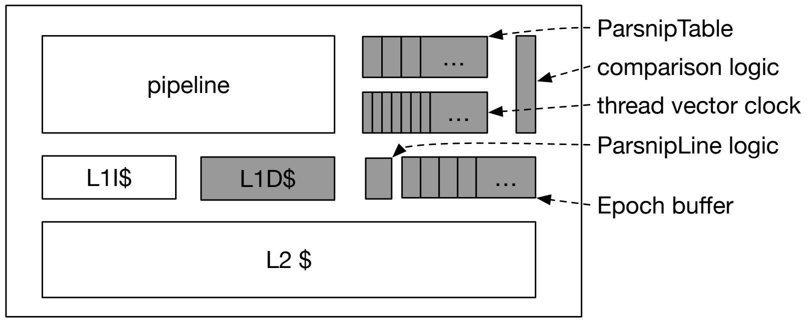


Figure 4.6: Parsnip additions (shaded) to a conventional core.

The reference count in a ParsnipTable entry tracks the number of ParsnipRefs in cache that point to that entry. Therefore, if a ParsnipRef leaves the data cache, the corresponding reference count is decremented. To implement this, an eviction handler is triggered when a ParsnipLine is evicted from the last-level data cache; inside this eviction handler, all reference counters corresponding to the indices of the ParsnipRefs being evicted are decremented.

Parsnip adopts the above ParsnipTable rejection and reference count policy as a simpler alternative to conventional preemptive eviction of least recently used ParsnipTable entries. LRU eviction of table entries is impractical because it requires changes to all ParsnipRefs referencing the victim entry, an unbounded set. Section 4.3.3 discusses refinements to the ParsnipTable management policy to improve utilization.

4.2.6. Parsnip Hardware Support

Figure 4.6 shows an overview of a Parsnip processor core, with shaded blocks showing the components Parsnip adds to a conventional design. As per-thread vector clocks are necessary in most runtime checks, Parsnip stores them on-chip. Parsnip also adds simple logic to compare a component of the per-thread vector clock with an epoch from a location's access history, and a small ParsnipTable to remove redundancy among the access histories of different memory locations (Section 4.2.5). Parsnip also adds custom logic to track the number of distinct last accesses in a ParsnipLine, to determine whether the line can be encoded in m:1 or 1:1 format (Section 4.2.3). Due to the limited capacity of ParsnipTables, some access histories may need to be persisted into software. To reduce the latency incurred for these updates, PARSNIP adds a hardware structure to buffer these updates (see Section 4.3.1). PARSNIP (like Radish) adds an extra read port to the L1 data cache to read data and metadata in parallel.

4.2.7. Parsnip Access Checks

When a program accesses a location x, PARSNIP first looks up x's Primary ParsnipRef in the data cache; if the Primary ParsnipLine is not in cache, a software handler will be invoked to complete the current check, after which the ParsnipLine of x will be updated. If the Primary ParsnipLine is in cache, the ParsnipLine logic decodes it and gets the Primary ParsnipRef of x. Depending on the type of the current access and the content of the Primary ParsnipRef, PARSNIP can decide whether all necessary information for the current check has been collected. If so, the check is complete, after which the Primary ParsnipRef of x is updated. Otherwise, PARSNIP continues to look up the Secondary ParsnipRef of x. If the Secondary ParsnipLine is not in cache, or the Secondary ParsnipRef still does not provide all necessary information required by the current check, a software handler will be invoked to handle the check and update the ParsnipRefs.

4.2.8. System-Level Considerations and ISA

The address of a ParsnipLine is derived from the physical address of the corresponding data. Because a ParsnipLine is the same size as a data cache line, cache indexing can occur in parallel with data address translation as with a conventional cache. Once the virtual tag of the data address has been translated to a physical tag by the TLB, the corresponding tag for the associated ParsnipLine can be computed directly. PARSNIP does not require a separate TLB access to obtain the tag for the ParsnipLine address.

Because ParsnipLines are physically-addressed, they do not need to be saved and restored on a context switch. Conventional address space isolation between processes keeps ParsnipLines isolated as well. Context switches must, however, save and restore ParsnipTable contents and per-core vector clocks.

PARSNIP's ISA support consists primarily of checked load and store instructions that trigger a race check when executed, similar to LARD [116]. This allows application code to interleave at fine granularity with code for a runtime system or trusted library, and allows users to easily opt-in to PARSNIP support. PARSNIP additionally requires the following work to be done in software: 1) handling ParsnipLine evictions from cache, 2) doing race checks in software when the hardware has insufficient information, and 3) updating the per-thread vector clocks on synchronization operations. User-level interrupts are used to quickly transfer control to software when these events occur.

An eviction handler is invoked when a ParsnipLine l leaves the cache hierarchy. The software handler must determine l's owning process (which may be de-scheduled). This is done with OS support by looking up l's corresponding data physical address in the OS "reverse frame map" which maps from physical pages to virtual pages. For every Long ParsnipRef containing a valid index *idx* into thread t's ParsnipTable, the handler gets the clock value c from thread t's ParsnipTable, decrements the entry's reference counter, and saves the epoch $c@t$ into a software representation. For Short ParsnipRefs, or Long ParsnipRefs with an invalid ParsnipTable index, the access history in software is already up-to-date and no further action needs to be taken.

A software check handler is called when the hardware has insufficient information for a runtime check (*e.g.*, the Primary ParsnipRef is not cached, or a write needs to check against the complete set of concurrent readers). The software check handler is invoked with the partial access histories from hardware (if available from the ParsnipTable), which, combined with the information stored in software, forms the complete access history of the memory location being checked. After finishing the current check, this handler also updates the Primary ParsnipRef and, if necessary, the Secondary ParsnipRef, in accordance with the updated access history.

In order for PARSNIP to track synchronization events across threads, the synchronization library must be modified such that PARSNIP's per-thread vector clocks are updated properly during each synchronization operation.

Execution Trace				Parsnip Access History for x							Synchronization History			
	Threads			ParsnipRefs		ParsnipTables (partial)			SW Metadata		C: Thread VCs			L: Lock VCs
Step	t_1	t_2	t_3	Primary	Secondary	t_1	t_2	t_3	W_x	R_x	t_1	t_2	t_3	l_1
0								$4 \mapsto (16, 1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
1	wr x			t_1, W, F, $[0]$		$0 \mapsto (1, 1)$		$4 \mapsto (16, 1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
2	rd x			t_1, W, F, $[0]$		$0 \mapsto (1, 1)$		$4 \mapsto (16, 1)$			$\{1@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	
3	rel l_1			t_1, W, F, $[0]$		$0 \mapsto (1, 1)$		$4 \mapsto (16, 1)$			$\{2@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
4	rd x			t_1, R, T, $[1]$	t_1, W, F, $[0]$	$0 \mapsto (1, 1)$ $1 \mapsto (2, 1)$		$4 \mapsto (16, 1)$			$\{2@t_1\}$	$\{1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
5		acq l_1		t_1, R, T, $[1]$	t_1, W, F, $[0]$	$0 \mapsto (1, 1)$ $1 \mapsto (2, 1)$		$4 \mapsto (16, 1)$			$\{2@t_1\}$	$\{1@t_1, 1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
6		rd x		t_2, R, T, $[1]$	t_1, W, T, $[0]$	$0 \mapsto (1, 1)$	$1 \mapsto (1, 1)$	$4 \mapsto (16, 1)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 1@t_2\}$	$\{16@t_3\}$	$\{1@t_1\}$
7		rel l_1		t_2, R, T, $[1]$	t_1, W, T, $[0]$	$0 \mapsto (1, 1)$	$1 \mapsto (1, 1)$			$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{16@t_3\}$	$\{1@t_1, 1@t_2\}$
8			acq l_1	t_2, R, T, $[1]$	t_1, W, T, $[0]$	$0 \mapsto (1, 1)$	$1 \mapsto (1, 1)$	$4 \mapsto (16, 1)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 16@t_3\}$	$\{1@t_1, 1@t_2\}$
9			! wr x	t_3, W, F, $[4]$	—	—	—	$4 \mapsto (16, 2)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 16@t_3\}$	$\{1@t_1, 1@t_2\}$
10			rel l_1	t_3, W, F, $[4]$	—	—	—	$4 \mapsto (16, 2)$		$\{2@t_1\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 17@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
11			rd x	t_3, R, T, $\perp$	t_3, W, F, $[4]$	—	—	$4 \mapsto (16, 2)$		$\{17@t_3\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 17@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
12			rel l_2	t_3, R, T, $\perp$	t_3, W, F, $[4]$	—	—	$4 \mapsto (16, 2)$		$\{17@t_3\}$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
13			wr x	t_3, W, F, $\perp$	—	—	—	$4 \mapsto (16, 1)$	$18@t_3$	—	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$
14		! rd x		$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\{2@t_1\}$	$\{1@t_1, 2@t_2\}$	$\{1@t_1, 1@t_2, 18@t_3\}$	$\{1@t_1, 1@t_2, 16@t_3\}$

Table 4.2: An example trace showing how Parsnip checks accesses to a memory location x.

4.2.9. Parsnip Example Trace

Table 4.2 gives an example trace illustrating how PARSNIP works for a memory location x. The first column indexes the trace. The next three columns show operations by three threads. The symbol "!" marks accesses on which a data race is reported. Remaining columns show status of analysis metadata *after* the corresponding operation is completed. Next is the PARSNIP access history, including ParsnipRefs, a partial ParsnipTable, and the access history stored by the software layer. The final columns show vector clocks tracking synchronization order. Vector clocks are notated as sets of epochs. We conflate cores with threads. Additional notation is introduced below.

At step 1, t_1 writes x. No primary ParsnipRef exists for x, so PARSNIP invokes a software handler. The software handler finds that there are no prior accesses to x in software access history, so the check succeeds. Free entry 0 from t_1's ParsnipTable is allocated to store the clock at which this access occurs (1, as obtained from the thread VC for t_1). ParsnipTable entries take the form *index* $\mapsto$ (*clock*, *ref count*). We show only some entries for each Parsnip-Table in Table 4.2. A ParsnipRef t_1, W, F, $[0]$ is initialized for x, showing that the last access was a write (W) by thread t_1. There is no secondary ParsnipRef (F). The ParsnipRef contains an index [0] into t_1's ParsnipTable, where the local clock of the access is recorded.

Lock operations in steps 3 and 5 induce happens-before ordering. Since the lock release increments t_1's epoch to $2@t_1$, the read at step 4 is in an epoch different from the epoch encoded in

x's primary ParsnipRef, so it needs to be recorded as the last access to x. Since this access is a read, information about the prior write must be retained. The ParsnipRef for the past access becomes the secondary ParsnipRef and a fresh primary ParsnipRef is recorded, with a freshly allocated entry [1] in the ParsnipTable, where the current clock (2) is stored.

The next read of x at step 6 is done by thread t_2, concurrently with the last read encoded in the primary ParsnipRef. To check for a potential write-read race, PARSNIP also checks the secondary ParsnipRef. Since there is space for information about only 2 accesses in hardware, the older read by t_1 is saved to software and the read by t_2 is recorded as a fresh primary ParsnipRef, using a newly allocated entry [0] in t_2's ParsnipTable. The target ParsnipTable is determined by the thread in the ParsnipRef.

Next, thread t_2 releases lock l_1 (step 7) and t_3 acquires it (step 8), establishing happens-before order. At step 9, the write by thread t_3 checks against both ParsnipRefs, but the secondary ParsnipRef indicates with its *hasNext* bit (T) that there is additional history in software. A software check is invoked and finds a data race with the read at $2@t_1$ (step 4). For illustration purposes, we assume PARSNIP continues past this data race report and records this access as usual. Since it is a write, all access history about x is obsolete, and it returns to a single primary ParsnipRef recording the write in epoch $16@t_3$, using entry [4] in t_3's ParsnipTable, which currently holds clock 16. Note that the software history retains information about an older read. PARSNIP could preemptively erase this, but there is no need since the new primary ParsnipRef shows with its *hasNext* bit (T) that there is neither a secondary ParsnipRef nor history in software, so future accesses will never inspect that history (even though doing so would not affect precision).

Thread t_3 then releases lock l_1 at step 10, incrementing its logical clock to 17. Thread t_3 reads x at step 11 in the same thread as the last access indicated by the primary ParsnipRef, but in a new epoch. However, the ParsnipTable of t_3 is still full (not shown) and a 17 entry is not available, so the clock value 17 cannot be stored. PARSNIP records a fresh primary ParsnipRef for x, assigned to t_3, but with the invalid index (notated $\perp$). PARSNIP also saves the full epoch value $17@t_3$ to software. Since the existing primary ParsnipRef indicated that there was neither a secondary ParsnipRef nor history in software, the software overwrites outdated history.

Lock l_2 is released by thread t_3 at step 12 (Table 4.2 does not show metadata for l_2 in L), followed by a write to x in t_3 at step 13. The check on this write completes after decoding the primary ParsnipRef alone. Even though the primary ParsnipRef indicates there is a secondary ParsnipRef, the current write is in the same thread as the last read encoded by the primary ParsnipRef, so the current write is race-free. Since t_3's ParsnipTable is still full (not shown), the full epoch value $18@t_3$ is persisted into software. As this is a write, all software read history is now stale and is erased. Finally, when t_2 reads x at step 14, the primary ParsnipRef lacks a table index, so PARSNIP performs a software check to check the last write epoch, finding a write-read race.

4.3. Optimizations in Parsnip

This section describes three key optimizations to reduce dependence on the high latency PARSNIP software layer in cases where on-chip resources are insufficient to recover or record necessary access history. These optimizations avoid unnecessary software interactions by buffering the transfer of evicted access history metadata to software (Section 4.3.1); prefetching access history information from software (Section 4.3.2); and varying the policy for eviction of ParsnipLines from cache (Section 4.3.3).

4.3.1. Buffering Evictions of Access History to Software

PARSNIP must persist access history information to software to preserve soundness in two cases that are neither common nor rare enough to ignore the high latency of the software layer: (1) on ParsnipTable rejections (Section 4.2.5) and (2) when recording a new read epoch in a RsAW-mode access history, where PARSNIP stores only the last read and last write in hardware and maintains information about other reads in software (Section 4.2.2).

To reduce the latency of persisting epochs to software, we extend the basic PARSNIP design to add a 32-entry coalescing *epoch store buffer* to each core (Figure 4.6). Each entry holds the address of the accessed data location, the memory access type (read/write), and the epoch in which the access occurred. PARSNIP buffers an entry when it needs to persist access history to software. The latency of inserting an entry into the buffer can be hidden by the cost of the corresponding ParsnipLine update. When the buffer fills, a software handler is invoked to drain all buffered entries to software.

When insufficient access history is available on-chip in ParsnipLines or ParsnipTables, PARSNIP queries the software layer only if it cannot find the required access history by snooping the epoch store buffer. The buffer also coalesces entries for the same address and access type. For example, if the buffer contains an entry for a read to data location x in epoch e_1 and a read to x in newer epoch e_2 must be persisted to software, the entries are coalesced, preserving only the most recent read to x in e_2. Upon last-level cache misses on a ParsnipLine, the software handler also snoops the epoch store buffers to ensure that any pending updates to x's access history is visible to later accesses.

4.3.2. Access History Prefetching and Prediction

When an access check by thread t for data location x finds insufficient in-hardware access history, PARSNIP invokes a software handler to resolve the check for location x from the full software-managed access history. To exploit spatial locality, PARSNIP additionally fills the ParsnipLine for x, setting access capabilities for all other locations that are tracked by the same ParsnipLine.

Rather than simply filling ParsnipRefs for other locations $y \neq x$ in the same ParsnipLine as x's ParsnipRef based on access history, the software handler *speculatively checks a future access* to y of the same type (read/write) and by the same thread t as the access that triggered the software check of y's neighbor, x. If the speculative check determines that such an access to y would be data race free according to the current access history, it fills a ParsnipRef for y with the access capability component set to describe this predicted future access, but leaves the ParsnipTable index component of the ParsnipRef *invalid*.

This optimization preserves soundness and completeness if the next access to y is predicted correctly, as the predicted future access check will resolve based on the ParsnipRef access capability for y and record the up-to-date epoch of this new access, achieving the same result as if it had dispatched the check to software.

If a speculative check mispredicts the next access to y and a different thread $u \neq t$ makes the next access to y, soundness and completeness are still preserved. An access to y by thread u will find the ParsnipRef access capability assigned to t with an invalid ParsnipTable index, thus triggering a search through the epoch store buffer (Section 4.3.1) or the software layer to find the most up-to-date access history for y. Since speculative checks update only the access capability of a ParsnipRef, no record of the predicted access is found in either the epoch store buffers or software. PARSNIP will thus complete a full access check against the same access history it would have checked without speculation. In sum, PARSNIP remains sound and complete with this optimization.

4.3.3. ParsnipTable and ParsnipLine Management

To ensure effective use of limited table space, ParsnipTable management must exploit locality and favor entries that are likely to be reused soon. However, the simple policy of reference counts and rejections established in Section 4.2.5 has limitations. ParsnipRefs for old, infrequently-used data that remain in the last-level cache may pin ParsnipTable entries with nonzero reference counts, leaving no table space for newer, frequently-used epochs, creating high rates of Parsnip-Table rejections and expensive software checks.

A simple alternative is to invalidate and persist a ParsnipLine to software as it leaves the L1 cache. ParsnipRefs for memory locations not accessed recently thus tend to be evicted from L1, decrementing reference counts and freeing more ParsnipTable entries to represent new epochs. However, even hot data locations in programs with large working sets experience frequent evictions and refills to the L1 or even L2. The associated ParsnipLines then suffer frequent round trips to and from to the software layer.

To balance ParsnipTable utilization and software costs, PARSNIP adaptively selects an L1 or LLC residence policy for ParsnipLines by tracking the rate of software checks in a window of recent events. Under policies that allow ParsnipLines to reside in the LLC, the heuristic responds to software checks due to excessive ParsnipTable rejections when old ParsnipRefs in the last-level cache pin too many useless ParsnipTable entries. Under L1-only ParsnipLine residence, the heuristic responds to software checks for repeatedly invalidated and refilled ParsnipLines in L1. A user may also mandate a cache residence policy based on profiling or experience. Extending Parsnip to support other heuristics is a promising avenue for future work.

4.4. Design Evaluation

To evaluate Parsnip's performance, we implemented a simulator using Intel PIN [66] to model a 16-core system with MESI coherence protocol, with a realistic memory hierarchy common in commodity processors. Cache lines are 64 bytes. Each core has an 8-way 32KB L1 cache and an 8-way 256KB L2 cache; all cores share a 16-way 32MB L3 cache. Latency of L1, local L2, remote L2, L3, and main memory accesses are 1, 10, 15, 35, and 120 cycles, respectively. The cache subsystem of the Parsnip simulator is implemented by extending the cache implementation in ZSim [99].

Each per-core ParsnipTable has 15 entries (165 bytes total), with each entry consisting of a 64-bit clock and a 24-bit reference count. Each per-core epoch store buffer has 32 entries (544 bytes total), with each entry holding a 64-bit address, a 64-bit clock, and an extra byte encoding both a tid and a r/w bit. Lookups in the local core's ParsnipTable cost 1 cycle; requesting an entry from a remote core's ParsnipTable takes 10 cycles. Latency of epoch store buffer insertions is hidden by the cycles of updating the corresponding ParsnipLine. Accesses to locations on the stack are assumed to be thread-local, and no data race checks are done for stack locations. Software checks and epoch-buffer drains cost 500 cycles plus the cost of any cache accesses triggered.

4.4.1. Experimental Setup

We evaluate Parsnip with 10 programs from the PARSEC 3.0 benchmark suite [9], including blackscholes, bodytrack, canneal, dedup, ferret, fluidanimate, raytrace, streamcluser, vips and x264. We omit the benchmarks facesim, which forks child processes as parallel workers, and swaptions, which has high variance across runs in our experiments. We report performance as the mean of 3 runs.

We ran four sets of experiments. (1) We ran performance experiments on the Parsnip simulator with 16 threads on 16 simulated cores and a corresponding baseline system without Parsnip extensions. (2) To compare the performance of Parsnip to that of the most closely related system, Radish [25], we ran performance experiments for Radish and a corresponding baseline system using the simulator made publicly available by the authors. (3) To evaluate Parsnip's scalability across cores, we ran performance experiments for Parsnip and the baseline system with 2, 4, 8, and 16 threads. (4) We ran additional performance and profiling experiments with 16 threads on 16 simulated cores to characterize the impact of individual features in the Parsnip system.

4.4.2. Performance

This section compares the performance of Parsnip with Radish [25], the most related work. Figure 4.7 shows the CPIs of Parsnip and Radish, normalized to the native executions of the respective simulator without data race detection extensions. Results shown here were collected when running with 8 threads and the simsmall input size.[2] We exclude raytrace from these results, since it did not run correctly on the Radish simulator in our experiments. We ran Radish with its relaxed asynchronous checking and unsafe type-safety optimization disabled [25] for the most direct comparison to Parsnip, which provides synchronous data race checks and does not make assumptions about type safety. Additionally, we ran Radish with its type-safety assumption enabled, represented by the Radish-ts bars in Figure 4.7. This enables better performance but

[2]Long running times and high memory usage of the Radish simulator made performance experiments for larger input sizes or thread counts infeasible on the Radish simulator in our experiments.

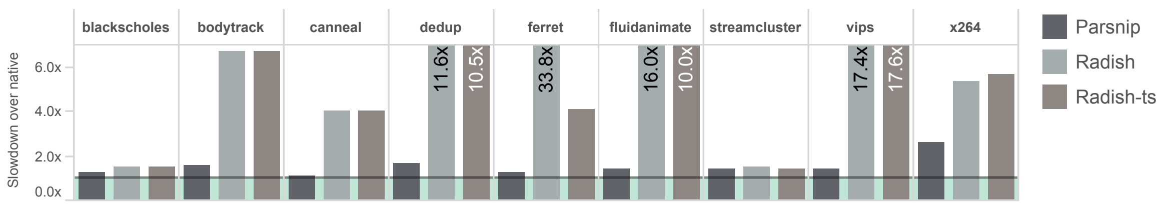


Figure 4.7: Slowdown of Parsnip, Radish, and Radish-ts, normalized to a system without data race detection support.

is unsound and incomplete in the presence of type-safety violations.

