Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems

Chang Seo Park

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-243
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-243.html

December 13, 2012

Copyright © 2012, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems

by

Chang Seo Park

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Koushik Sen, Chair
Professor Rastislav Bodík
Professor David Wessel

Fall 2012

Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems

Copyright 2012
by
Chang Seo Park

Abstract

Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems

by

Chang Seo Park

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Koushik Sen, Chair

Parallel and concurrent software sometimes exhibit incorrect behavior because of unintended interference between different threads of execution. Common classes of concurrency bugs include data races, deadlocks, and atomicity violations. These bugs are often non-deterministic and hard to find without sophisticated tools. We present Active Testing, a methodology to effectively find concurrency bugs that scales to large distributed memory parallel systems.

Active Testing combines the coverage and predictive power of program analysis with the familiarity of testing. It works in two phases: in the predictive analysis phase, a program is executed and monitored for potential concurrency bugs and in the testing phase, Active Testing re-executes the program while controlling the thread schedules in an attempt to reproduce the bug predicted in the first phase.

We have implemented Active Testing for multi-threaded Java programs in the CalFuzzer framework. We have also developed UPC-Thrille, an Active Testing framework for Unified Parallel C (UPC) programs written in the Single Program Multiple Data (SPMD) programming model combined with the Partitioned Global Address Space (PGAS) memory model. We explain in detail the design decisions and optimizations that were necessary to scale Active Testing to thousands of cores. We present extensions to UPC-Thrille that support hybrid memory models as well.

We evaluate the effectiveness of Active Testing by running our tools on several Java and UPC benchmarks, showing that it can predict and confirm real concurrency bugs with low overhead. We demonstrate the scalability of Active Testing by running benchmarks with UPC-Thrille on large clusters with thousands of cores.

To my father and mother,

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

List of Algorithms **vi**

List of Programs **vii**

1 Introduction **1**
- 1.1 Existing techniques to find concurrency bugs 1
- 1.2 Contributions 3
- 1.3 Outline 4

2 Common Concurrency Bugs **6**
- 2.1 Data race through an example 6
- 2.2 Deadlock through an example 9
- 2.3 Atomicity violation through an example 11

3 Active Testing **14**
- 3.1 Active Testing through an example 15
- 3.2 Concurrent system model 17
- 3.3 Phase I: Prediction 20
- 3.4 Phase II: Confirmation 22

4 Instantiations of Active Testing **25**
- 4.1 Data race detection for UPC 25
- 4.2 Deadlock detection 30
- 4.3 Atomicity violation detection 36

5 Implementation for Multi-threaded Programs **40**
- 5.1 CalFuzzer 40

5.2 Computing object abstractions 41
5.3 Yielding to avoid thrashing in reproducing deadlocks 45

6 Evaluation of Active Testing for Multi-threaded Programs 46
6.1 Summary of results 46
6.2 Deadlocks found 47
6.3 Atomicity violations found 49
6.4 Evaluation of object abstractions and yielding 53

7 Implementation for Distributed Memory Parallel Programs 57
7.1 Communication Avoiding Dynamic Analysis 57
7.2 UPC-Thrille 61
7.3 Extensions to hybrid memory models 68

8 Evaluation of Active Testing for Distributed Memory Programs 74
8.1 Evaluation of UPC-Thrille for data race detection 74
8.2 Evaluation of techniques for hybrid memory models 80

9 Related Work 87
9.1 Concurrency bug analyses for multi-threaded programs 87
9.2 Concurrency bug analyses for other programming models 89

10 Conclusion 91
10.1 Summary 91
10.2 Discussion 92

Bibliography 94

List of Figures

2.1 Execution trace of Program 2.1 7
2.2 Erroneous execution trace of Program 2.1 when `B` and `C` are aliased 8
2.3 Execution scenario and lock graph for Program 2.2 10
2.4 Execution scenario for Program 2.4 12

3.1 Memory access events and potential data race pairs of execution in Figure 2.2 . 16
3.2 Steps to reproduce data race in Program 2.1 16
3.3 Overview of the Active Testing framework 20

4.1 The regions of other threads concurrent with a shared access 28

6.1 Performance and effectiveness of DeadlockFuzzer variations 55

7.1 The three operation modes of Communication Avoiding Dynamic Analysis . . . 58
7.2 Data structures used to efficiently represent memory accesses in UPC-Thrille . . 62

8.1 Scalability of Active Testing up to 1024 cores 75
8.2 Communication and computation overhead of UPC data race detection 76
8.3 Breakdown of data race detection overhead for hybrid memory models 83
8.4 Average time for the insert and search operations in Interval Skiplist. 84
8.5 Scalability of the different sampling methods of UPC-Thrille 86

List of Tables

6.1 Results for each checker implemented in CalFuzzer 47

8.1 Results for race-directed Active Testing on a workstation 76
8.2 Statistics for UPC-Thrille with hierarchical sampling 79
8.3 Key for labels of hierarchical sampling strategies 80

List of Algorithms

3.1 RandomScheduler(c_0) 19
3.2 Phase1Scheduler(c_0) 20
3.3 CheckBeforeLockset(e) 21
3.4 CheckAfterLockset(e) 21
3.5 Phase2Scheduler(c_0, Abs) 23
3.6 CheckPartialLockset(e, Abs) 23
3.7 CheckFullLockset($e, Paused, Abs$) 24

4.1 CheckBeforeUPCRace(e) 29
4.2 CheckAfterUPCRace(e) 30
4.3 CheckBeforeDeadlock(e) 33
4.4 iGoodlock(D) 34
4.5 CheckPartialDeadlock($e, Cycle$) 35
4.6 CheckFullDeadlock($e, Paused, Cycle$) 35
4.7 CheckPartialAtomViol(e, Abs) 38
4.8 CheckFullAtomViol($e, Paused, Abs$) 38

7.1 FindRace(isl, e) 64
7.2 AddAccess(isl, e) 64
7.3 ThrilleRacerScheduler() 66
7.4 ThrilleTesterScheduler(s_1, s_2) 67

List of Programs

2.1 Example of a data race in a UPC program 7
2.2 Example of a deadlock in Java 9
2.3 Deadlock in Jigsaw 10
2.4 Example of an atomicity violation 11

4.1 Parallel knapsack implementation with data race. 26

5.1 Example code that creates objects inside a loop 43
5.2 An example program that may thrash often during reproduction 45

6.1 Atomicity violation in Java Collections 50
6.2 Initialization code causing spurious atomicity violation warnings 51
6.3 A common lock preventing atomicity violations 52
6.4 A benign atomicity violation 52

Acknowledgments

First, I would like to thank my Quals and dissertation committee—Professors Ras Bodik, Koushik Sen, David Wessel, and Kathy Yelick—for their insightful comments and feedback on my dissertation work. Special thanks goes to my advisor, Koushik Sen, for guiding me through my graduate school years for a successful completion. Without his motivation and energy, the deep intellectual discussions throughout the years, and invaluable advice for my research, I would not have made it this far.

I am grateful to my collaborators and co-authors—David Gay, Paul Hargrove, Costin Iancu, Pallavi Joshi, and Mayur Naik. Dr. Iancu, my mentor during my stay at Lawrence Berkeley National Labs, provided directions for research and technical insight for UPC-Thrille to reach the current level of maturity.

I would also like to thank Dimitra Giannakopoulou and Corina Pasareanu, my mentors at NASA Ames, for an enjoyable and stimulating internship; Jacob Burnim, Joel Galenson, Nick Jalbert, Gunho Lee, Yunsup Lee, Rhishikesh Limaye, Leo Meyerovich, Brad Miller, Daekyeong Moon, Christos Stergiou, my fellow graduate students for their feedback on my research, open-ended discussions, and collaboration on class projects; and last but not least, my family and friends for their emotional support during my long student years.

Parts of this dissertation are from previous papers [74, 49, 76] under copyright by ACM. This reuse follows the ACM Copyright Policy §2.5 Rights Retained by Authors and Original Copyright Holders. This research was supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.

Chapter 1

Introduction

Since the semiconductor industry hit the power wall [7], processor speeds are no longer doubling every two years, but instead the number of processing elements, also called *cores*, on a single chip is increasing. To make use of the multiple cores and maintain performance gains with increasingly parallel hardware, it is necessary to run programs in *parallel*, i.e. making simultaneous computations on each core. A program with many threads can be readily executed in parallel and utilize the increasing number of cores.

For high performance computing (HPC), parallelism in not new. Supercomputers with thousands of cores appeared in the 1990s, hence the need for parallel programs existed in the past. Nowadays, the ubiquity of parallelism is found from the lower end of computing on dual- and quad-core processors on smartphones to supercomputers with more than a million total cores. Thus, the necessity of writing parallel programs has become ubiquitous as well.

Writing parallel programs is harder than writing their sequential counterparts. For parallel programs, resources such as memory are shared and there may be unintended interference among the threads when accessing shared resources. Due to such interference, a parallel program can have different results in different executions depending on how the threads are scheduled. This is called *non-determinism*. Non-determinism makes it harder for the programmer to ensure the correctness of a concurrent program.

Bugs due to non-determinism are called non-deterministic bugs or *concurrency bugs*. Some common classes of concurrency bugs are data races, deadlocks, and atomicity violations. Undetected during testing, these bugs can manifest while running in a production environment. In this thesis, we develop practical tools and techniques for finding concurrency bugs in parallel programs.

1.1 Existing techniques to find concurrency bugs

There are many ways to help programmers write correct parallel programs. Model checking systematically explores all thread schedules for concurrency bugs. Programming languages for concurrency can have extended type systems and annotations that prevent writing erroneous code. Static analysis looks at the source code of a program and checks for bugs using various techniques such as data-flow analysis. Dynamic analysis examines an execution of a program to check for bugs in the execution or predict bugs that may happen in other executions. Testing runs the program with a given suite of inputs, usually with corresponding expected outputs to check against. These techniques mitigate the problem of writing incorrect concurrent programs with the following goals:

- Soundness: A technique is *sound* if it reports all bugs in the program, i.e. it does not have any false negatives.
- Completeness (Precision): A technique is *complete* or *precise* if it only reports real bugs, i.e. it does not have any false positives.
- Efficiency: A technique is *efficient* if it does not impose high overhead or have a prohibitively long analysis time.
- Scalability: A technique is *scalable* if it works for programs that have large code bases and/or run on large-scale systems.
- Automation: A technique is *automatic* if it requires little or no user intervention.

Model checking [26] verifies that a program follows its specification. By checking at each state exhaustively, model checking will either verify that the program is correct or report all violations of the specification, i.e. it is sound for bugs exhibiting behavior outside the specification. However, model checking fails to scale for large multi-threaded programs due to the exponential increase in the number of thread schedules with execution length (the state explosion problem).

Type and annotation based techniques [12, 34] help to avoid concurrency bugs at compile time by rejecting programs that do not type-check. They are sound, but they impose the burden of annotation on programmers. These annotations are often complex and/or tedious to write. Furthermore, these techniques are incomplete because some correct programs may be rejected if they cannot be well-typed in the system, requiring the programmer to rewrite the program or use workarounds.

Static analysis finds bugs in a program by reasoning about the code. Static program analyses often conservatively over-approximate bugs in the program such that bugs are not missed (sound), but they usually report many false positives (incomplete). Static analysis can be more precise using flow- and context-sensitive analysis, but this increases the computation and memory required for analysis and may not scale to large programs.

Dynamic analysis checks for bugs at run-time, by examining properties of a program during execution. Dynamic analyses are often precise, since bugs that they find are from real executions. However, dynamic analysis can add significant overhead to the program under analysis. Furthermore, dynamic analyses are unsound in nature, missing some bugs (false negatives), because it can only reason about code paths that are actually executed during analysis. Predictive dynamic analysis tries to reduce false negatives by extrapolating from an execution to predict bugs that may happen in other executions with different thread interleavings to increase coverage. These predictive techniques give both false negatives and false positives, requiring manual inspection to see if a concurrency bug is real or not. Nevertheless, they are effective in finding concurrency bugs because they can predict bugs that could potentially happen during a real execution.

Testing techniques, such as random testing or stress testing, are often widely used to check correctness of parallel programs. In random testing, threads are scheduled randomly at runtime; in stress testing, a program is executed with thousands of threads and other heavy workloads to increase the probability of hitting buggy schedules. Testing techniques are complete, efficient, and automatable. However, random testing fails to find concurrency bugs with high probability as they appear for very specific thread interleavings. Stress testing done in a given environment often fails to come up with interleavings that could happen in other environments, such as under different system loads. Stress testing does not try to explicitly control the thread schedules, but rather depends on the underlying operating system or virtual machine for thread scheduling and often ends up executing the same interleaving repeatedly.

1.2 Contributions

The main contribution of our work is that we make predictive dynamic analysis more precise and scalable while being efficient and automatic. We have developed a methodology called *Active Testing* which combines predictive dynamic analysis with testing. First, we take the predictive power of dynamic program analysis to find certain patterns during execution that may correspond to bugs. Then, like testing, we re-execute the program multiple times and observe failures and anomalous behavior. The main difference with traditional testing is that we actively control the thread schedules based on the information gathered from program analysis to direct testing towards the predicted bugs.

We have implemented a general framework for Active Testing which allows different dynamic analyses to be plugged in to find specific concurrency bugs. Existing dynamic analyses can be used to predict bugs; we have developed a few novel dynamic analyses of our own, such as a data race detector for distributed memory parallel programs because existing data race detection techniques do not scale well. A predictive dynamic analysis may generate a list of bugs with many false positives. Thus, we have also developed several *schedulers* for specific classes of concurrency bugs, which control the thread schedule to reproduce bugs. A scheduler tries to automatically reproduce the candidate bugs and report back to the programmer only the real bugs that it was able to reproduce. This removes the burden of the programmer to sift through all the reports and manually inspect if they are real bugs or not.

We have implemented Active Testing for Java, which uses threads and shared memory, and for UPC, a distributed memory parallel programming language. The implementation of the Active Testing framework for Java is called CalFuzzer. CalFuzzer includes dynamic analyses to predict and confirm data races, atomicity violations, and deadlocks. In this thesis, we only present the algorithms in CalFuzzer for prediction and confirmation of atomicity violations and deadlocks. The Active Testing framework for UPC, is called UPC-Thrille. We present the first dynamic data race detector able to handle distributed memory parallel programs and demonstrate scalability to over a thousand cores.

To make Active Testing scale to large distributed memory parallel systems, we developed several novel techniques to form the Communication Avoiding Dynamic Analysis framework. Dynamic analysis for multi-threaded programs, which usually require a central monitoring thread to collect and analyze the execution of threads, does not work well at scale when directly ported to distributed memory parallel systems. A central analysis thread incurs a huge communication overhead; our initial experiments showed that such an implementation of a central analysis thread fails to scale beyond a few nodes. We avoid a central analysis thread by distributing the analysis. We reduce communication overhead by coalescing analysis traffic to synchronization boundaries and using filtering and sampling techniques to avoid redundant information.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 starts by introducing examples of common concurrency bugs in programs. We present definitions and examples of a data race, a deadlock, and an atomicity violation. We show how the bugs may affect the execution of a program and lead to errors.

As pointed out in the previous section, various techniques for uncovering concurrency bugs have limitations such as large overheads or too many false positives. Our technique aims to overcome these shortcomings for an efficient, scalable, and precise push-button tool for finding concurrency bugs. Chapter 3 lays down the background theories and formal definitions for Active Testing, our general dynamic analysis framework to predict and confirm real concurrency bugs. We explain the two phases of Active Testing, the prediction phase and confirmation phase, which is common to the bug detection algorithms for specific concurrency bugs such as data races, deadlocks, and atomicity violations. In Chapter 4, we describe instantiations of Active Testing for these classes of bugs on top of this general two-phase framework. For data races, we use a barrier-aware lockset based algorithm to predict data races in SPMD programs. We use an augmented Goodlock algorithm to predict deadlocks in multi-threaded programs. For atomicity, we target a particular locking pattern to predict and confirm real atomicity violations.

In Chapter 5, we describe in detail our Active Testing framework for Java, named CalFuzzer. We also cover additional implementation details to increase the probability of reproducing concurrency bugs in phase II. We present two precise object abstraction techniques to pass contextual information from phase I to phase II to avoid *thrashing*, unnecessary pauses in execution. We also describe an optimization that uses *yield*s to improve the reproduction of deadlocks. Chapter 6 summarizes the evaluation of our tool on multi-threaded benchmarks. We give detailed descriptions of the bugs that our tool found and the limitations of our tool. We also present evaluation of the object abstraction techniques and yielding optimization, showing how they affect the reproduction of deadlocks.

To make Active Testing scale to large distributed memory systems, we structured the analyses for better load distribution among the nodes and less communication over the network. Chapter 7 presents the Communication Avoiding Dynamic Analysis framework and our Active Testing implementation for UPC, named UPC-Thrille. We focus on the implementation details of a general framework for distributed memory parallel programs and a data race detector for UPC. We also extend UPC-Thrille to handle hybrid memory models that deal with multiple abstractions of the same memory space. We classify the three different overheads associated with finding data races on distributed memory systems—instrumentation, computation, and communication. We present a hierarchical sampling technique that significantly reduces the instrumentation and computation overheads. Chapter 8 shows the results for the evaluation of our tools on distributed memory parallel benchmarks, with a description of the bugs that we found. We present results that show Active Testing can efficiently and scalably find concurrency bugs for distributed memory systems. We also compare our hierarchical sampling technique with others to emphasize its effectiveness.

In Chapter 9, we discuss related work. Chapter 10 concludes the thesis with a brief summary and discusses future research directions.

Chapter 2

Common Concurrency Bugs

In this chapter, we go through a brief overview of the common concurrency bugs in programs through examples. A data race happens when two threads try to access the same memory location with an unspecified order. A deadlock occurs when a system cannot make progress because all threads are waiting for a resource held by some other thread. Atomicity violations are bugs that happen when a programmer's assumption about the indivisibility of an operation is broken.

2.1 Data race through an example

One of the most common concurrency errors due to lack of synchronization is a *data race*. A parallel program has a data race if the program can reach a state during execution where two threads are about to access the same memory location and at least one of the accesses is a write. Depending on the order of access, i.e. read after write or write after read (or in case both of the accesses are writes, whichever one occurred last), the program may exhibit different behavior which in some cases may lead to erroneous executions. This *non-deterministic* behavior may result in bugs that happen only under very specific schedules, and thus hard to detect.

Data races are also harmful in that they may break *sequential consistency* [54] depending on the memory model of the underlying system. Sequential consistency is the notion that every execution of a program on multiple processors should be equivalent to some interleaved execution on a single processor, i.e. the operations have a single and total sequential order. If sequential consistency is violated, it may lead to unexpected and undefined behaviors of the program. [10]

Consider Program 2.1, a matrix-vector multiply routine written in Unified Parallel C (UPC). UPC [16] is an extension of the C language with a shared global address space and data parallel constructs. The `shared` keyword specifies pointers and arrays to shared memory (e.g. the input variables `A, B, C` are declared to be shared). Non-shared pointers and arrays, such as `sum` in line 4 are local and distinct to each thread. The `upc_forall`

```
1  void matvec(shared [N] double A[N][N], shared double B[N],
2              shared double C[N])
3  {
4    double sum[N];
5    upc_forall(int i = 0; i < N; i++; &C[i])
6    {
7      sum[i] = 0;
8      for(int j = 0; j < N; j++)
9        sum[i] += A[i][j] * B[j];
10   }
11   upc_forall(int i = 0; i < N; i++; &C[i])
12     C[i] = sum[i];
13 } // assert(C == A * B)
```

Program 2.1: Example of a data race in a UPC program

Initially,

A = $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and B = $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

```
             Thread 1                          Thread 2
1. sum[0] = 0;                     a. sum[1] = 0;
2. sum[0] += A[0][0] * B[0];       b. sum[1] += A[1][0] * B[0];
3. sum[0] += A[0][1] * B[1];       c. sum[1] += A[1][1] * B[1];
4. C[0] = sum[0];                  d. C[1] = sum[1];
```

Output

C = $\begin{pmatrix} 3 \\ 7 \end{pmatrix}$

Figure 2.1: Execution trace of Program 2.1

statement is a parallel-for loop that runs the loop body in different threads. The fourth argument denotes where to run the loop body. For example, the `upc_forall` loops in lines 5 and 11, run the loop body in the thread that owns the memory corresponding to the ith element of `C`. We give a more detailed description of UPC in Chapter 4.1.

If we run the example on two threads with $N = 2$, the execution trace of all memory loads and stores with arithmetic operations would look like Figure 2.1. The two threads

Initially,

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ and } B = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```
            Thread 1                                 Thread 2
1. sum[0] = 0;
                                         a. sum[1] = 0;
2. sum[0] += A[0][0] * B[0];
3. sum[0] += A[0][1] * B[1];
4. B[0] = sum[0];
                                         b. sum[1] += A[1][0] * B[0];
                                         c. sum[1] += A[1][1] * B[1];
                                         d. B[1] = sum[1];
```

Output

$$C = \begin{pmatrix} 3 \\ 13 \end{pmatrix}$$

Figure 2.2: Erroneous execution trace of Program 2.1 when `B` and `C` are aliased

execute in parallel, running their portion of the loop bodies. Since the writes made by the threads are to addresses distinct from each other, the result of the program is correct no matter how the shared memory accesses interleave with each other.

However, if the routine is called with `B` and `C` aliased (i.e., for an in-place multiplication), there are two data races: $\langle$ 3, d $\rangle$ (because 3 is a read from Thread 1, d is a write from Thread 2, and `&B[1] == &C[1]`) and $\langle$ 4, b $\rangle$ (similarly). One common misconception of the `upc_forall` loop is that all the loop bodies must terminate to continue (i.e., there is an implicit barrier at the end of the `upc_forall` statement), which is not true for UPC. Nothing prevents Thread 1 from executing 4 before Thread 2 finishes b or Thread 2 from executing d before Thread 1 finishes 3. Figure 2.2 is an execution trace where Thread 1 executes 4 then Thread 2 executes b. After Thread 1 makes the update to `B` (because it is aliased to `C`) at 4, Thread 2 reads the new value when computing the inner product at b, resulting in the wrong value.

This example illustrates several challenges for finding data races: 1) the number of possible interleavings can be very large; and 2) the number of memory access to analyze increases with the problem size. Even for such a small program with only two threads, there are 70 possible interleavings.[1] Five interleavings have 4 executing before b (e.g., Figure 2.2) and 10

[1]There are 5 places where a through d can go: before 1, between 1 and 2, between 2 and 3, between 3 and 4, and after 4. Picking from the 5 locations 4 times with replacement corresponds to $\binom{5+4-1}{4} = \binom{8}{4} = 70$.

```
Initially,
Object l1 = new Object();
Object l2 = new Object();

        Thread 1                   Thread 2
1  synchronized(l1) {        6  synchronized(l2) {
2    synchronized(l2) {      7    synchronized(l1) {
3      ...                   8      ...
4    }                       9    }
5  }                        10  }
```

Program 2.2: Example of a deadlock in Java

interleavings have d executing before 3. Thus, even though the program may be incorrect, 55 out of 70 interleavings (78.6%) execute the program computing the correct result. This illustrates why such bugs can be easily missed with stress testing or random testing, and why it requires a large amount of work to check all the interleavings exhaustively.

Instead of checking all interleavings, dynamic data race detectors like ours in Chapter 4.1 analyze the memory accesses to predict data races. However, if we run the program on many threads with a large data set, there may be an overwhelming number of memory accesses to analyze. We have developed novel sampling and filtering techniques to reduce the number of accesses to consider while retaining the ability to predict most or all data races in the program.

2.2 Deadlock through an example

Data races and other concurrency errors can be prevented with synchronization. One common synchronization primitive is a lock (also called a mutex, for **mut**ual **ex**clusion). A lock is an object that can be held by only one thread at a time, which can be used for mutual exclusion of threads in critical regions of code.

However, locks must be used with care; certain locking patterns may run into a *deadlock*, where all threads are blocked waiting to acquire a lock which is already held by some other thread, thus no progress is made. Consider example Program 2.2. Here, a deadlock situation can arise in the scenario illustrated in Figure 2.3.

Jigsaw, shown in Program 2.3, is a webserver which has this locking pattern that could deadlock. The nested synchronization corresponds to the statements in lines 7 and 8 for the main `httpd` thread and lines 4 and 6 for a `SocketClient` thread. The `SocketClientFactory` object corresponds to lock `l1` in Program 2.2 and the `SocketClientState` object in line 2 corresponds to lock `l2`. When the server shuts down, it calls cleanup code that shuts down the `SocketClientFactory`. The shutdown code holds a lock on the factory at line 7, and in turn attempts to acquire the lock on `csList` at line 8. On the other hand, when a

1. Thread 1 successfully acquires lock `l1` (line 1).
2. Thread 2 successfully acquires lock `l2` (line 6).
3. Thread 1 is waiting to acquire lock `l2` (line 2).
4. Thread 2 is waiting to acquire lock `l1` (line 7).
5. Both threads are blocked and cannot progress — a deadlock has occurred.

$$\begin{tikzcd}
\texttt{l1} \arrow[r, bend left, "\text{Thread 1}"] & \texttt{l2} \arrow[l, bend left, "\text{Thread 2}"]
\end{tikzcd}$$

Figure 2.3: Execution scenario and lock graph for Program 2.2

```
 1  class SocketClientFactory {
 2    SocketClientState csList;
 3    boolean clientConnectionFinished(...) {
 4      synchronized (csList) { decrIdleCount(); }
 5    }
 6    synchronized boolean decrIdleCount() {...}
 7    synchronized void killClients(...) {
 8      synchronized (csList) {...}
 9    }
10    void shutdown() {
11      killClients(...);
12    }
13  }
14
15  Thread 1: httpd                    Thread 2: SocketClient
16  SocketClientFactory factory;       SocketClientFactory pool;
17  factory.shutdown();                pool.clientConnectionFinished(...);
```

Program 2.3: Deadlock in Jigsaw

`SocketClient` is closing, it also calls into the factory to update a global count. In this situation, the locks are acquired in the opposite order: the lock on `csList` is acquired first at line 4, and then on the factory at line 6. Thus, when both events happen simultaneously, the webserver may deadlock and not shut down cleanly.

A deadlock can happen non-deterministically: in some cases, a programs runs to completion, and under some other thread schedule a deadlock occurs and can be detected by observing that a set of blocked threads are all waiting for a lock held by another thread in the set. In our running example (Program 2.2), a deadlock does not happen if the order of lock acquires is 1 2 6 7.