On all 9 PARSEC benchmarks on which we compare PARSNIP against Radish, PARSNIP runs faster than Radish. On average (geomean), PARSNIP slows the baseline CPI by 1.5x, whereas Radish's average slowdown is 6.9x. The maximum slowdown caused by PARSNIP is 2.6x on x264. Radish slows ferret by 33.8x. PARSNIP is on average 4.6x faster than Radish, and runs ferret 27.2x faster than Radish. Enabling Radish's type-safety assumption in Radish-ts yields a 1.3x boost over the safe version of Radish. Nonetheless, the optimized Radish-ts remains 3.4x slower than PARSNIP on average, and runs vips 12.3x slower than PARSNIP does.

4.4.3. Scalability

To evaluate PARSNIP's performance scaling over the number of threads of the target program, we ran PARSNIP experiments with 2, 4, 8 and 16 threads on 16 simulated cores with the simmedium input size. Slowdown of PARSNIP's CPI with respect to CPI of the baseline system is shown in Figure 4.8. On 4 benchmarks (blackscholes, canneal, fluidanimate, streamcluster), PARSNIP's overhead decreases with more threads. On 4 additional other benchmarks (bodytrack, dedup, ferret, x264), PARSNIP overhead increases with more threads. PARSNIP's overhead shows no significant difference on the remaining 2 benchmarks (raytrace and vips). For raytrace, PARSNIP shows no appreciable slowdown at all. PARSNIP's worst slowdown is 2.7x on x264 with 16 threads.

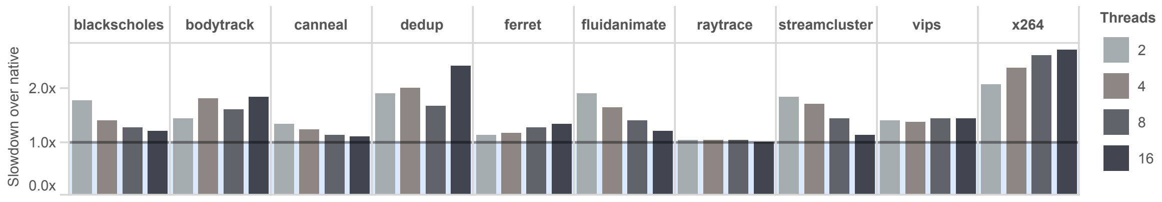


Figure 4.8: Slowdown of Parsnip with 2, 4, 8 and 16 threads, using the simmedium input.

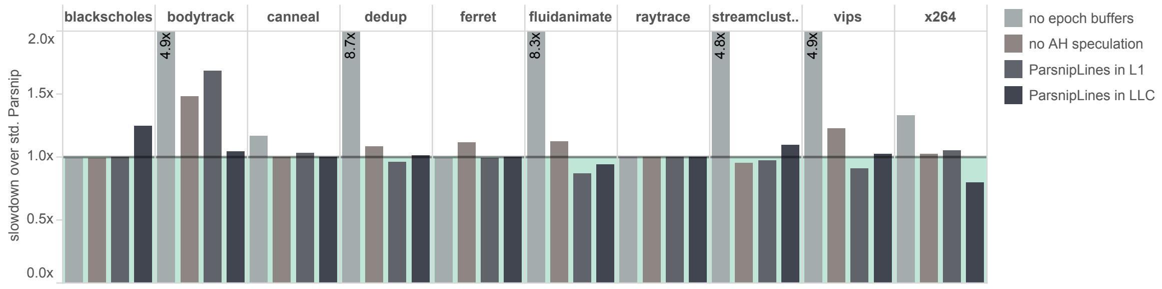


Figure 4.9: Slowdown of Parsnip with different optimizations disabled.

4.4.4. Effectiveness of Optimizations

This subsection presents results of several experiments conducted to evaluate the effectiveness of each optimization described in Section 4.3. Figure 4.9 shows the slowdown in CPI of several variants of Parsnip, as normalized to the standard Parsnip configuration. Epoch store buffers (described in Section 4.3.1) have the largest impact. Parsnip runs on average 3.7x slower with epoch store buffers disabled ("no epoch buffers"). Parsnip with access history prefetching and prediction (Section 4.3.2) offers an average 10% performance improvement versus without ("no AH speculation"). The baseline adaptive selection of the ParsnipLine cache residence policy (Section 4.3.3) shows comparable average performance with both the fixed L1 and LLC ParsnipLine residence policies ("ParsnipLines in L1, LLC"). However, the adaptive policy avoids the larger slowdowns of the L1 policy on bodytrack, and the LLC policy on blackscholes. In summary, all benchmarks see (often substantial) performance improvement from at least one of the optimizations described in Section 4.3.

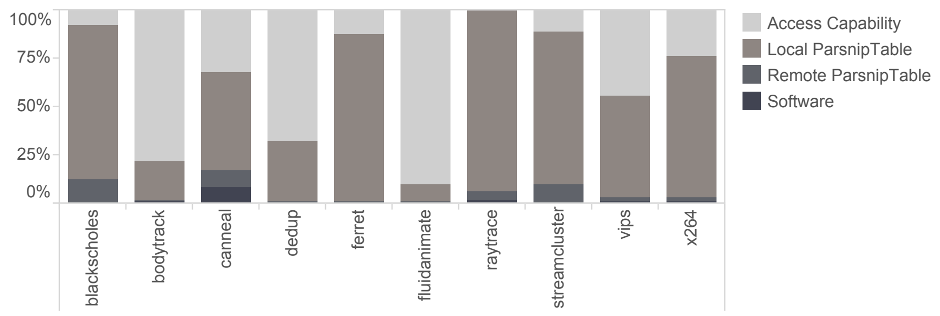


Figure 4.10: Breakdown of how often each access history source resolves a data race check.

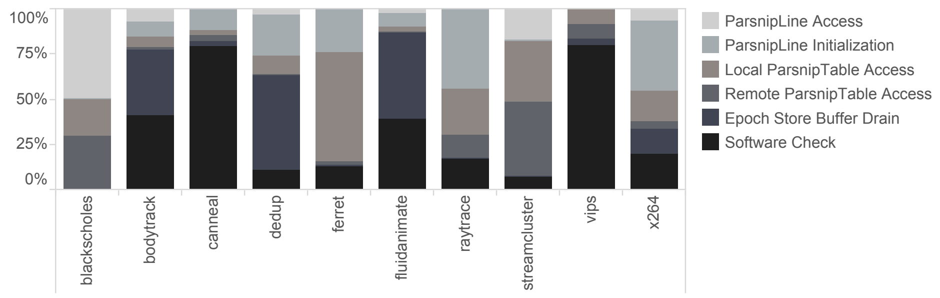


Figure 4.11: Contribution of architectural events to overall Parsnip overhead.

4.4.5. Architectural Characterization

We ran several characterization experiments to better understand the sources of Parsnip's overhead.

Figure 4.10 shows the percentage of data race checks that are resolved by each Parsnip mechanism. ParsnipRefs and the local core's ParsnipTable suffice to resolve at least 84% of checks in each benchmark. Access capabilities in ParsnipRefs alone resolve 68-90% of checks in bodytrack, dedup, and fluidanimate. Remote ParsnipTable lookups resolve a modest number of checks, up to 12% in blackscholes. With the exception of canneal, which requires software checks on 8% of accesses, all other benchmarks require software checks on <1% of accesses, with most <0.6%.

Figure 4.11 shows the breakdown of Parsnip's overhead from 6 different sources: (a) the cost of accessing ParsnipRefs in cache; (b) the cost of software handling when a ParsnipRef is not in cache; (c) the cost of runtime checks that complete after comparing with a clock in the local

Benchmark	Epoch buffer drains	Parsnip-Line evictions	Parsnip-Table rejections	SW checks
blackscholes	0.00	0.00	0.00	0.01
bodytrack	0.00	0.00	0.13	0.91
canneal	0.00	0.00	0.03	2.57
dedup	0.01	0.00	0.49	0.57
ferret	0.00	0.00	0.00	0.17
fluidanimate	1.38	0.00	79.15	0.34
raytrace	0.00	0.00	0.00	0.02
streamcluster	0.00	0.00	0.01	0.05
vips	0.00	0.00	0.02	0.45
x264	0.00	0.00	0.00	1.45

Table 4.3: Events per 1K instructions in Parsnip.

ParsnipTable; (d) the cost of runtime checks that finish after comparing with a clock in some remote ParsnipTable; (e) the cost of software check handlers; (f) the cost of saving the contents of an epoch buffer to software.

Table 4.3 shows the frequency of the most expensive architectural events in PARSNIP. With 32-entry epoch buffers, persisting epoch values to software is rare (column 2). The highest rate of epoch buffer drains is 1.38/1K instructions, in fluidanimate. ParsnipLines evictions are also rare (column 3). Column 4 shows that ParsnipTable rejections occur at a rate below 0.5/1K instructions in all benchmarks except fluidanimate. Despite its high rate of 79.15 ParsnipTable rejections per 1K instructions, most fluidanimate checks complete in hardware partly due to epoch buffers and the use of access capabilities (see Figure 4.10). Column 5 shows software checks per 1K instructions. While canneal and x264 have high software check rates of 2.57/1K instructions and 1.45/1K instructions, respectively, all other benchmarks execute fewer than 1 software check per 1K instructions.

Overall, PARSNIP's access history organization and optimizations are effective in allowing most runtime checks to be performed in hardware.

4.4.6. Hardware Overheads

We used CACTI 5.3 [109] to model the area and latency overheads of the hardware PARSNIP adds to a conventional processor core. PARSNIP's storage costs are modest: the epoch buffer and ParsnipTable for each core occupy less than 0.02 mm^2 in 32nm technology. For comparison, [112] states that a 2-wide in-order core at 32nm (excluding caches) occupies 0.1875 mm^2. PARSNIP does require an extra L1D$ read port to allow data and metadata to be read in parallel, increasing access latency by about 20%. In a superscalar design, PARSNIP could reuse an existing cache port but would reduce the number of memory operations that could be scheduled in parallel.

4.5. Conclusion

With the prevalence of parallel programs and systems, efficient data race detection is an increasingly important topic. However, previous software solutions incur high performance overheads, restricting their usability in real-world scenarios. Fast hardware race detectors either trade precision for performance, or make overly strict assumptions.

This Chapter discusses PARSNIP, a sound and complete data race detector that can track memory accesses at byte granularity efficiently. PARSNIP adds moderate hardware modifications to a conventional multicore processor, and does not rely on unsound type-safety assumptions. PARSNIP outperforms Radish, the leading hardware race detector, by 4.6x on average. PARSNIP incurs just 1.5x overhead over native execution on average.

CHAPTER 5

Practical Dynamic Data Race Detection for GPU

In recent years, graphics processing units (GPUs) have thoroughly permeated consumer processor designs. It is now essentially impossible to find a smartphone, tablet or laptop without a substantial integrated GPU on the processor die. Utilizing these omnipresent GPUs, however, remains a challenge. Writing correct parallel code, a notoriously difficult task, is exacerbated by the high degrees of parallelism that GPUs demand to attain high performance. GPU programming models have also grown more expressive over time to support increasingly general-purpose GPU (GPGPU) programming. This extra expressiveness, unfortunately, allows interesting classes of data races to arise. Such bugs can introduce complicated consistency model semantics and even undefined behavior into programs, making debugging difficult. Therefore, it is necessary to have a precise and efficient way to detect data races in real-world programs that run on GPUs.

This chapter focuses on discussing precise dynamic data race detection for GPU programs. First, a brief introduction to the GPU programming model is given. After that, we discuss challenges and opportunities in race detection for GPU. The discussion then continues to describe BARRACUDA, a dynamic data race detection system for CUDA programs, followed by the evaluation results of the system.

5.1. GPU (CUDA) Programming Model

This section provides a brief primer on the CUDA programming model, the most widely used model in GPU programming. A CUDA program that runs on a GPU is called a kernel. Nvidia's CUDA[81] programming model has two core abstractions: one for parallelism and one for the GPU hardware's memory hierarchy, as is shown in Figure 5.1. Parallel threads are expressed in a hierarchical structure known as a grid. A grid is composed of thread blocks, which are in

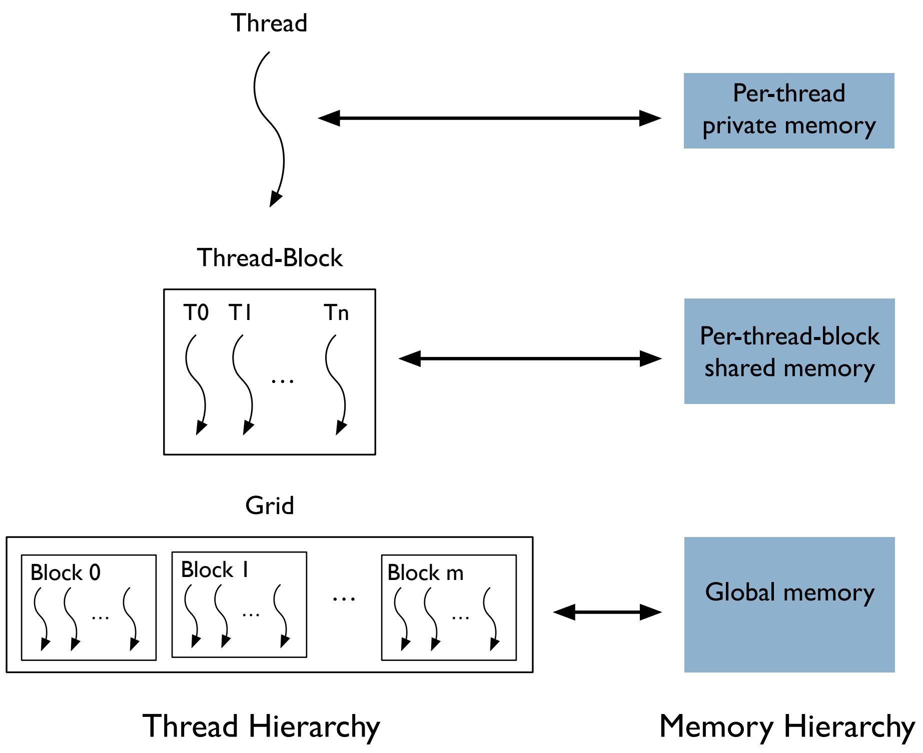


Figure 5.1: The CUDA thread and memory hierarchy.

turn decomposed into individual threads. The arrangement of blocks within the grid, or threads within a block, can be 1-dimensional, 2-dimensional or 3-dimensional as desired to fit a specific problem domain. Individual threads within a thread block are also grouped into warps, which often execute in lockstep on the GPU's SIMD hardware. Many thread blocks may be scheduled concurrently onto one streaming multiprocessor (SM), analogous to a CPU core, though a single thread block is never split across SMs.

CUDA programs are written in a variant of C/C++ and compiled to a high-level assembly language called PTX (Parallel Thread eXecution). All PTX instructions are SIMD instructions executed by an entire warp of threads. Scalar or sub-warp execution can be achieved via branches. If a branch condition does not evaluate the same way for every thread in a warp, *branch divergence* arises. Branch divergence is handled via a SIMT stack ([46],Section 5.4.3) that tracks the active threads within a warp.

CUDA's abstraction of memory provides a collection of memory spaces to each kernel. At the top level, global memory (also known as device memory) is accessible to all threads within a kernel. Shared memory is accessible only by threads within the same thread block. Though each thread block has access to shared memory, threads within thread block b1 see a different memory space than those within thread block b2. The final memory space is local memory, which is private to each individual thread. The semantics of these different memory spaces lead to vastly differing implementations and performance characteristics. Global memory is primarily backed by the GPU's off-chip DRAM, though in modern GPUs it can be cached in on-chip L2 caches and, in some cases, L1 caches as well. Shared memory, in contrast, is backed by on-chip L1 caches that are fast but limited to 48KB per thread block in the current version of CUDA. Local memory is backed by a combination of cacheable global memory and the GPU's register file.

CUDA provides a limited set of synchronization constructs for coordination across threads. The core construct is a thread-block-wide barrier called $\_\_syncthreads$, which is fully supported by all current CUDA race detectors. Lower-level synchronization primitives also exist in the form of atomic instructions and memory fence instructions, although these primitives per se are not considered reliable synchronization mechanisms in the programming model.

5.2. Challenges For Dynamic Data Race Detection on GPU

As described in Chapter 2, a parallel program can be modeled as a trace of events. The effects of synchronization operations can be modeled by the *happens-before* relation, which can be implemented using data structures like vector clocks. While these implementations can handle the scale of concurrency and synchronization in conventional CPU programming models, they struggle with the massive scale of GPU code.

In *happens-before* race detection, for example, each thread in the program has a vector clock, with the vector size equal to the number of threads (n^2 storage for n threads). GPU programs can easily reach millions of threads, requiring hundreds of gigabytes of storage for these vector clocks alone, leaving aside other race detection metadata. Such prohibitively high overhead prevents dynamic data race detection algorithms, such as vector-clock and FastTrack, to be applied directly to GPU programs.

5.3. Redundancy in GPU Race Detection

As shown in the previous chapters, dynamic data race detection generally requires keeping two pieces of metadata: 1) the per-location access histories for all shared memory locations, and 2) the per-thread states that encode logical times and synchronization events across threads. Chapters 3 and 4 have explained the high redundancy of the former for CPU programs, and have discussed the SlimFast and Parsnip systems that gain performance boost by reducing such redundancy. When it comes to race detection for the GPU programming model, the largest difficulty is how to handle the latter, i.e., the per-thread states for millions of threads.

In the classic vector-clock algorithm, the per-thread state to maintain is a vector clock, which is an $O(N)$ structure with a clock entry for each of the N threads within the program. Figure 5.2 gives an example execution on the left, with the vector clock for thread 1 on the right. The example kernel has 3 threads per warp, 2 warps per block, and 2 blocks. Shading of the example execution indicates each thread's logical time. In the example execution, thread T0 takes one branch of the *if* statement, whereas $T1$ and $T2$ takes another. In the vector clock of $T1$ (illustrated in the right half of Figure 5.2) the clock value for $T0$ is 1, whereas the values for $T1$ and $T2$ are 2. For the rest of the threads (external to the warp containing $T0$, $T1$, and $T2$), their values in the vector clock are all 0. In this case, even though the vector clocks sizes can be large if the program has a large number of threads, there are only a small number of distinct values within each per-thread vector clock (PTVC), which is a form of redundancy.

To measure how frequent such redundancy arises, a profiling experiment is conducted on a set of CUDA programs to count the distinct values within PTVCs. According to the data collected, roughly 90% of the time PTVCs have the same value for all threads external to a warp and either 1) the same value for all threads in a warp or 2) two distinct values, e.g., along the two branches of an if-else statement, as shown in Figure 5.2. This creates an opportunity for space

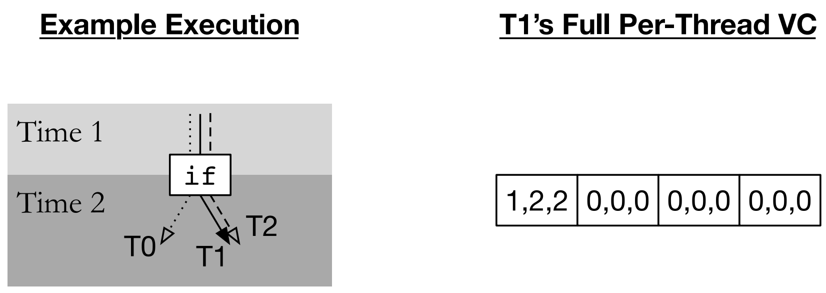


Figure 5.2: An example execution illustrating the redundancy in per-thread vector clocks.

savings by storing just a few clock values for each warp.

5.4. Barracuda Semantics

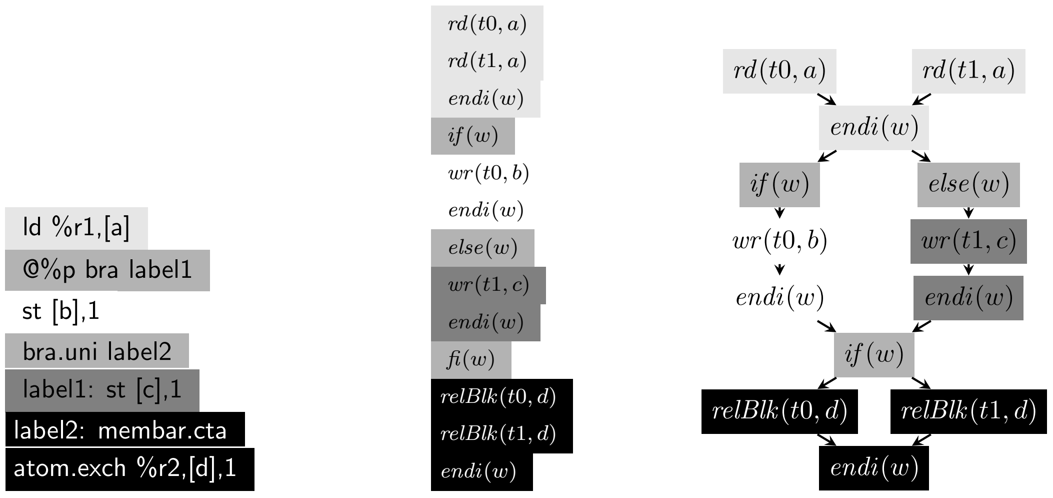
The core idea of BARRACUDA's design is to exploit the opportunity to cut down the overhead of PTVCs via a highly-compressed representation, thereby allowing conventional CPU-oriented dynamic race detection algorithms, such as FASTTRACK, to be adapted and applied to GPU programs. This section first describes how to model a CUDA execution as a trace, then introduces the operational semantics of BARRACUDA.

5.4.1. Modeling a CUDA Execution as a Trace

A program execution is modeled as a *trace*: a sequence of operations performed by a set of threads. Trace operations are an abstraction over the stream of dynamic PTX instructions to facilitate race detection. Our trace operations are:

- $rd(t, x)$ or $wr(t, x)$, in which a thread t reads or writes a location x

- $endi(w)$, used to model a warp w's lockstep execution
- $if(w)$, in which warp w begins executing a branch
- $else(w)$, in which w executes the else path of a branch
- $fi(w)$, in which w concludes its execution of a branch
- $bar(b)$, a barrier for all threads in thread block b
- $atm(t, x)$, in which t performs an atomic read-modify-write operation on a location x
- $acqBlk(t, x)$, $relBlk(t, x)$ or $arBlk(t, x)$, in which t acquires, releases (or both) a synchronization location x with a block-level memory fence
- $acqGlb(t, x)$, $relGlb(t, x)$ and $arGlb(t, x)$ behave like the block-level versions but with a global fence



(a) PTX instructions **(b) trace operations** **(c) synchronization order**

Figure 5.3: (a) Sample PTX instructions for a warp w with 2 threads, $t0$ and $t1$. (b) Shading shows the translation from PTX instructions into trace operations. (c) Arrows indicate synchronization order.

Our trace includes operations like acquires and releases, similar to high-level language consistency models like the C++ memory model [13]. Inferring trace operations involves heuristics (so that our traces are fundamentally approximations of the synchronization that actually occurred in an execution) and is complicated by the lack of an official CUDA memory consistency model to define illegal behavior. We describe below a useful set of rules for translating PTX instructions into trace operations.

While PTX instructions represent warp-level operations, *e.g.*, an entire warp performing a vector read from memory, for simplicity we model memory operations (reads, writes, atomics, acquires and releases) as *thread-level* operations so that we can consider an access to one memory location at a time. However, previous work has taken into account the fact that warps execute in a "lockstep" fashion [119] wherein all operations from warp instruction i complete before instruction $i + 1$ begins. Lockstep warp execution is indirectly acknowledged in the official CUDA documentation, stating that cores perform scheduling at warp granularity, a warp executes only one common instruction at a time, and warps are issued in program order [81, §4]. The treatment of diverging control flow within a warp also gives evidence that warps execute in lockstep (Section 5.4.3). However, the actual size of a warp can change across architectures, so portable CUDA code should eschew assumptions about warp size, or validate these assumptions at runtime. BARRACUDA's dynamic analysis checks for races based on the warp size of the current architecture, though in the future we could simulate the behavior of smaller/larger warps to find additional latent bugs.

We encode the end of warp w's instruction explicitly with an $endi(w)$ operation. A warp w performing a read of location x is thus translated into $rd(t, x)$ for each active thread t in w followed by $endi(w)$ (see the top part of Figure 5.3a & 5.3b).

Normally, given a dynamic trace, control flow constructs like loops, function calls, branches and so forth are only implicitly represented. However, we are the first to recognize that branches in GPUs have synchronization implications, so we include explicit branch operations in a trace. Unconditional control flow constructs like loops and function calls do not require such handling and are implicitly unrolled/inlined in the trace.

Control-flow operations are modeled at warp level as they manipulate the warp-level stack that tracks branches, and would be awkward to model at the level of individual threads. if, $else$ and fi operations are readily inferred from static PTX code by examining the targets of branch instructions (see Figure 5.3). All branches are encoded using if, $else$ and fi for simplicity: simpler constructs like an if statement (without an else) can be encoded via an empty else path.

We infer synchronization trace operations as follows. $bar(b)$ represents a block-wide barrier for thread block b, when every thread in b having executed the bar.sync PTX instruction (__syncthreads in CUDA). Atomic instructions (atom.* in PTX, or atomic* functions in CUDA) from a thread t to location x not immediately preceded or followed by a memory fence in static code become standalone $atm(t, x)$ operations (see Section 5.4.3 for more details).

If a store instruction is immediately preceded by a memory fence (membar.cta or membar.gl in PTX, __threadfence_block or __threadfence in CUDA[1]) in static code, the store plus the fence are bundled together into a release operation, with the scope (block or global) determined by the kind of fence used (see the $relBlk$ operation in Figure 5.3). Acquire operations arise similarly, from a load followed by a fence. An atomic instruction sandwiched between fences acts as both an acquire and a release (like $arBlk$). To identify errors in CUDA lock implementations, we treat the atom.cas and atom.exch PTX instructions specially. atom.cas performs a compare-and-swap, commonly used for obtaining a lock, and atom.exch performs a fetch-and-set, commonly used to free a lock. If atom.cas is followed by a fence, we treat them as an acquire. If atom.exch is preceded by a fence we treat them as a release.

Our inferences of acquire and release operations from PTX code are necessarily approximate, as other CUDA code may compile into something that looks to us like an acquire or release operation. If we infer an acquire/release where none existed in the original code, BARRACUDA may consider an execution safe when it actually contains a race. Interestingly, even the CUDA C/C++ API defines synchronization operations in terms of fences and loads/stores/atomics, instead of with high-level acquires and releases. Thus, some inference is necessary whether performing race detection at the PTX or the CUDA C/C++ level. We tuned our inference

[1]System-level fences are treated as global fences, as we focus on intra-kernel races.

of acquire/release operations based on litmus tests, documentation, and sophisticated code examples like threadFenceReduction from the CUDA SDK, and find that our policy avoids any incorrect atomic inference for our benchmarks.

We consider only *feasible* traces: those where (1) a warp-level memory instruction from warp w is represented in the trace as a consecutive sequence of memory operations, one for each active thread in w, (2) each of w's memory instructions is followed by an $endi(w)$ operation, and (3) branches are translated appropriately into $if(w)$, $else(w)$ and $fi(w)$ operations.

5.4.2. Synchronization Order

While operations appear in a total order in a trace, that order does not imply that the effects of operations can be linearized. Instead, we derive a partial order called *synchronization order* from a trace α, written $<_\alpha$, such that $a <_\alpha b$ when a must occur before b. Synchronization order is the transitive closure of the smallest relation such that $a <_\alpha b$ when a occurs before b in α and either:

- a and b are both performed by thread t (intra-thread program order); or
- one of a or b is $endi(w)$, and the other is by a thread that's both in w and active at the time the $endi$ is performed (intra-warp program order); or
- one of a and b is $bar(k)$, and the other is by a thread in k (barrier synchronization); or
- a and b are operations on the same synchronization location x where a is a release operation and b is an acquire operation, and both operations are either 1) at block scope within the same thread block or 2) at least one operation is at global scope (inter-thread synchronization)

Given this definition, a *data race* occurs when two operations a and b both access the same location, at least one of them is a write, they are not both atm operations (atomic operations do not race with each other, but also do not imply synchronization), and neither $a <_\alpha b$ nor $b <_\alpha a$ holds (so that the operations are seen as *concurrent* in the trace).

5.4.3. The Barracuda Algorithm

BARRACUDA checks for races by maintaining a tuple of metadata based on *vector clocks*. As we have introduced in Chapter 2, a vector clock V records a timestamp for each thread t in a system, written as $V(t)$. Recall that the standard comparison ($\sqsubseteq$), join ($\sqcup$) and increment (inc_t) operations on vector clocks are defined as:

$$\begin{array}{lcl} V \sqsubseteq V' & \text{iff} & \forall t.\ V(t) \leqslant V'(t) \\ V \sqcup V' & = & \lambda t.\ \max(\ V(t), V'(t)\ ) \\ inc_t(V) & = & \lambda u.\ \text{ if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \end{array}$$

The minimal vector clock has a 0 timestamp for each thread and is written $\bot_V$.

Just as with the FASTTRACK race detector [38], to save space *epochs* are sometimes used in place of vector clocks; an epoch $c@t$ is a reduced vector clock that holds a timestamp for just one thread, and is treated as a vector clock that is c for t and 0 for every thread other than t. Because epochs have a single non-zero entry, an epoch can be compared with a vector clock, or another epoch, in O(1) time using the $\preceq$ operator. We say $c@t \preceq V$ when $c \leqslant V(t)$. $\bot_e$ denotes a minimal epoch $0@t0$.

The BARRACUDA analysis state is a tuple (K, C, S, R, W). K_w is the per-warp stack for warp w that tracks branch divergence. Each stack entry is an *active mask* ($amask$ for short) which is the set of threads that are currently active.

C_t is a vector clock for the thread t: $C_u(t)$ records the last time at which the thread t synchronized with u. S_x represents the synchronization location x, which is an ordinary memory location since CUDA programs often use the same location to store data and for coordination. S_x is a map from thread block $\mapsto$ vector clock, recording the most recent logical time at which some thread from each thread block synchronized with x. R_x is the read metadata for a location x recording the most recent reads of x, which can be encoded either as an epoch or as a vector clock. W_x is a tuple of (write epoch, atomic-bit) for a location x, recording the time of the most recent write to x. The atomic-bit records whether the most recent write to x arose from an atomic operation or not. Epoch comparison $\preceq$ with write metadata W_x ignores the atomic bit. We write $E(t)$ for the epoch $C(t)@t$, the current epoch for thread t.

Our initial analysis state σ_0 is the tuple ($\lambda w.[\mathit{initActive}]$, $\lambda t.\mathit{inc}_t(\bot_V)$, $\lambda x, b.\bot_V$, $\lambda x.\bot_e$, $\lambda x.(\bot_e, \mathit{false})$). Each warp's initial active mask takes account of the number of threads requested for the grid, as the last warp of each thread block may be only partially full. Each thread initially has an empty vector clock with its own entry incremented, all synchronization locations have empty vector clocks for all blocks, and all memory locations have empty read epochs and empty write epochs without any previous atomic operations.

Basic Operations

Figure 5.4 gives the operational semantics for BARRACUDA for non-synchronization memory accesses and branches. For thread-level memory accesses, if a thread t is not active (due to a branch), t's operation is a NOP – no analysis state is updated. To avoid clutter, each rule implicitly checks that the current thread is active.

Read and write operations are handled essentially as with FASTTRACK. Totally-ordered reads can use a compact epoch representation (rule READEXCL), while concurrent reads require a vector clock (rule READSHARED). The first concurrent read, which necessitates a transition from an epoch to a vector clock, is handled by the READINFLATE rule. The WRITEEXCL rule handles totally-ordered writes, and WRITESHARED the first write after concurrent reads. Because the ENDINSN rule increments per-thread logical time after every instruction, the "same epoch" rules of FASTTRACK are not needed.

To faithfully model lockstep warp execution, the individual thread memory operations within a warp instruction run concurrently. This allows us to detect **intra-warp races**.[2] With intra-warp races, according to Nvidia, "the number of serialized writes that occur to that location varies depending on the compute capability of the device ... and which thread performs the final write is undefined" [81, §4.1]. Thus, intra-warp races can result in architecture-specific behavior and

[2]An intra-warp race is always a write-write race, as all active threads within the warp execute the same instruction and reads cannot race.

$$\frac{\begin{array}{l} R_x \in \mathit{VectorClock} \qquad W_x \preceq C_t \\ R' = R_x[t := C_t(t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)} \textsc{ ReadShared}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ R_x \preceq C_t \qquad W_x \preceq C_t \\ R' = R[x := E(t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)} \textsc{ ReadExcl}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ W_x \preceq C_t \qquad R_x = \mathit{clock}@t' \\ vc = \bot_v[t := C_t(t), t' := \mathit{clock}] \\ R' = R[x := vc] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)} \textsc{ ReadInflate}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ W_x \preceq C_t \qquad R_x \preceq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{false})] \end{array}}{(K, C, S, R, W) \Rightarrow^{wr(t,x)} (K, C, S, R', W')} \textsc{ WriteExcl}$$

$$\frac{\begin{array}{l} R_x \in \mathit{VectorClock} \\ W_x \preceq C_t \qquad R_x \sqsubseteq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{false})] \end{array}}{(K, C, S, R, W) \Rightarrow^{wr(t,x)} (K, C, S, R', W')} \textsc{ WriteShared}$$

$$\frac{\begin{array}{l} \mathit{amask} = K_w.\mathsf{peek}() \\ vc = \bigsqcup_{t \in \mathit{amask}} C_t \\ \forall t \in \mathit{amask}\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{endi(w)} (K, C', S, R, W)} \textsc{ EndInsn}$$

$$\frac{\begin{array}{l} \mathit{amask}_1, \mathit{amask}_2 = \mathsf{splitActive}(K_w.\mathsf{peek}()) \\ \mathit{stack}_1 = K_w.\mathsf{push}(\mathit{amask}_1) \\ \mathit{stack}_2 = \mathit{stack}_1.\mathsf{push}(\mathit{amask}_2) \\ K' = K[w := \mathit{stack}_2] \\ vc = \bigsqcup_{t \in \mathit{amask}_2} C_t \\ \forall t \in \mathit{amask}_2\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}_2\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{if(w)} (K', C', S, R, W)} \textsc{ If}$$

$$\frac{\begin{array}{l} \mathit{stack} = K_w.\mathsf{pop}() \\ K' = K[w := \mathit{stack}] \\ \mathit{amask} = \mathit{stack}.\mathsf{peek}() \\ vc = \bigsqcup_{t \in \mathit{amask}} C_t \\ \forall t \in \mathit{amask}\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{else(w),fi(w)} (K', C', S, R, W)} \textsc{ ElseEndif}$$

Figure 5.4: Barracuda basic operational semantics

nondeterminism. If all active threads within the warp write the same value to a location, we do not consider this a race as the documentation is clear that the outcome is well-defined. Our implementation detects and filters such "same-value" intra-warp races.

The $endi$ rule is used to join all active threads together after all thread-level memory operations complete, and then to fork the active threads again to support detection of future intra-warp races. Note that $endi$ operates only on active threads within a warp w; inactive threads (*e.g.*, those following a different control flow path) are logically concurrent with the active threads, as we explain next.

Branches on GPUs are handled via a hardware SIMT stack [46]. The top of the stack tracks which threads are active along the current control-flow path, and deeper entries support nested control flow. We explain the operation of the SIMT stack along with our semantics. When an if operation is encountered, the set of currently-active threads is split according to the branch condition into two sets: those threads active on the *then* path and those active on the *else* path. One of these sets may be empty due to the branch condition. The IF rule uses the splitActive function to capture the actual active masks. The *then* and *else* sets are represented as active masks that are pushed onto the stack K_w for the current warp w. The order in which they are pushed is arbitrary, but determines the order in which the paths will execute. In our IF rule, the *else* path is pushed first so the *then* path executes first. While Nvidia states that "the different executions paths have to be serialized" [81, §5.4.2] they do not define the order in which the serialization occurs. These semantics are similar to the way event handlers are treated in event-based concurrency systems like Android and JavaScript [97, 53, 8]. Our semantics treats different paths as concurrent so that we can identify **branch ordering races** between paths, though our modeling is conservative in that we do not exempt commutative paths. Branch ordering races are a new class of bugs not identified in previous work, and represent a subtle way in which a GPU program's correctness can implicitly rely on a given architecture and its SIMT stack implementation. Once the *then* and *else* active masks are determined, the IF rule joins and forks the *then* threads, capturing the fact that they are now concurrent with the *else* threads.

$else$ and fi operations are handled the same in our semantics. First we pop the SIMT stack to discard the *then*/*else* active mask, respectively, and perform a join and fork of the newly-active threads. For an $else$ operation, this models the beginning of the *else* path's execution which is logically concurrent with the *then* path. For a fi operation, this models threads from both the *then* and *else* paths restarting lockstep execution after their branching is complete.

Barriers and Atomic Operations

Figure 5.5 presents BARRACUDA's operational semantics for synchronization operations. $bar(b)$ is the simplest operation, representing a barrier for all threads within a thread block b. The BAR rule has an explicit predicate that all threads in b are active, as otherwise the Nvidia documentation states that "the code execution is likely to hang or produce unintended side effects" [81, §B.6]. Executing a bar operation with inactive threads, known as a **barrier divergence bug**, is detected as an error by BARRACUDA.

The INITATOM* rules handle an atomic operation on location x where the preceding write to x was non-atomic. These rules check for ordering with previous reads and the previous non-atomic write, as Nvidia states in the PTX documentation that "atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address." [82, §8.7.12.3]. While the documentation leaves the door open for stronger semantics for atomics on global memory, recent work [3] recommends that programmers avoid making both atomic and non-atomic accesses to the same global memory location, as doing so can exhibit relaxed consistency effects. We adopt a similar approach, and do not consider atomic and non-atomic accesses to synchronize with one another.

The ATOM* rules handle an atomic operation on x when the preceding write to x was another atomic. These rules check for ordering with preceding reads, but elide checks of the previous atomic write. Nvidia states that "Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations" [81, §B.12]. We capture

$$\frac{\begin{array}{l} \forall t \in b \ .\ \mathsf{active}(t) \\ vc = \bigsqcup_{t \in b} C_t \\ \forall t \in b \ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin b \ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{bar(b)} (K, C', S, R, W)} \ \textsc{Bar}$$

$$\frac{\begin{array}{l} W_x = (-, \mathsf{false}) \qquad R_x \in \mathit{Epoch} \\ W_x \preceq C_t \qquad R_x \preceq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{true})] \end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')} \ \textsc{InitAtomExcl}$$

$$\frac{\begin{array}{l} W_x = (-, \mathsf{false}) \qquad R_x \in \mathit{VectorClock} \\ W_x \preceq C_t \qquad R_x \sqsubseteq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{true})] \end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')} \ \textsc{InitAtomShrd}$$

$$\frac{\begin{array}{l} W_x = (-, \mathsf{true}) \qquad R_x \in \mathit{Epoch} \\ R_x \preceq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{true})] \end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')} \ \textsc{AtomExcl}$$

$$\frac{\begin{array}{l} W_x = (-, \mathsf{true}) \qquad R_x \in \mathit{VectorClock} \\ R_x \sqsubseteq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{true})] \end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')} \ \textsc{AtomShared}$$

$$\frac{C' = C[t := C_t \sqcup S_x[\mathsf{block}(t)]]}{(K, C, S, R, W) \Rightarrow^{acqBlk(t,x)} (K, C', S, R, W)} \ \textsc{AcqBlock}$$

$$\frac{\begin{array}{l} S' = S_x[\mathsf{block}(t) := C_t] \\ C' = C[t := \mathsf{incr}_t(C_t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{relBlk(t,x)} (K, C', S', R, W)} \ \textsc{RelBlock}$$

$$\frac{\begin{array}{l} C' = C[t := C_t \sqcup S_x[\mathsf{block}(t)]] \\ S' = S_x[\mathsf{block}(t) := C'_t] \\ C'' = C'[t := \mathsf{incr}_t(C'_t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{arBlk(t,x)} (K, C'', S', R, W)} \ \textsc{AcqRelBlk}$$

$$\frac{\begin{array}{l} vc = \bigsqcup_{b \in grid} S_x[b] \\ C' = C[t := C_t \sqcup vc] \end{array}}{(K, C, S, R, W) \Rightarrow^{acqGlb(t,x)} (K, C', S, R, W)} \ \textsc{AcqGlobal}$$

$$\frac{\begin{array}{l} \forall b \in grid \ .\ S'_x[b] = C_t \\ C' = C[t := \mathsf{incr}_t(C_t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{relGlb(t,x)} (K, C', S', R, W)} \ \textsc{RelGlobal}$$

Figure 5.5: Semantics for synchronization operations

this constraint by avoiding checks between atomic operations and also avoiding additions to synchronization order. Thus, atomics alone cannot be used to synchronize between threads.

Memory Fence Litmus Tests

To explore the semantics of inter-thread synchronization, we conducted a series of litmus tests on two Nvidia GPUs: a GRID K520 Kepler GPU, obtained via Amazon AWS, and a GTX Titan X Maxwell GPU on a local machine. We ran variations of the message-passing (**mp**) litmus test from Alglave *et al.* [3], with different combinations of fences in each thread. The variables x and y reside in global memory, the default cache operator is .cg (skipping the incoherent L1 cache), and each test thread runs in a distinct thread block. We utilized Alglave *et al.* and Sorensen *et al.* [3, 107]'s memory stress and thread randomization strategies to provoke weak consistency behavior.

```
    init: x = y = 0        final: r1=1 ∧ r2=0
1.1  st.global.cg [x],1   2.1  ld.global.cg r1,[y]
1.2  fence1               2.2  fence2
1.3  st.global.cg [y],1   2.3  ld.global.cg r2,[x]
```

observations per 1 million runs

fence1	*fence2*	K520	GTX Titan X
membar.cta	membar.cta	7,253	0
membar.cta	membar.gl	0	0
membar.gl	membar.cta	0	0
membar.gl	membar.gl	0	0

Figure 5.6: Memory fence litmus tests

Our results are presented in Figure 5.6, which shows that using a membar.cta in each thread allows non-sequentially-consistent (non-SC [59]) behavior to arise on the K520 GPU, though not on the GTX Titan X. Using a membar.gl in either thread resulted in SC behavior across all our tests on both GPUs. Our results are consistent with Alglave *et al.* [3], which ran only tests with the same fence in each thread. Of course, testing cannot prove the absence of weak behavior, but our results demonstrate that membar.cta is insufficient to implement synchronization between thread blocks.

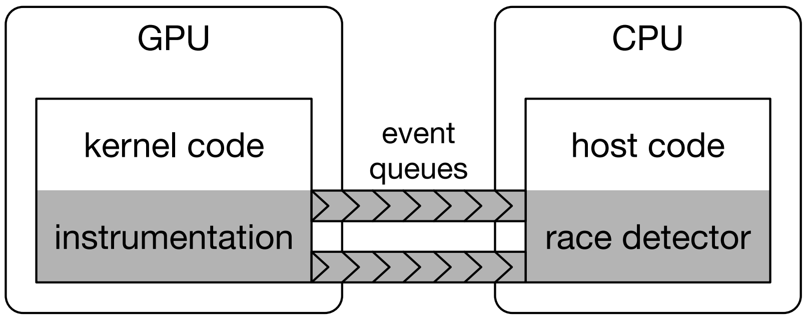


Figure 5.7: System overview: shading indicates the components of Barracuda.