```
public class Consumer {
  private LinkedList buffer;
  public synchronized void consume() {
    if(!buffer.isEmpty()) {
      Object data = buffer.remove();
      System.out.println(data.toString());
    }
  }
}
public class LinkedList ... {
  public synchronized boolean isEmpty() { ... }
  public synchronized Object remove() { ... }
  ...
}
```

Program 2.4: Example of an atomicity violation

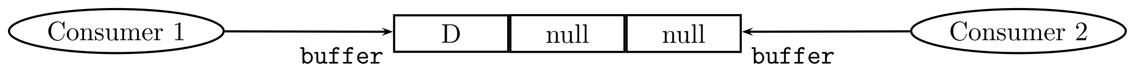
Thus, we need tools that check if a program could deadlock or not. A static deadlock detector [66, 103] can predict deadlocks by inspecting the locking behavior in code. We can also detect deadlocks with runtime monitoring by creating lock graphs. A lock graph has locks as nodes and the lock ordering relation as edges. Whenever a thread t attempts to acquire $lock_2$ when the last acquired lock still held is $lock_1$, an edge labeled t is added from node $lock_1$ to $lock_2$. For example, the diagram on the right of Figure 2.3 is the lock graph for Program 2.2. If there is a cycle in the lock graph with distinct labels along the edges, there is a potential for a deadlock. In our example program, there is a cycle $\langle \texttt{l1}, \texttt{l2}, \texttt{l1} \rangle$ and corresponds to the deadlock in the scenario above.

Static and predictive dynamic deadlock detectors often report too many false positives, i.e. they are imprecise. Active Testing reports only the deadlocks that it reproduces, reducing the burden of the programmer to confirm whether each report is real or not. For example, out of the 283 deadlocks in Jigsaw predicted by a dynamic deadlock analysis, we report only 29 of those that Active Testing was able to reproduce.

One challenge for reproducing deadlocks from lock graphs is that it is hard to know at run-time which lock acquires lead to a deadlock, as locks can be acquired in many contexts. We present a dynamic deadlock prediction algorithm in Chapter 4.2 with precise object abstractions that provides contextual information to help reproduce deadlocks.

2.3 Atomicity violation through an example

The absence of data races in a parallel program is not sufficient to ensure that it is free of non-deterministic bugs. *Atomicity* is a property of multi-threaded programs that enforces



Statement	Object	Returns
Consumer 1 calls `isEmpty()`	`buffer`: | D | null | null |	`false`
Consumer 2 calls `isEmpty()`	`buffer`: | D | null | null |	`false`
Consumer 1 calls `remove()`	`buffer`: | D | null | null |	D
Consumer 2 calls `remove()`	`buffer`: | null | null | null |	null
Consumer 2 calls `toString()`	`data`: null	Null Pointer Exception

Figure 2.4: Execution scenario for Program 2.4 where Consumer 1 and Consumer 2 share the buffer and both call `consume()`

non-interference. A block of code in a multi-threaded program is atomic if for every possible interleaved execution of the program there exists an equivalent execution with the same overall behavior where the atomic block is executed serially. In other words, the execution of the atomic block behaves as if it is not interleaved with actions of other threads. Therefore, if a code block is atomic, the programmer can assume that the execution of the code block by a thread cannot be interfered by any other thread. This helps programmers reason about atomic code blocks sequentially.

Consider Program 2.4 as an example. A `Consumer` class has an associated buffer where objects are stored for processing. Before consuming a data object, the buffer is first checked to see if there are any elements to process (line 4). Only when the buffer is non-empty, an element is removed and printed. All methods of the `LinkedList` class are synchronized, thus even if multiple consumers share the same buffer, there is no data race involved.

However, there may still be a problem in this program for some interleavings. In the execution scenario in Figure 2.4, two Consumer objects share a common buffer and both are calling the `consume()` method when there is only one item in the buffer. If the statements are executed in the given order, both Consumer objects will check if the buffer is empty and get a negative answer. Thus, they both try to remove an object from the buffer, but only one of them will get a data object (Consumer 1 in this case) and the other gets a null pointer. When the null data object is dereferenced by Consumer 2, an exception occurs.

This scenario is an exemplar "*time of check to time of use*" (TOCTTOU) bug, which is caused by changes in the system between the check of a state and the use of that state [94]. What we intended was for the check and remove from the buffer to be one indivisible operation, i.e. an *atomic* operation. This atomicity property has been violated in this case.

In most modern multi-threaded programming languages, atomicity is indirectly achieved through the use of locks. However, the use of locks cannot always ensure a code block to be atomic. Program 2.4 is an example of this problem. Thus, many existing program analysis techniques use heuristics to infer atomic blocks from locks and predict atomicity violations. This heuristic may lead to many false positives which increases the burden on the programmer to check each report manually. In Chapter 4.3, we present a dynamic atomicity violation analysis which reports only the real atomicity violations that actually happen in an execution, reducing this burden of confirming real atomicity violations.

Chapter 3

Active Testing

Numerous program analysis techniques [40, 72, 31] have been developed to *predict* concurrency bugs in multi-threaded programs by detecting violations of commonly used synchronization idioms. For instance, accesses to a memory location without holding a common lock are used to predict data races on the location, and cycles in the program's lock order graph are used to predict deadlocks. However, these techniques often report many false warnings because violations of commonly used synchronization idioms do not necessarily indicate concurrency bugs. Manually inspecting these warnings is often tedious and error-prone.

Active Testing [87, 74, 49] is a technique for finding *real* bugs in concurrent programs. Active Testing uses a randomized thread scheduler to verify if warnings reported by a predictive program analysis are real bugs. The technique works as follows. In the first phase, Active Testing uses an existing predictive off-the-shelf static or dynamic analysis, such as Lockset [84, 72], Atomizer [31], or Goodlock [40], to compute potential concurrency bugs. Each such potential bug is identified by an abstract state. For example, in the case of a data race, the abstract state consists of a pair of program statements that could potentially race with each other in some execution. In the second phase, for each potential concurrency bug, Active Testing runs the given concurrent program under random schedules. Further, Active Testing biases the random scheduling by pausing the execution of any thread when the thread partially satisfies the abstract state describing the potential concurrency bug. After pausing a thread, Active Testing also checks if a set of paused threads could exhibit a real concurrency bug. For example, in the case of a data race, Active Testing checks if two paused threads are about to access the same memory location and at least one of them is a write. Thus, Active Testing attempts to force the program to take a schedule in which the concurrency bug actually occurs.

In the following sections, we show informally how Active Testing works with a simple example. We also define the formal model of concurrent systems that we test. The model describes the minimal interface of concurrent systems used in our Active Testing algorithms. We describe in more detail the two phases of Active Testing and their generic schedulers.

3.1 Active Testing through an example

Consider the execution trace from Figure 2.2 (page 8) again. This execution trace of Program 2.1 (page 7) with `B` and `C` aliased is of great interest to us for two reasons: 1) it clearly shows that there is a race, and 2) it also shows that the result computed is incorrect. There is a race between events 4 and b, because the write from Thread 1 and the read from Thread 2 to the same address `B[0]` are temporally next to each other. This is ideally the kind of trace we would like to obtain from testing, but a trace that a random scheduler can rarely produce.

Active Testing tries to automatically create such execution traces that exhibit a data race. First, we run an imprecise dynamic analysis on an execution of the program to find potential data races that are present in the program. The analysis is a variant of a lockset based algorithm [84, 18, 72, 100]. The analysis checks if two threads could potentially access a memory location without holding a common lock. Specifically, the analysis observes all memory accesses that happen during an execution of the program and records the locks held during each such access. If there exists two accesses to the same memory location by different threads, a common lock is not held during the accesses, and at least one of the accesses is a write, then the analysis reports a potential data race. The analysis reports the pairs of statements where the threads access the memory location, respectively. Formally, the set of potential data races pairs reported by the analysis is defined as follows:

Definition 3.1 (Set of Potential Data Race Pairs: $\mathcal{D}_{P,E}$). *Given an execution E of a program P, let us denote a shared memory access* event *by a thread in the execution by $e = (m, t, l, a, s)$, where*

1. *m is the memory address range that is being accessed,*
2. *t is the thread accessing the memory address range.*
3. *l is the set of locks held by t at the time of access,*
4. *$a \in \{READ, WRITE\}$ is the access type, and*
5. *s is the label of the program statement that generates the memory access event.*

Let $\mathcal{A}_{P,E}$ be the set of all shared memory access events in the execution E. Then the set of potential data race pairs reported by the analysis is

$$\mathcal{D}_{P,E} = \{(s_1, s_2) \mid \exists e_1, e_2 \in \mathcal{A}_{P,E} \text{ such that } e_1 = (m_1, t_1, l_1, a_1, s_1) \ \wedge\ e_2 = (m_2, t_2, l_2, a_2, s_2) \\ \wedge\ m_1 \cap m_2 \neq \emptyset\ \wedge t_1 \neq t_2\ \wedge\ l_1 \cap l_2 = \emptyset \wedge\ (a_1 = WRITE \vee a_2 = WRITE)\}\ .$$

In our example execution trace of Figure 2.2, $\mathcal{A}_{P,E}$ and $\mathcal{D}_{P,E}$ are shown in Figure 3.1. Note that a race pair in $\mathcal{D}_{P,E}$ reported by the analysis can be a false warning because the analysis does not check if the two accesses are ordered by a synchronization operation. The analysis simply checks if the program adheres to the idiom that every memory access is consistently protected by a lock. As such, the analysis can report data races that did not actually happen in the execution E, but could happen in a different execution E' of the

$$\begin{array}{rl}
\mathcal{A}_{P,E} = & \{\ (\ [\text{A},\ \text{A}+8),\ \text{T1},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{A}+8,\ \text{A}+16),\ \text{T1},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B},\ \text{B}+8),\ \text{T1},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B}+8,\ \text{B}+16),\ \text{T1},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B},\ \text{B}+8),\ \text{T1},\ \texttt{WRITE},\ \{\},\ 12\ ), \\
& \phantom{\{}\ (\ [\text{A}+16,\ \text{A}+24),\ \text{T2},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{A}+24,\ \text{A}+32),\ \text{T2},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B},\ \text{B}+8),\ \text{T2},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B}+8,\ \text{B}+16),\ \text{T2},\ \texttt{READ},\ \{\},\ 9\ ), \\
& \phantom{\{}\ (\ [\text{B}+8,\ \text{B}+16),\ \text{T2},\ \texttt{WRITE},\ \{\},\ 12\ )\ \} \\
\hline
\mathcal{D}_{P,E} = & \{\ (9,\ 12)\ \}
\end{array}$$

Figure 3.1: Memory access events and potential data race pairs of execution in Figure 2.2

Statements 9 and 12 may race

	Statement of T1	Statement of T2	Paused	Scheduler action
a	7:`sum[0]=0`		-	pick T1 and execute
b		7:`sum[1]=0`	-	pick T2 and execute
c		(9:`sum[1]+=A[1][0]*B[0]`)	-	pick T2 and pause
d	9:`sum[0]+=A[0][0]*B[0]`		T2	pick T1 and execute
e	9:`sum[0]+=A[0][1]*B[1]`		T2	pick T1 and execute
f	(12:`B[0]=sum[0]`)	(9:`sum[1]+=A[1][0]*B[0]`)	T2	data race created
g	12:`B[0]=sum[0]`		-	pick T1 and execute
h		9:`sum[1]+=A[1][0]*B[0]`	-	pick T2 and execute
i		9:`sum[1]+=A[1][1]*B[1]`	-	pick T2 and execute
j		12:`B[1]=sum[1]`	-	pick T2 and execute

Figure 3.2: Steps to reproduce data race in Program 2.1

program under a different thread schedule. This predictive power of the analysis is crucial for increasing the coverage of our Active Testing technique.

To confirm that this prediction is true, we re-execute the program with a random scheduler, but with knowledge that statements 9 and 12 may race. We use a scheduler that serializes the execution, i.e. it picks one thread at a time to execute. Figure 3.2 shows the steps of the scheduler. At step a, the scheduler randomly picks T1 and executes the statement normally. At step b, the scheduler randomly picks T2 and executes the statement. At step c, the scheduler picks T2, but since it is about to execute statement 9 which may race with statement 12, it pauses the thread so that some other thread can reach statement 12. At step d, the scheduler can only pick T1, because T2 is paused. It is also about to execute a statement that may race with statement 12, but pausing T1 would stall the system by pausing all threads. Thus we execute the statement at T1, similarly at step e. At step f, T1 is about to execute statement 12. This is a statement that may race with statement 9, where T2 is paused. We check if the two threads are in race, and they are since T1 is about to write to `B[0]` and T2 is about to read from `B[0]`. Thus, we have created a real data race. From this point, we unpause all threads and continue to observe the remaining execution for any anomalies. After going through steps g–j, we have created an execution trace identical to Figure 2.2, showing the data race and the incorrect result.

The example illustrates the practical challenges of Active Testing. In phase I, 1) managing overhead of data collection at every memory access and synchronization operation; 2) managing overhead of exchanging the data collected between threads; and 3) efficiently reasoning about the information gathered to predict concurrency bugs. We address these challenges by presenting a hierarchical sampling technique for data race detection in Chapter 7.3; structuring scalable dynamic analyses to avoid communication in Chapter 7.1; and efficient algorithms for predicting concurrency bugs in Chapter 4.

In phase II, we need to strategically pick when to pause which thread. If done naively, we may incur large overheads by pausing threads unnecessarily at irrelevant points and reduce the probability of reproducing concurrency bugs. We developed techniques for precise object abstractions, discussed in Chapter 5.2, to help reproduce concurrency bugs by providing contextual information for when to pause threads.

3.2 Concurrent system model

We use a simple and general model of a concurrent system to describe our Active Testing methodology. This model can be used for shared memory systems, distributed memory parallel systems, and others as well. We consider a concurrent system to be composed of a finite number of threads. Each thread executes a sequence of labeled statements. A thread communicates with other threads using shared memory. At any point of program execution, a concurrent system is in some *state*, which internally contains the contents of shared memory and bookkeeping information for threads. Starting at the initial state c_0, a concurrent system evolves from one state to another when a thread executes a statement. We leave the detailed operational semantics of statements unspecified; for analysis and testing purposes, we do require a few operations to query and control the system. We assume that at any state of the concurrent system, we can make the following queries and operations.

- $\textsc{Enabled}(c) \rightarrow T$: Returns the set of threads T that are enabled in state c. A thread is disabled if it is waiting to acquire a lock already held by some other thread or waiting at a barrier.

- $\textsc{Alive}(c) \rightarrow T$: Returns the set of threads T that are alive (i.e., has not terminated) in state c. The state c is in a *stall state* when some threads are alive but none are enabled (i.e., $\textsc{Enabled}(c) = \emptyset \wedge \textsc{Alive}(c) \neq \emptyset$).

- Event$(c, t) \rightarrow e$: Returns the event e, which would be generated if thread t executes its next statement in state c.
- Execute$(c, t) \rightarrow c'$: Returns the new state c' after thread t executes its next statement in state c.

We consider the following events that are generated as statements are executed in the concurrent system. An event is generated as a state transitions into the next state.

Definition 3.2 (Events). *As a concurrent system evolves its state through execution of statements, the following externally observable transition* events *are generated.*

- `MEM`(t, s, m, a): Thread t at statement labeled s (same for all other events below) accessed memory range m, consisting of the start and end addresses, where the access type a is either `READ` or `WRITE`.
- `LOCK`(t, s, l): Thread t acquired lock l.
- `UNLOCK`(t, s, l): Thread t released lock l.
- `BARRIER_NOTIFY`(t, s): Thread t notified other threads that the barrier is ready to cross.
- `BARRIER_WAIT`(t, s): Thread t finished waiting at a barrier for notifications from other threads.
- `ATOMIC_ENTER`(t, s): Thread t entered an atomic section.
- `ATOMIC_EXIT`(t, s): Thread t left an atomic section.
- τ : any other internal events that we do not handle.

Given the above model of concurrent programs, we define a *happens-before* [53] relation to formally describe our bug detection algorithms. The *happens-before* relation requires the notion of *independence* of transitions.

Definition 3.3 (Independent Transitions). *If two transitions in a concurrent system do not interact with each other, then we call them* independent.

For example, a transition denoting the acquire of a lock l_1 by a thread t_1 is independent of a transition denoting the acquire of a lock l_2 by another thread t_2, if l_1 and l_2 are different locks.

Definition 3.4 (Dependent Transitions). *Two transitions are said to be* dependent*, if they are not independent.*

Algorithm 3.1: RandomScheduler(c_0)

```
Input: the initial state $c_0$
1 $c := c_0$;
2 while Enabled($c$) $\neq \emptyset$ do
3     $t$ := a random thread in Enabled($c$) ;
4     $c$ := Execute($c, t$) ;
5 if Alive($c$) $\neq \emptyset$ then
6     print "ERROR: system stall";
```

Transitions from the same thread are always dependent on each other. Similarly, the acquire or release of a lock by one thread is dependent on the acquire or release of the same lock by another thread. Two accesses (i.e. read or write) of a memory location are dependent if at least one of the accesses is a write.

A sequence of transitions represents the execution of a concurrent system. Specifically, $\tau = t_1 t_2 \ldots t_n$ is a *transition sequence* if there exists states $c_0, c_1, \ldots, c_n$ such that c_0 is the initial state and

$$c_0 \xrightarrow{t_1} c_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} c_n \ .$$

Definition 3.5 (Happens-before: $\preceq$)**.** *The* happens-before *relation $\preceq$ for a transition sequence $\tau = t_1 t_2 \ldots t_n$ is defined as the smallest relation such that*

1. *if t_i and t_j are dependent and $1 \leq i \leq j \leq n$, then $t_i \preceq t_j$, and*
2. *$\preceq$ is transitively closed.*

Thus $\preceq$ is a partial order relation.

We use a *scheduler* to get an execution trace of a concurrent system. Algorithm 3.1 is an example of a simple random scheduler which randomly selects an enabled thread and executes a single statement of that thread. The main loop in line 2 continues execution of a statement from a random thread that is enabled. When no thread is enabled, the scheduler makes the final check that all threads have terminated in line 5. If this is not the case, then the system is in a state that cannot make progress while some thread has not yet terminated. We call this error state a *system stall*.

Figure 3.3 shows an overview of the Active Testing framework. In the following sections, we describe generic schedulers for Phase I (Section 3.3) and Phase II (Section 3.4) of Active Testing. These schedulers require functions specific to predicting and confirming a class of concurrency bug. We use lockset-based data race detection as an example in the next sections. Chapter 4 describes these functions for predicting and confirming data races, deadlocks, and atomicity violations.

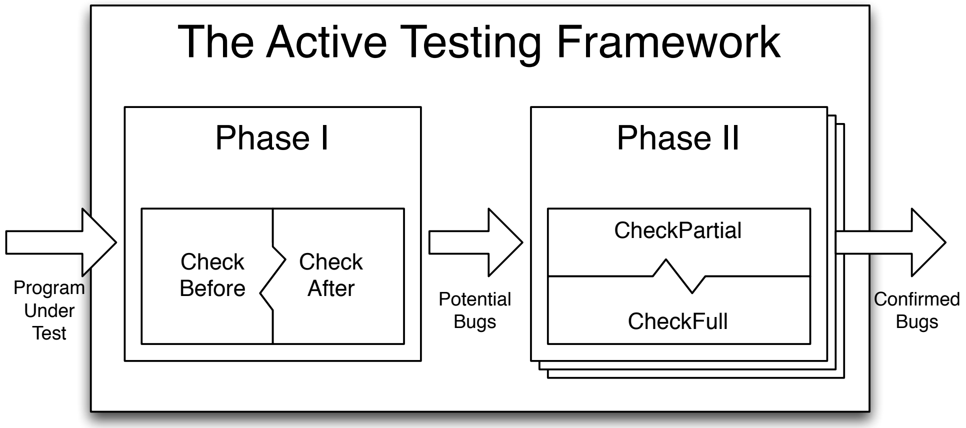


Figure 3.3: The Active Testing Framework takes a concurrent program under test and returns confirmed bugs. Internally, it finds potential bugs in the first phase and confirms and reports real bugs in the second phase.

Algorithm 3.2: PHASE1SCHEDULER(c_0)

```
Input: the initial state $c_0$
Output: set of abstract states
1  $c := c_0$;
2  $Abs := \emptyset$;
3  while ENABLED($c$) $\neq \emptyset$ do
4      $t :=$ a random thread in ENABLED($c$) ;
5      $e :=$ EVENT($c, t$) ;
6      $Abs := Abs \cup$ CHECKBEFOREX($e$) ;
7      $c :=$EXECUTE($c, t$);
8      $Abs := Abs \cup$ CHECKAFTERX($e$) ;
9  if ALIVE($c$) $\neq \emptyset$ then
10     print "ERROR: system stall";
11 return $Abs$;
```

3.3 Phase I: Prediction

The Phase I scheduler shown in Algorithm 3.2 is a general way to dynamically observe and gather information about the execution of a program. Each dynamic analysis may have a variety of data structures and algorithms, but we generalize this problem as inserting two functions before and after each statement execution. The two function calls are at lines 6 and 8 in PHASE1SCHEDULER. Each of these functions return *abstract states* which describe certain states of the program that *potentially* correspond to bugs. We collect these states

Algorithm 3.3: CheckBeforeLockset(e)

```
  Initially: $Acc = \emptyset$, the set of all memory accesses in current execution,
             $\forall t \in T.\ L(t) = \emptyset$, the set of locks held by thread t
  Input: Event $e$ the concurrent system is about to generate
1 switch $e$ do
2     case $MEM(t, s, m, a)$
3         $Acc := Acc \cup (m, t, L(t), a, s)$;
4         return $\{(s, s') \mid \exists(m', t', l', a', s') \in Acc.\quad m \cap m' \neq \emptyset \ \wedge\ t \neq t'$
          $\wedge\ L(t) \cap l' = \emptyset \ \wedge\ (a = WRITE \ \vee\ a' = WRITE)\}$;
5     case $LOCK(t, s, l)$
6         $L(t) := L(t) \cup l$;
7 return $\emptyset$;
```

Algorithm 3.4: CheckAfterLockset(e)

```
  Initially: $L$ is shared with CheckBeforeLockset
  Input: Event $e$ the concurrent system has just generated
1 switch $e$ do
2     case $UNLOCK(t, s, l)$
3         $L(t) := L(t) \setminus l$;
4 return $\emptyset$;
```

until the end when we pass off the set of abstract states to the next phase, which confirms whether these states can occur in a real execution of the program and correspond to real concurrency bugs.

For example, let us consider the lockset-based predictive data race analysis from Section 3.1. The analysis implemented in our framework works as follows. We use Phase1Scheduler (Algorithm 3.2) with the functions CheckBeforeLockset (Algorithm 3.3) and CheckAfterLockset (Algorithm 3.4). This is an online analysis that checks for potential data races at every memory access and returns the statement pairs that may race as abstract states describing the bug. Before each memory access (lines 2–4 of CheckBeforeLockset), we add the access event into the set of all memory accesses so far. We check if there is a previous event in the set that accessed an overlapping memory address ($m \cap m' \neq \emptyset$), from a different thread ($t \neq t'$), without a common lock held ($L(t) \cap l' = \emptyset$), and either the previous or current access is a write ($a = \texttt{WRITE} \ \vee\ a' = \texttt{WRITE}$). If such an access is found, we return the abstract state that describes the data race: the pair of statements that make the accesses. To keep track of the locks held during a memory access, we update the set of locks held by each thread before acquiring a lock (lines 5–6 of CheckBeforeLockset) and after releasing a lock (lines 2–3 of CheckAfterLockset). We need not consider any events other than memory accesses and locking related events.

We show in Chapter 4 other algorithms predicting concurrency bugs written in such manner: CheckBeforeUPCRace and CheckAfterUPCRace for finding data races and CheckBeforeDeadlock for finding deadlocks.

3.4 Phase II: Confirmation

Once we obtain a set of abstract states from Phase I, we check one by one whether these abstract states can occur in an execution and correspond to a real concurrency bug. An abstract state tries to capture the essence of a concurrency bug: two or more threads that are in certain concrete states simultaneously. For example, in our lockset-based data race detection example, a pair of statements is an abstract state that represents racy accesses. Another example is a lock graph containing a cycle to represent a potential deadlock.

One reason why concurrency bugs non-deterministically occur only in certain executions is because of the simultaneity condition: only particular thread schedules allow these threads to be in their respective states at the same time. By controlling the scheduler, we make a best effort to recreate these buggy states by delaying certain threads to meet the simultaneity condition. However, recreating the abstract state is not always enough. We need to make sure that the abstract state created corresponds to a real concurrency bug. Once we have two threads about to execute the statements in the pair predicted as a race, we need to confirm that they are indeed in race by checking if the two accesses are to overlapping memory addresses.

The Phase II scheduler shown in Algorithm 3.5 is a general way to recreate a state corresponding to concurrency bugs. As in the Phase I scheduler, it requires functions particular to a class of concurrency bug that check states of an execution. CheckPartialX returns true when a thread is in a state which could potentially be part of a concurrency bug state. For example, CheckPartialLockset (Algorithm 3.6) returns true when the next event of a thread is a memory access predicted to be in a race. CheckFullX examines a set of threads to see if they are in a real concurrency bug state. In CheckFullLockset (Algorithm 3.7), we make sure that we are in a state where a real data race is about to happen by checking that the memory addresses overlap and the accesses each correspond to one of the statements in the pair.

Phase2Scheduler guides the execution of a concurrent system to reach a potential buggy state (*Abs*) reported from Phase I. We randomly pick and execute enabled threads that are not *Paused* by the analysis (line 7). Certain threads are paused to increase the chance of threads being in the abstract state simultaneously. Line 5 is a safeguard that prevents the scheduler from artificially stalling the system by pausing all enabled threads.

For the thread that was selected, we check whether the next event of the thread partially satisfies the abstract state (line 9). If this is true, we check if the current thread along with the paused threads correspond to a real concurrency bug (line 10). We report an error if we are in such a state. Since we have confirmed the bug, we unpause all threads and

Algorithm 3.5: PHASE2SCHEDULER(c_0, *Abs*)