Inter-thread Synchronization

To realize inter-thread synchronization, release and acquire operations must be used. These rules update the S_x metadata, a map from thread blocks to vector clocks for a location x, which is used to propagate synchronization order. The AcqBlock rule is similar to a lock acquire in a CPU program in that the current thread t joins its vector clock C_t with the vector clock for the synchronization location, but scoped to the particular thread block in which t resides. The RelBlock rule accordingly updates S_x only for the current block. A $relBlk$ in block b_1 followed by an $acqBlk$ in block b_2 thus does *not* contribute to synchronization order, as non-SC behavior is possible in this case (Section 5.4.3).

Our litmus tests show that a global fence in just one message-passing thread results in SC behavior. The AcqGlobal rule thus joins the vector clocks for all blocks in S_x, while RelGlobal sets the vector clocks for all blocks in the grid. This ensures that a global release/acquire in one block can synchronize with an acquire/release in any other block, even if the latter operation is at block scope.

5.5. Implementation

Figure 5.7 shows an overview of the Barracuda system. The Barracuda implementation takes advantage of the structure of modern heterogeneous systems by offloading much of the race detection analysis to the host CPU, instead of performing race detection directly on the GPU device which would substantially worsen the performance of the target kernel. There are two benefits to our hybrid GPU+CPU approach. First, the host is typically underutilized during kernel execution, as it waits for the results of the kernel. Second, the host is better suited to the memory-intensive work of race detection as a modern multicore can easily have 1-2 orders of magnitude more DRAM than a modern GPU does. A kernel running under Barracuda logs all GPU memory accesses, both global and shared, to queues in GPU memory. These queues are then consumed by host-side threads which do the actual race-checking.

We describe the implementation of Barracuda in three steps. First, we describe our dynamic instrumentation framework. Then, we explain the GPU memory access logging mechanism. Finally, we describe the implementation of the host-side race detector.

5.5.1. Dynamic Instrumentation

Barracuda supports all modern versions of CUDA, including 7.5 and 8. We implemented our own binary instrumentation framework because existing frameworks, such as GPU Ocelot [33] and GPU Lynx [34], do not support CUDA SDK versions 5.0 or newer. We also considered the SASSI machine-level instrumentation framework [84] but opted against it because it is closed-source and did not support adequate hooks on synchronization instructions.

Barracuda is implemented as a shared library. The library is injected into the target process using LD_PRELOAD. It intercepts the __cudaRegisterFatBinary() function call, loads the embedded CUDA fat binary, strips out any architecture-specific binary entries, and extracts and decompresses the architecture-neutral PTX assembly code contained in the fat binary. This PTX code is then fed into the instrumentation engine which performs three operations:

- **Merging the GPU-side logging framework**. The GPU-side logging framework is written in regular CUDA. This code is compiled into PTX at build time and stored inside the Barracuda library. At runtime, the logging code is merged with the application's PTX code.
- **Unique thread id calculation**. We add PTX code to the beginning of every kernel to combine the three-dimensional block id and thread id's into a globally unique value. For

the rest of the paper we will refer to this 64-bit value as the TID. All device functions are modified to accept this TID as an additional argument so that the TID is always available for logging calls.

- **Memory and synchronization logging**. We scan the PTX source code and add logging calls to all load, store, atomic, fence, and barrier instructions. As described in Section 5.4.1, we infer high-level *acquire* and *release* operations from the PTX code. Special care is required for predicated instructions: we transform the predicated instruction into a branch and a non-predicated instruction, so that the logging call is covered by the branch. To detect intra-branch races we also add logging calls to all branch convergence points. To reduce logging overhead we avoid some repeated logging of accesses (within the same basic block) to the same memory location, as in some CPU race detection schemes [41] (in particular, we do not log an access to the address in a register if the value of the register has not been changed since the last logged access).

Once the application PTX code is instrumented, the data structures within the CUDA runtime are modified to point to the newly-generated fat binary that includes only the instrumented PTX code. We then relinquish control back to __cudaRegisterFatBinary() and our modified PTX is loaded, JIT-compiled into machine code, and loaded onto the GPU.

The BARRACUDA shared library is also responsible for initializing the GPU-side memory structures used by BARRACUDA. We reserve a configurable percentage (50% by default) of the GPU's global memory for the shared GPU-CPU queues, and invoke a kernel to initialize this region.

Special care has to be given to device resets, as these will free the memory backing BARRACUDA's queues. When BARRACUDA intercepts a cudaDeviceReset call, it delays the reset until the queues are fully drained. It also raises an internal flag so BARRACUDA is reinitialized the next time any CUDA library call is intercepted.

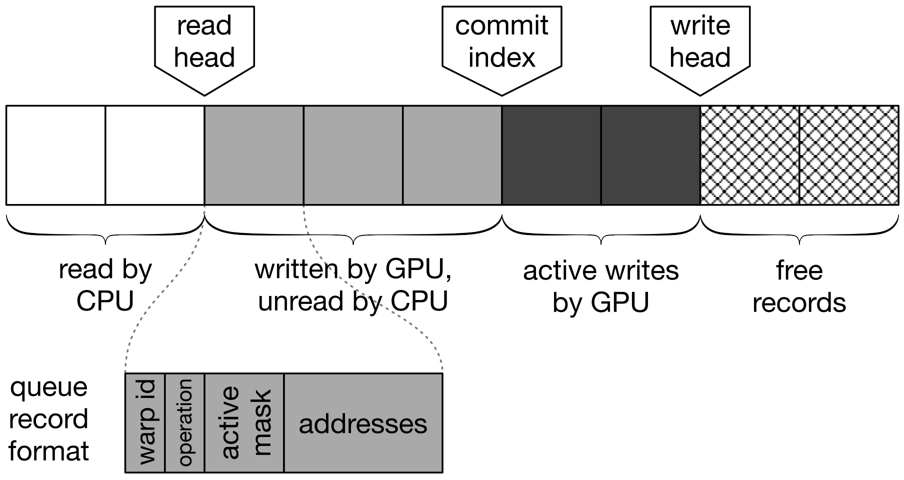


Figure 5.8: A Barracuda queue, for communicating events from the GPU to the host race detector.

5.5.2. Device-side Logging

Barracuda's GPU logging facility has been designed to be minimal and fast. The core of the GPU-side logging algorithm is a lock-free queue of fixed-size records (Figure 5.8). The queue contents are tracked via three pointers: a *write head*, a *commit index*, and a *read head*, which track the next entry available for writing by the GPU logging instrumentation, for transferring from the GPU to the host, and for reading by the host race detector, respectively. The queue uses a virtual indexing scheme with monotonically increasing indices, which are mapped to physical locations by taking their modulus with the queue size. The queue is considered full when the write head is *queue-size* entries ahead of the read head. Log records are modeled closely on the trace operations given in Section 5.4.1, except that, for efficiency, a record contains the operation for an entire warp. Each record contains fields identifying the warp, the operation, a 32-bit mask of active threads, and 32 entries for the addresses accessed by each thread in the warp (for memory operations). Records are a fixed $16 + 8 \times 32 = 272$ bytes in size.

To best take advantage of the memory architecture of the GPU we allocate multiple queues, which can achieve orders of magnitude better throughput than using a single queue. Each thread block sends events to a single queue, though multiple thread blocks may use the same queue. We found the optimal organization to be $\approx 1.1 - 1.5$ queues per SM (*i.e.*, GPU core). A significant benefit of mapping each thread block to a single queue is that locking can sometimes be avoided in the host-side race detector. For example, the host race detector uses one CPU thread per queue, so operations on shared memory (which are private to a thread block) will always be processed by the same CPU thread, avoiding the need for locking.

Logging operations on the device are performed cooperatively by all threads within a warp w. To log an operation, the first active thread within w is selected as the leader t_l. t_l reserves an empty slot in the queue, waiting for the CPU to drain queue entries if necessary. Then t_l shares the index of this empty slot with the other threads in w, and all threads record their individual memory addresses in parallel. t_l then fills in the warp ID, operation type, and active mask, and makes the completed record visible to the CPU by bumping the commit index (Figure 5.8). Logging operations use CUDA's system-level fences to ensure proper memory ordering between the GPU and the CPU.

5.5.3. Host-side Detector

We implement the BARRACUDA race detection algorithm on the host side. Each GPU queue is allocated a corresponding host thread and GPU stream. Queue draining is the mirror image of the logging algorithm, with the read head used instead of the write head. The detector processes each dequeued event according to the rules given in Section 5.4.3.

Thread VC Compression

One of BARRACUDA's key innovations is a more efficient mechanism for tracking the per-thread vector clocks (PTVCs) used to record when each thread has synchronized with each other thread in the program (the C_t state from Section 5.4.3). In a race detector for conventional multithreaded programs, these PTVCs consume O(n^2) space, where n is the number of threads in the program, but n is at most a few tens of threads in practice. With GPU programs, in contrast, kernels can easily utilize *more than one million* threads, which entails crippling space overheads. Fortunately, there is often massive redundancy among the entries in each PTVC. Accordingly, BARRACUDA employs an adaptive scheme for compressing PTVCs, mirroring the

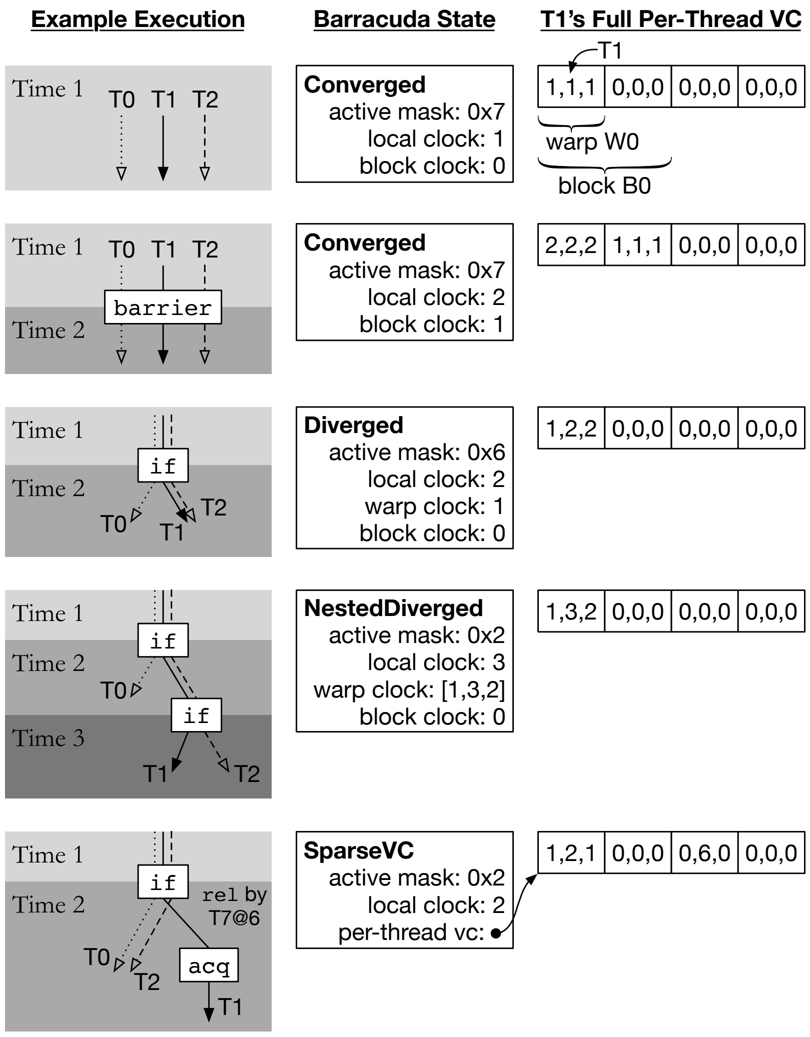


Figure 5.9: Barracuda's four per-thread VC formats. The example kernel has 3 threads per warp, 2 warps per block, and 2 blocks. Shading of the example execution indicates each thread's logical time. The right side shows the equivalent full per-thread VC that the Barracuda state implicitly represents.

GPU thread hierarchy.

In our analysis of CUDA programs, we discovered that roughly 90% of the time PTVCs have the same value for all threads external to a warp and either 1) the same value for all threads in a warp or 2) two distinct values, *e.g.*, along the two branches of an if-else statement. This creates an opportunity for space savings by storing just a few clock values for each warp. BARRACUDA's PTVC compression is lossless, and always functionally equivalent to a full vector clock.

In BARRACUDA, PTVCs are managed at warp granularity because it is often the case that threads in a warp have identical PTVCs. PTVCs can be in one of four formats, as Figure 5.9 illustrates. The simplest case is the CONVERGED format, used when all threads in a warp are executing in lockstep. Consider the PTVC for thread T1 from warp W0 (top execution of Figure 5.9). The PTVCs for T0, T1 and T2 are managed collectively at warp granularity. The *active mask* of 0x7 indicates that all 3 threads in W0 are active, and the *local clock* gives the logical time each thread in W0 has for itself. The *block clock* indicates that the threads in W0 have never synchronized with the other threads in B0. All other PTVC entries are implicitly 0, indicating that the threads in W0 have not synchronized with other threads outside their block.

The second execution in Figure 5.9 shows the impact of a block-level barrier. The block clock for W0 is 1 after the barrier, representing the fact that all threads in W0 synchronized with all other threads in the block at time 1. The threads in W0 then move on after the barrier to time 2.

The third execution illustrates the DIVERGED format, which is used to handle the common case of non-nested control flow. T0 takes one path after the if statement, and threads T1 and T2 the other path. The mask is updated to reflect the active threads (just T1 and T2) along the current path. We introduce a new *warp clock* field to track the last time the active threads in W0 synchronized with the inactive threads in W0, which was at time 1 before the if statement. Synchronization with threads outside the warp is handled via the block clock as in the CONVERGED format.

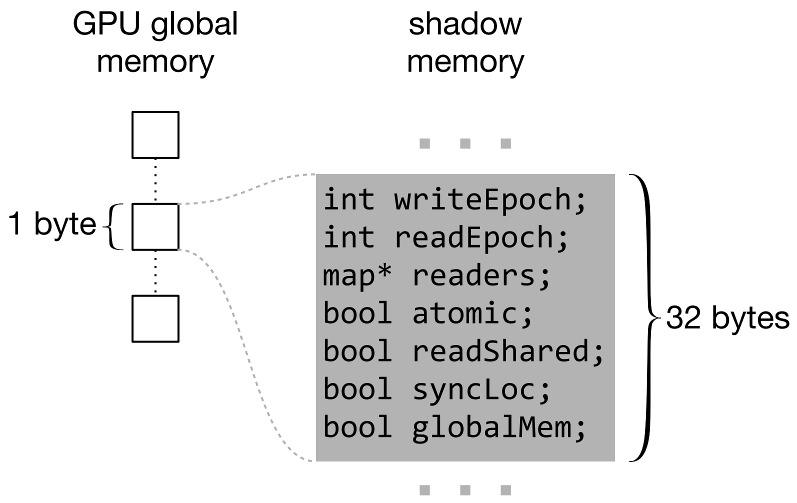
The fourth execution illustrates the NestedDiverged format, which is used to handle nested control flow. Here, the warp clock field is generalized to a vector clock to track the precise times at which the active threads synchronized with each other thread in W0. Thus, the warp clock field is a vector clock with the vector size equal to the number of threads in a warp. Due to the nested if statements, T1 is the only currently active thread, and it last synchronized with T0 at time 1 and T2 at time 2. Synchronization with threads outside the warp is handled via the block clock as in the Converged format.

The final execution illustrates the fully-general SparseVC PTVC format, which is simply an unordered map from threads to clocks. Using a map instead of a vector clock is more efficient because, typically, the entries for most threads are zero. In this example, T1 is the only thread that acquires a lock l which was previously released by thread T7 at time 6. T7 is in a completely different thread block than T1. This point-to-point synchronization between arbitrary, individual threads requires tracking clock values precisely at thread granularity.

Barracuda's PTVC management is integrated with a stack that mirrors the GPU's *reconvergence stack* [46], to reduce redundant tracking of the active mask. Each stack entry is 16 bytes and contains the fields listed in the "Barracuda State" portion of Figure 5.9. Whenever a reconvergence operation occurs, we merge the two divergent cases in the stack by joining their PTVCs (the ElseEndif rule from Figure 5.4). Barracuda checks for opportunities to use a simpler PTVC format at branches and at reconvergence, as further compression is often possible in these cases.

Barriers

Block-level barriers are the most common CUDA synchronization operation. When a barrier operation occurs, we take a block-wide join of all PTVCs (the Bar rule from Figure 5.5). We optimize this case by continually tracking the highest clock value for each block, so that at a barrier we can simply broadcast this value to each thread's PTVC.



```
int writeEpoch;
int readEpoch;
map* readers;
bool atomic;
bool readShared;
bool syncLoc;
bool globalMem;
```

Figure 5.10: Barracuda shadow memory format. The format is the same for global (shown) and shared memory locations.

Shadow Memory

The host race-detector maintains a shadow memory containing per-location race detection metadata (Figure 5.10). This metadata contains a last-write epoch and a last-read epoch (as in FastTrack), and, for locations that have been concurrently read, an unordered map from TIDs to clocks that acts as a sparse vector clock. We do not extensively optimize per-location VCs (unlike per-thread VCs) as the case of shared readers is extremely rare in all the CUDA code we examined. Per-location metadata also contains a spinlock and a set of flags for memory location attributes: whether the location was last accessed by an atomic operation (the *atomic bit* from Section 5.4.3), has been read concurrently by multiple threads, is used as a synchronization location, and is in global or shared memory. All together, per-location metadata occupies 28 bytes, but 64-bit alignment forces the object to be padded to 32 bytes. Thus, host-side memory usage is 32x that of the GPU, but CPU memory is usually much more abundant than GPU memory. Memory consumption could be substantially decreased if all GPU memory accesses are 2- or 4-byte aligned. Although most of the benchmarks we tested access memory exclusively at 4-byte granularity, Barracuda uses 1-byte granularity for generality.

We allocate shadow memory for shared and global memory differently. Shared memory consumption is small (16, 32 or 48KB per thread block in current versions of CUDA) and known at kernel launch, so we preemptively allocate shadow memory for shared memory. A kernel with 512×1024 threads each having 16KB of shared memory requires just $16384 \times 512 \times 32 = 256\text{MB}$ of CPU memory).

Global memory consumption, on the other hand, is not known at kernel launch as allocations can occur concurrently with kernel execution. Thus, for tracking accesses to global memory, we allocate shadow memory on-demand in response to a kernel's actual global memory accesses. We maintain shadow memory for the GPU's global memory using a page table where each page holds shadow memory to track 1MB of the GPU's global memory. When a global memory location is accessed by the GPU we check if it belongs to an allocated page. If not, we lock the root of the page table and allocate a new page of shadow memory.

When loads, stores and standalone atomic operations are processed, BARRACUDA begins by retrieving the appropriate shadow structures for all the addresses accessed by active threads. The code implements the BARRACUDA algorithm as described in Section 5.4.3. If a race is detected, the offending TIDs are examined to classify the race as a *divergence race*, an *intra-block race* or *inter-block race*.

A memory location x accessed with acquire and release operations is deemed a synchronization location and tracked specially. GPU code usually has few such synchronization locations, and many programs have none, so storing them in shadow memory would be wasteful. Instead they are stored in their own map (the S_x map from Section 5.4.3). For each synchronization location x we maintain a collection of VCs, representing the various times at which different thread blocks synchronized on x. Each of these per-block VCs is compressed via the scheme described above in Section 5.5.3. For each synchronization location x, the ACQGLOBAL, RELGLOBAL, ACQBLOCK, RELBLOCK rules are thus implemented via join operations between the PTVCs and the per-block VCs of x.

5.6. Evaluation

In this section we evaluate BARRACUDA along two axes: ability to detect races precisely, and performance overhead.

5.6.1. Experimental Setup

Our experimental machine is a dual-socket system with two Xeon E5-2620v4 processors, each with 8 cores running at 2.1GHz, and 128GB of RAM. The machine additionally has an Nvidia GTX Titan X GPU which uses the Maxwell architecture and has 12GB of RAM, and 3072 threads across 24 SMs running at 1GHz. The GPU is connected via PCIe x16. The machine uses Ubuntu 16.04 (Linux 4.4.0), Nvidia CUDA Toolkits 7.5 and 8, and version 367.48 of the Nvidia drivers. All benchmarks were built with Nvidia's nvcc compiler, using the flags -cudart=shared -arch=sm_35 -O.

BARRACUDA is evaluated with the following benchmarks: we use bfs, backprop, dwt2d, gaussian, hotspot, hybridsort, kmeans, lavamd, needle, nn, pathfinder and streamcluster from Rodinia version 3.1 [15]; hashtable from GPU-TM [47, 110]; bfs from SHOC [23]; dxtc and threadFenceReduction from the Nvidia CUDA SDK 7.5 samples; we also use block_radix_sort, block_reduce, block_scan, device_partition_flagged, device_reduce, device_scan, device_select_flagged, device_select_if, device_select_unique and device_sort_find_non_trivial_runs from Nvidia's CUB SDK 1.4.1 samples. Performance measurements are the average of 10 runs, with a full GPU reset between each run.

5.6.2. Concurrency Bug Suite

To evaluate the correctness of BARRACUDA, we constructed a CUDA concurrency bug suite consisting of 66 small CUDA programs that exhibit subtle data races or race-free behavior via global memory, shared memory, within and across warps and blocks, and using a variety of atomic and memory fence instructions to implement locks, whole-grid barriers and flag synchronization. BARRACUDA reports races (or the absence of a race) correctly for all 66 of our tests. Nvidia's

CUDA-racecheck [83] reports correctly on only 19 tests, sometimes reporting races where there are none (with intra-warp synchronization), missing races on global memory, and even hanging on the tests involving spinlocks.