```
Input: the initial state $c_0$ and the abstract state $Abs$ to check for
$c := c_0$;
$Paused := \emptyset$;
$prob := 1.0$;
while ENABLED($c$) $\neq \emptyset$ do
    if ENABLED($c$) $= Paused$ then
        Remove a random thread from $Paused$;        // Prevents artificial stall
    $t :=$ a random thread in ENABLED($c$) $\setminus Paused$ ;
    $e :=$ EVENT($c, t$) ;
    if CHECKPARTIALX(e, Abs) then
        if CHECKFULLX(e, Paused, Abs) then
            Report error;
            $Paused := \emptyset$;
            $prob := 0$;
        else if RANDOM() $< prob$ then
            $Paused := Paused \cup T$;
            $prob := prob \times$ BACKOFF;
            continue;
    $c :=$ EXECUTE($c, t$);
if ALIVE($c$) $\neq \emptyset$ then
    print "ERROR: system stall";
```

Algorithm 3.6: CHECKPARTIALLOCKSET(e, *Abs*)

```
Input: Event $e$ the concurrent system is about the generate and the abstract state
       $Abs$ to confirm
$(s_1, s_2) := Abs$;
switch $e$ do
    case MEM($t, s, m, a$)
        return $s = s_1 \lor s = s_2$;
    otherwise
        return $false$;
```

prevent pausing threads hereafter such that we can observe the remaining execution for any anomalies.

If the event partially satisfies the abstract state but the full state does not correspond to a concurrency bug, it may be that all the threads required for the bug have not reached their certain states yet. Thus, we add the selected thread to the *Paused* set so that other threads may catch up. Since the constant pausing and unpausing of threads may reduce

Algorithm 3.7: CheckFullLockset($e, Paused, Abs$)

```
Input: Event $e$, the current set of $Paused$ threads, and the abstract state $Abs$ to
       confirm
1 $(s_1, s_2) := Abs$;
2 switch $e$ do
3     case $MEM(t, s, m, a)$
4         return $\exists t' \in Paused.$ Event$(t') = MEM(t', s', m', a') \ \wedge \ (m \cap m' \neq \emptyset) \wedge$
          $((s = s_1 \wedge s' = s_2) \vee (s = s_2 \wedge s' = s_1))$;
5     otherwise
6         return $false$;
```

execution speed drastically, we employ an *exponential backoff* scheme to randomly decide whether to delay the thread or not (line 14). Starting with a pause probability of 1, we gradually decrease this probability exponentially after each attempt to recreate the buggy state by delaying a thread.

In the next chapter, we describe two more bug reproduction algorithms using functions that fit Phase2Scheduler. CheckPartialDeadlock and CheckFullDeadlock reproduce deadlocks; CheckPartialAtomViol and CheckFullAtomViol detect and reproduce atomicity violations in a single phase.

Chapter 4

Instantiations of Active Testing

In this chapter, we present algorithms used in the two phase of Active Testing to detect and confirm concurrency bugs. First, we present a data race detection and confirmation algorithm for programs written in Unified Parallel C (UPC), an extension of C for writing parallel programs. Then, in section 4.2, we describe an algorithm to detect and confirm deadlocks in Java programs. Finally, in section 4.3, we introduce an algorithm to detect and confirm atomicity violations for Java programs in a single phase. This algorithm is different from the previous two in that it does not require the prediction phase to find abstract potential buggy states. It discovers and confirms these abstract states while running only the confirmation phase of Active Testing.

4.1 Data race detection for UPC

Unified Parallel C

Unified Parallel C (UPC) is a parallel extension to ISO C 99 for high performance computing. UPC uses the Single Program Multiple Data (SPMD) parallelism model and provides a Partitioned Global Address Space: the memory is partitioned in a thread local heap and a global heap. All threads can access memory residing in the global heap, while access to the local heap is allowed only for the owner thread. The global heap is logically partitioned between threads and each thread is said to have local affinity with its sub-partition. The language extends the C type system with the `shared` qualifier to denote pointer accesses to the global address space. Global memory can be accessed either using shared pointer dereferences (load and store) or using bulk communication primitives (`memget()` and `memput()`). The language provides synchronization primitives, namely locks, barriers, and split-phase barriers.

Locks and semaphores are common synchronization constructs in shared memory programming models (e.g. pthreads), but SPMD programs also utilize barriers (and their split-phase versions) for bulk synchronization. When a thread reaches a barrier statement, it

```
int build_table(int nitems, int cap, shared int *T,
                shared int *w, shared int *v) {
  int wj, vj;
  wj = w[0];
  vj = v[0];
  upc_forall(int i = 0; i < wj; i++; &T[i])
    T[i] = 0;
  upc_forall(int i = wj; i <= cap; i++; &T[i])
    T[i] = vj;
  upc_barrier;
  ...
}
int main(int argc, char** argv) {
  ...
  upc_forall(i = 0; i < nitems; i++; i) {
    weight[i] = 1 + (lrand48()%max_weight);
    value[i]  = 1 + (lrand48()%max_value);
  }
  best_value = build_table(nitems, capacity, total, weight, value);
  ...
}
```

Program 4.1: Parallel knapsack implementation with data race.

cannot proceed onto the next statement until all others threads have also reached the barrier. This effectively splits a program execution into multiple *phases* and all threads run in the same phase at a given time.

Utilizing the time while waiting for other threads in barriers, we can increase parallelism by doing some useful local work. UPC has *split-phase barriers*, in which a thread first *notifies* the other threads that they can continue through the barrier and then after doing some local work, if available, it *waits* for notification from the other threads. The events `BARRIER_NOTIFY`$(\cdot)$ and `BARRIER_WAIT`$(\cdot)$ in Definition 3.2 are generated respectively, when a thread notifies other threads and when a thread finished waiting for other threads at a barrier.

Phase I: Race Prediction Phase

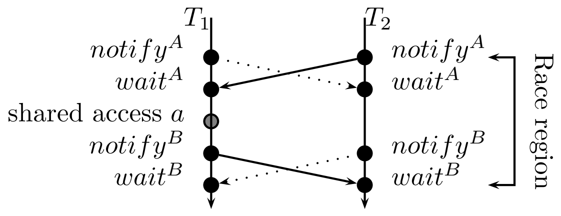
Data races happen when in an execution two threads are about to access the same memory location, at least one access is a `write`, and no ordering is imposed between these concurrent accesses. Figure 4.1 is a partial listing for a UPC program that computes the “0-1 knapsack problem" in parallel using dynamic programming. Although not apparent at first look, there are two data races in this program that can lead to incorrect results. The first data race is between lines 4 and 16, and the second between lines 5 and 17. Since a `upc_forall` statement is not a collective operation (i.e., there is no implicit synchronization at the beginning or end of the loop), there is no ordering enforced between the write to `weight[0]` in line 16 and the read from it in line 4. If the read happens before the write, the table is incorrectly initialized and result in an incorrect computation. The second race is similar to the first. Ironically, the program has been assigned for years as homework for graduate parallel programming courses at UC Berkeley. The bug has been reported by students, which was detected by our tools and independently confirmed by others.

To predict this bug in the example program, we can use the lockset-based predictive dynamic analysis from Chapter 3. However, in lieu of lock based synchronization, scientific programs tend to use barrier synchronization. A barrier partitions the program execution into different phases and prevents a thread from advancing to the next phase before all other threads have completed the phase. Due to this kind of synchronization, our phase I analysis reports a large number of false warnings. In order to eliminate these false warnings, we propose a modification to our analysis, called *Barrier Aware May-Happen-in-Parallel Analysis.*

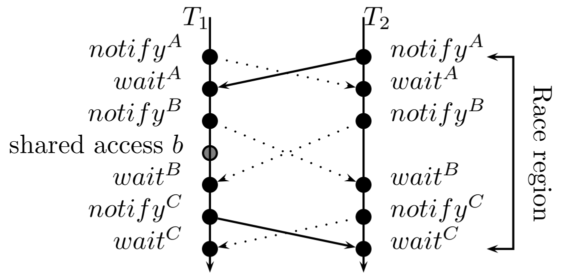
Barrier Aware May-Happen-in-Parallel Analysis

In order to hide communication latency in clusters, split phase barriers are provided in the UPC language. Non-blocking collectives serve a similar purpose in other languages. A split phase barrier in UPC is implemented by a pair of calls `upc_notify` and `upc_wait`. As long as there are no data conflicts, threads can execute arbitrary code in between this pair of calls, also called the *local computation phase* as opposed to the *global computation phase* which is everywhere outside the pair of calls. In principle, only local work should be done in the local computation phase for proper synchronization but this is not enforced.

We illustrate which phases of a program can execute concurrently, using a subset of the *happens-before* relation $\preceq$ from Definition 3.5. In UPC, a thread cannot progress from a `upc_wait` call until notifications from all other threads arrive. Thus, the `upc_notify` calls of the other threads must happen before a matching wait on any thread. According to this happens-before relation for split-phase barriers, a shared access a can happen concurrently with an access from another thread in the region starting from the notify that matches the first wait before a and ending at the wait matching the first notify after a. Figure 4.1 illustrates this scenario for two threads. The arrows denote the happens-before relation induced by the barrier synchronization behavior, with the event pairs affecting the ordering of the shared access indicated as solid arrows. The upper diagram shows the region of thread T_2 that can happen in parallel with a shared access of thread T_1 between a `wait` and `notify`. Following the split-phase barrier semantics, the shared access cannot happen before T_2 has notified $wait^A$ of T_1, and similarly T_2 cannot go beyond $wait^B$ before T_1 executes $notify^B$.



(a) Region concurrent with a shared access after a `wait` (global computation)



(b) Region concurrent with a shared access after a `notify` (local computation)

Figure 4.1: The regions of other threads concurrent with a shared access

The lower diagram shows that a shared access of T_1 between a `notify` and `wait` can happen in parallel with a larger region of T_2.

Based on the happens-before relation of `notify` and `wait` statements, we derive a *may-happen-in-parallel* relationship between program blocks and incorporate barrier awareness in the race detection analysis. For our purposes, we only consider split-phase barriers, since a single-phase barrier can be expressed as a consecutive `notify` and `wait` with no statements in between.[1]

Definition 4.1 (May-happen-in-parallel: $||$). *Let each thread t_i have a barrier phase counter $p_i \in \mathbb{N}$. Initially, $\forall i.\ p_i = 0$. After a thread executes each* ***`notify`*** *and* ***`wait`****, the phase counter is increased by 1. Two phases p_1 and p_2 may happen in parallel, denoted as $p_1 \mid\mid p_2$, if*

$$p_2 \in \left[2\left\lfloor\frac{p_1}{2}\right\rfloor - 1, 2\left\lfloor\frac{p_1+1}{2}\right\rfloor + 1\right] \ .$$

The formula in Definition 4.1 unifies the fact that even phases (global computation) can race with phases ± 1 and odd phases (local computation) can race with phases ± 2. Note that $p_1 \mid\mid p_2$ is a necessary condition for all statements in p_1 to be concurrent with statements in p_2. We now extend the lockset algorithm with the barrier aware may-happen-in-parallel analysis.

[1]This is how the Berkeley UPC Runtime defines single-phase barriers:
```
#define upc_barrier (x) { upc_notify(x); upc_wait(x); }
```

Algorithm 4.1: CheckBeforeUPCRace(e)

```
Initially: $Acc = \emptyset$, the set of all memory accesses in current execution,
           $\forall t \in T.\ L(t) = \emptyset$, the set of locks held by thread t, and $\forall t \in T.\ bp(T) = 0$,
           the barrier phases of threads.
Input: Event $e$ the concurrent system is about to generate
1 switch $e$ do
2     case $\mathtt{MEM}(t, s, m, a)$
3         $Acc := Acc \cup (m, t, L(t), a, bp(t), s)$;
4         return $\{(s, s') \mid \exists (m', t', l', a', p', s') \in Acc.\quad m \cap m' \neq \emptyset \ \wedge\ t \neq t'$
          $\wedge\ L(t) \cap l' = \emptyset \ \wedge\ (a = \mathtt{WRITE} \ \vee\ a' = \mathtt{WRITE}) \ \wedge\ bp(t) \mid\mid p'\}$;
5     case $\mathtt{LOCK}(t, s, l)$
6         $L(t) := L(t) \cup l$;
7 return $\emptyset$;
```

Definition 4.2 (Set of Barrier Aware Potential Data Race Pairs: $\bar{\mathcal{D}}_{P,E}$)**.** *We extend the memory access event in Definition 3.1 by adding a field for the barrier phase p of thread t: $e = (m, t, l, a, p, s)$. The set of potential data race pairs reported by our analysis is*

$$\begin{aligned}\bar{\mathcal{D}}_{P,E} = \{(s_1, s_2) \mid \exists e_1, e_2 \in \mathcal{A}_{P,E} \text{ such that } p_1 || p_2 \wedge \\ m_1 \cap m_2 \neq \emptyset \wedge t_1 \neq t_2 \wedge l_1 \cap l_2 = \emptyset \wedge \\ (a_1 = WRITE \vee a_2 = WRITE)\}\ .\end{aligned}$$

The first phase of Active Testing executes the program once to build the set of possible data races $\bar{\mathcal{D}}_{P,E}$ from the above definition. Each thread builds a trace of memory accesses $e = (m, t, l, a, p, s)$ with respect to the barrier phases and set of locks held at that program point. Each thread maintains this trace for the portion of the global heap with local affinity. For any global memory reference in T_1, Phase I performs a query operation on the trace of the thread responsible for maintaining state for that region, e.g. T_2. If an outstanding conflicting access is found on T_2, the detection algorithm has identified a potential data race, i.e. it has found a statement pair (s_1, s_2) in $\bar{\mathcal{D}}_{P,E}$.

The functions CheckBeforeUPCRace (Algorithm 4.1) and CheckAfterUPCRace (Algorithm 4.2) calculate $\bar{\mathcal{D}}_{P,E}$ online. They are called from Phase1Scheduler (Algorithm 3.2). As in the lockset-based race detection example from Chapter 3, we record memory accesses and check for conflicts with previous accesses (lines 2–4 of Algorithm 4.1) and maintain lock (lines 5–6 of Algorithm 4.1 and lines 2–3 of Algorithm 4.2) and barrier phase (lines 4–7 of Algorithm 4.2) information. The abstract state returned by the functions is a statement pair that may potentially race.

Algorithm 4.2: CheckAfterUPCRace(e)

```
Initially: $L$ and $bp$ are shared with CheckBeforeUPCRace
Input: Event $e$ the concurrent system has just generated
1 switch $e$ do
2     case UNLOCK($t, s, l$)
3         $L(t) := L(t) \setminus l$;
4     case BARRIER_NOTIFY($t, s$)
5         $bp(T) := bp(T) + 1$;
6     case BARRIER_WAIT($t, s$)
7         $bp(T) := bp(T) + 1$;
8 return $\emptyset$;
```

Phase II: Race Confirmation Phase

From Phase I, we get the set $\bar{\mathcal{D}}_{P,E}$. For each statement pair in $\bar{\mathcal{D}}_{P,E}$, there is a possibility that it may be a false positive due to lack of application specific semantic information, e.g. usage of custom synchronization operations. In the second phase of Active Testing, we re-execute the program and actively control the thread schedule in an effort to confirm the real data races. Specifically, for each statement pair (s_1, s_2) in $\bar{\mathcal{D}}_{P,E}$, we try to create an execution state where two threads are about to execute s_1 and s_2, respectively, and they are about to access the same memory location, and at least one of the accesses is a write. Such an execution is evidence that the data race over the statements s_1 and s_2 is a real race.

The two functions required by Algorithm 3.5 for confirmation of data races in UPC are the same as CheckPartialLockset (Algorithm 3.6) and CheckFullLockset (Algorithm 3.7). In the same manner of confirming lockset-based data races, CheckPartialUPCRace pauses threads whenever they reach a statement in (s_1, s_2) and CheckFullUPCRace confirms any real races between the thread selected to be executed and any of the paused threads.

4.2 Deadlock detection

In this section, we present DeadlockFuzzer [49], which follows the active testing methodology to predict and confirm deadlocks. The DeadlockFuzzer algorithm also consists of two phases. In the first phase, we execute a multi-threaded program and find potential deadlocks that could happen in some execution of the program. This phase uses a modified Goodlock [9] algorithm, called *informative Goodlock*, or simply iGoodlock, which identifies potential deadlocks even if the observed execution does not deadlock. We call the modified algorithm *informative* because we provide suitable debugging information to identify the cause of the deadlock—this debugging information is used by the second phase to create real deadlocks with high probability. A limitation of iGoodlock is that it can give false positives because it does not consider the happens-before relation between the transitions in an execution. As a result, the user is required to manually inspect such potential deadlocks. The second phase removes this burden from the user. In this phase, a random thread scheduler is biased to generate an execution that creates a real deadlock reported in the previous phase with high probability. We next describe these two phases in more detail.

Phase I: Deadlock prediction

In this section, we present a formal description of iGoodlock. The algorithm observes the execution of a multi-threaded program and computes a *lock dependency relation* (defined below) and uses a transitive closure of this relation to compute potential deadlock cycles. The algorithm improves over generalized Goodlock algorithms [9, 3] in two ways. First, it adds contextual information to a computed potential deadlock cycle. This information helps to identify the program locations where the deadlock could happen and also to statically identify the lock and thread objects involved in the deadlock cycle. Second, we simplify the generalized Goodlock algorithm by avoiding the construction of a lock graph, where locks form the vertices and a labeled edge is added from one lock to another lock if a thread acquires the latter lock while holding the former lock in some program state. Unlike existing Goodlock algorithms, iGoodlock does not perform a depth-first search, but computes the transitive closure of the lock dependency relation.

Given a multi-threaded execution σ, let L_σ be the set of locks that were held by any thread in the execution and T_σ be the set of threads in the execution. Let $\mathcal{C}$ be the set of all statement labels in the multi-threaded program. We define the *lock dependency* relation of a multi-threaded execution as follows.

Definition 4.3 (Lock dependency relation)**.** *Given an execution σ, a* lock dependency relation *D_σ of σ is a subset of $T_\sigma \times 2^{L_\sigma} \times L_\sigma \times \mathcal{C}^*$ such that $(t, L, l, C) \in D_\sigma$ iff in the execution σ, in some state, thread t acquires lock l while holding the locks in the set L, and C is the sequence of labels of **`LOCK`** statements that were executed by t to acquire the locks in $L \cup \{l\}$.*

Definition 4.4 (Lock dependency chain)**.** *Given a lock dependency relation D,* a lock dependency chain
$\tau = \langle(t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$ is a sequence in D^* such that the following properties hold.

1. *for all distinct $i, j \in [1, m]$, $t_i \neq t_j$, i.e. the threads $t_1, t_2, \ldots, t_m$ are all distinct objects,*
2. *for all distinct $i, j \in [1, m]$, $l_i \neq l_j$, i.e. the lock objects $l_1, l_2, \ldots, l_m$ are distinct,*
3. *for all $i \in [1, m-1]$, $l_i \in L_{i+1}$, i.e. each thread could potentially wait to acquire a lock that is held by the next thread,*
4. *for all distinct $i, j \in [1, m]$, $L_i \cap L_j = \emptyset$, i.e. each thread t_i should be able to acquire the locks in L_i without waiting.*

Definition 4.5 (Potential deadlock cycle). *A lock dependency chain*

$$\tau = \langle(t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$$

is a potential deadlock cycle *if* $l_m \in L_1$.

Note that the definition of a potential deadlock cycle never uses any of the C_i's in D_σ to compute a potential deadlock cycle. Each C_i of a potential deadlock cycle provides us with information about program locations where the locks involved in the cycle were acquired. This is useful for debugging and is also used by the confirmation phase to determine the program locations where it needs to pause a thread.

The iGoodlock algorithm is language independent and can be adapted to most multi-threaded programming languages without modification. For simpler exposition, we describe it in terms of the Java language. Java is an object-oriented language where everything is an object[2], including locks and threads. Each lock and thread object involved in a potential deadlock cycle is identified by its unique ID, which is typically the *address* of the object. The unique ID of an object, being based on dynamic information, can change from execution to execution. Therefore, the unique ID of an object cannot be used by Phase II of Active Testing to identify a thread or a lock object across executions. In order to overcome this limitation, we compute an abstraction of each object. An abstraction of an object identifies an object by static program information. For example, the label of a statement at which an object is created could be used as its abstraction. Such an abstraction of an object does not change across executions. However, multiple objects could map to the same abstraction. We describe two object abstraction computation techniques that are more precise in Section 5.2. In this section, we assume that $\texttt{abs}(o)$ returns some abstraction of the object o. In Java, locks are *re-entrant*, i.e., a thread may re-acquire a lock it already holds. In our algorithms, we ignore a transition $\texttt{LOCK}(t, s, l)$ if t re-acquires a lock l.[3]

Given a potential deadlock cycle $\langle(t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$, iGoodlock reports the abstract deadlock cycle $\langle(\texttt{abs}(t_1), \texttt{abs}(L_1), \texttt{abs}(l_1), C_1), \ldots, (\texttt{abs}(t_m), \texttt{abs}(L_m), \texttt{abs}(l_m), C_m)\rangle$. The confirmation phase takes such an abstract deadlock cycle and tries to create a real deadlock corresponding to the cycle with high probability.

iGoodlock is a combination of an online and offline analysis. We next describe how we compute the lock dependency relation during a multi-threaded execution (online) and how we compute all potential deadlock cycles given a lock dependency relation (offline).

Computing the lock dependency relation of a multi-threaded execution

With CheckBeforeDeadlock (Algorithm 4.3), we can compute the lock dependency relation during an execution using the general Phase I Scheduler (Algorithm 3.2). For

[2]except primitive data such as `int`s and `double`s

[3]This is implemented by associating a usage counter with a lock which is incremented whenever a thread acquires or re-acquires the lock and decremented whenever a thread releases the lock. A $\texttt{LOCK}(l)$ is considered whenever the thread t acquires or re-acquires the lock l and the usage counter associated with l is incremented from 0 to 1.

Algorithm 4.3: CheckBeforeDeadlock(e)

```
Initially: $\forall t.\ Lockset(t) = \langle \cdot \rangle$, a map from each thread to the stack of locks it holds;
           $\forall t.\ Context(t) = \langle \cdot \rangle$, a map from each thread to the stack of labels of
           statements where the it acquired the currently held locks.
Input: Event $e$ the concurrent system is about to generate
Output: A lock dependency relation
1  $ret := \emptyset$;
2  switch $e$ do
3      case LOCK($t, s, l$)
4          $Context(t) := \text{push}(Context(t), s)$;
5          $ret := \langle t, Lockset(t), l, Context(t) \rangle$;
6          $Lockset(t) := \text{push}(Lockset(t), l)$;
7      case UNLOCK($t, s, l$)
8          $Context(t) := \text{pop}(Context(t))$;
9          $Lockset(t) := \text{pop}(Lockset(t))$;
10 return $ret$;
```

each thread, we maintain two stacks that keep track of the locks held by a thread and the statement labels (program counter) where each lock was acquired. As seen in the algorithm, maintaining these stacks to compute the lock dependency relation is quite simple. At the end of execution, we obtain the full lock dependency relation D observed during that execution. This relation is passed to the iGoodlock algorithm for offline processing.

Note that in Algorithms 4.3, 4.5, and 4.6, we use a general stack data structure. We define a stack below in a pure-functional style.

Definition 4.6 (Stack). *A stack is an ordered collection of elements. An empty stack is denoted as $\langle \cdot \rangle$. The contents of a stack is enumerated within angled brackets, starting with the top-most element. For example, $S = \langle a, b, c \rangle$, denotes a stack S with a as its top-most element and c the bottom-most. A stack supports the following operations:*

- **push**$(S, i) \to S'$: Returns the stack S' with the item i *push*ed onto the top of stack S. For example, push$(\langle a, b, c \rangle, d)$ returns $\langle d, a, b, c \rangle$. push$(S, i)$ is also abbreviated as $i :: S$.

- **pop**$(S) \to S'$: Returns the stack S' with the top-most element *pop*ped off the stack S. For example, pop$(\langle a, b, c \rangle)$ returns $\langle b, c \rangle$.

- **top**$(S) \to i$: Returns the *top*-most element of the stack S. For example, top$(\langle a, b, c \rangle)$ returns a.

Algorithm 4.4: iGoodlock(D)

```
Input: lock dependency relation $D$
Output: potential deadlock cycles
1  $i := 1$;
2  $D^i := D$;
3  while $D^i \neq \emptyset$ do
4      for each $(t, L, l, C) \in D$ and each $\tau$ in $D^i$ do
5          if $\tau, (t, L, l, C)$ is a dependency chain by Definition 4.4 then
6              if $\tau, (t, L, l, C)$ is a potential deadlock cycle by Definition 4.5 then
7                  report abs($\tau, (t, L, l, C)$) as a potential deadlock cycle;
8              else
9                  add $\tau, (t, L, l, C)$ to $D^{i+1}$;
10     $i := i + 1$;
```

Computing *potential* deadlock cycles iteratively

Let D^k denote the set of all lock dependency chains of D that has length k. By definition, $D^1 = D$. iGoodlock computes potential deadlock cycles by iteratively computing $D^2, D^3, D^4, \ldots$ and finding deadlock cycles in those sets. The iterative algorithm for computing potential deadlock cycles is described in iGoodlock (Algorithm 4.4).

Note that in iGoodlock we do not add a lock dependency chain to D^{i+1} if it is a deadlock cycle (lines 6–9). This ensures that we do not report complex deadlock cycles, i.e. deadlock cycles that can be decomposed into simpler cycles.

In iGoodlock, a deadlock cycle of length k gets reported k times. For example, if

$$\langle (t_1, L_1, l_1, C_1), (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m) \rangle$$

is reported as a deadlock cycle, then

$$\langle (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m), (t_1, L_1, l_1, C_1) \rangle$$

is also reported as a cycle. In order to avoid such duplicates, we put another constraint in Definition 4.4: the unique ID of thread t_1 must be less than the unique ID of threads $t_2, \ldots, t_m$.

Phase II: Deadlock confirmation

In the second phase of DeadlockFuzzer, it can execute a multi-threaded program using a random scheduler such as Algorithm 3.1. Starting from the initial state, it randomly picks an enabled thread and executes its next statement. When the system reaches a state that has no enabled threads, it checks if any of the threads are still alive. If so, it reports a system stall. A stall could happen due to a resource deadlock (i.e. deadlocks that happen

Algorithm 4.5: CheckPartialDeadlock($e, Cycle$)

```
Input: Event $e$ and the $Cycle$ to confirm
Initially: $\forall t.\ Lockset(t) = \langle\cdot\rangle$ and $\forall t.\ Context(t) = \langle\cdot\rangle$ as defined in Algorithm 4.3.
1 switch $e$ do
2     case LOCK($t, s, l$)
3         $Lockset(t) := \text{push}(Lockset(t), l)$;
4         $Context(t) := \text{push}(Context(t), s)$;
5         return $\langle abs(t), abs(Lockset(t)), abs(Context(t))\rangle \in Cycle$;
6     case UNLOCK($t, s, l$)
7         $Lockset(t) := \text{pop}(Lockset(t))$;
8         $Context(t) := \text{pop}(Context(t))$;
9 return $false$;
```

Algorithm 4.6: CheckFullDeadlock($e, Paused, Cycle$)

```
Input: Event $e$, the current set of $Paused$ threads, and the $Cycle$ to confirm
Initially: $Lockset$ and $Context$ are shared with CheckPartialDeadlock
1 $n := |Cycle|$;
2 $t_n := e.t$;
3 return $\exists t_1, \ldots, t_{n-1} \in Paused.\ \text{top}(Lockset(t)) \in \text{pop}(Lockset(t_1)) \wedge$
  $\forall i \in [1, n-1].\ \text{top}(Lockset(t_i)) \in \text{pop}(Lockset(t_{i+1}))$;
```

due to locks) or a communication deadlock (i.e. a deadlock that happens when each thread is waiting for a signal from some other thread in the set). DeadlockFuzzer only considers resource deadlocks.

A key limitation of this approach is that it may not create real deadlocks very often. Following the Active Testing template, we implement functions CheckPartialDeadlock (Algorithm 4.5) and CheckFullDeadlock (Algorithm 4.6) enabling DeadlockFuzzer to bias the random scheduler so that potential deadlock cycles reported by iGoodlock are created with high probability.

The abstract state passed to these functions is the potential deadlock cycle that we wish to confirm. The maps *Lockset* and *Context* are as defined in CheckBeforeDeadlock (Algorithm 4.3) and shared between the two functions. The Phase II scheduler executes the multi-threaded program like the simple random scheduler, except that it performs some extra work when it encounters a lock acquire or lock release statement. If a thread t is about to acquire a lock l in the context C, then if $(\texttt{abs}(t), \texttt{abs}(l), C)$ is present in `Cycle`, the scheduler pauses thread t before t acquires lock l, giving a chance for another thread, which is involved in the potential deadlock cycle, to acquire lock l subsequently. This ensures that the system creates the potential deadlock cycle `Cycle` with high probability.

During the confirmation phase of DeadlockFuzzer, it maintains three data structures: *Lockset* that maps each thread to a stack of locks that are currently held by the thread, *Context* that maps each thread to a stack of statement labels where the thread has acquired the currently held locks, and *Paused* which is a set of threads that has been paused by DeadlockFuzzer. *Paused* is initialized to an empty set, and *Lockset* and *Context* are initialized to map each thread to an empty stack.

DeadlockFuzzer runs in a loop until there is no enabled thread. At termination, DeadlockFuzzer reports a system stall if there is at least one active thread in the execution. Note that DeadlockFuzzer only catches resource deadlocks. In each iteration of the loop, DeadlockFuzzer picks a random thread t that is enabled but not in the *Paused* set. If the next statement to be executed by t is not a lock acquire or release, t executes the statement and updates the state as in the simple random scheduling algorithm (see Algorithm 3.1). If the next event to be generated by t is $\texttt{LOCK}(t, s, l)$, s and l are pushed to $Context(t)$ and $Lockset(t)$, respectively. DeadlockFuzzer determines if t needs to be paused in order to get into a deadlock state in CheckPartialDeadlock (Algorithm 4.5. Specifically, it checks if $(\texttt{abs}(t), \texttt{abs}(Lockset(t)), Context(t)$ is present in *Cycle*. Only if t can be part of a deadlock cycle, DeadlockFuzzer checks if the acquire of l by t could lead to a deadlock using CheckFullDeadlock in Algorithm 4.6. CheckFullDeadlock goes over the current lock set of each paused thread and sees if it can find a cycle. If a cycle is discovered, then DeadlockFuzzer has created a *real* deadlock. If there is no cycle, then t is added to *Paused*. If the next event to be generated by t is $\texttt{UNLOCK}(t, s, l)$, then we pop from both $Lockset(t)$ and $Context(t)$.

At the end of each iteration, it may happen that the set *Paused* is equal to the set of all enabled threads. This results in a state where DeadlockFuzzer has unfortunately paused all the enabled threads and the system cannot make any progress. We call this *thrashing.* DeadlockFuzzer handles this situation by removing a random thread from the set *Paused.* A thrash implies that DeadlockFuzzer has paused a thread in an unsuitable state. DeadlockFuzzer should avoid thrashing as much as possible in order to guarantee better performance and improve the probability of detecting real deadlocks. Sections 5.2 and 5.3 present techniques to reduce thrashing.

4.3 Atomicity violation detection

In this section, we describe AtomFuzzer, which tries to create atomicity violations in executions on-the-fly. It follows the Active Testing template, but is different from others in that it does not have a prediction phase: only Phase II is required to find and confirm real atomicity violations. We use a robust notion of atomicity, called *causal atomicity*, introduced by Farzan et al. [27].

Definition 4.7 (Causal Atomicity)**.** *A block of code B of a thread is* causally atomic *if there is no execution where a transition of another thread happens-after the beginning of B and happens-before another transition that is within the same block B.*

The definition of causal atomicity implies that an atomicity violation occurs in an execution if there are three transitions t_1, t_2, and t_3 such that

1. t_1 and t_3 are transitions of the same thread and are within the same atomic block,
2. t_2 is a transition of another thread, and
3. t_1 happens-before t_2 and t_2 happens before t_3.

The goal of AtomFuzzer is to create such atomicity violations.

The randomized active atomicity violation detection algorithm

We are interested in detecting a special class of causal atomicity violations where the transitions t_1, t_2, and t_3 involved in the violation are acquires of the same lock l. There are three reasons for focusing on this particular pattern. First, our experiments demonstrate that this is a very common atomicity violation pattern and real-world programs often show this buggy pattern. Second, since we focus on lock acquires only, the runtime overhead of AtomFuzzer is pretty low compared to other tools. Third, we believe that this pattern does not capture some other common situations where t_1, t_2, and t_3 are accesses to the same memory location. However, we argue that a data race over the memory location implies the remaining patterns and can be detected by a race detector. Assume that t_1, t_2, and t_3 are accesses to the same memory location m. If these accesses are not in data race, then each of these accesses is surrounded by a common lock, say l. Let t'_1, t'_2, and t'_3 be transitions denoting the acquire of the lock l before the transitions t_1, t_2, and t_3, respectively. The transitions t'_1, t'_2, and t'_3 then form the above mentioned atomicity violation pattern. Therefore, if there is no race among the transitions t_1, t_2, and t_3, then the resulting atomicity violation pattern is the same as the pattern we are interested in.

In AtomFuzzer, we consider the `ATOMIC_ENTER`(t, s) and `ATOMIC_EXIT`(t, s) events generated by the concurrent system. Normally, we will assume that the `ATOMIC_ENTER`$(\cdot)$ and `ATOMIC_EXIT`$(\cdot)$ transitions will be introduced by programmers to annotate the entry and exit of atomic blocks in a concurrent system. Also, we consider locks to be re-entrant, as in Section 4.2.

AtomFuzzer looks for the atomicity violation pattern described above while running the Phase II scheduler (Algorithm 3.5) with the functions CheckPartialAtomViol (Algorithm 4.7) and CheckFullAtomViol (Algorithm 4.8). In particular, whenever AtomFuzzer finds that a thread t is inside an atomic block and is about to acquire a lock l that has been previously acquired and released inside the same atomic block, AtomFuzzer issues an atomicity violation warning (Line 10 in Algorithm 4.7). Such a warning would be given by any other static or dynamic atomicity checking tool. However, such a warning can be spurious unless one can show that some other thread can acquire and release the same lock l immediately before the thread t acquires the lock. AtomFuzzer tries to create this scenario by changing the default scheduler behavior. Specifically, AtomFuzzer *pauses* the

Algorithm 4.7: CHECKPARTIALATOMVIOL(e, Abs)

```
Input: Event $e$ and the abstract state $Abs$ to confirm
1  $(AlreadyAcquired, InsideAtomic) := Abs$;
2  switch $e$ do
3      case $ATOMIC\_ENTER(t, s)$
4          $InsideAtomic := InsideAtomic \cup t$ ;
5      case $ATOMIC\_EXIT(t, s)$
6          $InsideAtomic := InsideAtomic \setminus t$ ;
7          $AlreadyAcquired := AlreadyAcquired \setminus \{(t', l) : t' = t\}$;
8      case $LOCK(t, s, l)$
9          if $t \in InsideAtomic$ then
10             if $(t, l) \in AlreadyAcquired$ then
11                 print "Warning: atomicity violation possible at ", s;
12                 return true;
13             else
14                 $AlreadyAcquired := AlreadyAcquired \cup (t, l)$;
15         if $\exists t'.\ (t', l) \in AlreadyAcquired$ then
16             return true;
17 return $false$;
```

Algorithm 4.8: CHECKFULLATOMVIOL(e, $Paused$, Abs)

```
Input: Event $e$, the current set of $Paused$ threads, and the abstract state $Abs$ to
       confirm
1  $(AlreadyAcquired, InsideAtomic) := Abs$;
2  switch $e$ do
3      case $LOCK(t, s, l)$
4          return $\exists t' \in Paused.\ \text{EVENT}(t') = LOCK(t', s', l') \wedge l' = l$;
```

execution of the thread t just before it acquires the lock l and allows the other threads to execute. If at any point in the execution, ATOMFUZZER discovers that some other thread has acquired the lock l, then ATOMFUZZER flags an atomicity violation error because it has created a scenario showing an atomicity violation.

The algorithm can produce three kinds of outputs:

1. **Warnings:** These are potential atomicity violations. Existing tools, such as Atomizer, already produce these warnings. We do not show these warnings to the user and we only use them for experimental evaluation.

2. **Errors:** These are real atomicity violations, i.e. ATOMFUZZER has actually created an execution showing the violations. Sometimes atomicity violations may not result

in bugs because they are benign. Moreover, if we are using the heuristic that all synchronized public methods are atomic blocks, then an atomicity violation may not result in a bug because the heuristic may not match the user intention.

3. **Bugs/Exceptions:** These are uncaught exceptions or program crashes that result due to real atomicity violations. If we are using the heuristic, then they indicate that the found atomicity violation is a real bug.

In the algorithm we maintain three sets: *Paused* to maintain information about the threads that we have paused in an effort to create an atomicity violation, *InsideAtomic* to keep track of threads that are already inside an atomic block, and *AlreadyAcquired* to keep track of locks that a thread has already acquired and released while inside its current atomic block. These sets are initially empty. Since AtomFuzzer does not have a predictive phase reporting potential buggy states, we consider the pair of sets *AlreadyAcquired* and *InsideAtomic* as the abstract state we wish to confirm.

At every state AtomFuzzer picks a random enabled thread t, such that the thread has not been paused for the purpose of creating an atomicity violations. If a thread t executes `ATOMIC_ENTER`(t, s) (see line 3 in Algorithm 4.7), then we add t to the set *InsideAtomic* to record the fact that the thread is now inside an atomic block. Similarly, if a thread t executes `ATOMIC_EXIT`(t, s) (see line 5 in Algorithm 4.7), then we remove t from the set *InsideAtomic* to indicate the fact that the thread is no longer inside an atomic block. We also clear all entries corresponding to the thread t in the set *AlreadyAcquired.*

The key component of the AtomFuzzer algorithm kicks in if the randomly picked transition is `LOCK`(t, s, l) of thread t. There are two ways a thread is part of an atomicity violation. First, is to be the thread that is inside of an atomic block that acquires, releases, and then re-acquires the same lock, but have another thread violate atomicity by acquiring the lock between the release and acquire of the lock. This is the case that is handled in line 10 of Algorithm 4.7. Whenever a thread is about to reacquire a lock inside an atomic block, we issue a warning that an atomicity violation could happen, following Lipton's reduction algorithm [58], and pause the thread to wait for another thread to possibly acquire the lock.

The other case is for the atomicity-violating thread to acquire a lock that has been released inside an atomic block and about to be acquired again. This case is handled in line 15 of Algorithm 4.7. Whenever a thread could possibly participate in an atomicity violating situation, we check if the lock acquire is an actual atomicity violation in Algorithm 4.8.

Chapter 5

Implementation for Multi-threaded Programs

In this chapter, we describe an extensible tool for Active Testing of concurrent Java programs, called CalFuzzer. CalFuzzer provides a framework for implementing various predictive dynamic analyses to obtain a set of abstract program states involved in a potential concurrency bug. We have implemented three such techniques in CalFuzzer: a hybrid race detector [72], the Atomizer algorithm [31] for finding potential atomicity violations, and iGoodlock [49] which is a more informative variant of the Goodlock algorithm [40] for detecting potential deadlocks. More generally, CalFuzzer provides an interface and utility classes to enable users to implement additional such techniques.

CalFuzzer also provides a framework for implementing custom schedulers for Active Testing. We call these custom schedulers *active checkers*. We have implemented three active checkers in CalFuzzer for detecting real data races, atomicity violations, and deadlocks. More generally, CalFuzzer allows users to specify an arbitrary set of program statements in the concurrent program under test where an active checker should pause. Such statements may be thought of as *concurrent breakpoints* [75].

We have applied CalFuzzer to several real-world multi-threaded Java programs comprising a total of 600K lines of code and have detected both previously known and unknown data races, atomicity violations, and deadlocks. CalFuzzer could also be extended to detect other kinds of concurrency bugs, such as missed notifications and atomic set violations.

5.1 CalFuzzer

We have implemented the CalFuzzer [47] Active Testing framework for Java. CalFuzzer (available at `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`) instruments Java bytecode using the SOOT compiler framework [96] to insert callback functions before or after various synchronization operations and shared memory accesses.[1] These callback functions are used to implement various predictive dynamic analyses and active checkers. Each predictive dynamic analysis implements an interface called `Analysis`. The methods of these interface implement the CheckBeforeX and CheckAfterX functions in Algorithm 3.2. Likewise, each active checker is implemented by extending a class called `ActiveChecker` which implements the CheckPartialX and CheckFullX functions in Algorithm 3.5. The methods of these two classes are called by the callback functions.

The framework provides various utility classes, such as `VectorClockTracker` and `LocksetTracker` to compute vector clocks and locksets at runtime. Methods of these classes are invoked in the various callback functions described above. These utility classes are used in the hybrid race detection [72] and iGoodlock [49] algorithms; other user defined dynamic analyses could also use these classes.

The instrumenter of CalFuzzer modifies all bytecode associated with a Java program including the libraries it uses, except for the classes that are used to implement CalFuzzer. This is because CalFuzzer runs in the same memory space as the program under analysis. CalFuzzer cannot track lock acquires and releases by native code and can therefore go into a deadlock if there are synchronization operations inside uninstrumented classes or native code. To avoid such scenarios, CalFuzzer runs a low-priority monitor thread that periodically polls to check if there is any deadlock. If the monitor discovers a deadlock, then it removes one random transition from the *paused* set.

CalFuzzer can also go into livelocks. Livelocks happen when all threads of the program end up in the *paused* set, except for one thread that does something in a loop without synchronizing with other threads. We observed such livelocks in a couple of our benchmarks including `moldyn`. In the presence of livelocks, these benchmarks work correctly because the correctness of these benchmarks assumes that the underlying Java thread scheduler is fair. In order to avoid livelocks, CalFuzzer creates a monitor thread that periodically removes those transitions from the *paused* set that are waiting for a long time.

5.2 Computing object abstractions

A key requirement of active checkers is that it should know where a thread needs to be paused. For example, DeadlockFuzzer needs to know if a thread t that is trying to acquire a lock l in a context C could lead to a deadlock. DeadlockFuzzer gets this information from iGoodlock, but this requires us to identify the lock and thread objects that are the "same" in the iGoodlock and DeadlockFuzzer executions. This kind of correlation cannot be done using the address (i.e., the unique ID) of an object because object addresses change across executions. Therefore, we propose to use object abstraction—if two objects are the

[1]We decided to instrument bytecode instead of changing the Java virtual machine or instrumenting Java source code because Java bytecode changes less frequently than JVM and Java source may not be available for libraries.

same across executions, then they have the same abstraction. We assume $\texttt{abs}(o)$ computes the abstraction of an object.

There could be several ways to compute the abstraction of an object. One could use the label of the statement that allocated the object (i.e. the allocation site) as its abstraction. However, that would be too coarse-grained to distinctly identify many objects. For example, if one uses the factory pattern to allocate all thread objects, then all of the threads will have the same abstraction. Therefore, we need more contextual information about an allocation site to identify objects at finer granularity.

If we use a coarse-grained abstraction, then active checkers will pause unnecessary threads before events such as acquiring some unnecessary locks. This is because all these unnecessary threads and unnecessary locks might have the same abstraction as the relevant thread and lock, respectively. This will in turn reduce the effectiveness of an active checker, as it will more often remove a thread from the `Paused` set due to the unavailability of any enabled thread. We call this situation *thrashing*. Our experiments with DeadlockFuzzer show that if we use the trivial abstraction, where all objects have the same abstraction, then we get a lot of thrashing. This in turn reduces the probability of creating a real deadlock. On the other hand, if we consider too fine-grained abstractions for objects, then we will not be able to tolerate minor differences between two executions, causing threads to pause at fewer locations and miss deadlocks. We next describe two abstraction techniques for objects that we have found effective in our experiments. Note that good object abstractions can help other checkers in the CalFuzzer framework reproduce bugs.

To obtain more knowledge of the context in which objects are created, we consider the following three *internal events* that are irrelevant to concurrency.

- $\texttt{CALL}(t, s, m)$: Thread t at statement labeled s (same for all other events below), called method m.
- $\texttt{RETURN}(t, s, m)$: Thread t return from method m.
- $\texttt{NEW}(t, s, o, T, o')$: Thread t created a new object o of dynamic type T inside the body of a method whose `this` argument evaluates to o'.

Abstraction based on k-object-sensitivity

Given a multi-threaded execution and $k > 0$, let $o_1, \ldots o_k$ be the sequence of objects such that for all $i \in [1, k-1]$, o_i is allocated by some method of object o_{i+1}. We define $\texttt{abs}^O_k(o_1)$ as the sequence $\langle s_1, \ldots, s_k \rangle$ where s_i is the label of the statement that allocated o_i. $\texttt{abs}^O_k(o_1)$ can then be used as an abstraction of o_1. We call this *abstraction based on k-object-sensitivity* because of the similarity to k-object-sensitive static analysis [63].

In order to compute $\texttt{abs}^O_k(o)$ for each object o during a multi-threaded execution, we instrument the program to maintain a map `CreationMap` that maps each object o to a pair (o', s) if o is created by a method of object o' at the statement labeled s. This gives the following straightforward runtime algorithm for computing `CreationMap`.

```
 1  main() {
 2    for (int i=0; i<5; i++)
 3      foo();
 4  }
 5  void foo() {
 6    bar();
 7    bar();
 8  }
 9  void bar() {
10    for (int i=0; i<3; i++)
11      Object l = new Object();
12  }
```

Program 5.1: Example code that creates objects inside a loop

- At the event $\texttt{NEW}(t, s, o, T, o')$, add $o \mapsto (o', s)$ to `CreationMap`.

One can use `CreationMap` to compute $\texttt{abs}^O_k(o)$ using the following recursive definition:

$$\begin{array}{rr} \texttt{abs}^O_k(o) = \langle\rangle & \text{if } k = 0 \text{ or } \texttt{CreationMap}[o] = \bot \\ \texttt{abs}^O_{k+1}(o) = s :: \texttt{abs}^O_k(o') & \text{if } \texttt{CreationMap}[o] = (o', s) \end{array}$$

When an object is allocated inside a static method, it will not have a mapping in `CreationMap`. Consequently, $\texttt{abs}^O_k(o)$ may have fewer than k elements.

Abstraction based on light-weight execution indexing

Given a multi-threaded execution, a $k > 0$, and an object o, let $m_n, m_{n-1}, \ldots, m_1$ be the call stack when o is created, i.e. o is created inside method m_1 and for all $i \in [1, n-1]$, m_i is called from method m_{i+1}. Let us also assume that s_{i+1} is the label of the statement at which m_{i+1} invokes m_i and q_{i+1} is the number of times m_i is invoked by m_{i+1} in the context $m_n, m_{n-1}, \ldots, m_{i+1}$. Then $\texttt{abs}^I_k(o)$ is defined as the sequence $[s_1, q_1, s_2, q_2, \ldots, s_k, q_k]$, where s_1 is the label of the statement at which o is created and q_1 is the number of times the statement is executed in the context $m_n, m_{n-1}, \ldots, m_1$.

For example in Program 5.1, if o is the first object created by the execution of `main`, then $\texttt{abs}^I_3(o)$ is the sequence $[11, 1, 6, 1, 3, 1]$. Similarly, if p is the last object created by the execution of `main`, then $\texttt{abs}^I_3(p)$ is the sequence $[11, 3, 7, 1, 3, 5]$. The idea of computing this kind of abstraction is similar to the idea of execution indexing proposed in [104], except that we ignore branch statements and loops. This makes our indexing light-weight, but less precise.

In order to compute $\texttt{abs}_k^I(o)$ for each object o during a multi-threaded execution, we instrument the program to maintain a thread-local scalar d to track its depths and two thread-local maps `CallStack` and `Counters`. We use $\texttt{CallStack}_t$ to denote the `CallStack` map of thread t. The above data structures are updated at runtime as follows.

- Initialization:
  - for all t, $d_t \Leftarrow 0$
  - for all t and s, $\texttt{Counters}_t[d_t][s] \Leftarrow 0$
- At event $\texttt{CALL}(t, s, m)$
  - $\texttt{Counters}_t[d_t][s] \Leftarrow \texttt{Counters}_t[d_t][s] + 1$
  - push s to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][s]$ to $\texttt{CallStack}_t$
  - $d_t \Leftarrow d_t + 1$
  - for all c, $\texttt{Counters}_t[d_t][s] \Leftarrow 0$
- At event $\texttt{RETURN}(t, s, m)$
  - $d_t \Leftarrow d_t - 1$
  - pop twice from $\texttt{CallStack}_t$
- At event $\texttt{NEW}(t, s, o, T, o')$
  - $\texttt{Counters}_t[d_t][s] \Leftarrow \texttt{Counters}_t[d_t][s] + 1$
  - push s to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][s]$ to $\texttt{CallStack}_t$
  - $\texttt{abs}_k^I(o)$ is the top $2k$ elements of $\texttt{CallStack}_t$
  - pop twice from $\texttt{CallStack}_t$

Note that $\texttt{abs}_k^I(o)$ has $2k$ elements, but if the call stack has fewer elements, then $\texttt{abs}_k^I(o)$ returns the full call stack.

Chapter 6.4 shows our evaluation on the effectiveness of these two object abstraction strategies for DeadlockFuzzer. We also compare against ignoring the abstractions and contextual information altogether to show the usefulness of good object abstraction for lower overhead and better probability of reproducing deadlocks.