5.6.3. Standard Benchmarks

	2	3	4	5
benchmark	static insns	total threads	global mem MB	races found
BFS	281	1,000,448	155	
Backprop	272	1,048,576	9	
DWT2D	35,385	2,304	6,644	3 global
Gaussian	246	1,048,576	124	
Hotspot	338	473,344	119	
Hybridsort	906	32,768	252	1 shared
Kmeans	384	495,616	252	
Lavamd	1,320	128,000	965	
Needle	1,006	495,616	64	
Nn	234	43,008	188	
Pathfinder	285	118,528	155	7 shared
Streamcluster	299	65,536	188	
BFS	770	1,024	68	3 global
Hashtable	193	64	103	3 global
dxtc	1,578	1,048,576	17	120 shared
ThreadFenceRed	5,037	16,384	787	12 shared
block_radix_sort	2,174	128	66	
block_reduce	2,456	1,024	70	
block_scan	4,451	128	118	
d_partition_flagged	2,834	128	66	
d_reduce	2,397	128	66	
d_scan	1,661	128	65	
d_select_flagged	2,615	128	66	
d_select_if	2,508	128	66	
d_select_unique	2,484	128	66	
d_sort_find_non_triv	16,479	128	66	

Table 5.1: The benchmarks used with Barracuda.

Table 5.1 gives more detail about each benchmark used in our evaluation. Column 2 lists the number of static PTX instructions in each program. Column 3 lists the total number of threads used within the largest kernel in each program. Four benchmarks launch more than 1 million threads to run on the GPU simultaneously, and several launch many hundred thousand. Column 4 lists the total global memory used by each benchmark, which is typically small with the exception of DWT2D. There is plenty of space in global memory for BARRACUDA to allocate its queues without impinging on the application. Finally, column 5 lists the number of races

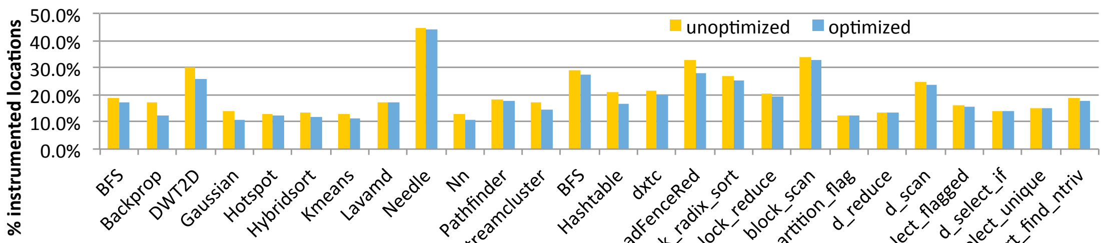


Figure 5.11: The percentage of static PTX instructions instrumented by Barracuda before (left bars) and after instrumentation pruning (right bars).

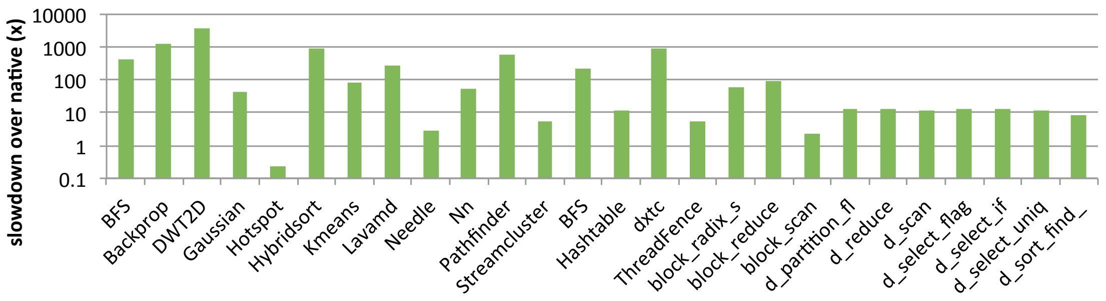


Figure 5.12: The performance overhead of Barracuda, normalized to native execution. Note the log y-axis.

found by Barracuda for each benchmark, and whether the races are in shared memory or global memory. Previous software-based race detectors for CUDA [118, 119, 83, 75] focus on shared memory, and so will not be able to detect the 9 races in global memory that Barracuda finds.

Figure 5.11 shows the fraction of the static instructions in each benchmark that are instrumented by Barracuda. Because arithmetic instructions don't require instrumentation with Barracuda, and they typically comprise the bulk of the instructions in a GPU kernel, Barracuda never instruments more than half of the instructions among our benchmarks. The blue bars in Figure 5.11 show how many fewer instructions are instrumented thanks to Barracuda's intra-basic-block logging optimizations (Section 5.5.1).

Figure 5.12 shows the performance overhead that Barracuda incurs, normalized to native execution. Barracuda's dynamic binary instrumentation approach, which maximizes compatibility with existing CUDA binaries, can incur significant performance overheads. On DWT2D, Barracuda's slowest benchmark, the overhead is 3700x, though the relative overhead for this and many other benchmarks is exacerbated by short running times: DWT2D executes natively in just 90ms. dxtc is Barracuda's slowest benchmark in absolute time and completes in 20 minutes, which is too slow for interactive use but more than fast enough for usable debugging.

hotspot with Barracuda reliably runs significantly faster than with native execution. We have traced this issue to differing JIT compilation decisions with and without Barracuda, but have not yet pinpointed the issue further.

5.6.4. Bugs Discovered

Here we describe some of the bugs we discovered with Barracuda. In the hashtable benchmark, each thread stores a value in a random key in the hashtable. Each hashtable bucket is protected by a fine-grained lock. The program uses an atomicCAS without a fence to synchronize access to each bucket. Barracuda detects two bugs: first, since there is no fence for the atomicCAS, it can be reordered with other operations that manipulate the hashtable bucket. Second, releasing the bucket lock occurs via a non-atomic write without a fence. The hashtable data structures reside in global memory, so the bug is not discoverable by tools that focus only on shared memory [118, 119, 83, 75].

Another interesting race comes from the bfs SHOC benchmark [23]. The graph data structure in bfs is stored in global memory. As multiple threads traverse the graph, they track the distance of each node from the starting node. These updates are performed without atomic operations or fences, and the writes can occur concurrently from multiple blocks. A flag is also concurrently set to 1 from multiple threads. While the CUDA documentation states that multiple writes, from threads within a warp, to the same location are serialized [81, §4.1], no such guarantees are stated for writes beyond a warp.

5.7. Conclusion

In this chapter, we have demonstrated Barracuda, a precise dynamic data race detector for CUDA GPU programs. Barracuda maintain the per-thread metadata required by a vectorclock based algorithm using highly compressed data structures, making the algorithm practical to run on GPU programs, which often have millions of threads.

CHAPTER 6

More Efficient Data Race Detection for GPU

In Chapter 5, we have presented the BARRACUDA system, which allows GPU programs to be analyzed for data races. However, as shown in the evaluation results, BARRACUDA can high incur runtime overhead, limiting its application. Although the loss-less compression scheme of PTVCs enables BARRACUDA to apply a CPU-oriented vector clock based algorithm to GPU programs, this algorithm is unaware of some unique characteristics of the GPU programming and execution model. In particular, synchronization in GPU programs is much more structured than general C/C++ programs that run on CPU, which may enable a lighter-weight scheme to check for conflicting accesses. Also, GPU programs are often optimized for memory efficiency, which makes some unique memory access patterns common in these programs. Knowledge of such access patterns can potentially enable a dynamic race detector to maintain its metadata more efficiently.

This chapter presents CURD, a system that applies the intuitions mentioned above to the design of a new dynamic race detection algorithm.

6.1. The Curd System

With this work, we seek a CUDA race detector that provides both precision and acceptable performance. We leverage the open-source BARRACUDA design as a starting point, but find that its focus on supporting a wide range of synchronization primitives makes it a poor match for common-case GPU code, which generally uses just simple barriers to synchronize. We describe a new race detector, CURD, that is highly optimized for this common case but retains the ability to fall back to the BARRACUDA algorithm to preserve precision. We describe a simple static analysis that can decide, before a kernel runs, which race detection algorithm to use. This hybrid approach provides a significant performance improvement for many programs without compromising precision.

CURD's central approach to race detection is to record the read and write sets of synchronization-free regions (SFRs) of code, and then perform set intersections to detect locations for which a data race occurred, *i.e.*, the location was accessed by two distinct threads, at least one of which wrote to the location. While using read/write-set tracking for race detection has been explored before [78, 57, 119, 118], CURD innovates in several key respects. First, CURD utilizes an efficient set representation that is attuned to the typical access patterns of GPU kernels, ensuring that kernels that are well-optimized for the GPU memory hierarchy have very compact read/write set representations with CURD. Second, CURD aggressively exploits the GPU's parallelism in its internal operations. Third, CURD's hybrid organization allows it to avoid "overpaying" for race detection on programs that eschew complicated synchronization.

When a programmer decides to instrument a program with CURD to perform race detection, a simple static analysis runs to examine which race detection algorithm to use. This analysis has been implemented in LLVM and examines the bytecodes of a GPU kernel, looking for atomic and fence instructions. If one of these instructions is present, CURD uses the BARRACUDA algorithm [30] to preserve precision in the face of low-level synchronization. Otherwise, CURD uses an alternative algorithm that is optimized for common-case GPU code that synchronizes only via barriers. In the remainder of this paper, we focus on this alternative algorithm, which we refer to as CURD for simplicity.

Figure 6.1 shows CURD's high-level operation. To detect intra-block races, CURD maintains two sets, TRS(t) and TWS(t), for each thread t to track the locations read and written by t, respectively. CURD tracks accesses to each memory space (shared or global) separately, so there are in fact two read sets and two write sets for each thread. For simplicity, we do not distinguish between these sets in the discussion below except in the special cases (like barriers) where they are treated differently. To detect inter-block races on global memory, for each block b, BRS(b) and BWS(b) sets are maintained to save the aggregated sets of reads & writes performed by the block b.

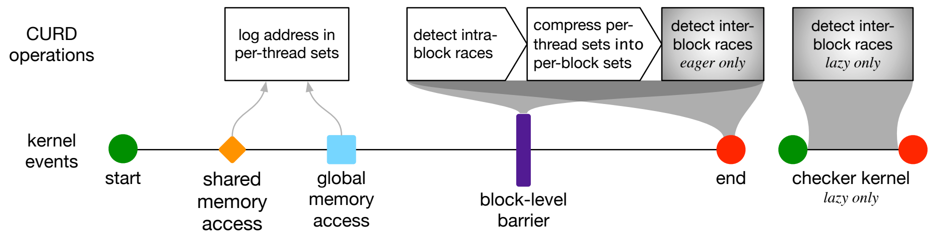

CURD operations
log address in per-thread sets
detect intra-block races
compress per-thread sets into per-block sets
detect inter-block races *eager only*
detect inter-block races *lazy only*
kernel events
start
shared memory access
global memory access
block-level barrier
end
checker kernel *lazy only*

Figure 6.1: High-level operation of Curd in response to various kernel events.

6.1.1. Access Logging

When a thread t reads/writes a memory location in shared or global memory, CURD records the memory region being accessed in TRS(t)/TWS(t). Each memory region is a pair (a, s), where a is an address within a particular memory address space (either shared or global memory), and s is an unsigned integer. The pair represents a region of memory starting at address a and extending for the next s contiguous bytes. In addition, CURD can track the source code location (file and line) for each region's most recent read/write.

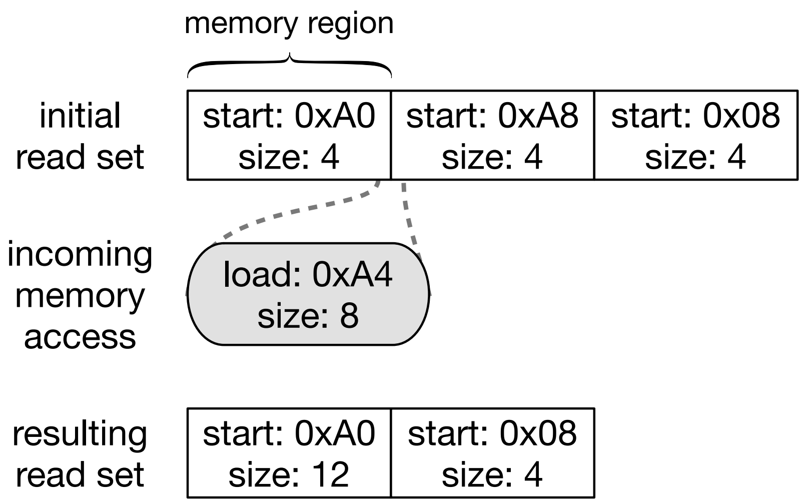


Figure 6.2: Curd's read and write sets are implemented as an array of disjoint, non-adjacent memory regions.

Each read or write set is a collection of regions that obey the following invariant: memory regions stored inside these sets are always disjoint and non-adjacent. This invariant is maintained on insertion operations.[1] If a region being inserted into a set overlaps with, or is adjacent to, an

[1]We will actually relax this invariant later for efficiency, but it is convenient for expository purposes to uphold

existing region, recursive merging occurs to keep the set elements disjoint and non-adjacent. GPU DRAM and caches provides maximum bandwidth when threads access adjacent data. Thus, by constantly compressing adjacent memory regions, the space consumption of CURD's read/write sets is typically quite small for well-optimized CUDA programs.

A per-thread read or write set is implemented as an array of structs, where each struct represents a memory region. Memory regions within the array are not ordered, so a region containing a particular address may lie anywhere within the array. The memory regions in use are densely packed at the beginning of the array, allowing for simple bump allocation of new regions and efficient clearing of the array.

Figure 6.2 shows an example of adding a load access into an existing read set. The initial read set contains three memory regions. The incoming load is adjacent to the first region, and overlaps with the second. Thus, the first and second regions can be merged into a larger region in the resulting read set.

Inserting a new memory region r into a set is performed as follows. r is checked against each memory region in the set to see whether r can be merged with an existing region in the set. If r is disjoint and non-adjacent to all existing regions, r is appended to the array of regions. If r can be merged with another region r' (*i.e.* r is adjacent to or overlaps with r'), we merge r with r' and then recursively explore the remaining regions of the set for further merging opportunities that may have opened up. After merging regions, source information is updated to reflect this most recent access. Note that regions in the already-explored prefix of the set cannot be candidates for further merging. We know that such "prefix regions" were already disjoint and non-adjacent with respect to both r' (because of the set's invariant) and r (because we did not choose to merge r earlier), so prefix regions must also be disjoint and non-adjacent with respect to $r \cup r'$.

Note that when two regions merge, the source information of one of the merging regions is overwritten. If a race is detected on a memory region r, CURD reports the source-level information

it for now.

of the most recent access to r, which may differ from the actual access involved in the race (though we found this problem rare in practice, see paragraph 6.2). However, on a race CURD additionally reports the memory addresses on which the race was detected, and these addresses are always precisely those involved in a race.

In CUDA programs, to fully exploit the available memory bandwidth, threads within a warp need to access adjacent memory locations. This motivates a "striped" organization of the per-thread sets as shown in Figure 6.3. The first elements of each per-thread set, for each thread t within

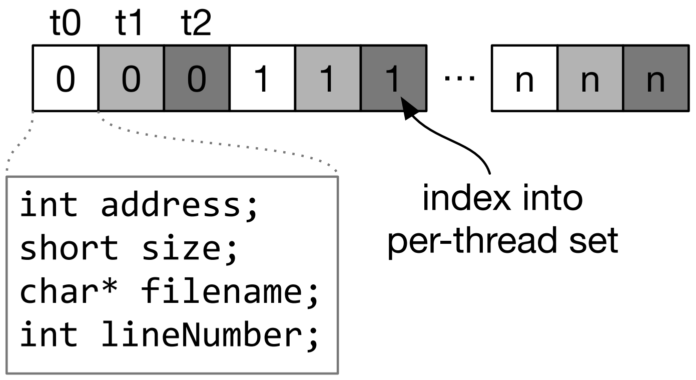


Figure 6.3: Curd organizes per-thread read/write sets to maximize global memory coalescing by interleaving entries from each thread. In the example shown, there are three threads per block.

a block b, appear first. They are followed by the second elements for each thread, the third elements for each thread, and so on.

6.1.2. Intra-block Race Detection

When the threads in a block reach a barrier operation, we can perform intra-block race detection using the TRS(t) and TWS(t) for each thread t in a block B. We write $a \cap b$ to denote the intersection of two sets a and b. If two sets have a non-empty intersection, we say that the sets **overlap**. For each thread in B, we logically intersect read and write sets to detect read-write races, and then we intersect write sets with one another to detect write-write races. In a race-free CUDA program, these intersections should all be empty. Once intra-block race detection is finished, each thread t can safely discard its read and write sets for shared memory, as no

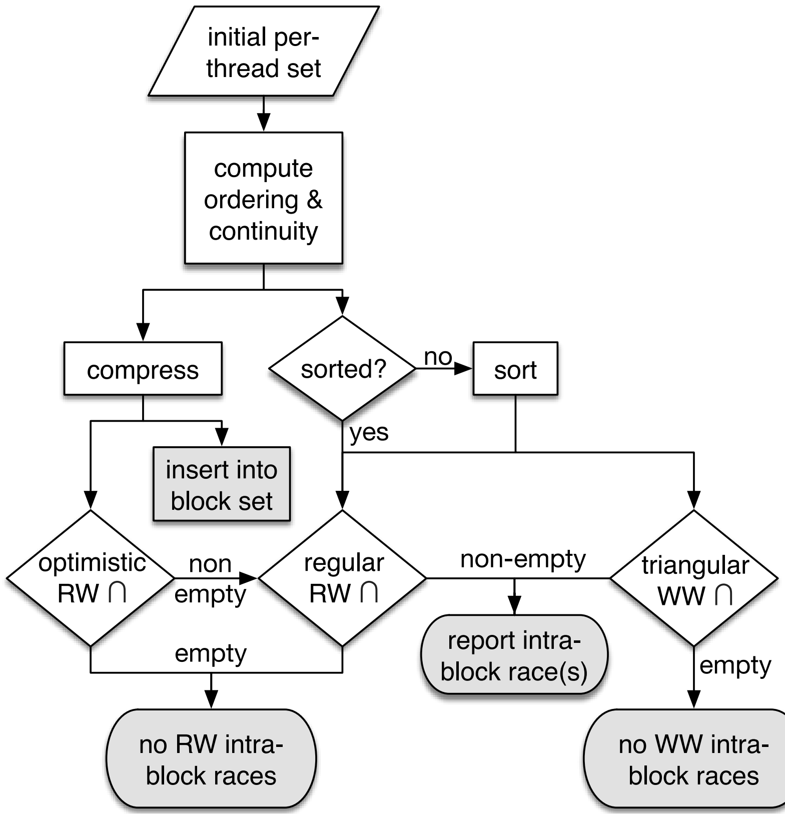


Figure 6.4: The steps involved in intra-block race detection and adding per-thread sets to per-block sets.

inter-block races are possible in shared memory and any future shared memory accesses from the block will be well-ordered with t's current accesses due to the barrier. Global memory TRS(t)/TWS(t) need to be added to BRS(t)/BWS(t), respectively, and then they, too, are cleared.

While intra-block race detection and adding to the block sets are logically two separate steps, their implementation is integrated for efficiency. Figure 6.4 gives an overview. We first discuss adding to the block sets, then RW intra-block checks, and then WW intra-block checks.

Per-thread Set Compression

Per-thread sets are compressed before being added to the per-block sets to reduce time and space overheads. TRS(t) and TWS(t) are treated identically, so we refer just to TRS(t) in this section.

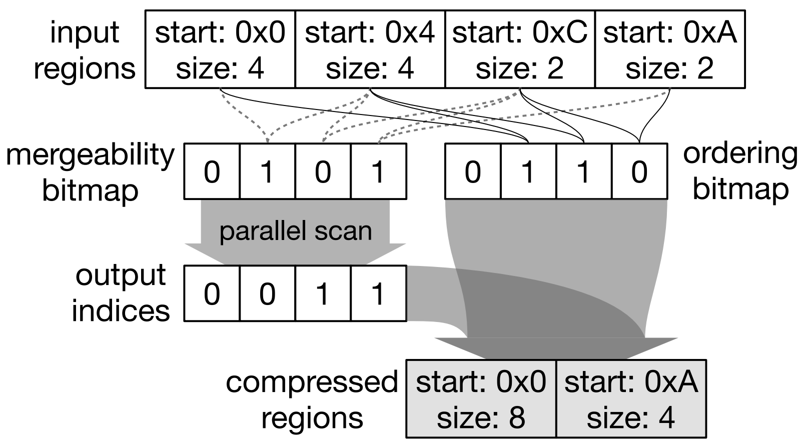


Figure 6.5: An example of how per-thread sets are compressed in parallel.

Compression produces a compressed read set CRS(b) from the read sets for all threads in the block b, *i.e.* $CRS(b) = \forall t \in b \cup TRS(t)$. CRS($b$) is a new set, so the original TRS(t) are preserved, which is important later for intra-block race detection. Compression operates directly on the striped TRS(t) layout (Figure 6.3), and runs in parallel via the following algorithm (Figure 6.5). First, an *ordering bitmap* is computed, identifying whether each region is located in memory after its left neighbor. Next, a *mergeability bitmap* is computed, identifying whether each region is adjacent to or overlaps with its left neighbor. Neighboring regions are often mergeable because, for well-optimized kernels, adjacent threads (more precisely, threads within the same warp) often access adjacent memory regions to ensure coalesced global memory accesses. Next, a parallel scan of the mergeability bitmap computes the indices to use in the output set CRS(b). Finally, using the output indices and the ordering bitmap, we identify the beginning and end of each mergeable sequence of input regions, and compress them into a single output region.

Once CRS(b) is computed, it is inserted into BRS(b). This insertion does not exhaustively search BRS(b) for potential merges, instead examining just k regions. We found that bounded search yielded better performance in practice, even though it makes BRS(b) larger. While bounded search also violates the disjoint and non-adjacent property for region sets, it does not impact correctness because set intersections still operate on all regions in a set.