```
1: Thread 1                 8: Thread 2
2:   synchronized(l1) {     9:   synchronized(l1) {
3:     synchronized(l2) {  10:
4:     }                   11:   }
5:   }                     12:   synchronized(l2) {
                           13:     synchronized(l1) {
                           14:     }
                           15:   }
```

Program 5.2: An example program that may thrash often during reproduction

5.3 Yielding to avoid thrashing in reproducing deadlocks

Object abstractions is one way to help reduce thrashing; this in turn helps increase the probability of reproducing a concurrency bug. We show another potential reason for a lot of thrashings using an example and propose a solution to partly avoid such thrashings.

iGoodlock reports a potential deadlock cycle in Program 5.2. In the active random deadlock checking phase, if `Thread 1` is paused first (at line 3) and if `Thread 2` has just started, then `Thread 2` will get blocked at line 9 because `Thread 1` is holding the lock `l1` and it has been paused and `Thread 2` cannot acquire the lock. Since we have one paused and one blocked thread, we get a thrashing. DeadlockFuzzer will un-pause `Thread 1` and we will miss the real deadlock. This is a common form of thrashing that we have observed in our benchmarks.

In order to reduce the above pattern of thrashing, we make a thread yield to other threads before it starts entering a deadlock cycle. Formally, if $(\texttt{abs}(t), \texttt{abs}(l), C)$ is a component of a potential deadlock cycle, then DeadlockFuzzer will make any thread t' with $\texttt{abs}(t) = \texttt{abs}(t')$ yield before a statement labeled c where c is the bottom-most element in the stack C. For example, in the above code, DeadlockFuzzer will make `Thread 1` yield before it tries to acquire lock `l1` at line 2. This will enable `Thread 2` to make progress (i.e. acquire and release `l1` at lines 9 and 11, respectively). `Thread 2` will then yield to any other thread before acquiring lock `l2` at line 12.

We show in our evaluations (see Chapter 6.4) that this optimization is useful in some cases to increase the probability of reproducing deadlocks.

Chapter 6

Evaluation of Active Testing for Multi-threaded Programs

In this chapter, we discuss the results of running our Active Testing tool for Java on several benchmarks. We summarize the overheads and bugs reported from the various analyses and checkers in the CalFuzzer framework running on real-world concurrent Java programs and libraries totaling over 600K lines of code. We discuss some of the deadlocks (Section 6.2) and atomicity violations (Section 6.3) reported from our tool in more detail and present evidence to support the decision of using the lightweight indexing abstraction in Section 6.4.

6.1 Summary of results

We evaluated CalFuzzer on a variety of Java programs and libraries. The following programs were included in our benchmarks: moldyn, a molecular dynamics simulation benchmark from the Java Grande Forum; cache4j, a fast thread-safe implementation of a cache for Java objects; sor, a successive over-relaxation benchmark, and hedc, a web-crawler application, both from ETH [100]; jspider, a highly configurable and customizable Web Spider engine; and jigsaw, W3C's leading-edge Web server platform. We created a test harness for jigsaw that concurrently generates simultaneous requests to the web server, simulating multiple clients, and administrative commands (such as "shutdown server") to exercise the multi-threaded server in a highly concurrent situation.

The libraries we experimented on include Java Collections, Java logging facilities (`java.util.logging`), and the Swing GUI framework (`javax.swing`). Another widely used library included in our benchmarks is the Database Connection Pool (DBCP) and Collections components of the Apache Commons project. We created general test harnesses to use these libraries with multiple threads.

Table 6.1 summarizes some of the results of running Active Testing on several real-world concurrent Java programs comprising over a total of 600K lines of code. Further details are available in [87, 74, 49]. Note that the bugs reported by the active checkers (RaceFuzzer,

		Avg. run time (s)				Number of reported bugs								
Benchmark	LoC	Norm	RF	DF	AF	HR	RF	K	iG	DF	K	Az	AF	K
moldyn	1,352	2.07	42.4	-	-	5	2	0	0	-	-	-	-	-
Apache CFB[1]	3,370	0.139	-	-	0.309	-	-	-	-	-	-	1	1	0
cache4j	3,897	2.19	2.61	-	35	18	2	-	-	-	-	1	1	0
Java Logging	4,248	0.166	-	0.493	-	-	-	-	3	3	2	-	-	-
ArrayList	5,886	0.16	0.24	7.07[2]	0.30	14	7	-	9	9	-	1	1	0
jspider	10,252	4.62	4.81	-	51	29	0	-	0	0	-	28	4	0
sor	17,718	0.163	0.23	-	1.0	8	0	0	0	0	-	0	0	0
hedc	25,024	0.99	1.11	-	1.8	9	1	1	0	0	-	3	0	1
DBCP	27,194	0.60	-	1.4	-	-	-	-	2	2	2	-	-	-
weblech	35,175	0.81	1.36	-	17.01	27	2	1	-	-	-	25	0	0
jigsaw	160,388	-	-	-	-	547	36	-	283	29	-	60	2	1
Java Swing	337,291	4.69	-	28.1	-	-	-	-	1	1	1	-	-	-

Table 6.1: Average execution time and number of bugs reported for each checker implemented with the CalFuzzer framework (LoC: Lines of Code, Norm: Uninstrumented code, RF: RaceFuzzer, DF: DeadlockFuzzer, AF: AtomFuzzer, HR: Hybrid Race Detection, K: Previously known real bugs, iG: iGoodlock, Az: Atomizer). [1]CFB: CircularFifoBuffer). [2]DF seems to perform poorly, but a different harness with a higher normal run time (2.86s) was used.

AtomFuzzer, and DeadlockFuzzer) are real, whereas those reported by the dynamic analyses (hybrid race detector, Atomizer, and iGoodlock) could be false warnings. Although Active Testing may not be able reproduce some real bugs, all previously known real bugs were reproduced, with the exception of AtomFuzzer (see [74] for a discussion on its limitations).

The run-time overhead of CalFuzzer is from 1.1x to 20x. Normally, the slowdown is low since only the synchronization points and memory accesses of interest are instrumented. However, in some cases the slowdown is significant—this is caused when CalFuzzer pauses redundantly at an event. We use precise abstractions (see Section 5.2) to distinguish relevant events, which lessens redundant pauses as shown in Section 6.4.

6.2 Deadlocks found

DeadlockFuzzer found a number of previously unknown and known deadlocks in our benchmarks. Two previously unknown deadlocks were found in Jigsaw. One is shown in Figure 2.3, when the http server shuts down. Another similar deadlock occurs when a `SocketClient` kills an idle connection. It involves the same locks, but are acquired at different program locations. iGoodlock provided precise debugging information to distinguish between the two contexts of the lock acquires.

The deadlock in the Java Swing benchmark occurs when a program synchronizes on a `JFrame` object, and invokes the `setCaretPosition()` method on a `JTextArea` object that is a member of the `JFrame` object. The sequence of lock acquires that leads to the deadlock is as follows. The `main` thread obtains a lock on the `JFrame` object, and an `EventQueue` thread which is also running, obtains a lock on a `BasicTextUI$BasicCaret` object at line number 1304 in *javax/swing/text/DefaultCaret.java*. The `main` thread then tries to obtain a lock on the `BasicTextUI$BasicCaret` object at line number 1244 in *javax/swing/text/Default-Caret.java*, but fails to do so since the lock has not been released by the `EventQueue` thread. The `EventQueue` thread tries to acquire the lock on the `JFrame` object at line number 407 in *javax/swing/RepaintManager.java* but cannot since it is still held by the `main` thread. The program goes into a deadlock. This deadlock corresponds to a bug that has been reported at `http://bugs.sun.com/view_bug.do?bug_id=4839713`.

One of the deadlocks that we found in the DBCP benchmark occurs when a thread tries to create a `PreparedStatement`, and another thread simultaneously closes another `PreparedStatement`. The sequence of lock acquires that exhibits this deadlock is as follows. The first thread obtains a lock on a `Connection` object at line number 185 in *org/a-pache/commons/dbcp/DelegatingConnection.java*. The second thread obtains a lock on a `KeyedObjectPool` object at line number 78 in *org/apache/commons/dbcp/PoolablePrepared-Statement.java*. The first thread then tries to obtain a lock on the same `KeyedObjectPool` object at line number 87 in *org/apache/commons/dbcp/PoolingConnection.java*, but cannot obtain it since it is held by the second thread. The second thread tries to obtain a lock on the `Connection` object at line number 106 in *org/apache/commons/dbcp/PoolablePrepared-Statement.java*, but cannot acquire it since the lock has not yet been released by the first thread. The program, thus, goes into a deadlock.

The deadlocks in the Java Collections Framework happen when multiple threads are operating on shared collection objects wrapped with the `synchronizedX` classes. For example, in the `synchronizedList` classes, the deadlock can happen if one thread executes `l1.addAll(l2)` concurrently with another thread executing `l2.retainAll(l1)`. There are three methods, `addAll()`, `removeAll()`, and `retainAll()` that obtain locks on both `l1` and `l2` for a total of 9 combinations of deadlock cycles. The test cases for Java Collections are artificial in the sense that the deadlocks in those benchmarks arise due to inappropriate use of the API methods. We used these benchmarks because they have been used by researchers in previous work (e.g. Williams et al. [103] and Jula et al. [50]), and we wanted to validate our tool against these benchmarks.

Since DeadlockFuzzer is unsound, if it does not classify a deadlock reported by iGood-lock as a real deadlock, we cannot definitely say that the deadlock is a false warning. For example, in the Jigsaw benchmark, the informative Goodlock algorithm reported 283 dead-locks. Of these 29 were reported as real deadlocks by DeadlockFuzzer. We manually looked into the rest of the deadlocks to see if they were false warnings by iGoodlock, or real deadlocks that were not caught by DeadlockFuzzer. For 18 of the cycles reported, we can say with a high confidence that they are false warnings reported by the iGoodlock algorithm. These cycles involve locks that are acquired at the same program statements, but by different threads. There is a single reason why all of these deadlocks are false positives. The deadlocks can occur only if a `CachedThread` invokes its `waitForRunner()` method be-fore that `CachedThread` has been started by another thread. This is clearly not possible in an actual execution of Jigsaw. Since iGoodlock does not take the happens-before relation between lock acquires and releases into account, it reports these spurious deadlocks. For the rest of the cycles reported by iGoodlock, we cannot say with reasonable confidence if they are false warnings, or if they are real deadlocks that were missed by DeadlockFuzzer.

6.3 Atomicity violations found

Since we do not have atomicity annotations in our benchmark programs, we use the heuristic that any code block that is `synchronized` is atomic in our experiments. The same assumption has been made in the Atomizer tool [31]. A rationale behind this assumption is that programmers often surround a code block with `synchronized` to achieve mutual exclusion, i.e. to ensure that the data inside the code block is accessed without interference from other threads. In other words, programmers often assume that a synchronized code block will behave atomically. Note that this assumption might give some false warnings if a synchronized block need not be atomic in a program. However, this does not affect our general claim that AtomFuzzer gives no false warnings—if the programmer properly annotates atomic blocks, then we get no false warnings.

AtomFuzzer discovered several previously unknown atomicity violations in the JDK 1.4.2 synchronized classes `LinkedList`, `ArrayList`, `HashSet`, `TreeSet`, and `LinkedHashSet`. All these violations lead to uncaught exceptions; therefore, these violations indicate real bugs. Java provides wrappers to Collection classes to make them *thread-safe* in a concurrent program. For example, `java.util.Collections.synchronizedSet(Set s)` wraps a `java.util.Set` object, so that operations on a set are protected by a mutex. AtomFuzzer discovered real atomicity violations in the `removeAll` method.

For example, if we have two `SynchronizedSets` `l1` and `l2`, and call `l1.removeAll(l2)` then the following methods in Program 6.1 get called: `Collections.removeAll` calls `AbstractCollection.removeAll`, which calls `Collections.contains`. The while loop inside `AbstractCollection.removeAll` should be atomic because any changes to `l2` triggers a `ConcurrentModificationException`. However, the statement `c.contains()` inside the loop calls `Collections.contains`, which acquires and releases a lock at each iteration. This allows an atomicity violation in `Collections.removeAll`.

The errors discovered in the Apache Commons Collections library are also due to non-synchronized use of iterators. For example, the `CircularFifoBuffer` is a wrapper class for an circular buffer, which in turn extends from JDK `AbstractCollection`. `CircularFifoBuffer` is a synchronized Buffer, which locks the underlying collection before each operation. However, the implementation of the underlying collection (an `AbstractCollection` in this case) does not take synchronization into account, and uses the iterator in an unsafe manner. This causes an atomicity violation error which results in an exception.

Since AtomFuzzer is unsound, we cannot definitely say that an atomicity violation warning is not an error if AtomFuzzer has not classified the warning as an error. Similarly,

```
Collections.java:
    public boolean removeAll(Collection coll) {
       synchronized(mutex) {
          return c.removeAll(coll);
       }
    }
    public boolean contains(Object o) {
       synchronized(mutex) {
          return c.contains(o);
       }
    }

AbstractCollection.java:
    public boolean removeAll(Collection c) {
       boolean modified = false;
       Iterator e = iterator();
       while (e.hasNext()) {
          if(c.contains(e.next())) {
             e.remove();
             modified = true;
          }
       }
       return modified;
   }
```

Program 6.1: Atomicity violation in Java Collections

we cannot definitely say that an atomicity violation error is not a bug if AtomFuzzer has not classified the error as a bug. For example, in the case of `jspider`, AtomFuzzer reports 28 warnings, 4 errors, and 0 bugs. We cannot definitely say that the remaining 24 warnings are not errors and that the 4 errors are not bugs. In order to better understand the effectiveness of our technique, we manually analyzed the 28 warnings and 4 errors and found that all the remaining warnings (i.e. the warnings that were not classified as errors by AtomFuzzer) are not errors and all the reported errors are not bugs. The results of our manual analysis show that AtomFuzzer is relatively complete for `jspider`; however, they do not imply that AtomFuzzer is complete for all concurrent programs. We next give a detailed description of the results of our manual analysis.

We found that there are two key reasons why some of the warnings are not errors. First, we found that many times the `main` thread releases and acquires a lock while inside an atomic block, but before creating any thread. Therefore, no other thread can interleave between the release and acquire of the lock. However, Lipton's reduction algorithm will give an atomicity

```
public static void initialize() {
   synchronized { // inferred as an atomic block
      synchronized(L) {
         // do something
      }
      synchronized(L) {
         // do something
      }
   }
}

public static void main(String args[]) {
  initialize();
  (new SomeThread()).start();
}
```

Program 6.2: Initialization code causing spurious atomicity violation warnings

violation warning. A simplified code snippet that gives such atomicity violation warning is shown in Program 6.2.

Second, we found that in some situations all atomic blocks that access a particular lock (say L) are synchronized by a common lock (say L'.) In such situations, no other thread can acquire and release the lock L while a thread is in an atomic block and is accessing the same lock L. Therefore, a real atomicity violation cannot happen although Lipton's reduction algorithm will give an atomicity violation warning. A simplified code snippet that gives such atomicity violation warning is shown in Program 6.3.

We observed that there are two reasons for getting atomicity errors that are not bugs. First, we observed that our heuristic for identifying atomic blocks does not match the user intention in some situations. For example, in `jspider` some of the `run` methods that are entry method for threads are `synchronized`. Because of the heuristics we used, AtomFuzzer treats them as atomic. However, it is unrealistic to assume that the entry method of a thread is atomic because such an assumption would make the entire thread atomic. In some other situations we found that a code block has been synchronized over a lock l because the thread calls l.`notify()` or l.`wait()` inside the block. In such scenarios the block should not treated as atomic because a call to l.`wait()` would naturally violate the atomicity assumption by releasing the lock l. However, since our heuristic treats any synchronized block as atomic, we get a false atomicity error report. We can remove these false error reports by modifying our heuristic such that it does not treat such synchronized blocks as atomic.

AtomFuzzer reports some atomicity errors which we have found to be benign. An example of such a benign error is shown in Program 6.4. Here any interleaving of calls to

```
public void foo() {
   synchronized(L') { // inferred as an atomic block
      synchronized(L) {
         // do something
      }
      synchronized(L) {
         // do something
      }
   }
}

Thread 1                 Thread 2
foo();                    foo();
```

Program 6.3: A common lock preventing atomicity violations

```
public static synchronized int getUniqueId() {
  count++;
  return count;
}

Thread 1                 Thread 2
// atomic {              getUniqueId();
  getUniqueId();         ...
  ...
  getUniqueId();
// } end atomic
```

Program 6.4: A benign atomicity violation `getUniqueId()` is correct because the semantics of `getUniqueId()` allows such interleavings. We found such a benign atomicity violation error in `cache4j`.

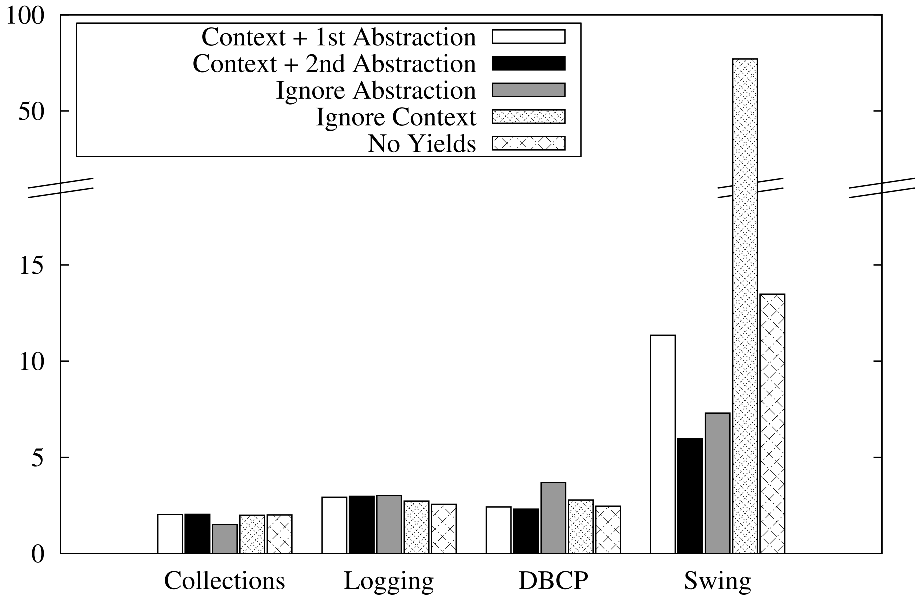
6.4 Evaluation of object abstractions and yielding

We conducted additional experiments to evaluate the effectiveness of various design decisions for DeadlockFuzzer. We tried variants of DeadlockFuzzer: 1) with abstraction based on k-object-sensitivity, 2) with abstraction based on light-weight execution indexing, 3) with the trivial abstraction (all objects map to the same abstract object), 4) without context information, and 5) with the optimization in Section 5.3 turned off. Figure 6.1 summarizes the results of our experiments. Note that the results in Table 6.1 correspond to variant 2, which uses the light-weight execution indexing abstraction, context information, and the yielding optimization in Section 5.3. We found this variant to be the best performer: it created deadlocks with higher probability than any other variant and it ran efficiently with minimal number of thrashings.

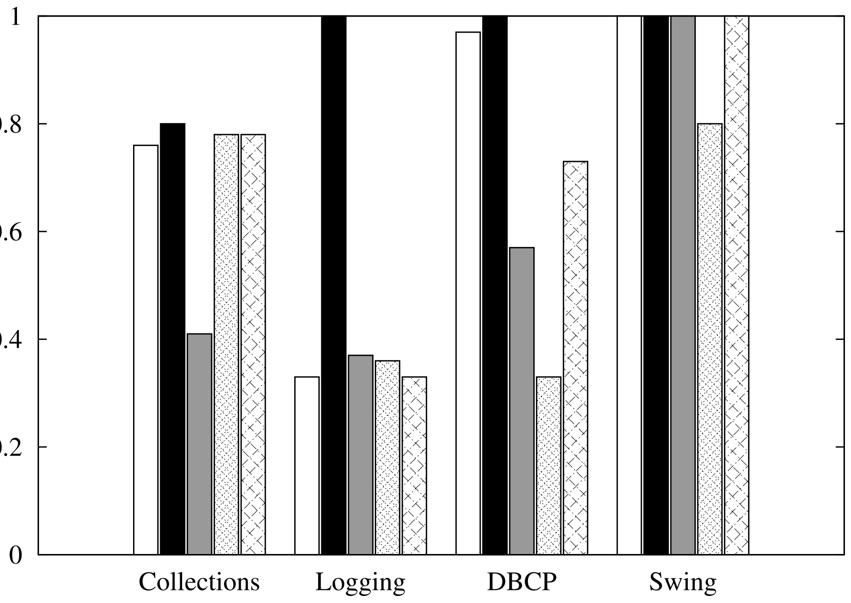
Graph (a) shows the correlation between the variants of DeadlockFuzzer and average run time. Graph (b) shows the probability of creating a deadlock by the variants. Graph (c) shows the average number of thrashings encountered by each variant. Graph (d) shows the correlation between the number of thrashings and the probability of creating a deadlock.

Graph (a) shows that variant 2, which uses execution indexing, performs better than variant 1, which uses k-object-sensitivity. Graph (b) shows that the probability of creating a deadlock is the highest for variant 2 on our benchmarks. The difference is significant for the Logging and DBCP benchmarks. Ignoring abstraction entirely (i.e. variant 3) led to many thrashings in Collections and decreased the probability of creating a deadlock. Graph (c) on the Swing benchmark shows that variant 2 has the least amount of thrashing. Ignoring context information increased the thrashing and the run time overhead for the Swing benchmark. In the Swing benchmark, the same locks are acquired and released many times at several different program locations during the execution. Hence, ignoring the context of lock acquires and releases leads to a huge amount of thrashing.

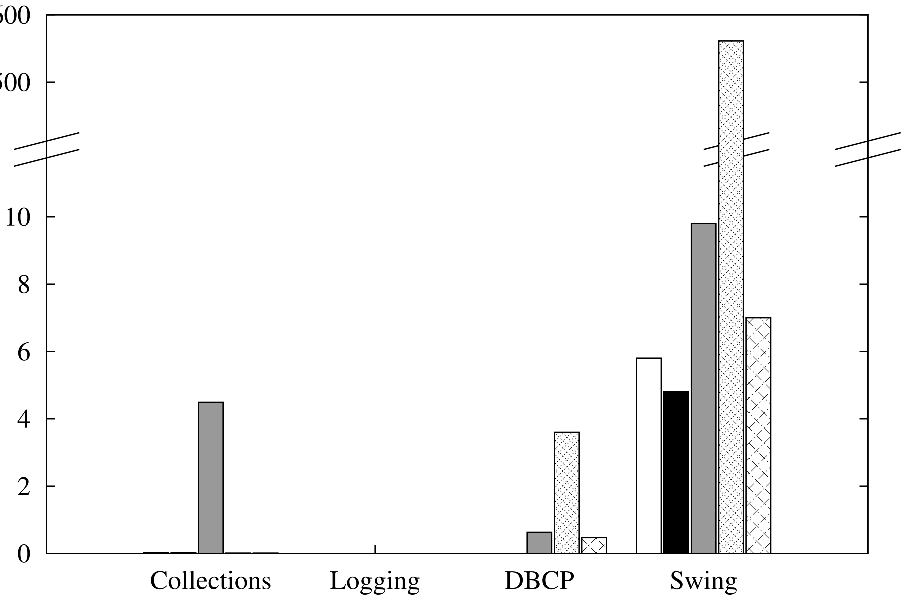
Graph (a), which plots average run time for each variant, shows some anomaly. It shows that variant 3 runs faster than variant 2 for Collections—this should not be true given that variant 3 thrashes more than variant 2. We found that without the right debugging information provided by iGoodlock, it is possible for DeadlockFuzzer to pause at wrong locations but, by chance, introduce a real deadlock which is unrelated to the deadlock cycle it was trying to reproduce. This causes the anomaly in graph (a) where the run-time overhead for Collections is lower when abstractions are ignored, but the number of thrashings is larger. The run time is measured as the time it takes from the start of the execution to either normal termination or when a deadlock is found. DeadlockFuzzer with our light-weight execution indexing abstraction faithfully reproduces the given cycle, which may happen late in the execution. For more imprecise variants such as the one ignoring abstractions, a different deadlock early in the execution may be reproduced, thus reducing the run time.



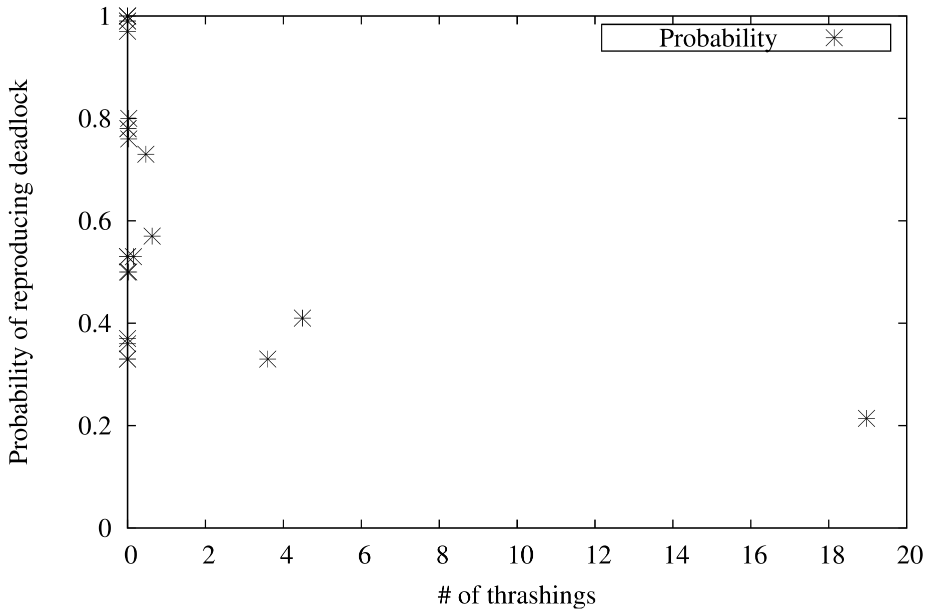
(a) Run time for each variation of contextual information



(b) Probability of reproducing deadlock for each variation normalized to uninstrumented run



(c) Average number of thrashings per run for each variation



(d) Deadlock reproduction probability plotted against number of thrashings for all variants of DeadlockFuzzer on all benchmarks

Figure 6.1: Performance and effectiveness of DeadlockFuzzer variations

Graph (d) shows that the probability of creating a deadlock goes down as the number of thrashings increases. This validates our claim that thrashings are not good for creating deadlocks with high probability. The second variant tries to reduce thrashings significantly by considering context information and object abstraction based on execution indexing, and by applying the yield optimization in Section 5.3.

Chapter 7

Implementation for Distributed Memory Parallel Programs

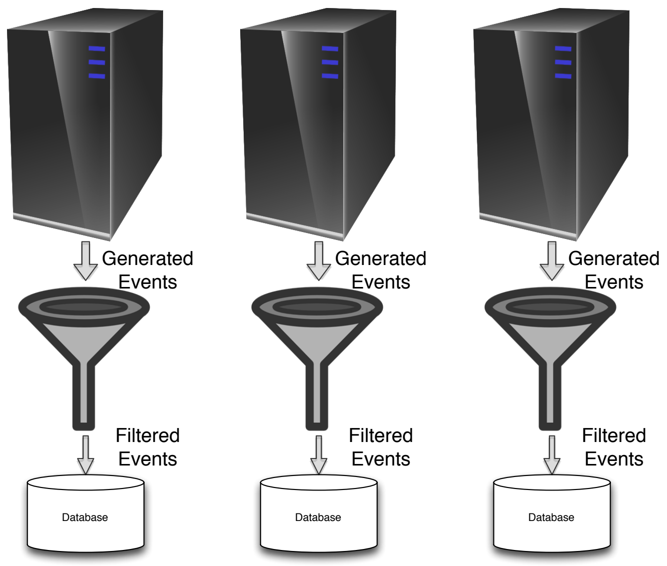
Program analyses for multi-threaded programs do not work well when directly ported to distributed memory systems. Distributed memory programs run at a much larger scale than multi-threaded programs that run on a single node with one or two multi-core processors, making a central monitoring thread unscalable. Also, because of the increased overhead of communication over the network, transmitting analysis information among threads too frequently will clog the network and decrease performance dramatically. Reasoning about bulk memory transfers, instead of word-level accesses, are necessary as well.

We propose a general structure for scalable distributed analysis called the Communication Avoiding Dynamic Analysis (CADA) framework. We divide an analysis into three modes suitable for scalable distributed analysis. UPC-Thrille [76] is an Active Testing tool for UPC, structured in the CADA framework. We discuss implementation details of data race prediction and confirmation for distributed memory parallel programs. We extended UPC-Thrille to handle hybrid memory models using two new techniques for scalability — persistent alias locality heuristic and hierarchical sampling.

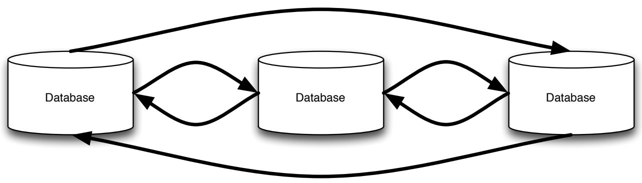
7.1 Communication Avoiding Dynamic Analysis

To address the challenges of writing an efficient and scalable dynamic analysis for large-scale systems, we propose a Communication Avoiding Dynamic Analysis (CADA) framework. We generalize two key properties of a scalable analysis: 1) an analysis should be distributed in nature, without the need of a central thread and 2) communication between threads should be minimized. Based on these properties, we derive the structure of an CADA and some common optimizations that can be performed independent of a particular analysis for communication avoidance.

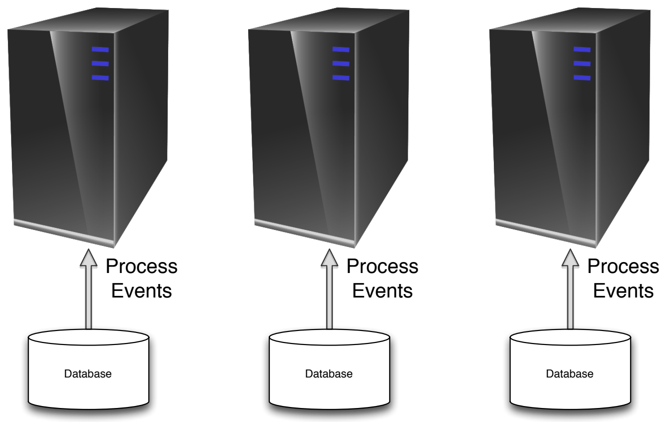
The structure of a distributed analysis is crucial for its scalability. Unlike program analyses for multi-threaded programs, a central analysis thread is prohibitive because of



(a) Local gathering mode



(b) Data distribution mode



(c) Distributed computation mode

Figure 7.1: The three operation modes of Communication Avoiding Dynamic Analysis

load imbalance and concentrated communication traffic. Thus, we need to distribute, as evenly as possible, the computation and communication required in the analysis among the threads in the distributed system to achieve scalability.

An analysis operates in three modes: a local information gathering mode where information required for the analysis is gathered and stored in a local database, a global data distribution mode where analysis data is bucketed and sent to the respective thread responsible for each bucket, and a distributed computation mode where the actual analysis of data happens. We show this general structure in Figure 7.1.

Local gathering mode

For large scale distributed systems, latency and bandwidth issues with communication among threads become a big performance problem. An analysis for software on such platforms have the same problems and need to follow best practices, such as communication avoidance, in order to remain scalable.

The first thing we need to remove from a distributed analysis, coming from a program analysis for multi-threaded programs, is a central monitoring thread that collects events from all the threads in order to perform some analysis. Communication among threads cannot be completely removed, because of the necessity for a global analysis. Thus, our efforts are mainly put into minimizing communication as much as possible.

We take advantage of the fact that SPMD programs are broken up into multiple phases by barriers. As seen in Figure 4.1, a shared access could only race with its neighboring phases. Thus, a check for a conflict among accesses must be made by all the accesses within a barrier phase, with all the neighboring barrier phases. Instead of checking for conflicts at each access, the query for conflict checking can be made at a synchronization point, all at once. This reduces the total overhead of all the conflict checking queries, by combining multiple communication into one.

Between synchronization points, the data generated by events are collected by individual threads and stored locally in a database. The contents and format of the data is analysis dependent, and left unconstrained in the framework. All collected data is held locally, until the next data distribution mode. The effect of this is already great in reducing the communication latency, but it is desirable to reduce the amount (i.e., total bytes of data) of communication as well, in order to reduce the increase in bandwidth usage caused by the analysis.

From our experiences, we have learned that some data gathered from the events are redundant. There could be data that is completely identical (e.g., two reads of one thread from the same memory address range) or data that dominate others (e.g., a write to a memory range by one thread masks a read to the same or sub-range of the memory by the same thread, for purposes of detecting races). A filter, specific to an analysis, can be employed by the analysis writer to compact the data gathered locally by a thread, which will eventually be communicated to other threads.

A simple yet effective filter that is useful to many analyses is the *sampling filter*. For programs with tight inner loops, multiple events might be generated by a statement inside the loop. Although the generated data may not be redundant, the significance of the data drop as the event is generated hundreds or thousands of times.

Data distribution mode

Once data is gathered from all threads locally, it must be distributed among the threads for analysis. How the data is distributed affects the overhead of the analysis, both in terms of computation and communication. Data should be distributed as evenly as possible, because 1) the amount of data assigned to each thread is the amount of work that the thread needs to perform, and 2) communication overhead can be significant if every thread tries to send large data to one particular thread. Unbalanced work loads can cause unnecessary delays for all threads, and slow down the application under analysis.

The data is divided up by some bucketing function that maps the meta-data to a particular thread. Data can be bucketed through hashing, or by some other function of the meta-data. It is desirable that the bucketing is as even as possible, to get better load balancing. For example, in our data race detection work for UPC (Section 7.2), the bucketing function is based on the affinity of the memory location that was being accessed, i.e. the destination location is calculated as

$$\texttt{upc\_threadof}(e.\mathit{memory}) \qquad \text{where } e \text{ is the memory access event .}$$

Although this bucketing scheme does not guarantee even distribution, it follows the natural communication pattern of the original application (requesting the thread which has affinity to the shared memory for access). In other words, if the application makes shared memory accesses in a balanced way, the analysis will not incur any more imbalance while distributing data for analysis.

The framework takes care of sending the data to the appropriate destination thread based on the bucketing function. Because the database is analysis specific, it is left mostly unconstrained except that the framework requires the analysis writer to supply data serialization and unserialization functions to convert data to and from byte streams for transfer.

Distributed computation mode

With the data distributed among the threads in the data distribution mode, each thread performs analysis on its bucket of data. This mode of operation is essentially taking advantage of the parallel hardware on the system to do a distributed analysis. All the data necessary for analysis should be transferred into each thread's local memory (or in the thread's portion of the global heap) in the data distribution mode.

Again, assuming that the data (hence, the workload) is distributed evenly, the analysis should scale well with the size of the system. The total overhead of the analysis on the application program is still dependent on how much computation is required from the analysis.

However, forbidding the use of inter-thread communication in the distributed computation mode should minimize the effects of the analysis on the scalability of the application.

7.2 UPC-Thrille

We have implemented the Active Testing framework for the Berkeley UPC [14] compiler called UPC-Thrille [76] available at `http://upc.lbl.gov/thrille`. We support all operations provided by the UPC v1.2 language specification: memory accesses through pointers to shared, bulk transfers (e.g. `upc_memput`), lock operations, and barrier operations. The framework itself is implemented in the UPC programming language and it can be easily ported to other UPC compiler/runtime implementations, such as Cray UPC. We next describe the implementation of the two phases of Active Testing.

Implementation of race prediction phase

The runtime instrumentation redefines all memory access and synchronization operations by adding "before" and "after" calls into our analysis framework. For example, for any data access we add THRILLE_BEFORE(type, address) and THRILLE_AFTER(type, address) calls before and after the actual data access statement, respectively. When linking the application with our runtime, a write to shared memory `p[i] = 3` translates into:

```
THRILLE_BEFORE(write, p+i);
upcr_put_pshared_val(p+i, 3);
THRILLE_AFTER(write, p+i);
```

During execution, each thread maintains a trace of memory accesses to a particular portion of the shared heap. Whenever a thread accesses the shared heap it has to inform the maintainer of that particular region. Overall, during phase I there are two sources of program slowdown: 1) querying a potentially large access trace and; 2) transmitting the query data over the network. In the rest of this section we describe optimizations designed to reduce the overhead of these operations.

Data structures to represent memory accesses: The data structure to represent memory accesses needs to efficiently support both single address queries as well as the address range queries associated with bulk transfers. Previous work on data races focuses on word level memory accesses and uses hash tables to find conflicting addresses. To efficiently find overlapping intervals, we use interval skip lists (IS-list) [38].

Skip lists [78] are an alternative to balanced search trees with probabilistic balancing. They are much easier to implement and perform very well in practice. Skip lists are essentially linked lists augmented with multiple forwarding pointers at each node. The number of forwarding pointers is called the *level* of a node, which is randomly assigned with a geometric distribution when a node is inserted into the list. Higher level nodes are rare, but can skip

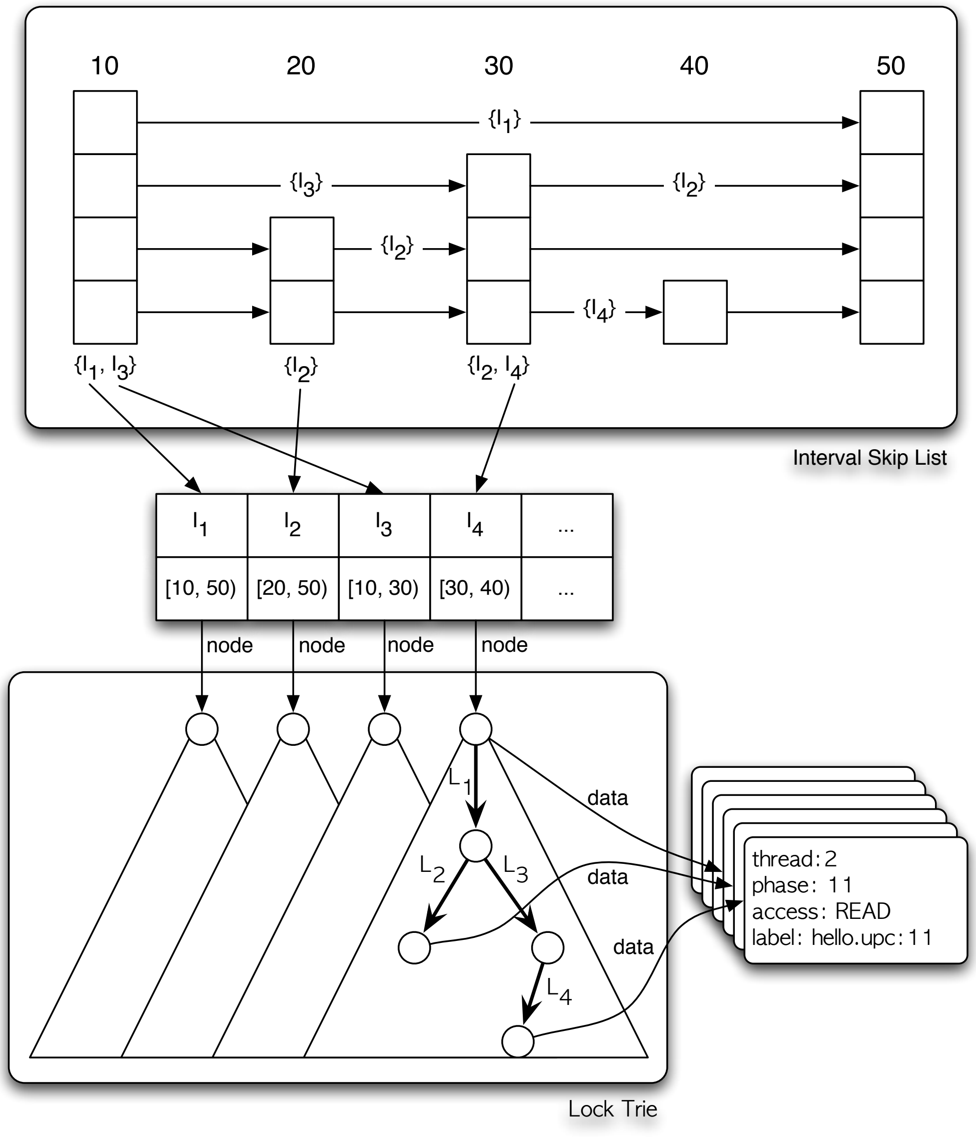


Figure 7.2: Data structures used to efficiently represent memory accesses in UPC-Thrille

over many nodes, contributing to the performance of skip lists (expected time $O(\log n)$ for *insert, delete,* and *search*). Skip lists are also space efficient: a node can be stored in memory with an average of four machine words, one word for the key, one word for the pointer to data, and an average of two forward pointers with $p = 0.5$ as the success probability parameter for the geometric distribution.

An IS-list is essentially a skip list for all the endpoints of intervals it contains, with edges representing the interval spanned by the endpoints. Each node and edge is marked with intervals that cover them. To efficiently handle overlapping queries with minimal space overhead, only the highest level edges that are sub-intervals of I (i.e. edge $(n_1, n_2) \subseteq I$ and $\nexists$ edge (n'_1, n'_2). $(n_1, n_2) \subset (n'_1, n'_2) \subseteq I$) need to be marked, and a node n is marked with I if any of the incoming or outgoing edges of that node is marked with I and $n \in I$. For example, in Figure 7.2, interval $I_2 = [20, 50)$, has markers on the second level forward edge of node 20 and the third level forward edge of node 30, because there are no higher level edges that are contained in I_2.

The operations on IS-lists are time and space efficient. Inserting an interval takes expected time $O(\log^2 n)$ where n is the number of intervals in the IS-list. A search for intervals overlapping a point can be found in expected time $O(\log n + L)$, where L is the number of matching intervals, and searching for intervals overlapping an interval takes expected time $O(\log^2 n + L)$. The expected space required for n intervals is $O(n \log n)$.

Figure 7.2 is an overview of how the database of shared memory accesses is represented. Each memory access event $e = (m, t, l, a, p, s)$ is first grouped by the address range m and inserted into the IS-list. Each interval is associated with a lock trie that represents the locks l held during e. Each node in the trie represents an access with all the locks in the path from the root held. For example, the root of the trie represents an access to m without any locks held. A trie is used to represent locks to efficiently search for accesses racing with e, by only following edges not included in $e.l$. Algorithm 7.1 shows the full steps for finding racing accesses in the trie.

Query Coalescing: The race prediction phase has to perform remote queries at each individual remote memory access to check for conflicts. On a cluster, this amounts to performing additional data transfers for each transfer present in the application.

In our implementation, each thread tracks all the remote accesses locally and delays all the queries until barrier boundaries. Inside barriers, threads coalesce the query data by memory region into larger messages and perform point-to-point communication with the maintainer of each region. Upon receiving information from all other threads, each thread independently computes all conflicting access that happen within its region of the global shared heap.

Our implementation performs both communication - communication and communication - computation overlap using a *software pipelining* approach. Transfers for query data are asynchronously initiated at a barrier operation and overlapped with each other. These transfers are allowed to proceed until the program executes the next barrier, at which point

Algorithm 7.1: FINDRACE(isl, e)

```
Input: IS-list $isl$ and access $e$
Output: An access in $isl$ that races with $e$
1  I ← Intervals in $isl$ that overlap with $e.m$;
2  foreach $interval\ i \in I$ do
3      N ← { i.node };
4      foreach $node\ n \in N$ do
5          if $n.data \neq NULL \wedge n.data.t \neq e.t \wedge n.data.p \mid\mid e.p \wedge (n.data.a = WRITE \vee e.a = WRITE)$ then /* a race is found */
6              $e' = (i, \{l\text{: } l \in$ path from $n$ to root$\}, n.data)$;
7              return $e'$
8          else /* only traverse lock edges not held by $e$ */
9              foreach $c \in n.children$ do
10                 if $c.lock \notin e.l$ then N ← N ∪ {c};
```

Algorithm 7.2: ADDACCESS(isl, e)

```
Input: IS-list $isl$ to add access $e$ while reporting races and removing stronger accesses
1  if $\nexists e' \in isl\ s.t.\ e' \sqsubseteq e$ then
2      if $r \leftarrow$ FINDRACE$(isl,\ e)$ then
3          Report r;
4      else
5          Insert $e$ into $isl$;
6          Remove $\forall e' \sqsupset e$ from $isl$;
```

they are completed, new transfers are initiated and queries are performed for the requests just completed.

Extended Weaker-Than Relation: Keeping track of all shared memory access can incur high space overhead for threads and increase the amount of communication required between threads. Thus, we prune redundant information about accesses that do not contribute to finding additional data races. For example, if a thread reads and writes to the same memory region, only the write information is required, because any races with the read would also imply a race with the write. Similarly, a narrower memory region, an access with more locks held, and a memory region accessed by a lesser number of threads are all redundant information. Formally, the *weaker-than* relation between memory accesses introduced by Choi et al [18] identifies the accesses that can be pruned. We extend this relation to handle memory ranges.

Definition 7.1 (Extended Weaker-Than: $\sqsubseteq$)**.** *For two memory access events e_1 and e_2,*

$$e_1 \sqsubseteq e_2 \Leftrightarrow e_1.m \supseteq e_2.m \ \wedge \ e_1.t \sqsubseteq e_2.t \ \wedge \\ e_1.l \subseteq e_2.l \ \wedge \ e_1.a \sqsubseteq e_2.a$$

where

$$t_i \sqsubseteq t_j \Leftrightarrow t_i = t_j \vee t_i = * \quad (\textit{multiple threads}) \\ a_i \sqsubseteq a_j \Leftrightarrow a_i = a_j \vee a_i = WRITE$$

Only the weakest accesses are stored locally and sent to other threads at barriers. Also, when conflicting races are computed, the weaker-than relation is used in Algorithm 7.2 to prune redundant accesses from multiple threads.

Exponential Backoff: We can further prune redundant access information by dynamically throttling instrumentation on statements that are being repeatedly executed. For each static access label (file, line number, variable), we keep a probability for considering these accesses for conflict detection, initially all set to 1.0. Whenever a data race is detected on a statement, we set the probability to 0, effectively disabling instrumentation for that statement for the rest of the execution. Each time an access is recorded, we reduce the probability by a *backoff factor*, eventually disabling instrumentation for this statement after multiple unsuccessful attempts at finding a conflicting access. For our experiments, we used a backoff factor of 0.9 which was a good balance for effectively finding potential data races with low overhead. In Section 8.1, we discuss the performance gains achieved by these optimizations.

Algorithm 7.3 is the complete scheduling algorithm for race prediction. The global access list is the communication channel to send shared access information between threads. Each thread also maintains local IS-lists to keep track of its memory accesses separately for each phase and affinity. The probability of considering each program statement is initially set to 1.0. Lines 4–9 are the actions performed for each memory access. We probabilistically add the shared access information to the thread's *local* IS-list, while pruning all but the weakest accesses (Algorithm 7.2). Lines 10–15 handle lock acquires and releases. In case of a notify statement (lines 16–20), before notifying the other threads (line 19), we make sure that all pending asynchronous transfers are complete (line 17) and initiate asynchronous transfers of accesses in the current phase (line 18). For wait statements (lines 21–37), we first wait for all other threads (line 22), and then initiate asynchronous transfers of accesses in the current phase (line 23). In lines 24–36, we check for barrier aware potential data race pairs (Definition 4.2) in the previous barrier phases based on local information (islist) and information received from other threads (global_acc_list).

Implementation of race confirmation phase

After collecting potential data race pairs from phase I, we run the race testing phase on each pair to confirm they are real and observe the effects of the race in isolation. Algorithm

Algorithm 7.3: ThrilleRacerScheduler()

```
Initially,
Global: $\forall t \in \mathcal{T}, p \in \mathbb{N}, o \in \mathcal{T}.\ global\_acc\_list[t,p,o] = \emptyset$
Thread local: $p = 0$, $L = \emptyset$, $\forall s.\ prob[s] = 1.0$, and $\forall p \in \mathbb{N}, o \in \mathcal{T}.\ islist[p,o] = \emptyset$
while $i$ := next instruction of thread $t$ do
    switch $i$ do
        case $i$ = MEM(m,a,s)
            $e \leftarrow (m, t, L, a, p, s)$;
            if random() < prob[s] then
                AddAccess($islist[p, owner(m)], e$);
                $prob[s]$ *= BACKOFF;
            Execute $i$;
        case $i$ = LOCK(l)
            Execute $i$;
            $L \leftarrow L \cup l$;
        case $i$ = UNLOCK(l)
            $L \leftarrow L \setminus l$;
            Execute $i$;
        case $i$ = UPC_NOTIFY
            Synchronize all pending transfers;
            foreach $t' \neq t$ do Asynchronously send $islist[p,t']$ to $global\_acc\_list[t,p,t']$;
            Execute $i$;
            $p$++;
        case $i$ = UPC_WAIT
            Execute $i$;
            foreach $t' \neq t$ do Asynchronously send $islist[p,t']$ to $global\_acc\_list[t,p,t']$;
            /* Check for races among all accesses in phase $p-2$ */
            foreach $t' \neq t$ do
                foreach $e \in global\_acc\_list[t', p-2, t]$ do
                    AddAccess($islist[p-2, t], e$);
            /* Check races in phases p-2, p-3 */
            foreach $e \in islist[p-2, t]$ do
                AddAccess($islist[p-3, t], e$);
            /* Check races in phases p-2, p-1 */
            foreach $e \in islist[p-2, t]$ do
                AddAccess($islist[p-1, t], e$);
            foreach $t' \neq t$ do
                foreach $e \in global\_acc\_list[t', p-1, t]$ do
                    AddAccess($islist[p-1, t], e$);
            $p$++;
        otherwise
            Execute $i$;
```

Algorithm 7.4: ThrilleTesterScheduler(s_1, s_2)

```
   Input: Potential race pair $s_1, s_2$
1  Initially,
   Global: $Prob = 1.0, \forall t \in \mathcal{T}.\ sem[t] = 1,\ pending[t] = NULL$
2  while $i$ := next instruction of thread $t$ do
3      if $Prob > 0 \wedge i = MEM(m,a,s)$ then
4          if $s = s_1 \vee s = s_2$ then
5              if $\exists t'.\ pending[t'].m \cap m \neq \emptyset \wedge (pending[t'].a = WRITE \vee a = WRITE) \wedge$
               $pending[t'].s \neq s$ then
6                  Report race between threads $t$ and $t'$;
7                  $sem[t']$.signal();
8                  $Prob = 0$;
9              else if $random() < Prob$ then
10                 $pending[t] = (m, a, s)$;
11                 $sem[t]$.wait(TIMEOUT);
12                 $pending[t]$ = NULL;
13                 $Prob$ *= BACKOFF;
14     Execute $i$;
```

7.4 shows the complete scheduling algorithm. We use an exponential backoff optimization similar to phase I to keep the overhead of phase II low and achieve scalability. `bupc_sems`, an extension in the Berkeley UPC runtime, are used as semaphores to control the execution order of threads. Each thread announces the memory access it is currently pending on in the global data structure *pending*. Here, we only need to consider the shared memory accesses in the program (line 3). If the statement label matches one of the statements in the potential data race pair (line 4), we first check if any other thread is pending on the other statement (line 5). If one is found to be pending on an access with an overlapping memory address and either access is a write, we report a real race and signal the other thread to proceed. Once a real race is found, we disable testing (line 8) and continue execution of the program normally and observe if any errors occur due to the race.

If no other pending thread meeting the race criteria is found, we probabilistically post the information about the access (line 10) and pause the thread for some time (line 11). At line 12, either a real race was found and the semaphore released by the other thread (line 7) or the timeout could have expired. In either case, we no longer announce this thread as pending on the memory access and reduce the probability of pausing.

Phase II is not guaranteed to be a sound approach—it cannot confirm all real data races. However, the approach was able to confirm all previously known races in our benchmarks.

7.3 Extensions to hybrid memory models

Well optimized UPC programs usually cast `pointers-to-shared` (e.g. `shared int *`) to C proper pointers (e.g. `int*`) and Section 7.2 misses a large class of data races introduced by memory aliases. Furthermore, the presence on non-blocking communication operations [11] introduces another class of data races. As non-blocking communication is a background asynchronous activity that can be overlapped with computation, memory accesses within a thread can race with the communication operations initiated by the same thread. MPI programs face a similar problem when using the MPI_Isend/IRecv non-blocking communication primitives. A complete solution for finding both "traditional" and races introduced by non-blocking communication needs to track all the memory references, including those using C pointers, as well as communication calls.

In this section, we discuss the overhead of data race detection and how it is exacerbated when dealing with hybrid memory models that alias the same address space. We describe two techniques — persistent alias locality heuristic and hierarchical sampling — that effectively reduces the overhead for data race detection in programs using hybrid memory models.

The overhead of data race detection

Runtime overhead due to instrumentation is recognized as a problem that dynamic race detectors have to address. Commercial tools for C programs such as the Intel Thread Checker or the Sun Thread Analyzer, usually provide full coverage at the expense of 600X execution slowdown [93] on scientific OpenMP programs with small memory footprints. Average overheads on other scientific programs for the Intel Thread Checker have been reported [83] around 200X and as high as 485X.

Sampling techniques have been introduced by Arnold and Ryder [5] and later adopted in other bug finding tools [6, 61] for parallel programs. The efficacy of these techniques is determined by the granularity of the instrumented code region and the sampling strategy. Tools [6] for finding bugs in programs running on managed runtimes (e.g. Java) tend to use instruction level sampling; the additional instrumentation overhead is not perceived as unacceptable since the runtime already manages object metadata and access. These systems usually observe up to 3X slowdowns for non-scientific applications and data is not available for HPC applications. The equivalent of instruction sampling is performed in distributed memory tools such as DAMPI [99] and UPC-Thrille which track communication calls.

Recently, Marino et al [61] proposed a technique to coarsen the sampling control from instruction level to function level. They use a compiler to generate instrumented and uninstrumented versions of functions and select the appropriate copy at run-time. The instrumented version of a function monitors every memory reference during its execution. Their LiteRace tool introduces up to 3X overhead while providing good coverage on non-scientific programs; it has not been evaluated on scientific programs. In the rest of this thesis, we refer to this technique as function sampling. One reason that function sampling outperforms instruction sampling is that it amortizes the cost of tracking memory references better:

function sampling executes one branch/decision per function call while instruction sampling executes one branch/decision per instruction.

Several sampling strategies have been proposed and evaluated for non-scientific programs. Random sampling has been shown to provide poor coverage. SWAT [39] detects memory leaks and uses an approach where the execution of code segments is sampled at a rate inversely proportional to their execution frequency. LiteRace uses a bursty sampler, where the execution of a function is sampled initially at a 100% rate and the sample rate is progressively reduced until it reaches a lower bound. Both approaches try to give priority to regions of code rarely executed and give priority to the first execution of any code region.

The implementation of UPC-Thrille described in Section 7.2 uses instruction-level instrumentation with a bursty sampling similar to LiteRace. UPC-Thrille instruments every memory access performed using `pointer-to-shared`, either at word granularity or using bulk `memget/memput` memory operations.

In order to provide a complete data race detection solution we have modified UPC-Thrille and the Berkeley UPC compiler to track all memory references, including all references through C proper pointers. We provide a well optimized implementation of instruction sampling that makes extensive use of C macro-definitions to eliminate function call overheads for the instrumentation code. Every memory reference is examined using a bursty sampling strategy. We have also implemented function sampling with the same bursty strategy.

For any sampled memory reference, the implementation checks whether the address resides within a thread's private address space or within the global address space. This check requires integration with the UPC runtime memory management module and it is an expensive operation, common to PGAS languages. References to the private address space are ignored as they cannot race. Global references are inserted into the UPC-Thrille internal data structures and further checked against other references.

We distinguish three types of overhead for data race detection: 1) *instrumentation overhead* is introduced by the checks to prune the non-interesting data accesses; 2) *computation overhead* is introduced by the operations on internal data structures to manage the interesting accesses; and 3) *communication overhead* is introduced by the exchange of conflicting accesses between threads. Thus, private references contribute only to instrumentation overhead while global references also contribute to the computation and communication overhead.

We have performed experiments with instruction sampling using the default backoff factor of 0.9 and with multiple function sampling strategies. Our results indicate that for the NAS Parallel Benchmarks function sampling is not a scalable strategy. For most benchmarks class B and up, experiments when instrumenting only the first invocation of a function did not terminate: some exhausted the available memory while some were manually terminated after observing 1000X slowdown. Results for instruction level sampling indicate that this approach is able to find races with up to 65X slowdown. Detailed results are presented in Section 8.2.

This behavior contradicts the intuition that function sampling scales better than instruction sampling. The performance reversal is caused by the granularity of control over instrumentation being too coarse: as loops within these benchmarks execute billions of references, function sampling tracks billions of references.

As function level sampling does not work and instruction sampling introduces a 65X overhead that is unacceptable when running at scale, our implementation uses two techniques to reduce the number of tracked memory references without sacrificing the precision of the analysis.

The first optimization reduces the overhead of instrumentation by exploiting the insight that *aliases are persistent* in PGAS programs: once one is created it will point in the same region (private or global) for a long period of time. Using this we can eliminate the overhead introduced by looking up the physical memory layout inside the language runtime.

The second optimization reduces overhead using hierarchical sampling. By combining function and instruction sampling we amortize the cost of instrumentation while retaining fine grained control over the number of events sampled.

Exploiting the persistence of locality in aliases

PGAS languages, such as UPC, Titanium, CAF, Chapel and X10, provide the abstraction of a shared memory address space. Data residing in this space is accessible through references to variables that have a particular type, e.g. `pointer-to-shared` type in UPC or `global` in Titanium[1].

The memory management inside any PGAS language runtime is complex due to the need to provide globally addressable memory and to support data layouts, e.g. block cyclic layouts. Thus, a reference to a global object is orders of magnitude [46] more expensive than a local reference, through a C pointer in the UPC case. Application developers aggressively cast global references to local and compiler optimizations [56, 51, 57] have been explored to *privatize* global references.

For every local memory reference, the data race detection code needs to perform the inverse up-cast operation and check whether the address is globally visible. This operation is also orders of magnitude more expensive than a regular memory load/store.

In our implementation we limit the number of up-casts performed at run-time using the intuition that aliases/locality are persistent: during the program execution a reference will access only the private space or only the global space, independent of its static data type. This assumption allows the analysis to determine at run-time the locality of any reference only once and cache the result for the rest of the execution. In our implementation, we add a shadow variable to cache the locality of every memory reference expression.

The persistence of locality assumption is valid in all of our test programs and it does not decrease the precision of the analysis. The heuristic may lead to false negatives (miss real data races) when the underlying assumption is not valid for the program. However, the technique can be trivially generalized for programs with a more dynamic behavior. As casts in PGAS languages are complicated and are implemented as runtime calls, any casting

[1]Actually, in Titanium any reference is global by default and the language provides local qualifiers.

call can be modified to invalidate the locality information cached. The performance of this approach is determined by the ratio of casts to memory references performed by the program at run-time. The additional overhead for realistic programs is likely to be negligible in practice.

Hierarchical sampling

For every memory reference there are two sources of run-time overhead. Instrumentation overhead is introduced to decide whether the reference should be recorded and computation overhead is introduced when recording the reference in the tool's internal data structures. By reducing the instruction sampling rate, one can clearly reduce overhead, but at the expense of program coverage. To provide both low overhead and good coverage we propose a hierarchical sampling approach which combines the fine grained control of instruction sampling with the overhead amortization provided by function sampling. By using a good hierarchical sampling strategy, we can reduce the instrumentation overhead while retaining the ability to sample from a diverse context with less redundancy. Using the concept of code regions, we formally define instrumentation and hierarchical sampling.

Definition 7.2 (Code regions)**.** *We inductively define code regions. By definition, the smallest unit of a code region is a memory reference (read or write). A code region is a reference or a sequence of one or more code regions. The entire program is the largest code region. Each code region R has a label, denoted as $\#R$.*

Functions, loop bodies, basic blocks etc. are examples of code regions. We assume structured code, i.e. that all code regions are properly nested.

Definition 7.3 (Region stack)**.** *During program execution, a region stack RS is maintained. Similar to a call stack, when a region R is entered, the label of the region $\#R$ is pushed to RS. When exiting a region, the last label is popped from the stack. At the beginning of a program execution, RS is initially empty.*

Definition 7.4 (Instrumentation)**.** *Instrumentation is a transformation of a code region $R \to R^{inst}$.*

If R is a memory reference (base case),