Checking for Read-Write Intra-Block Races

A sufficient, but not necessary, condition for the absence of intra-block read-write races within a block b is $CRS(b) \cap CWS(b) = \varnothing$. As the compressed sets are small, this check is much faster than checking for intersections among TRS(t) and TWS(t) for each pair of threads. Moreover, the ordering bitmap from Section 6.1.2 and bit-twiddling instructions quickly reveal whether CRS(b) and CWS(b) are sorted, which then permits linear-time set intersection.

If $CRS(b) \cap CWS(b) \neq \varnothing$, it may be because of a true intra-block race or because a thread t both read and wrote some location x. Thus, we must go back and perform intersections among the individual TRS(t) and TWS(t). To make these intersections faster, we first sort any unsorted TRS(t) and TWS(t) sets in-place. With n threads in a block, this requires $O(n^2)$ intersections, each of which can run in linear time. However, the optimistic RW check is effective at reducing the need for this more expensive check.

Checking for Write-Write Intra-Block Races

To check for write-write races in a SPMD fashion, each write set is compared with every other write set. However, half of these intersections are unnecessary as set intersection is commutative. Considering a matrix with each row and column representing a write set, the strict upper triangular matrix represents the necessary intersections. Performing only these intersections improves performance as both memory traffic and intersections are reduced by half. This "triangular check" design also minimizes branch divergence [60] in the code, which saps performance on GPU SIMD hardware. Since the TWS(t) are sorted from read-write checking, individual set intersections can run in linear time.

6.1.3. Inter-Block Race Detection

For detection of inter-block races, an important design decision is when to do the checks. The CURD design explores two options: a *lazy* scheme (CURD-Lazy) which defers inter-block race detection until the kernel terminates, and an *eager* scheme (CURD-Eager) which performs inter-block race detection at each barrier. Note that intra-block race detection always occurs eagerly, at each barrier. Both lazy and eager inter-block race detection are fully precise, in the sense that no data race on location x will ever be missed, nor will a false race ever be reported. Both schemes report the precise addresses involved in any data races. The lazy approach has lower implementation complexity and can be faster in some circumstances, but has less-timely detection because races are not reported until the end of the kernel when source information may have been overwritten during previous merges. In contrast, performing inter-block checks eagerly allows races to be caught earlier, potentially giving users more accurate source-level information about the race.

We discuss the lazy scheme first, as it is conceptually simpler than the eager detection scheme. The inter-block detection scheme also affects how the results of per-thread set compression (Section 6.1.2) are used, as we explain below.

Curd-Lazy

In the lazy scheme, inter-block race checks are not conducted until the target kernel terminates. As the kernel executes, at each barrier, a block b unions all of its per-thread sets with each other (Section 6.1.2) *and* with a single cumulative BRS(b) (or BWS(b)) as shown in the top of Figure 6.6. BRS(b) and BWS(b) are implemented exactly as the per-thread sets are, maintaining disjointedness and non-adjacency among all constituent regions. The location A, which is accessed both before and after a barrier in Figure 6.6, is represented only once in the final per-block set.

Checks for inter-block races are done in a separate checker kernel launched after the termination of the target kernel. After the checker kernel terminates, global memory used for block read/write sets can be freed. Similar to intra-block race checks, in the lazy scheme, inter-block races are detected in a read-write pass and a write-write pass for each block b in the grid G.

Each inter-block check involves $O(B^2)$ intersection operations of block read/write sets in total. To parallelize these operations, each block is responsible for checking its own read/write set with the write set of all other blocks, involving $O(B)$ intersections. Within each block, the operations

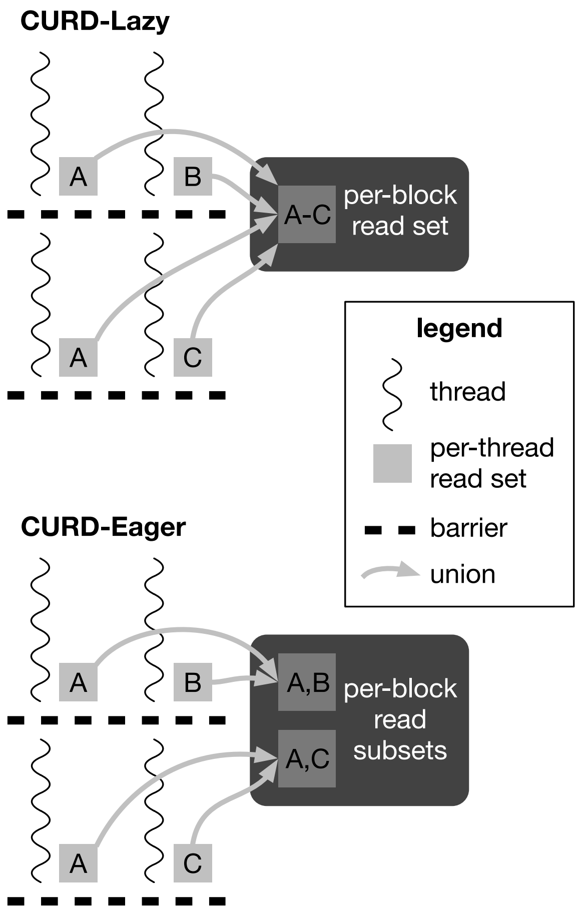


Figure 6.6: Curd combines per-thread sets into per-block sets differently in the lazy (top) versus eager (bottom) schemes. The two threads shown here are part of the same block. Write sets are elided for simplicity but are handled analogously.

are distributed evenly among N threads, with each thread performing $O(B/N)$ intersections.

One benefit of the lazy strategy is a simplified implementation. As all block sets are read-only after the target kernel terminates, no synchronization is necessary inside the inter-block checker kernel.

Eager Scheme

To detect inter-block races before kernel termination, CURD also offers an eager detection strategy. In this scheme, checks for inter-block races are performed at the end of each synchronization-free region.

The simplest version of the eager scheme would intersect all the per-block sets at each barrier, exactly as the lazy scheme does. However, this would entail a significant number of redundant comparisons as the per-block sets change only partially after each barrier. Thus, we adopt a more sophisticated incremental detection scheme that reduces redundancy by handling the updates to a per-block set separately.

A key challenge in the design of eager inter-block detection is coordinating concurrent access to per-block sets across blocks. While all threads in b_0 are at a barrier when b_0 begins inter-block detection, another block b_i may be actively executing and as a result $BRS(b_i)$ and $BWS(b_i)$ may be changing. To avoid the need for complex concurrency control, BRS(b_0) is organized as a *list* of read subsets $BRS(b_0)[0] \ldots BRS(b_0)[n]$ (and BWS(b_0) similarly). Within each read subset, memory regions are disjoint and non-adjacent. However, there are no such guarantees across subsets. At each barrier, b_0 compresses its threads' read sets into a compressed read subset (Section 6.1.2) and appends this subset to BRS(b_0) as shown in the bottom part of Figure 6.6. The same memory location A is accessed before and after a barrier so A is stored twice, within each subset of the per-block set.

Once a read subset is appended to BRS(b_0), it becomes immutable. Thus, when a block b_i wants to check against BRS(b_0), b_i records the current number of subsets in BRS(b_0) and intersects with them. b_0 may concurrently append additional subsets, but these will be checked at b_i's next barrier (or at the end of the kernel). A per-block spinlock is used to protect the size of each block's BRS(b) and BWS(b) sets, to ensure that remote threads see a consistent view of the set during their checks. Performing intersections with a read subset does not require synchronization, however, as these subsets are immutable.

To avoid redundant intersections, when b_0 reaches a barrier we partition BRS(b_0) in two: a prefix BRS(b_0)$[0 : -1]$ of read subsets from before the current SFR began, and a suffix BRS(b_0)$[-1]$ containing just the read subset from the current SFR.[2] BWS(b_0) can be split similarly. BRS(b_i)

[2]We adopt Python's array slicing syntax for identifying sub-lists of per-block sets. $[a : b]$ indicates the slice from index a up to, but not including, index b. The special index -1 indicates the last index of a list. Thus, the slice $[0 : -1]$ contains all elements except the last element in the list. A slice may omit the second index, *e.g.*, $[a :]$, in which case the slice begins at index a and continues through the end of the list.

for a remote block b_i can be partitioned at some index k, where BRS(b_i)$[0 : k]$ is a prefix of b_i's read subsets up to but not including subset k, and BRS(b_i)$[k :]$ is a suffix of the read subsets from index k onward through the end of the list. After performing an inter-block check, a block b_0 remembers the prefix BRS(b_i)$[0 : k]$ of read subsets that have been checked so as to avoid redundant intersections with these subsets in the future. We will explain how k is tracked for each block shortly, but first we illustrate how eager inter-block race checks work assuming k is already known.

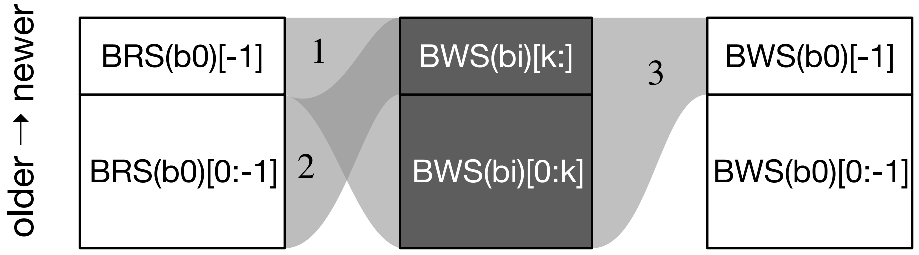


Figure 6.7: Curd-Eager inter-block race checks involve three set intersections between each distinct pair of blocks. This example illustrates the checks between blocks b_0 and b_i.

When b_0 reaches a barrier, the race checking required between b_0 and some other block b_i is shown in Figure 6.7. The checks proceed in three steps. **Step 1:** We intersect b_0's current read subset with all writes from b_i. **Step 2:** We intersect all of b_0's previous read subsets with the suffix write subsets from b_i, eliding $BRS(b_0)[-1] \cap BWS(b_i)[k :]$ which was performed in Step 1. **Step 3:** We intersect b_0's current writes with all writes from b_i. Note that $BRS(b_0)[0 : -1] \cap BWS(b_i)[0 : k]$, and $BWS(b_0)[0 : -1] \cap BWS(b_i)[0 : k]$, have been performed by b_0 during some previous barrier. At b_0's first barrier, $BRS(b_0)[0 : -1]$ and $BWS(b_0)[0 : -1]$ are empty so no intersection is necessary. The intersection $BWS(b_i)[k :] \cap BWS(b_0)[0 : -1]$ will be performed by b_i at its next barrier. We use multiple threads within a block to accelerate these set intersections, by having each thread read a different element of, say, $BWS(b_i)$ in parallel and compare it with each other element of $BRS(b_0)[-1]$.

To determine the pivot k which divides subsets we have checked already from those we have not yet checked, each block maintains an array alreadyChecked with an element for each other block in the grid. These elements contain the appropriate k value for each other block. Because read subsets are immutable once appended to some BRS(b), the length of BRS(b) is non-decreasing over time and thus the k values are as well. alreadyChecked can also be thought of as a kind of vector clock, tracking the time at which a block last coordinated with each other block.

6.2. Implementation

Like many other compiler-based race detectors (*e.g.*, LLVM's ThreadSanitizer race detector for CPU programs [103]), CURD consists of two major parts: an instrumentation pass and a detection library. CURD uses LLVM to instrument memory accesses and barriers in kernel code, as well as kernel launches in host code. These locations are instrumented with calls into the CURD runtime library, which is linked together with the application.

Shadow Memory management At present, CURD allocates its shadow memory on the host side, via cudaMalloc(). The main reason why the allocation was not done dynamically on the device is that CURD needs a global memory region visible to all blocks, and across kernels. It is simpler to set this up before the kernel launch to ensure that the memory region is communicated to all threads and blocks. An additional reason to avoid device-side allocation is that we need a handle to the shadow memory on the host side, and memory dynamically allocated on the device is not accessible to host code. Besides the shadow memory allocated in global memory, several read-only configuration parameters which CURD reads during online race detection (*e.g.*, the sizes of shared memory per block and total global memory), are copied into constant memory.

It is non-trivial to decide the size of shadow memory to be allocated. To keep things simple, the current version of CURD allocates each data structure (*e.g.*, BRS(b)/BWS(b), TRS(t)/TWS(t)) with a fixed size, which can be configured at runtime via environment variables. We will investigate more sophisticated ways of preallocating the shadow memory in future work. After a kernel terminates, all detected races are reported to the user, after which the shadow memory allocated is freed.

Source-level information To provide useful information about the location of data races, CURD stores the source file name and line number with each memory region in the thread and block sets. Since only one piece of the source information can be kept per region, region merges cause information to be lost about one of the regions involved in the merge. To track whether such information loss about any source locations has occurred, each memory region structure maintains a *complete* bit that indicates whether the stored source level information is complete. When a memory region is initially constructed, this bit is set. When two regions are merged into a single region, the bit of the resulting region is set if and only if 1) the bit is set in both merging regions and 2) the source-level information stored in the two regions is identical. When a race is detected when comparing two regions, CURD reports to the user whether it has information about all the source file locations that may be involved in the race, by inspecting the *complete* bits.

We verified the races found on the benchmarks in our evaluation of CURD (Section 6.3), and the source-level information for all reported races are complete. This is intuitive as 1) CUDA has a SPMD programming model, where many threads execute the same program statement and 2) mergeable regions are often accessed at the same program statement, as these accesses are often to adjacent global memory regions to permit coalescing.

Reducing impact on occupancy One potential side effect of instrumenting CUDA programs is that the instrumented kernel may have reduced achievable occupancy on the GPU, as calls to functions in CURD may increase the maximum number of registers and amount of shared memory a thread may use. Reduced occupancy can slow the instrumented program. To minimize this effect, functions in the CURD library avoid the use of shared memory when possible, and set a limit for the maximum number of registers when compiling the CURD library.

Pruning instrumentation to local and constant memory In addition to shared and global memory, CUDA programs also frequently use another two address spaces: local and constant memory. We found in the development that accesses to local and constant memory can dominate all accesses on some programs (e.g. heartwall from Rodinia), so we develop a data-flow analysis pass that tries to statically indentify local and constant memory pointers, to avoid unnecessarily instrumenting these memory locations.

Caching Thread-Set Insertions Recall that when the program accesses a memory region r, CURD first searches for any mergeable region in the corresponding thread set before inserting r into the set. Typically, programs exhibit temporal and spatial locality when accessing their data, so the efficiency of the lookups into the thread sets can be improved if each search starts from the most recently added/merged region. Therefore, we optimized the thread-set insertion routine such that 1) when a merging of memory regions occurs, move the merged region to the last position of the thread-set and 2) each search in a thread set starts from the last element of the thread-set and moves forwards and 3) only a bounded number of elements are examined in each search. These modifications make thread-set insertion faster because the newly merged/inserted region is more likely to be reused in subsequent insertions and the insertion procedure can return within a bounded number of steps, even if no mergeable region exists in the thread sets.

Note that since insertions into the thread sets now only examine a bounded number of elements, it is possible for a thread set to have overlapping/adjacent elements. This does not hurt soundness, but might make the thread sets larger than necessary and potentially introduces more work during intra-block checks and compression into block sets. We found in our experiments that the benefit of cached set insertions outweighs the cost of the potential extra work, as the cached thread-set insertions speed up all benchmarks used in our evaluation.

Metadata Reuse Across Kernel Launches Many GPU programs launch a large number of kernels, each of which does a relatively small amount of work. The Caffe machine learning framework is an excellent example of this design as it uses frequent kernel launches (and terminations) as a form of global synchronization in lieu of atomics. CURD incurs non-trivial startup costs to allocate and initialize its metadata, which would normally be required before every kernel launch. As an optimization, we allow CURD's metadata (which resides in global memory) to persist across kernel launches, avoiding constant deallocation and reallocation. This change improves the performance of Caffe by 7x.

6.3. Evaluation of Curd

6.3.1. Experimental Setup

To evaluate how effective and efficient CURD is in terms of detecting global and shared memory races in GPU programs, we test CURD on a wide range of 53 benchmarks, as shown in Table 6.1. We include programs from the NVIDIA CUDA SDK version 7.5 samples (indicated by an N after the benchmark name in Table 6.1), Rodinia 3.1 (R) [15], NVIDIA's CUB SDK 1.4.1 samples (C), Parboil 2.5 (P) [108], Gunrock 0.4 (G) [111], and Caffe, a widely-used machine learning framework, evaluated by running the AlexNet and OverFeat models. Of these benchmarks, only threadFenceReduction uses low-level atomics and fences for synchronization, necessitating the use of the BARRACUDA algorithm to maintain precision. The remaining 52 benchmarks run with the CURD algorithm described in Section 6.1.

Our experimental machine is a dual-socket system with two Xeon E5-2620v4 processors, each with 8 cores running at 2.1GHz, and 128GB of RAM. The machine additionally has an Nvidia GTX Titan X GPU which uses the Maxwell architecture and has 12GB of RAM, and 3072 threads across 24 SMs running at 1GHz. The GPU is connected via PCIe 3.0 x16. The machine uses Ubuntu 16.04 (Linux 4.4.0), Nvidia CUDA Toolkits 8, and version 367.48 of the Nvidia drivers. All benchmarks were built with Nvidia's nvcc compiler included in CUDA 8.0, using the flags -cudart=shared -arch=sm_50 -O3. During compilation, the NVVM IR generated by nvcc was intercepted when a special function call is done by the libNVVM library, which enables us to invoke our instrumentation pass developed based on LLVM 3.4 (which is compatible with the NVVM IR). After that, the instrumented IR is sent back to the toolchain of nvcc. This way we can utilize the handy support for separate compilation and linking of CUDA files, which is not yet supported by LLVM. Performance results shown are the geomean of 5 runs.

6.3.2. Performance

Across all workloads, CURD-Eager incurs an average slowdown of 2.89x, whereas CURD-Lazy incurs 2.88x slowdown over native execution. CURD-Eager's overhead ranges from 1.1x (de-

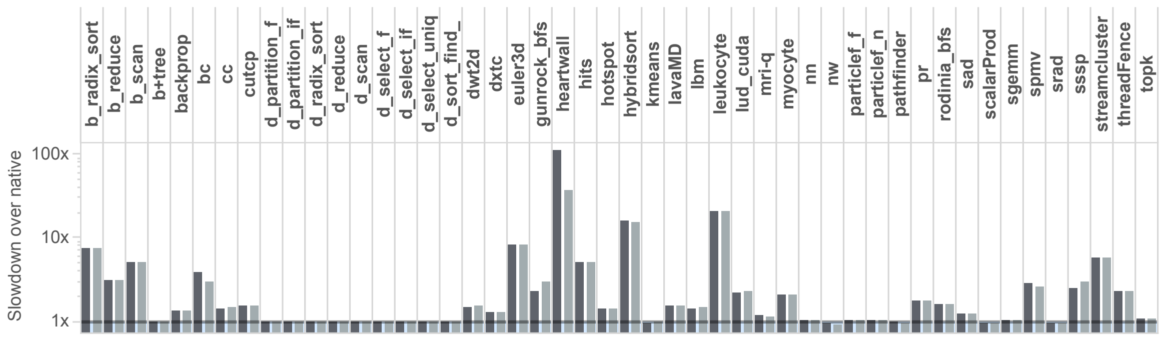


Figure 6.8: Slowdown over native execution (lower is better) for Curd-Lazy (light bars) and Curd-Eager (dark bars). Note the log scale for the y-axis.

vice_scan) to 111x (heartwall), while CURD-Lazy slows the programs by 1.1x (device_scan) to 37.9x (heartwall). heartwall is something of an outlier for CURD, as the next slowest benchmark (leukocyte) incurs just a 23.9x slowdown with CURD-Lazy and 24.8x with CURD-Eager. We investigate heartwall's performance in more detail in Section 6.3.3. Of particular note is CURD's performance on the Caffe machine learning framework, where CURD incurs an acceptable overhead of 14x for this important class of workload.

Overall, CURD-Lazy and CURD-Eager perform similarly but CURD-Lazy has a noticeable performance edge on several programs. We envision programmers first running with CURD-Lazy to quickly discover if a program has any races, and then, if necessary, using CURD-Eager to get more precise debugging information.

As the BARRACUDA system is the most related work to CURD, we evaluate BARRACUDA and CURD under the same environment. However, BARRACUDA only succeeds in running 24 programs among all the benchmarks we evaluate; for other benchmarks, it fails either due to runtime error or insufficient memory. Figure 6.9 shows CURD's speedup over BARRACUDA. Comparing their geomeans, CURD outperforms BARRACUDA by 17.4x. CURD sees a maximum speedup of 1435x on DWT2D. On 21 programs, CURD runs faster than BARRACUDA, with a minimum 1.8x speedup (nw). BARRACUDA is 6.4x faster than CURD on hotspot, where BARRACUDA consistently outperforms even native execution due to BARRACUDA's inadvertent effects on the PTX JIT compiler.

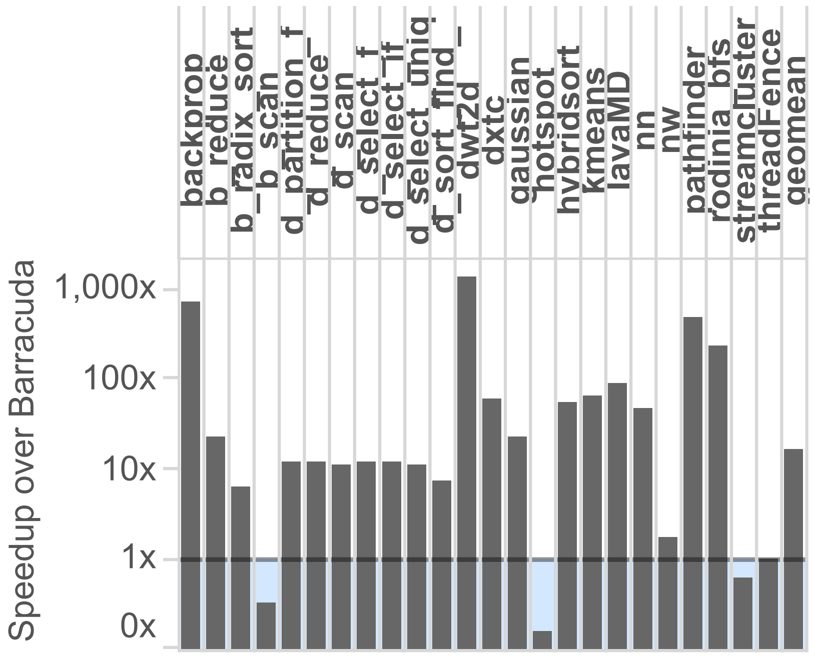


Figure 6.9: Curd-Lazy's speedup over Barracuda (higher is better). Note the log scale on the y-axis.