```
               if check-reference(#R :: RS) then
R^inst =       |  log(#R)
               R
```

Else, if R is a sequence of regions $[R_1, R_2, \ldots, R_n]$,

```
$R^{inst}$ =    if $\textit{check-region(\#R :: RS)}$ then
                    $RS = \#R :: RS;$
                    $[R_1^{inst}, R_2^{inst}, \ldots, R_n^{inst}];$
                    $RS = pop(RS)$
                else
                    $[R_1, R_2, \ldots, R_n]$
```

By specializing the *check-reference* and *check-region* and choosing the region granularity, we can implement multiple sampling algorithms. For example, instruction sampling with an exponential backoff (strategy **I** in the experiments presented in Section 8.2), is implemented as the following functions. The map $p : label \rightarrow \mathbb{R}$ contains the (dynamic) sampling probabilities of regions.

$\forall \#R \in Statements.\ p(\#R) = 1.0$

```
$\textit{check-reference(\#R :: RS)} =$
if $\textit{rand()} < \textit{p(\#R)}$ then
    $p(\#R) *= BACKOFF\_FACTOR;$
    return $true$
else
    return $false$
```

$\textit{check-region(x)} = true$

Function sampling as introduced by the LiteRace [61] implementation is defined as follows. The region is a whole function and the *sample-strategy* function depends on the strategy of sampling, such as a fixed probability, random or an adaptive strategy.

$\textit{check-reference(x)} = true$

$\textit{check-region(\#R :: RS)} = \textit{sample-strategy}(\#R)$

Intuitively, the *check-reference* function decides what events should be logged at run-time, while the *check-region* function provides control over the granularity of these decisions. We propose a hierarchical sampling strategy that combines instruction sampling with function sampling. The combination of hierarchical sampling with the aliasing run-time heuristic is referred to as **HA** and described as follows. This implementation uses exponential backoff at both individual reference and function granularity.

```
$\forall \#R \in \mathit{Statements} \cup \mathit{Functions}.\ p(\#R) = 1.0$

$\mathit{check\text{-}reference(\#R :: RS) =}$
if $p > 0 \wedge rand() < p(\#R)$ then
    if $\mathit{is\text{-}local\text{-}access(R)}$ then
        // locality persistence heuristic
        $p(\#R) = 0;$
        return $false;$
    else
        $p(\#R) *= STMT\_BACKOFF\_FACTOR;$
        return $true;$
else
    return $false$