To put CURD's performance in an appropriate context, we also run Nvidia's commercial race detector CUDA-Racecheck (from CUDA SDK 8) and measure its performance on our benchmarks. Figure 6.10 shows CURD's speedup over CUDA-Racecheck. Since CUDA-Racecheck only checks for races on shared memory, we exclude programs that do not use shared memory, and compare the performance on the 35 benchmarks that use shared memory. For these benchmarks, CURD-Lazy is on average 2.1x faster than CUDA-Racecheck, with a maximum 461x speedup on dxtc. CURD runs slower than CUDA-Racecheck on block_reduce, bc, cc, dwt2d, pr, salsa, sssp.

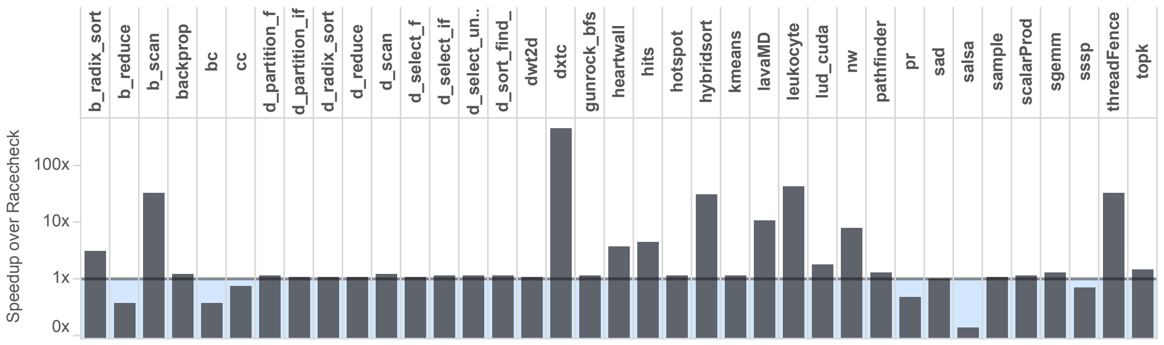


Figure 6.10: Curd-Lazy's speedup over CUDA-Racecheck (higher is better). Note the log scale on the y-axis.

On these benchmarks, CURD typically spends a significant amount of time in tracking races on global memory (see Figure 6.11), while CUDA-Racecheck ignores global memory entirely. Some of CUDA-Racecheck's overhead is likely due to its use of dynamic binary instrumentation instead of CURD's compiler-based approach. However, CURD currently employs essentially no static analysis and focuses on algorithmic optimizations instead, which are likely quite applicable to the CUDA-Racecheck implementation as well.

6.3.3. Additional Characterization

In this section, we evaluate some additional aspects of CURD's performance. First, Figure 6.11 shows the breakdown of CURD's overhead from four sources: 1) instrumentation; 2) shared memory checking; 3) intra-block global memory checking; and 4) inter-block global memory checking. We measure the cost of each component by comparing performance after disabling each component (except instrumentation, which is always required). We excluded programs where CURD incurs negligible overhead, as the breakdown of overhead is difficult to measure using this method on such programs.

Table 6.1 shows detailed characterization data for our benchmarks. Column 2 shows the lines of CUDA code in each benchmark. Column 3 shows the maximum number of threads for each program, which occasionally exceeds 1M and is reliably in the hundreds of thousands. Column 4 shows the maximum number of entries ever found in a block set or thread set accordingly; if only one number is shown, the max block set and thread set sizes are the same. These sets are typically just tens of regions, and never more than a few thousand. Column 5 shows the percentage of all per-thread sets that were naturally sorted, of all the sets encountered across barriers and kernel exits. The vast majority naturally appear in sorted order due to the predominant GPU memory access patterns. Column 6 shows the "high-water mark" amount of shadow memory used by CURD, which is typically small but occasionally reaches into the GBs.

The instrumentation overhead measures the cost of instrumenting loads and stores with an empty callback. On heartwall, CURD's slowest benchmark, this is 84% of its total overhead. Shared

Table 6.1: Details of each benchmark and its behavior with Curd-Lazy.

	2	3	4	5	6	7
program *Suite*	CUDA LoC	#max threads	max BS/TS	sorted PTS %	SMem (MB)	races
b+tree R	874	2.6M	33	71	832	
backprop R	337	1M	52	80	375	
bc G	885	98K	117 / 12	50	36	1s
bfs R	349	66K	1 / 11	68	294	6g
bfs G	855	1M	23 / 14	100	427	2s
block_radix_sort C	327	128	2 / 21	99	1	1s
block_reduce C	294	1K	2 / 64	100	2	
block_scan C	348	1K	29 / 34	100	4	1s
caffe	6856	10K	4110 / 121	98	36	
cc G	678	123K	324 / 8	100	30	1s,1g
cutcp P	1280	15K	32 / 241	92	114	2s
dev_part_flagged C	246	128	34 / 10	77	¡1	
dev_part_if C	257	128	34 / 10	67	¡1	
dev_radix_sort C	226	128	2 / 4	99	¡1	
dev_reduce C	193	128	1	100	¡1	
dev_scan C	199	128	34 / 12	97	¡1	1s
dev_sel_flagged C	246	128	34 / 3	90	¡1	
dev_sel_if C	255	128	34 / 3	88	¡1	
dev_sel_unique C	234	128	34 / 3	90	¡1	
dev_sort_fnt_runs C	384	128	23	95	¡1	
dwt2d R	2559	37K	544	100	288	
dxtc N	820	688K	2 / 16	100	528	1s
gaussian R	470	7K	16	100	6	
heartwall R	2171	16K	2605	94	2327	1s,1g
hits G	431	25K	80 / 50	100	38	1s
hotspot R	339	473K	28	99	109	
hotspot3D R	324	66K	81	100	1095	
huffman R	1185	16K	16 / 8	79	4	
hybridsort R	1145	16K	1025	100	1456	2s
kmeans R	518	496K	34	100	423	
lavaMD R	541	128K	126	99	63	
lbm P	1018	2.2M	20	100	1318	
leukocyte R	657	12K	176	100	14	5s
lud R	401	58K	48	98	333	
mri-q P	376	66K	3 / 5	99	10	
myocyte R	5413	64	6 / 8	54	2	1g
nn R	322	43K	1	99	3	
nw R	453	2K	34	99	134	
particlef_float R	870	1K	622	100	8	2g
particlef_naive R	696	1K	6	99	¡1	
pathfinder R	236	119K	439	99	358	
pr G	1116	246K	18 / 52	100	390	1s
scalarProd N	271	33K	4 / 32	52	31	1s
sad P	1440	13K	360 / 90	99	35	
salsa G	456	131K	189 / 33	100	132	
sample G	496	33K	2	50	2	
sgemm P	417	1K	1 / 16	100	7232	
srad R	566	2.1M	113	99	2209	
spmv P	977	12K	3407 / 131	63	48	
sssp G	747	49K	69 / 14	99	21	2s
streamcluster R	1518	66K	2631 / 516	50	1032	
threadFenceRed N	1021	8K	66 / 128	98	32	1s
topk G	449	33K	2 / 49	100	49	

memory checking adds an additional 4% overhead for heartwall, intra-block global memory checks add 1%, and the remaining 11% results from inter-block global memory checks. After investigating the instrumented bitcode of heartwall, we found that most instrumented accesses are to local or constant memory, which introduces unnecessary overhead. CURD includes a dynamic check to ignore local and constant memory accesses, as these cannot be involved in data races, but the cost of these checks accumulates rapidly. Although the data-flow analysis we mentioned in Section 6.2 already prunes some of local or constant memory accesses, the address space of some pointers cannot be statically decided. One direction to improve this is to do a more aggressive inter-procedural analysis to decide the address space of more pointers statically. We leave this as a future work.

On 21 programs in all, the main source of CURD's overhead is instrumentation. These programs may also benefit from the address space analysis that would help heartwall. Monitoring shared memory is the most significant source of overhead on eight programs. On eight other programs, checking intra-block races on global memory contributes the most to the overall overhead, while the cost of checking inter-block races dominates on just three programs (myocyte, spmv and streamcluster).

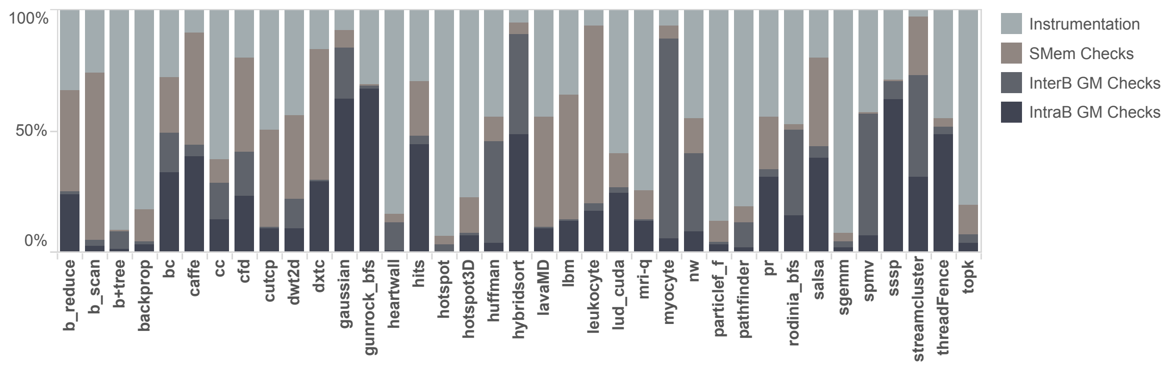


Figure 6.11: How much different parts of Curd-Lazy contribute to its performance overhead.

6.3.4. Data Races Detected

Across our benchmark programs, CURD detects 35 races in total (Column 7 in Table 6.1). Among them are 24 shared memory races, 5 intra-block global memory races, and 6 inter-block global memory races. The shared memory races have been validated with CUDA-Racecheck. For benchmarks that BARRACUDA can successfully run, we have also cross-validated the shared memory and global memory races detected by CURD and BARRACUDA. Benchmarks are run with both CURD-Eager and CURD-Lazy, and the two schemes detect the same set of races. After investigating the races reported by CURD, we group the races into the following categories:

- **Shared memory races without any synchronization:** CURD finds 15 such shared memory races in 13 programs. Among these races, three, including one in leukocyte and two in hybridsort, are write-write conflicts involve multiple threads writing to the same shared memory location within a barrier-free-region. According to the CUDA language specification, the order of these writes are undefined, so such conflicts form data races. The rest of the 12 races are write-read conflicts between threads in different warps within the same barrier-free-region, such as the 5 races shown in leukocyte. Depending on the actual execution order of the read and write operations, the read may return different values.

- **Warp-level synchronization:** CURD issues warnings for fragile code that relies on intra-warp synchronization. We found 9 such issues in our benchmarks. In future GPU architectures, such code may break if the warp size changes or additional warp scheduling flexibility is introduced [46, 73]. In fact, with the release of CUDA 9.0 and introduction of the Volta architecture, threads within a warp are no longer executed in lock-step unconditionally; as a result, code relying on warp-level synchronization now needs to be modified in order to behave correctly on the new generations of GPU hardware.

- **Intra-block global memory races:** One such race is detected in heartwall and one in cc, while 3 others are found in bfs (from Rodinia). Among these races, The ones in bfs are particularly interesting in that they involve indirect memory accesses: the index of a write into an array a is obtained by first reading an element from another array b; unfortunately, two different threads can read the same value x from elements of array b, then concurrently write $a[x]$, forming a race condition. These kinds of indirect memory accesses are particularly challenging to reason about, for both humans and static analysis techniques.

- **Inter-block global memory races:** In total, CURD finds 6 such races in the benchmarks we used, including three in bfs, and one each in myocyte, cc and particlefilter_float. These races were easy to verify as these three programs do not have any inter-block synchronization, so these conflicting global memory accesses are clearly race conditions.

6.4. Conclusion

In this chapter, we have presented the CURD system, a new data race detector for CUDA programs. CURD has been implemented in LLVM and applied to a wide range of benchmark kernels. CURD detects data races in both global and shared memory, unlike many previous race detectors that focus exclusively on shared memory. Simultaneously, CURD provides a substantial performance boost over previous work such as Nvidia's CUDA-Racecheck system. The key to CURD's performance advantages are its compiler-based instrumentation, an efficient representation of read/write sets that is compact in the common case of well-tuned GPU code, and GPU-specific optimizations to maximize memory bandwidth utilization. We believe CURD is robust enough and fast enough to have a positive impact on how CUDA developers debug their code.

CHAPTER 7

Related Work

Dynamic data race detection has been studied extensively in the past a few decades, and there exist a large number of published work that are relevant to the systems discussed in this dissertation. In this chapter, we discuss prior research that is most related to the core topic of this dissertation, i.e., dynamic data race detection for CPU and GPU programs.

7.1. CPU Data Race Detection

Many dynamic data race detection algorithms have been proposed for general programs that run on CPUs. Existing dynamic data race detectors can be software-only, hardware-assisted, or adopt a hybrid design.

In software-only solutions, the most closely related work to SlimFast is the FastTrack [39] system, with which we have already compared extensively. The SlimState[115] system seeks to reduce metadata redundancy for dynamic race detection, just as SlimFast does. However, SlimState removes redundancy only within individual arrays. SlimState also delays race checks until synchronization operations, improving compression but losing information about where exactly races occur in the code. In contrast, FastTrack and SlimFast detect races eagerly, reporting a precise code point involved in each data race, greatly facilitating debugging. The RedCard[41] system provides static analysis to accelerate dynamic race detection, merging the metadata for two memory locations that are always accessed together in a release-free span. This analysis targets metadata space usage, as SlimFast does, but is limited by the conservatism of static analysis. For the Java Grande benchmarks used in both evaluations, RedCard saves no space on lufact, montecarlo, series, sor, or sparsematmult while SlimFast reduces the heap by at least 2x on each. RedCard reduces metadata alone by 1.5x on moldyn and 7x on crypt, while SlimFast reduces metadata by 300,000x and 12x, respectively.

In Song and Lee[106], race detection metadata that is identical across consecutive memory locations can be coalesced into a single metadata object, though redundancy across non-consecutive locations can remain. Moreover, coalescing can introduce false races as an access to a location x appears as an access to all locations whose metadata is coalesced with x's. Accordion clocks[19] reduce metadata space usage by removing vector clock entries for terminated threads. FastTrack makes the use of vector clocks infrequent and thus dilutes the benefit of accordion clocks, though they remain complementary to SlimFast as we do not reduce the space consumption of vector clocks.

Several forms of sampling-based dynamic race detection have been proposed. Such schemes trade off soundness [49, 14, 70, 32, 28] for reduced performance overheads. In contrast, SlimFast reduces the memory and performance overheads of data race detection without sacrificing precision. Lockset-based race detection [27, 100], an alternative to the happens-before data race detection algorithm, reports false races on some common programming idioms like privatization but can also detect with a single execution some races that would require multiple executions with happens-before. Other work has generalized happens-before race detection to detect more races from a single execution [105, 102, 16] at the cost of decreased performance.

Several systems exist for detecting data races in structured parallel programs such as fork-join programs [58, 67, 75], series-parallel Cilk programs [35], async-finish programs [96] or programs with asynchronous callbacks [90, 98, 53]. None of these algorithms exploit metadata redundancy.

Another type of dynamic data race detectors is hardware assisted detection systems like PARSNIP. The work most closely related to PARSNIP is the Radish system[25] for hardware-accelerated sound and complete race detection. We compare extensively to Radish in Chapter 4. The LARD system [116] showed that naive usage of even sound and complete hardware data race detection can result in false and missed data races due to interactions with layers of the system stack like the OS and language runtime. LARD also demonstrates how to convey sufficient information across these layers to restore precision. Other hardware race detectors [92, 76, 120, 78, 80, 94, 93, 55]sacrifice soundness and completeness in favor of simpler and faster hardware.

The Vulcan[77] and Volition[95] hardware architectures have been proposed for detecting and recovering from sequential-consistency violations, which arise from two or more cyclically-coupled data races. These schemes provide sequential consistency at the instruction level by detecting the underlying data races that can violate SC. Both schemes implement precise data race detection, however, it is only needed during a short window within which instruction reordering can occur, which simplifies the implementation and allows for almost-negligible performance overheads. Other hardware schemes enforce stronger memory consistency models design to preserve sequential consistency or related properties[65, 71, 104, 101]. SC violation detectors ignore data races where at least one access occurs outside of the current detection window. Sound techniques like PARSNIP can find these additional races, making debugging easier and supporting a wide range of race-detection clients such as record-and-replay and deterministic execution. HARD [120] is proposed to provide hardware support for lockset race detection to reduce its performance overheads.

7.2. GPU Data Race Detection

Several prior schemes have been proposed for detecting data races in GPU programs. Boyer et al.[75] analyze CUDA programs for data races and inefficient memory accesses. Their race analysis is restricted to shared memory only, and does not take account of atomics or memory fences. GRace[67] proposed a dynamic analysis to find intra-warp races and inter-warp races via shared memory, using static analysis to prune instrumentation when possible. GMrace[118] detects the same kinds of errors as GRace, but with improved running time. Neither GRace nor GMrace detect any inter-block concurrency bugs, nor bugs related to global memory, atomics or memory fences. LDetector [64] can find concurrency bugs via both shared and global memory, but it uses value-based checking to detect writes so it may miss bugs that involve a thread overwriting a location with the location's existing value. LDetector does not handle atomics or memory fences. HAccRG[50] is a hardware-based data race detector for GPUs. It provides coverage of both shared and global memory and also memory fences. To keep hardware overheads low, however, HAccRG does not track all readers for a given location, which can lead to missed races.

The structured data parallel nature of many GPU kernels makes them well-suited to static verification. The GPUVerify system [7, 6, 4, 5, 18, 17, 21] uses SMT solving to find data races and barrier divergence. GPUVerify is sound (it does not miss real bugs) up to the CUDA features it supports, though it occasionally reports false races and does not support memory fences or indirect memory accesses. PUG [32] also uses SMT solving to find races and barrier divergence bugs, though its abstractions can cause it to be both unsound and incomplete in some cases. The GKLEE[63] and KLEE-CL[20] systems use dynamic symbolic execution to find bugs in GPU kernels, and GKLEE has been extended to handle atomic operations[63], but it is difficult to scale symbolic execution beyond small kernels. Leung et al.[61] check for data races and determinism of GPU kernels leveraging the insight that most of a kernel's execution is independent of its input parameters, leaving only a portion of the kernel that requires dynamic checking. Their dynamic analysis does not, however, handle kernels with atomics or memory fences. Though completeness remains a challenge for static analysis techniques, leveraging verification machinery to filter dynamic instrumentation could be a powerful and complementary optimization for systems like BARRACUDA.

Several papers have focused on elucidating the memory consistency models of GPU architectures. Work targeting AMD GPUs[51, 48, 114] has culminated in the Heterogeneous System Architecture (HSA) formal memory consistency model[52] adopted by AMD and other GPU manufacturers. In comparison, work on formalizing CUDA's consistency model is still nascent and driven by 3rd party researchers. Recent work has explored the CUDA memory model via litmus tests: Alglave et al. [3] present an axiomatic memory consistency model for Nvidia GPUs, and Sorensen et al. [107] identify fuzz testing strategies to expose concurrency bugs on GPUs. Our definition of synchronization order is informed by this prior work. In contrast to work on litmus testing, we have pursued a new safety property for CUDA that can be dynamically checked without the need for fuzzing or program-specific invariants.

CHAPTER 8

Conclusions

This dissertation has demonstrated and evaluated a series of work on precise dynamic data race detection. We started by presenting SLIMFAST, a scheme to identify and reduce common metadata redundancy in dynamic race detector for CPU programs(Chapter 3), showing the performance limitation of software-only data race detectors. We then described PARSNIP (Chapter 4), a work motivated by SLIMFAST's high overhead, that brings the runtime overhead of dynamic race detectors to less than 1.5x. We believe this is a solid step towards always-on dynamic race detection, which can enable useful high-level applications such as Data Race Exceptions[2, 31, 65, 71]. The focus of the dissertation then shifts to data race detection for GPU programs, by first introducing BARRACUDA (Chapter 5), a tool that adapts a CPU-oriented algorithm to run for GPU programs, followed by a presentation of CURD (Chapter 6), a race-detection system that is optimized for the common synchronization mechanisms and access patterns.

8.1. Summary of Techniques

One idea that appeared repeatedly in the systems presented in this dissertation, is to identify and reduce redundancy in dynamic race detection. Specifically, SLIMFAST identifies and removes metadata redundancy of access histories on-the-fly, reducing memory overhead and boosting runtime performance. PARSNIP de-duplicates metadata in hardware, utilizing existing cache system to manage metadata for race detection. BARRACUDA makes it possible to scale vector-clock based data race detector design for CPU programs to GPU programs, which can have millions of threads, by avoiding redundancy in per-thread metadata. In the CURD system, a number of optimizations aim at avoiding redundant computation. Redundancy often arise in data race detectors, due to different kinds of locality and correlation, and reducing such redundancy can often translate into higher efficiency without hurting precision.

Another general technique widely used by the systems presented in this dissertation is to exploit parallelism in hardware whenever possible. For example, the Parsnip design rests on existing multi-core cache system design, and implements the common operations of dynamic race detection in hardware. The Barracuda implementation take advantage of the often-idle CPU when a kernel is running on GPU, and utilize the CPU and main memory to do online analysis for data races. The Curd system, on the other hand, exploits the parallelism on the GPU to do the computation required by race checking to maximize efficiency.

Finally, a simple yet effective approach that is widely used in all works of this dissertation is caching. Despite its simplicity, caching proves useful in various scenarios. SlimFast uses caching to speed up the metadata look-ups; Parsnip exploits extensive caching in its management of metadata; Barracuda implicitly benefits from the caching effects in its events logging; Curd uses caching to reduce device memory traffic and to boost look-ups into the sets.

8.2. Limitations

The ultimate goal of the work in this dissertation is to make precise dynamic race detection efficient enough to be always-on, for both CPU and GPU applications, thereby making parallel programming and debugging easier. However, there is still a long way to go to achieve this end goal. In particular, systems demonstrated in this dissertation are limited by several factors.

Perhaps the most important limiting factor is performance overhead. As shown by the evaluation results in Chapter 3, 4, 5, 6, although our work improves the performance of data race detection, a precise software-only race detector can slow down a program by orders of magnitude. Such high overhead makes a race detector impractical to apply in many production scenarios.

Another limitation is the hardware requirements. In particular, Parsnip requires modifications on commodity processors, which can take an unpredictably long time to happen. The efficiency of the Barracuda implementation can also be impacted by the efficiency of hardware interface (e.g. PCI-E) between GPU and CPU. The Curd detector, being a GPU application itself, needs to have detailed knowledge about the specifics of the target GPU hardware, to maximize performance.

Finally, a general limitation of almost all pure dynamic race detector, is that only races that occur in an observed execution can be detected. For some feasible races that only manifest under some rare scheduling conditions, it may be difficult for dynamic race detectors to catch them. Also, as the execution of a program is monitored dynamically, the disturbance caused by the monitoring on the execution, no matter how small, can change a program's behavior, making it a challenge to detect "Heisenbugs".

8.3. Looking forward

It is our particular hope that the performance of precise data race detectors can continue to improve and their applicable scope becomes increasingly broad, and always-on data race detection can ultimately become a reality. Although we are not there yet, we hope the work and results presented in this dissertation can provide some useful reference for future endeavors.

BIBLIOGRAPHY

[1] Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. ACM Transactions on Programming Languages and Systems 28(2), 207–255 (Mar 2006), http://doi.acm.org/10.1145/1119479.1119480

[2] Adve, S.: Data races are evil with no exceptions. Communications of the ACM 53(11), 84 (Nov 2010), http://portal.acm.org/citation.cfm?doid=1839676.1839697

[3] Alglave, J., Batty, M., Donaldson, A.F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T., Wickerson, J.: Gpu concurrency: Weak behaviours and programming assumptions. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 577–591. ASPLOS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2694344.2694391

[4] Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson, A.F., Ketema, J., Liew, D., Qadeer, S.: Engineering a static verification tool for gpu kernels. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 226–242. Springer-Verlag New York, Inc., New York, NY, USA (2014), http://dx.doi.org/10.1007/978-3-319-08867-9_15

[5] Bardsley, E., Donaldson, A.F.: Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels. In: Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430. pp. 230–245. Springer-Verlag New York, Inc., New York, NY, USA (2014), http://dx.doi.org/10.1007/978-3-319-06200-6_18

[6] Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: A verifier for gpu kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 113–132. OOPSLA '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2384616.2384625

[7] Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for gpu kernels. ACM Trans. Program. Lang. Syst. 37(3), 10:1–10:49 (May 2015), http://doi.acm.org/10.1145/2743017

[8] Bielik, P., Raychev, V., Vechev, M.: Scalable Race Detection for Android Applications. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. OOPSLA (2015)

[9] Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University (Jan 2008)

[10] Biswas, S., Cao, M., Zhang, M., Bond, M.D., Wood, B.P.: Lightweight data race detection for production runs. In: Proceedings of the 26th International Conference on Compiler Construction. pp. 11–21. ACM (2017)

[11] Biswas, S., Zhang, M., Bond, M.D., Lucia, B.: Valor: Efficient, software-only region conflict exceptions. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 241–259. OOPSLA 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2814270.2814292

[12] Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanovi, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. pp. 169–190. OOPSLA '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1167473.1167488

[13] Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08. p. 68. Tucson, AZ, USA (2008), http://portal.acm.org/citation.cfm?doid=1375581.1375591

[14] Bond, M.D., Coons, K.E., McKinley, K.S.: PACER: Proportional Detection of Data Races. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10. p. 255. Toronto, Ontario, Canada (2010), http://portal.acm.org/citation.cfm?doid=1806596.1806626

[15] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. pp. 44–54. IEEE (2009)

[16] Chen, F., Rou, G.: Parametric and Sliced Causality. In: Proceedings of the 19th International Conference on Computer Aided Verification. pp. 240–253. CAV'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1770351.1770387

[17] Chong, N., Donaldson, A.F., Kelly, P.H., Ketema, J., Qadeer, S.: Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications. pp. 605–622. OOPSLA '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2509136.2509517

[18] Chong, N., Donaldson, A.F., Ketema, J.: A sound and complete abstraction for reasoning about parallel prefix sums. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 397–409. POPL '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2535838.2535882

[19] Christiaens, M., Bosschere, K.D.: TRaDe: Data Race Detection for Java. In: Proceedings of the International Conference on Computational Science-Part II. pp. 761–770. ICCS '01, Springer-Verlag, London, UK, UK (2001), http://dl.acm.org/citation.cfm?id=645456.654536

[20] Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of opencl code. In: Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing. pp. 203–218. HVC'11, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-34188-5_18

[21] Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of gpu kernels. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. pp. 270–289. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-37036-6_16

[22] Daly, C., Horgan, J., Power, J., Waldron, J.: Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark suite. In: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande. pp. 106–115. JGI '01, New York, NY, USA (2001), http://doi.acm.org/10.1145/376656.376826

[23] Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 63–74. GPGPU-3, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1735688.1735702

[24] Devietti, J., Lucia, B., Ceze, L., Oskin, M.: DMP: Deterministic Shared Memory Multiprocessing. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09). p. 85. Washington, DC, USA (2009), http://portal.acm.org/citation.cfm?doid=1508244.1508255

[25] Devietti, J., Wood, B.P., Strauss, K., Ceze, L., Grossman, D., Qadeer, S.: RADISH: always-on sound and complete RAce Detection In Software and Hardware. In: Proceedings of the 39th Annual International Symposium on Computer Architecture. pp. 201–212. ISCA '12, IEEE Computer Society, Washington, DC, USA (2012), http://dl.acm.org/citation.cfm?id=2337159.2337182

[26] Dice, D.: Biased locking in hotspot. https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot (Aug 2006)

[27] Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging. pp. 85–96. PADD '91, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/122759.122767

[28] Effinger-Dean, L., Lucia, B., Ceze, L., Grossman, D., Boehm, H.J.: IFRit: interference-free regions for dynamic data-race detection. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 467–484. OOPSLA '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2384616.2384650

[29] Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: Barracuda: Binary-level analysis of runtime races in cuda programs. In: ACM SIGPLAN Notices. vol. 52, pp. 126–140. ACM (2017)

[30] Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: Barracuda: Binary-level analysis of runtime races in cuda programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 126–140. PLDI 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3062341.3062342

[31] Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware java runtime. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 245–255 (Jun 2007), http://doi.acm.org/10.1145/1273442.1250762

[32] Erickson, J., Musuvathi, M., Burckhardt, S., Olynyk, K.: Effective data-race detection for the kernel. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–16. OSDI'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1924943.1924954

[33] Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., Schwan, K.: A framework for dynamically instrumenting gpu compute applications within gpu ocelot. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. pp. 9:1–9:9. GPGPU-4, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1964179.1964192

[34] Farooqui, N., Kerr, A., Eisenhauer, G., Schwan, K., Yalamanchili, S.: Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures. In: Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on. pp. 58–67. IEEE (2012)

[35] Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in Cilk programs. In: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures - SPAA '97. pp. 1–11. Newport, Rhode Island, United States (1997), http://portal.acm.org/citation.cfm?doid=258492.258493

[36] Fidge, C.: Logical time in distributed computing systems. IEEE Computer 24(8), 28–33 (Aug 1991), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=84874

[37] Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 256–267. POPL '04, New York, NY, USA (2004), http://doi.acm.org/10.1145/964001.964023

[38] Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design

and implementation - PLDI '09. p. 121. Dublin, Ireland (2009), http://portal.acm.org/citation.cfm?doid=1542476.1542490

[39] Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. Communications of the ACM 53(11), 93–101 (Nov 2010), http://doi.acm.org/10.1145/1839676.1839699

[40] Flanagan, C., Freund, S.N.: The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 1–8. PASTE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806672.1806674

[41] Flanagan, C., Freund, S.N.: RedCard: Redundant Check Elimination for Dynamic Race Detectors. In: Proceedings of the 27th European Conference on Object-Oriented Programming. pp. 255–280. ECOOP'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39038-8_11

[42] Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08. p. 293. Tucson, AZ, USA (2008), http://portal.acm.org/citation.cfm?doid=1375581.1375618

[43] Force, U.C.: Final report on the august 14th blackout in the united states and canada. Department of Energy and National Resources Canada (2004)

[44] Friedemann Mattern: Virtual Time and Global States of Distributed Systems. In: Parallel and Distributed Algorithms (1989)

[45] Frumkin, M.A., Schultz, M., Jin, H., Yan, J.: Implementation of the NAS Parallel Benchmarks in Java NAS. Tech. Rep. NAS-02-009, NASA Advanced Supercomputing Division (2002), https://www.nas.nasa.gov/publications/npb.html

[46] Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.M.: Dynamic warp formation and scheduling for efficient gpu control flow. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 407–420. MICRO 40, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/MICRO.2007.12

[47] Fung, W.W.L., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for gpu architectures. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 296–307. MICRO-44, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2155620.2155655

[48] Gaster, B.R., Hower, D., Howes, L.: Hrf-relaxed: Adapting hrf to the complexities of industrial heterogeneous memory models. ACM Trans. Archit. Code Optim. 12(1), 7:1–7:26 (Apr 2015), http://doi.acm.org/10.1145/2701618

[49] Greathouse, J.L., Ma, Z., Frank, M.I., Peri, R., Austin, T.: Demand-driven Software Race Detection Using Hardware Performance Counters. In: Proceedings of the 38th Annual

International Symposium on Computer Architecture. pp. 165–176. ISCA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2000064.2000084

[50] Holey, A., Mekkat, V., Zhai, A.: Haccrg: Hardware-accelerated data race detection in gpus. In: Proceedings of the 2013 42Nd International Conference on Parallel Processing. pp. 60–69. ICPP '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/ICPP.2013.15

[51] Hower, D.R., Hechtman, B.A., Beckmann, B.M., Gaster, B.R., Hill, M.D., Reinhardt, S.K., Wood, D.A.: Heterogeneous-race-free memory models. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 427–440. ASPLOS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2541940.2541981

[52] HSA Foundation: HSA Memory Consistency Model, http://www.hsafoundation.com/html/HSA_Library.htm#SysArch/Topics/03_Memory/_chpStr_HSA_memory_consistency_model.htm

[53] Hsiao, C.H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C.L., Pokam, G.A., Chen, P.M., Flinn, J.: Race Detection for Event-driven Mobile Applications. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 326–336. PLDI '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2594291.2594330

[54] Huang, J., Rajagopalan, A.K.: Precise and Maximal Race Detection from Incomplete Traces. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (2016)

[55] Huang, R., Halberg, E., Ferraiuolo, A., Suh, G.: Low-overhead and high coverage run-time race detection through selective meta-data management. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 96–107 (Feb 2014), raceSMM

[56] Jackson, J.: Nasdaqs facebook glitch came from race conditions (2012)

[57] Ji, W., Lu, L., Scott, M.L.: TARDIS: Task-level Access Race Detection by Intersecting Sets. In: Proceedings of the 4th Workshop on Determinism and Correctness in Parallel Programming (WODET '13) (2013)

[58] John Mellor-Crummey: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91. pp. 24–33. Albuquerque, New Mexico, United States (1991), http://portal.acm.org/citation.cfm?doid=125826.125861

[59] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers C-28(9), 690–691 (Sep 1979), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1675439

[60] Lee, Y., Grover, V., Krashinsky, R., Stephenson, M., Keckler, S.W., Asanovic, K.: Exploring the design space of spmd divergence management on data-parallel architectures. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 101–113 (Dec 2014)

[61] Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying gpu kernels by test amplification. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 383–394. PLDI '12, ACM, New York, NY, USA (2012)

[62] Leveson, N.G., Turner, C.S.: An investigation of the therac-25 accidents. Computer 26(7), 18–41 (1993)

[63] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: Gklee: Concolic verification and test generation for gpus. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 215–224. PPoPP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2145816.2145844

[64] Li, P., Ding, C., Hu, X., Soyata, T.: LDetector: A Low Overhead Race Detector For GPU Programs. In: Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming (WODET '14) (2014)

[65] Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm, H.J.: Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In: Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10. p. 210. Saint-Malo, France (2010), http://portal.acm.org/citation.cfm?doid=1815961.1815987

[66] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. PLDI '05, New York, NY, USA (2005), http://doi.acm.org/10.1145/1065010.1065034

[67] Mai Zheng, Vignesh T. Ravi, Feng Qin, Gagan Agrawal: GRace: a low-overhead mechanism for detecting data races in GPU programs. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. New York, NY (2011)

[68] Mansky, W., Peng, Y., Zdancewic, S., Devietti, J.: Verifying dynamic race detection. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs. pp. 151–163. ACM (2017)

[69] Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05. pp. 378–391. Long Beach, California, USA (2005), http://portal.acm.org/citation.cfm?doid=1040305.1040336

[70] Marino, D., Musuvathi, M., Narayanasamy, S.: LiteRace: Effective Sampling for Lightweight Data-Race Detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09. p. 134. Dublin, Ireland (2009), http://portal.acm.org/citation.cfm?doid=1542476.1542491

[71] Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: DRFX: a simple and efficient memory model for concurrent programming languages. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10. p. 351. Toronto, Ontario, Canada (2010), http://portal.acm.org/citation.cfm?doid=1806596.1806636

[72] Mathew, J.A., Coddington, P.D., Hawick, K.A.: Analysis and Development of Java Grande Benchmarks. In: Proceedings of the ACM 1999 Conference on Java Grande. pp. 72–80. JAVA '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/304065.304101

[73] Meng, J., Tarjan, D., Skadron, K.: Dynamic warp subdivision for integrated branch and memory divergence tolerance. In: Proceedings of the 37th Annual International Symposium on Computer Architecture. pp. 235–246. ISCA '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1815961.1815992

[74] Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (Jun 2004), http://dx.doi.org/10.1109/TPDS.2004.8

[75] Michael Boyer, Kevin Skadron, Westley Weimer: Automated Dynamic Analysis of CUDA Programs. In: Workshop on Software Tools for MultiCore Systems (2008)

[76] Milos Prvulovic: CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In: Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture (2006)

[77] Muzahid, A., Qi, S., Torrellas, J.: Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 363–375. MICRO-45, IEEE Computer Society, Washington, DC, USA (2012), http://dx.doi.org/10.1109/MICRO.2012.41

[78] Muzahid, A., Surez, D., Qi, S., Torrellas, J.: SigRace: Signature-Based Data Race Detection. In: Proceedings of the 36th annual international symposium on Computer architecture. pp. 337–348. ISCA '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1555754.1555797

[79] Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. pp. 308–319. PLDI '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1133981.1134018

[80] Nistor, A., Marinov, D., Torrellas, J.: Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 541–552. MICRO 42, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1669112.1669180

[81] Nvidia: CUDA C Programming Guide v7.5, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[82] Nvidia: Parallel Thread Execution ISA Version 4.3, http://docs.nvidia.com/cuda/parallel-thread-execution/

[83] Nvidia: Racecheck Tool (2016), http://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool

[84] Nvidia: SASSI Instrumentation Tool for NVIDIA GPUs (2016), https://github.com/NVlabs/SASSI

[85] Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: Efficient Deterministic Multithreading in Software. In: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09. p. 97. Washington, DC, USA (2009), http://portal.acm.org/citation.cfm?doid=1508244.1508256

[86] Oracle: Oracle berkeley db java edition 6.4.25. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index-093405.html (12 2015)

[87] Peng, Y., DeLozier, C., Eizenberg, A., Mansky, W., Devietti, J.: Slimfast: Reducing metadata redundancy in sound and complete dynamic data race detection. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 835–844. IEEE (2018)

[88] Peng, Y., Grover, V., Devietti, J.: Curd: a dynamic cuda race detector. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 390–403. ACM (2018)

[89] Peng, Y., Wood, B.P., Devietti, J.: Parsnip: performant architecture for race safety with no impact on precision. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 490–502. ACM (2017)

[90] Petrov, B., Vechev, M., Sridharan, M., Dolby, J.: Race detection for web applications. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 251–262. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254095

[91] Pozniansky, E., Schuster, A.: Efficient on-the-fly data race detection in multithreaded C++ programs. In: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 179–190. PPoPP '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/781498.781529

[92] Prvulovic, M., Torrellas, J.: ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In: Proceedings of the 30th annual international symposium on Computer architecture. pp. 110–121. ISCA '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/859618.859632

[93] Qi, S., Muzahid, A.A., Ahn, W., Torrellas, J.: Dynamically detecting and tolerating if-condition data races. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 120–131 (Feb 2014)

[94] Qi, S., Otsuki, N., Nogueira, L.O., Muzahid, A., Torrellas, J.: Pacman: Tolerating asymmetric data races with unintrusive hardware. In: Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture. pp. 1–12. HPCA '12, IEEE Computer Society, Washington, DC, USA (2012), http://dx.doi.org/10.1109/HPCA.2012.6169039

[95] Qian, X., Torrellas, J., Sahelices, B., Qian, D.: Volition: Scalable and Precise Sequential Consistency Violation Detection. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 535–548. ASPLOS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2451116.2451174

[96] Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Scalable and precise dynamic datarace detection for structured parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 531–542. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254127

[97] Raychev, V., Vechev, M., Sridharan, M.: Effective Race Detection for Event-driven Programs. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. OOPSLA (2013)

[98] Raychev, V., Vechev, M., Sridharan, M.: Effective Race Detection for Event-driven Programs. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 151–166. OOPSLA '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2509136.2509538

[99] Sanchez, D., Kozyrakis, C.: Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In: Proceedings of the 40th Annual International Symposium on Computer Architecture. pp. 475–486. ISCA '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2485922.2485963

[100] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4), 391–411 (Nov 1997), http://portal.acm.org/citation.cfm?doid=265924.265927

[101] Segulja, C., Abdelrahman, T.S.: Clean: A race detector with cleaner semantics. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture. pp.

401–413. ISCA '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2749469.2750395

[102] Sen, K., Rou, G., Agha, G.: Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In: Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems. pp. 211–226. FMOODS'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11494881_14

[103] Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data Race Detection in Practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp. 62–71. WBIA '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1791194.1791203

[104] Singh, A., Marino, D., Narayanasamy, S., Millstein, T., Musuvathi, M.: Efficient Processor Support for DRFx, a Memory Model With Exceptions. In: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems. pp. 53–66. ASPLOS '11, New York, NY, USA (2011), http://doi.acm.org/10.1145/1950365.1950375

[105] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 387–400. POPL '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2103656.2103702

[106] Song, Y.W., Lee, Y.H.: Efficient data race detection for c/c++ programs using dynamic granularity. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp. 679–688 (May 2014)

[107] Sorensen, T., Donaldson, A.F.: Exposing errors related to weak memory in gpu applications. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 100–113. PLDI '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2908080.2908114

[108] Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., vLi Wen Chang, Anssari, N., Liu, G.D., mei W. Hwu, W.: Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign (March 2012), http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf

[109] Thoziyoor, S., Muralimanohar, N., Ahn, J.H., Jouppi, N.P.: CACTI 5.1. Tech. Rep. HPL-2008-20, Hewlett-Packard Labs, http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html

[110] W. W. L. Fung et al.: http://www.ece.ubc.ca/ wwlfung/code/kilotm-gpgpu_sim.tgz (2013), http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpu_sim.tgz

[111] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 11:1–11:12. PPoPP '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2851141.2851145`

[112] Wentzlaff, D., Beckmann, N., Miller, J., Agarwal, A.: Core count vs cache size for manycore architectures in the cloud. Tech. Rep. MIT-CSAIL-TR-2010-008, MIT (Feb 2010), `http://hdl.handle.net/1721.1/51733`

[113] Wester, B., Devecsery, D., Chen, P.M., Flinn, J., Narayanasamy, S.: Parallelizing Data Race Detection. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (2013)

[114] Wickerson, J., Batty, M., Beckmann, B.M., Donaldson, A.F.: Remote-scope promotion: Clarified, rectified, and verified. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 731–747. OOPSLA 2015, ACM, New York, NY, USA (2015), `http://doi.acm.org/10.1145/2814270.2814283`

[115] Wilcox, J., Finch, P., Flanagan, C., Freund, S.N.: Array shadow state compression for precise dynamic race detection. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. ASE '15 (2015)

[116] Wood, B.P., Ceze, L., Grossman, D.: Low-level detection of language-level data races with lard. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 671–686. ASPLOS '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2541940.2541955`

[117] Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the twentieth ACM symposium on Operating systems principles. pp. 221–234. SOSP '05, ACM, New York, NY, USA (2005), `http://doi.acm.org/10.1145/1095810.1095832`

[118] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Gmrace: Detecting data races in gpu programs via a low-overhead scheme. IEEE Transactions on Parallel and Distributed Systems 25(1), 104–115 (Jan 2014)

[119] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Grace: A low-overhead mechanism for detecting data races in gpu programs. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. pp. 135–146. PPoPP '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1941553.1941574`

[120] Zhou, P., Teodorescu, R., Zhou, Y.: HARD: Hardware-Assisted Lockset-based Race Detection. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. pp. 121–132. HPCA '07, Washington, DC, USA (2007), `http://dx.doi.org/10.1109/HPCA.2007.346191`