$\mathit{check\text{-}region(\#R :: RS) =}$
if $p > 0 \wedge rand() < p(\#R)$ then
    $p(\#R) *= FUNC\_BACKOFF\_FACTOR;$
    return $true$
else
    return $false$
```

In Section 8.2 we present our evaluation results that show overheads for single-level and hierarchical sampling. Hierarchical sampling is the most efficient with lowest overhead. We also show that the persistent alias locality heuristic is necessary in addition to hierarchical sampling for some benchmarks to maintain low overheads.

Chapter 8

Evaluation of Active Testing for Distributed Memory Programs

In this chapter, we evaluate the effectiveness of Active Testing for distributed memory parallel programs. We apply the race detector and tester from Chapter 7 on several benchmark UPC programs. We discuss the computation and communication overheads caused by the analysis and explain the races found in detail.

To evaluate our techniques for hybrid memory models, we ran our experiments with the persistent alias locality on and off for a variety of sampling strategies including hierarchical sampling, which showed the best performance.

8.1 Evaluation of UPC-Thrille for data race detection

We evaluate data race detection on UPC fine-grained and bulk communication benchmarks. For implementations using bulk communication primitives we use the NAS Parallel Benchmarks (NPB) [68, 69], releases 2.3, 2.4, and 3.3. The fine-grained benchmarks reflect the type of communication/synchronization that is present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling.

The *guppie* benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction. The amount of work is static and evenly distributed among threads at execution time. The *mcop* benchmark solves the matrix chain multiplication problem [21]. This is a classical combinatorial problem that captures the characteristics of a large class of parallel dynamic programming algorithms. The matrix data is distributed along columns, and communication occurs in the form of accesses to elements on the same row. The *psearch* benchmark performs parallel unbalanced tree search [73]. The benchmark is designed to be used as an evaluation tool for dynamic load balancing strategies.

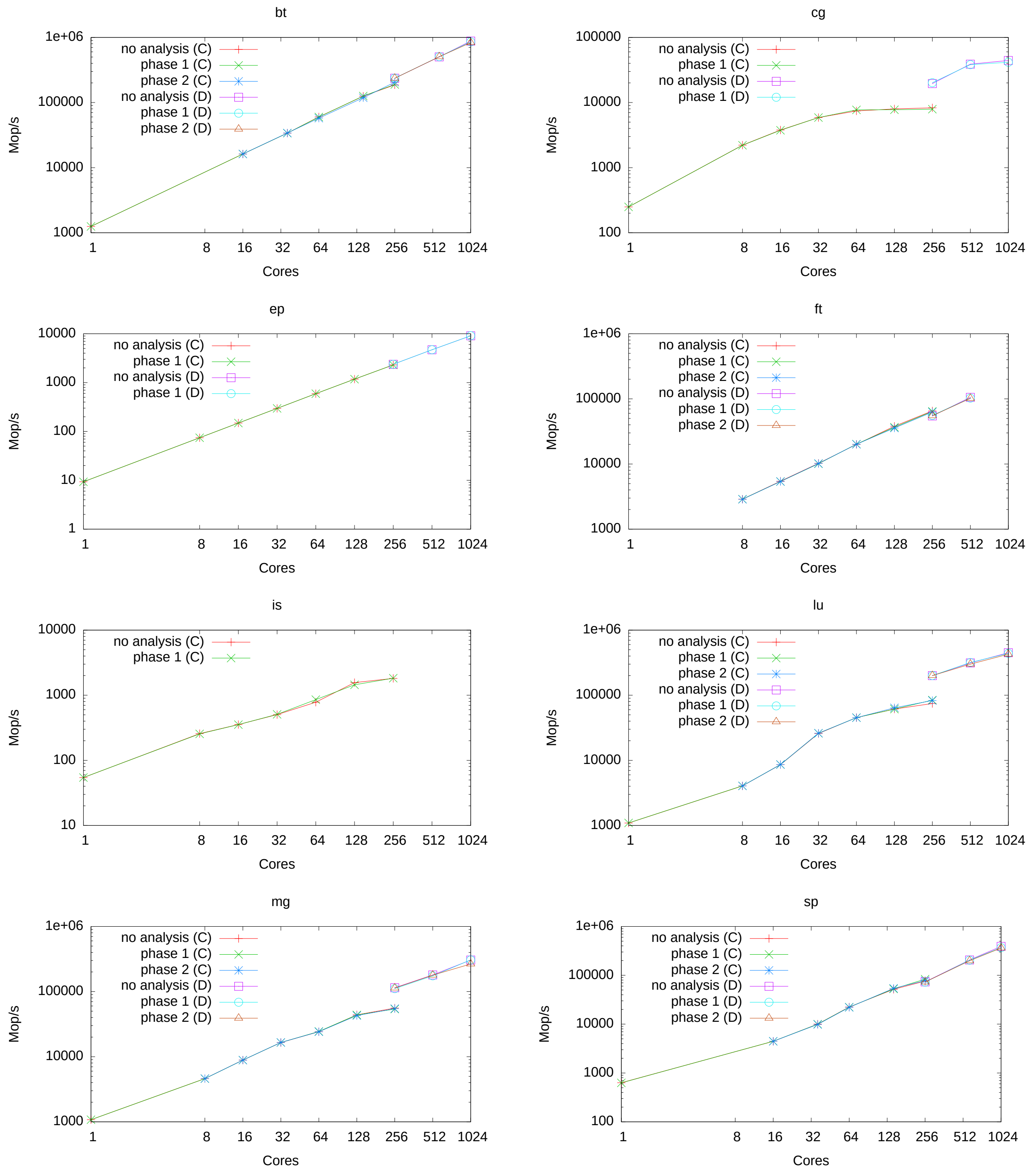


Figure 8.1: Scalability of Active Testing up to 1024 cores for NPB classes C and D. Class D was not available for benchmark IS. We were unable to run FT class C on 1 core and class D on 1024 cores.

				ThrilleRacer			ThrilleTester		
Benchmark		LoC	Runtime	RT	OH	Pot. races	Avg.RT	Avg.OH	Conf. races
guppie		277	2.094s	2.346s	1.120x	157 (2)	2.129s	1.017x	2
knapsack		191	2.099s	2.412s	1.149x	2 (2)	2.136s	1.018x	2
laplace		123	2.101s	2.444s	1.163x	0 (0)	-	-	-
mcop		358	2.183s	2.198s	1.007x	0 (0)	-	-	-
psearch		777	2.982s	3.037s	1.018x	11 (3)	3.095s	1.038x	2
NAS Parallel Bench.	FT 2.3	2306	8.711s	9.243s	1.061x	25 (2)	9.131s	1.048s	2
	CG 2.4	1939	3.812s	3.831s	1.005x	0 (0)	-	-	-
	EP 2.4	763	10.022s	10.109s	1.009x	0 (0)	-	-	-
	FT 2.4	2374	7.036s	7.045s	1.001x	6 (1)	7.334s	1.042x	1
	IS 2.4	1449	3.073s	3.106s	1.011x	0 (0)	-	-	-
	MG 2.4	2314	4.895s	5.045s	1.031x	9 (2)	4.955s	1.012x	2
	BT 3.3	9626	48.78s	49.04s	1.005x	40 (8)	49.15s	1.008x	0
	LU 3.3	6311	37.05s	37.22s	1.005x	0 (0)	-	-	-
	SP 3.3	5691	59.56s	59.70s	1.002x	32 (8)	61.36s	1.030x	0

Table 8.1: Results for race-directed Active Testing on a 4 core workstation. We present results for NPB class A.

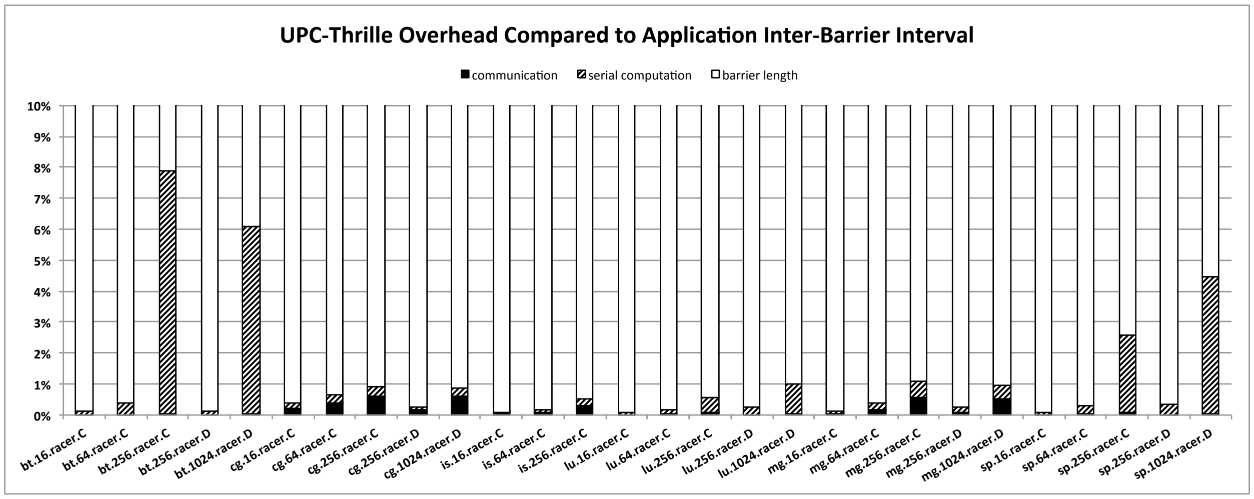


Figure 8.2: Communication and computation overhead of UPC data race detection

UPC-Thrille on Shared Memory Systems

In Table 8.1 we show a summary of results obtained for UPC-Thrille when running the benchmarks on a quad-core 2.66GHz Intel Core i7 workstation with 8GB RAM. We report the total lines of source code (LoC) for each benchmark and the runtime of the original program without any analysis. We report the runtime (RT) and overhead(OH) of the program with race prediction (phase I) enabled and the average runtime and overhead for program re-execution with race confirmation (phase II).

The total number of potential races predicted by phase I is reported in column six. This number reflects the number of racing memory accesses performed throughout the application execution. The number in parentheses represents the unique pairs of racing statements reported by UPC-Thrille that are associated with the runtime races. Each pair of racing statements identified in phase I is tested in phase II and we report the number of confirmed races in the last column.

The race prediction phase added for most benchmarks only a small runtime overhead of up to 15%. The overhead of the race confirmation phase is determined by the granularity of the delays (pauses) introduced in the thread schedule, as well as by the dynamic count of such pauses. For all experiments, the overhead of phase II was negligible when using a delay of 10ms.

Our results demonstrate that UPC-Thrille is able to precisely detect and report the races present in the benchmarks evaluated. For all these benchmarks, the races manifest regardless of the concurrency of the execution or the actual input set. Any testing run at any concurrency will uncover the same set of races.

Races Found

guppie: These races are expected in the program, as random updates are made to a global table. One race is between a read of a table entry from one thread and a write to the same entry from another thread. The other race is between two writes from different threads to the same location in the table.

knapsack: This is our example program in Figure 4.1. Through Active Testing, we can confirm that both races are indeed real. Furthermore, by controlling the order in which the race is resolved, we can force the program into an error. If either initial value is read before the write, the verification check of the solution fails.

psearch: The races are in code that implements work stealing. Shared variables hold the amount of stealable work available for each thread. A real race can result in work stealing to fail, but does not affect the correctness of the program because of carefully placed mutexes. One of the predicted races is unrealizable because of this custom synchronization.

NPB 2.4 FT: This race is real but benign, as all threads initialize the variable `dbg_sum` to 0 at the same time.

NPB 2.4 MG: Two shared variables are read by each thread and then reset to 0 by thread 0. It seems highly suspicious that there is no barrier to wait for all the reads. We reproduced both the races but it did not affect the solution computed. After inspecting the code, the variables are actually used only by thread 0 for reporting purposes.

NPB 2.3 FT: The races in this version is quite devious. The accesses to a shared variable `dbg_sum` is protected by a lock `sum_write`. Then how could the accesses be racing? It turns out that each thread holds a different lock, because the wrong function `upc_global_lock_alloc()` was called to allocate the lock — a different global lock is returned to each calling thread. `upc_all_lock_alloc()`, a collective function that returns the same lock to all threads, should have been used instead.

NPB 3.3 BT and SP: The races predicted by phase I in these benchmarks cannot be confirmed in phase II, because these benchmarks use custom synchronization operations. Both benchmarks implement point-to-point synchronization operations where one thread performs a write operation (`write a`) followed by a `upc_fence` (null strict memory reference), while another thread is polling on the value of the variable `a`. The false positives are not caused by the strict operation per se, but by lack of semantic information about the application.

Scalability of UPC-Thrille on Distributed Memory Systems

Some applications might have races that occur only at certain concurrency levels; thus, the overall scalability of UPC-Thrille is important. In Figure 8.1 we present results for NPB class C scaled up to 256 cores and class D up to 1024 cores on the Franklin [35] Cray XT4 system at NERSC. The nodes contain a quad-core AMD Budapest 2.3GHz processor, connected with a Portals interconnect. The latency for this network using Berkeley UPC [14] is around 11μs for eight byte messages.

For all benchmarks, we plot the original application speedup (*no analysis*) and the speedup with race prediction enabled (*phase II*). For benchmarks with races predicted, we also present the scalability of the confirmation phase (*phase II*). The average slowdown of both phase I and phase II are less than 1%. The maximum slowdown observed for phase I was 8.1% for IS class C at 128 cores. For phase II, MG class D at 1024 cores had the maximum slowdown of 15%. These results are obtained with an exponential backoff of 0.9.

For most of the benchmarks, the race prediction and confirmation phases scale well. For phase I, our implementation introduces overhead mostly in barrier operations. Figure 8.2 compares the average per-barrier overhead of Active Testing with the average application inter-barrier time, noted as *barrier length.* The *barrier length* is computed as the total application original execution time divided by the number of barriers. The average per-barrier overhead of Active Testing is computed as the total overhead divided by the number of barriers and we further sub-divide it into computation and communication overhead. *Serial computation* is the average overhead of the IS-lists lookup while *communication* is the average overhead of communication added by UPC-Thrille. Our implementation overlaps UPC-Thrille specific communication with both internal lookups and the whole computation

				Overhead				
Bench	LoC	Runtime(s)	#Races	NL	HA.5	IA	FA0	I
guppie	271	19.070	2(2) + 0(0)	54.9%	54.2%	53.7%	DNF	74.9%
psearch	803	0.697	3(1) + 2(2)	2.48%	10.8%	666%	8.01%	6490%
BT 3.3	9698	189.48	7(0) + 3(1)	0.574%	1.16%	77.6%	DNF	-
CG 2.4	1654	39.573	0(0) + 1(1)	1.09%	27.6%	57.6%	DNF	2579%
EP 2.4	678	54.453	0(0) + 0(0)	-0.618%	0.805%	2.09%	4.74%	111%
FT 2.4	2289	62.663	2(2) + 0(0)	0.601%	30.1%	121%	DNF	2744%
IS 2.4	1426	5.130	0(0) + 0(0)	0.376%	119%	159%	DNF	1201%
LU 3.3	6348	155.997	0(0) + 44(2)	-0.425%	-	75.7%	DNF	-
MG 2.4	2229	18.687	2(2) + 4(0)	0.336%	176%	632%	DNF	2020%
SP 3.3	5740	247.937	10(0) + 3(1)	0.160%	0.861%	29.1%	DNF	-

Table 8.2: Statistics for the NAS Parallel Benchmarks class C, guppie and psearch running on 16 cores. We report the races found as A(B) + C(D), where A represents the number of races detected by the original UPC-Thrille tool with B of them confirmed, and C represents the additional number of races detected with our extensions with D of them confirmed through phase II. Some execution overheads are omitted (-), due to configuration errors.

already present in the application between barriers. Thus, *communication* measures only residuals after overlapping and in most cases lookups are the main cause of slowdown.

For reference, CG class C running on 256 cores (CG-C-256) executes on average a barrier every 335μs, CG-D-256 every 2.7ms, while CG-D-1024 executes one at 1ms intervals. In all these cases, the race prediction phase has a low overhead, accounting for a few percent slowdown. These considerations indicate that Active Testing has a good scalability potential.

All optimizations applied provide good performance benefits. *Weaker-than* and the exponential back-off reduce the volume and frequency of communication operations. For example, in MG class C running on 64 cores, there were a total of 4.7M shared accesses (memput, memget) for all threads. With *weaker-than*, we pruned 2.1M accesses (46%), mostly shared reads because writes are weaker than reads (Definition 7.1). With exponential backoff (factor=0.9), we further pruned to 53K accesses but were still able to predict all the data races as before. Each thread communicated an average of 10 bytes per barrier. The maximum bytes sent by a thread at a barrier was 23KB, but this number went down after the effects of dynamic throttling kicked in.

Format	S[A][F]
Sampling	H: hierarchical, I: instruction-level, F: function-level NL: no instrumentation on local accesses
Alias	Indicates the use of the persistence alias heuristic
Factor	Back-off factor for sampling at function level
Example	HA.5: Hierarchical sampling with alias heuristic and back-off factor of 0.5

Table 8.3: Key for labels of hierarchical sampling strategies

8.2 Evaluation of techniques for hybrid memory models

We evaluate our techniques for hybrid memory models on the same benchmarks as in Section 8.1. The difference is that we also account for all indirect loads and stores made by each thread that may potentially alias the global portion of the heap.

The experimental results are obtained on a Cray XE6 system [45] composed of nodes containing two twelve-core AMD MagnyCours 2.1 GHz processors. The system has two nodes attached to a Gemini network interface card, forming an overall 3-D torus network with 6,384 nodes. The network provides a bandwidth of 9.375 GBytes/sec per direction in 10 directions. The maximum injection bandwidth per node is 20GB/s.

We evaluate the performance of our data race detection tool on 10 UPC programs written in different programming styles. Table 8.2 summarizes the results for running the benchmarks on a single Cray XE6 node with 16 threads. For each benchmark we evaluate the overhead of several configurations of the tool. Instruction sampling is denoted by **I** and for this configuration we report results with the default setting of 0.9 instruction backoff factor as in the previous section. Function sampling is denoted by **F**, while hierarchical function and instruction sampling is denoted by **H**. For hierarchical sampling, instructions are sampled with the default values for **I**, while the numbers at the end of the label denote the function backoff factor. Thus, **H1** is identical to **I** (always samples functions), while with **H0** we sample only the first invocation of any function. At the mid-point **H.5** the probability of sampling a function invocation decays from 1 by 0.5 each time the function is sampled; for long running programs the sampling probability converges to 0. The letter **A** in the configuration name denotes applying the aliasing heuristic to that particular sampling method. Table 8.3 summarizes the labeling of sampling strategies.

Comparison of Sampling Techniques

We illustrate the differences between the different tool configurations using the CG benchmark. These trends are representative for the whole suite of benchmarks we examined. For reference, the original UPC-Thrille tool adds 8% runtime overhead when instrumenting only communication calls (labeled as **NL** in the graphs for No-Local). Our implementation finds one new race in the implementation of this benchmark when compared to the original UPC-Thrille.

Figure 8.3 presents the tool performance when applied to the CG benchmark classes A and D running on 16 and 2048 cores respectively. The benchmark implements an iterative method and Class A solves a problem with a small memory footprint (MBs) in few iterations, while class D solves a large (GBs) problem. Other previous shared memory data race detectors [84, 61, 81, 32] have been scaled at most up to 16 cores and on applications using small data sets. LiteRace is validated on a four core system, while the tool presented by Raman et al [81] has been scaled up to 16 cores.

Instruction level sampling **I** of all memory references adds a 3600% overhead to the CG benchmark execution. This is obtained using the default sampling backoff factor of 0.9 to find races when instrumenting only shared memory accesses through runtime calls. The overhead can be reduced by decreasing the sampling frequency, at the expense of coverage.

Function level sampling **F.5** introduces a 2900% overhead for class A, lower than the 3600% overhead of **I**. A comparison of the overhead breakdown for **F** and **I** illustrates the fundamental differences between the two methods. **I** introduces almost all overhead (3600%) in instrumentation, with less than 3% for computation and communication overhead combined, while **F.5** adds only 112% instrumentation overhead. This large difference validates the common intuition that function level sampling amortizes the cost of deciding what references to track. On the other hand, **F.5** exhibits a large 2800% computation overhead to record and reason about the memory references that are actually tracked. The computation overhead for **I** is very small at less than 2%. This behavior is explained by the temporal distribution of tracked memory accesses during the program execution. UPC-Thrille uses a combination of lockset based and happens-before analysis that requires tracking all memory references between two `barrier` statements. Function level sampling exhibits a clustered behavior, where many memory references are tracked for a short period of time. Instruction sampling spreads the tracking of memory references more evenly over the program execution. Thus, the behavior of function sampling is determined by the scalability of the tool's internal data structures, while the behavior of instruction sampling is determined by the speed of classifying whether a memory access is to the global heap or not. We discuss the scalability of data structures below.

Hierarchical sampling **H.5** provides better performance than both function and instruction sampling and exhibits 2550% overhead. Most of this overhead is instrumentation overhead.

Adding the aliasing heuristics to any of the tool methods greatly improves performance. The overhead of instruction sampling is reduced from 3600% to 105% with **IA**. The overhead of hierarchical sampling is reduced from 2550% with **H.5** to 99% with **HA.5** and from 294% with **H0** to 17% with **HA0**. The lowest overhead of data race detection for the CG class A benchmark running on 16 cores is obtained by the **HA** approach.

Similar trends are observable when scaling the problem and running class D on 2048 cores. For this particular configuration, the **F** and **FA** methods do not finish due to memory and time constraints. **I** exhibits a 259% overhead, while all hybrid and instruction level methods with the aliasing heuristic exhibit less than 15% slowdown.

Implementation Overheads

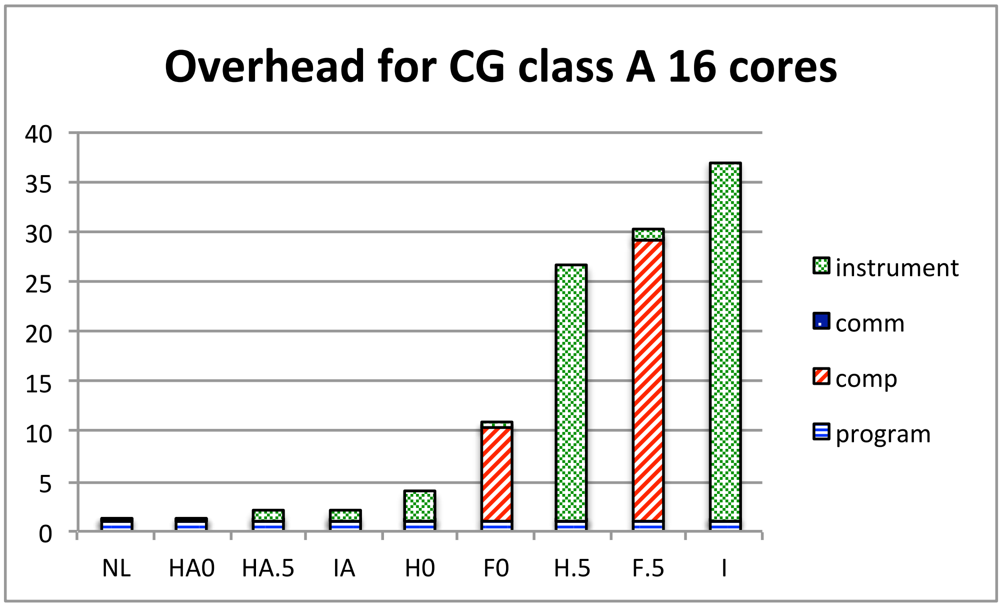
Previous work on data race detection focuses on word-level memory accesses and only require keeping track of conflicting addresses. These tools usually use hash table data structures internally. For scientific programs with bulk communication operations (PGAS or MPI), data races on full memory ranges can occur during execution. UPC-Thrille uses an efficient Interval Skiplist [38] data structure to represent memory ranges that demonstrate good performance when sampling shared memory accesses made through the runtime.

As the performance of function sampling is clearly hampered by the internal data structure overhead, we evaluate the scalability using micro-benchmarks for the insertion and search operations. The time complexity of these algorithms is dependent on the number of elements in the data structure and the distribution of the intervals. We evaluate performance across a range of list sizes and interval distributions: sequential, reverse sequential, strided and uniform random. Sequential streams are often encountered in code that performs data structure initialization, and are present in all of our benchmarks. Strided accesses occur in the Fast Fourier Transform code NAS FT, while random accesses of the form `a[b[i]]` appear in sparse methods of NAS CG and sorting in NAS IS, as well as *guppie*. For a real-world perspective, we also measure the average number of memory intervals that are recorded in our benchmarks.

Figure 8.4 presents the measured performance on one core of the Cray XE6 system. For a uniform random distribution of 20,000 ranges, the average insert time is 12 μs and the average search time is 1.3 μs. For a more regular distribution of ranges such as a sequential one (e.g. [0, 10), [10, 20), [20, 30), . . .), the insertion and search times were higher at 114μs and 2.4μs, respectively. This is a weakness of the Interval Skiplist which relies on randomness of data for balancing link levels. The effect can be offset by adding some irregularity, such as inserting a mix of two different sequential streams. In the application benchmark, the memory access stream does have irregularity, and as illustrated by the results for the stream of memory access from MG: inserts are on average 45μs and searches 0.54μs.

When using instruction sampling for the application benchmarks, the Interval Skiplists never grew too large. They remained at under 1000 unique ranges, thus the insert and search times of the Interval Skiplist do not contribute largely to the overhead. On the other hand, when using function sampling the data structures grew above 10^6 entries, at which point we stopped the execution due to the very large overheads already accumulated.

Instruction sampling pays a higher cost for classifying a memory reference but it naturally throttles the number of references recorded at any time. Function sampling performs a fast classification while having to record a large number of references. Reference classification has a constant overhead independent of the number of references already recorded, while recording overhead scales with the number of references. This difference explains why function sampling scales worse than instruction sampling for scientific programs. For reference,



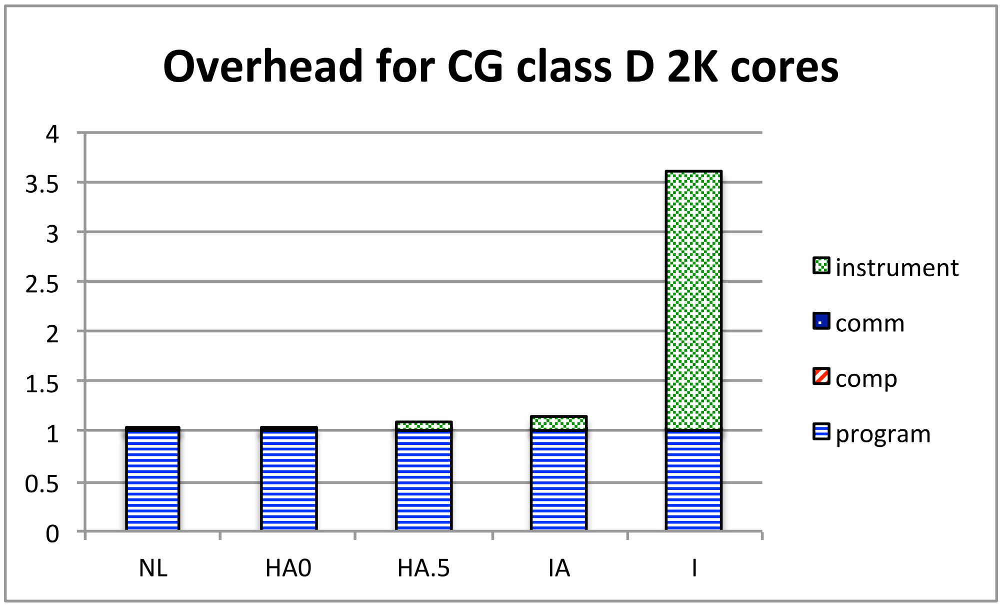
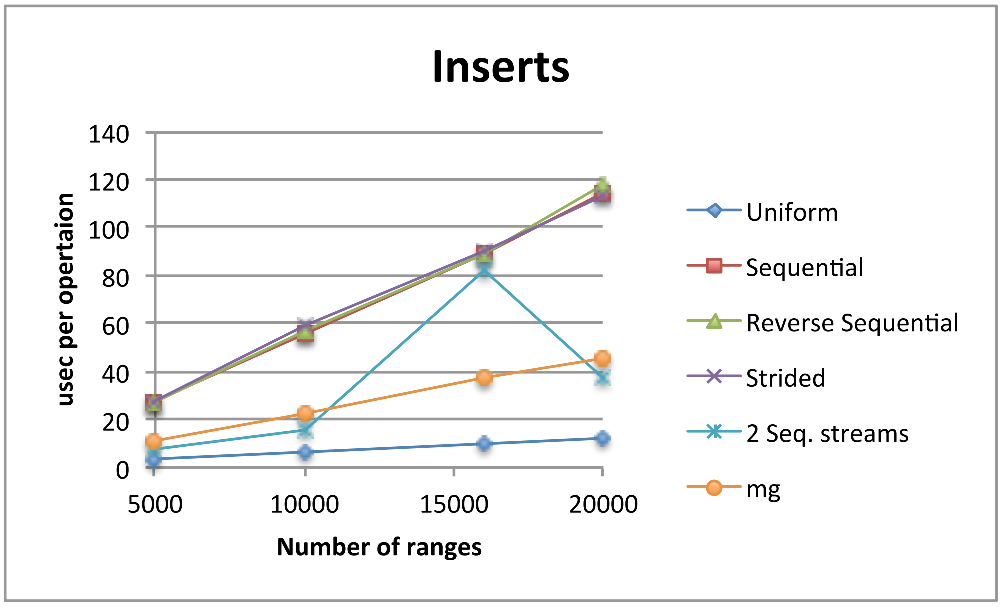


Figure 8.3: Breakdown of data race detection overhead for the CG class A benchmark running on 16 cores and class D running on 2048 cores. The **F** and **FA** configurations did not finish for the class D experiment. At the mid-point **HA.5** the probability of sampling a function invocation decays from 1 to 0, by 0.5 each time a function invocation is instrumented.



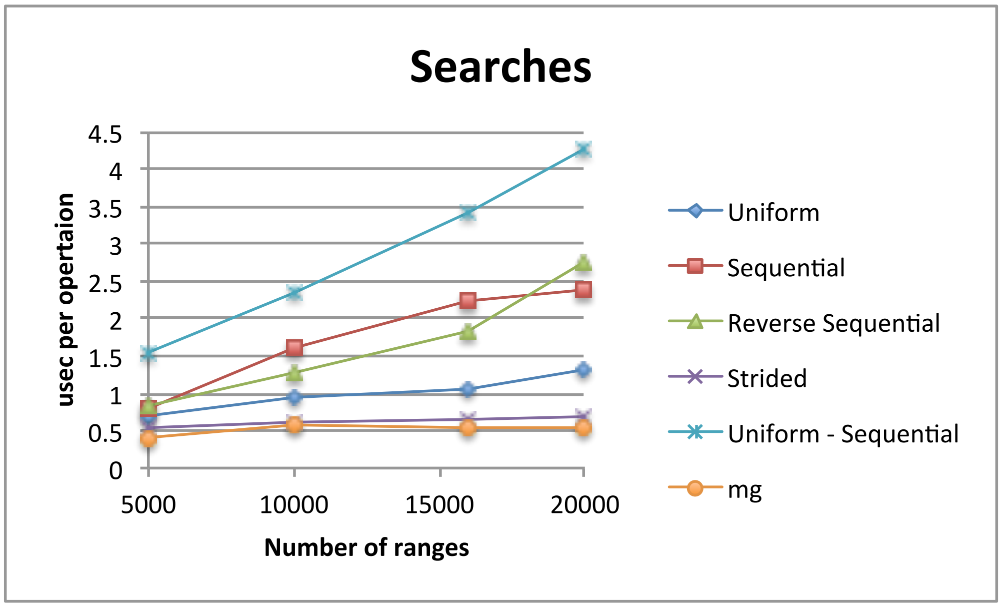


Figure 8.4: Average time for the insert and search operations in Interval Skiplist.

when running on the Cray XE6, the average instrumentation overhead per reference is 1ns, the average memory classification is 45ns, the average computation overhead per reference is 500ns while the average communication overhead per reference is 60μs.

Scalability Aspects of Data Race Detection

The trends discussed for the CG benchmark are illustrative of the behavior of data race detection for all the other applications in our workload.

Function sampling (**F** or **FA**) is faster than instruction sampling (**I** or **IA**, respectively) for problems using small datasets, such as class A of the NAS Parallel Benchmarks. When increasing the data set size to B, C and D, function sampling in any flavor does not terminate, while the highest overhead observed for instruction sampling is 6500%. From all benchmarks considered, the only exception happens for *psearch* and EP where **F** is roughly twice as fast than **I**. *psearch* is a tree search benchmark which performs a constant and small amount of work per function, independent of the problem size: this is a common characteristic to many commercial applications. EP is an "Embarrassingly Parallel" benchmark where no global memory accesses are made and thus none need to be tracked. The performance reversal observed for most benchmarks contradicts the common intuition that function sampling performs better than instruction sampling.

Hierarchical sampling **H** performs better than both instruction sampling **I** and function sampling. While it does reduce overhead, we observe slowdowns as high as 2000% which is still unacceptable when running at scale.

Applying the aliasing heuristic reduces the overhead of data race detection for both instruction level and hierarchical sampling. The maximum slowdown observed by **IA** is 1000% while the maximum slowdown for **I** is 6500%. Similar results are observed for **HA** when compared to **H**.

Figure 8.5 shows the performance of our approach when performing strong scaling experiments for the classes C and D of the NAS Parallel Benchmarks. For all experiments, the lowest overhead is introduced by the **HA** configuration and we are able to find all the races with less than 50% runtime overhead when running up to 2048 cores. In the case of the NAS Parallel Benchmarks class C on 16 cores, the weighted average overhead for all the benchmarks with **HA.5** was 11.9%. Overall, instrumentation overhead contributes the most to the slowdown caused by data race detection. The computation overhead in the scalable versions of **IA** and **HA** is small. At large scale, the communication overhead is also small due to the techniques in Section 7.2.

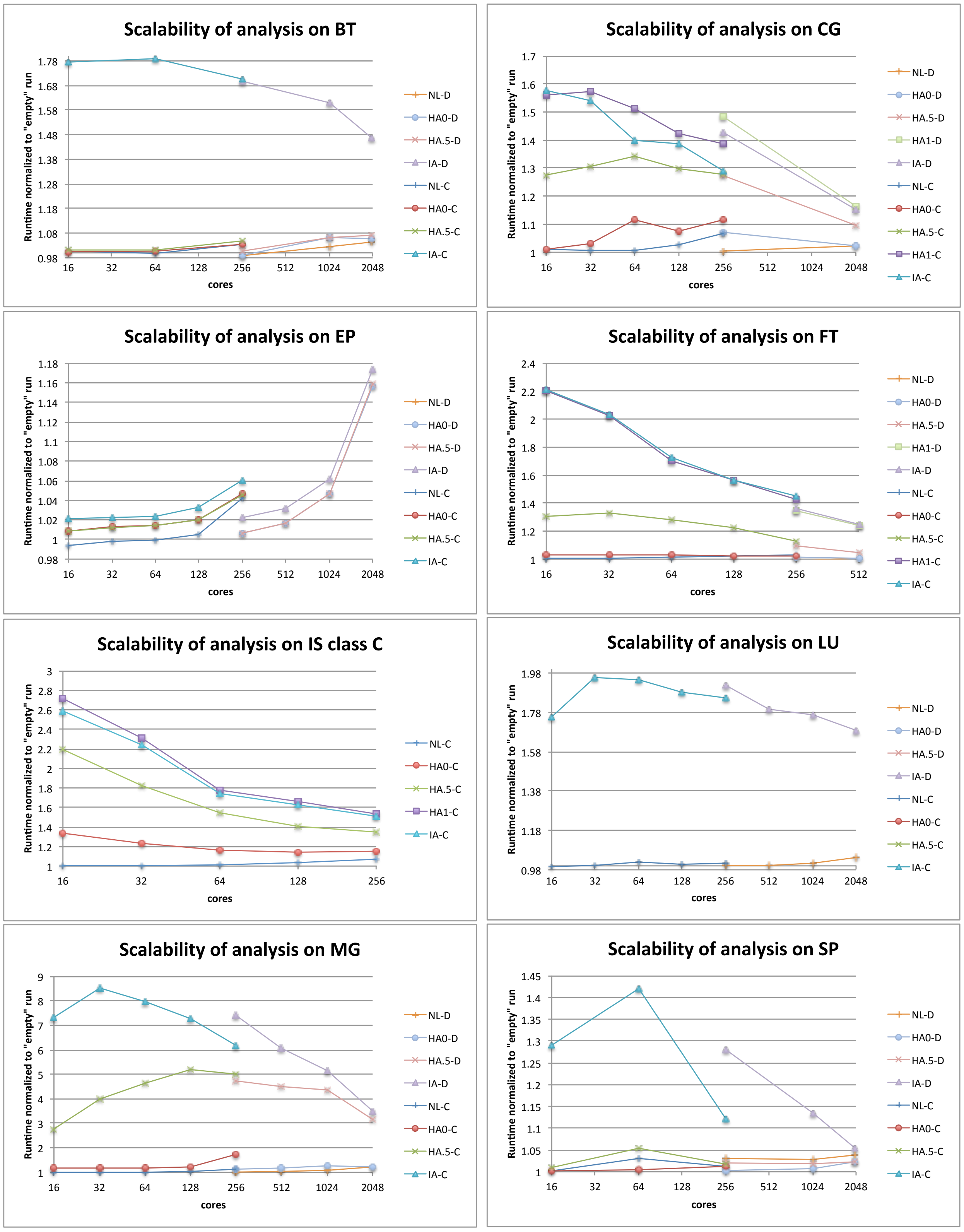


Figure 8.5: Scalability of the sampling methods on NPB classes C and D. The overhead of instruction sampling **I** is very high compared to the others and has been omitted.

Chapter 9

Related Work

We provide additional references to related work in the field of concurrency bug finding for multi-threaded shared memory programs and other programming models.

9.1 Concurrency bug analyses for multi-threaded programs

Dynamic techniques for finding concurrency bugs can be classified into two classes: predictive techniques and precise techniques. Predictive dynamic techniques [84, 100, 18, 71, 2, 40, 9, 49, 101, 102, 31, 29, 30] could predict concurrency bugs that did not happen in a concurrent execution; however, such techniques can report many false warnings. Phase I of our Active Testing uses predictive techniques. Precise dynamic techniques, such as happens-before race detection [85, 23, 1, 19, 62, 82, 20, 67, 32] and atomicity monitoring [105, 59, 28, 33], are capable of detecting concurrency bugs that actually happen in an execution. Therefore, these techniques are precise, but they cannot give as good coverage as predictive dynamic techniques.

Dynamic techniques need to address the challenge of high runtime overhead. Sampling approaches to reduce instrumentation overhead have been discussed throughout this paper. Techniques to reduce the computation overhead have been explored as well. Choi et al [18] discuss static analysis techniques to reduce the overhead of data race detection for Java programs. As alias and pointer analysis for C based programs is notoriously conservative, these techniques need to be supplemented by the runtime techniques presented in Section 7.3. Recently, Raman et al [81] describe a scalable implementation for data race detection in Habanero Java programs implemented using fine-grained structured parallelism. Their benchmarks are equivalent to our fine-grained benchmarks, while our NAS benchmarks use coarse grained interactions. They report analysis overheads as high as 10$\times$ and provide valuable data about the scalability of other state of the art race detectors for multi-threaded programs: Eraser [84] and FastTrack [32]. They report slowdowns as high as 100$\times$ for the latter.

Static verification [25, 42, 79, 15] and model checking [26, 44, 36, 41, 98, 64] or path-sensitive search of the state space are alternative approaches to finding bugs in concurrent programs. Model checkers, being exhaustive in nature, can often find all concurrency related bugs in concurrent programs. Unfortunately, model checking does not scale with program size. Several other systematic and exhaustive techniques [13, 17, 91, 88] for testing concurrent and parallel programs have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points. To detect generalized (resource and communication) deadlocks [48], a model of a program based on synchronization is extracted and run through a model checker. CHESS [64, 65] tames the scalability problem of model checking by bounding the number of preempting context switches to small numbers. However, CHESS is not directed towards finding common concurrency bugs quickly—it is geared towards systematic search and better coverage.

Randomized algorithms for model checking have also been proposed. For example Monte Carlo Model Checking [37] uses a random walk on the state space to give a probabilistic guarantee of the validity of properties expressed in linear temporal logic. Randomized depth-first search [24] and its parallel extensions have been developed to dramatically improve the cost-effectiveness of state-space search techniques using parallelism. A randomized partial order sampling algorithm [86] helps to sample partial orders (i.e. non-equivalent executions) almost uniformly at random. Race directed random testing [87], the precursor to Active Testing, uses an existing dynamic analysis tool to identify a set of pairs of statements that could potentially race in a multi-threaded execution. Each such pair is then used to bias a random scheduler so that the statements in the pair can be executed temporally next to each other.

A few techniques have been proposed to confirm potential bugs in concurrent programs using random testing. Havelund et al. [8] uses a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. However, they assume that the thread and object identifiers do not change across executions. Similarly, ConTest [70] uses the idea of introducing noise to increase the probability of the occurrence of a deadlock. It records potential deadlocks using a Goodlock algorithm. To check whether a potential deadlock can actually occur, it introduces noise (using `yield`, `sleep`, `wait` (with timeout)) during program execution to increase the probability of exhibition of the deadlock. Our work differs from ConTest in the following ways. ConTest uses only locations in the program to identify locks. We use context information and object abstractions to identify the run-time threads and locks involved in the deadlocks; therefore, our abstractions give more precise information about run-time objects. Moreover, these techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. As such, they cannot pause a thread as long as required to reproduce real bugs. We explicitly control the thread scheduler to create the potential deadlocks, instead of adding timing noise to program execution.

Shacham et al. [89] have combined model checking with lockset based algorithms to prove the existence of real races. CTrigger [77] uses trace analysis, instead of trying out all possible schedules, to systematically identify (likely) feasible unserializable interleavings for the purpose of finding atomicity violations. SideTrack [106] improves monitoring for atomicity violations by generalizing an observed trace.

A couple of techniques have been proposed to prevent deadlocks from happening during program execution, and to recover from deadlocks during execution. When a buggy program executes and deadlocks, Dimmunix [50] records the deadlock pattern. During program execution, it tries to prevent the occurrence of any of the deadlock patterns that it has previously observed. Rx [80] proposes to recover programs from software failures, including deadlocks, by rolling them back to a recent checkpoint, and re-executing the programs in a modified environment.

9.2 Concurrency bug analyses for other programming models

So far there have been a lot of research effort to verify and test concurrent and parallel programs written in Java and C/pthreads for non-HPC platforms; the huge body of literature listed above supports this fact. There have also been effort to test and verify HPC programs, mostly focused on C/MPI programs.

ISP [95] is a push-button dynamic verifier capable of detecting deadlocks, resource leaks, and assertion violations in C/MPI programs. DAMPI [99] overcomes ISP's scalability limitations and scales to thousands of MPI processes. Like ISP, DAMPI only tests for MPI Send/Recv interleavings, but runs in a distributed way. In contrast to our work, DAMPI instruments and reasons only about the ordering of Send/Recv operations with respect to the MPI ranks, and not about the memory accessed by these operations. Both ISP and DAMPI assume that program input is fixed. TASS [92] removes this limitation by using symbolic execution to reason about all possible inputs to a MPI program, but it is work only at inception.

MPI messages can be intercepted and analyzed for bugs and anomalies. Intel MessageChecker [22] does a post-mortem analysis after collecting message traces, while MARMOT [52] and Umpire [97] check at runtime. Our proposed Active Testing technique targets finding memory bugs in HPC programs and has to extend the previous approaches with techniques to reason about local memory accesses in conjunction with communication operations.

Some of the existing work for Single Program Multiple Data (SPMD) programs uses static analysis, e.g. barrier matching [4, 107] or single value analysis [43]. Static analysis requires extensive compiler support, often lacks whole program information, and reports a large number of false positives. Debugger based approaches [60] also face challenges finding concurrency bugs due to their non-determinism.

Recently, GPGPU (General Purpose Graphical Processing Unit) programming is increasingly used for data parallel and scientific applications. GRace [108] is data race detector for

GPU programs using a two-phase approach, like Active Testing. First, it uses static analysis to reduce instrumentation on statements with addresses that are pre-determined at compile-time to be race-free. Then, a dynamic checker logs and analyzes shared memory accesses at run-time. GRace takes advantage of the more structured thread scheduling and Single Instruction Multiple Data (SIMD) execution model of GPUs to remove false positives. GKLEE [55] is a concolic verifier and test generator for GPU programs. By running GPU programs on a symbolic virtual machine (VM) and exploring different branches, it can detect data races, deadlocks, and other performance issues. GKLEE also generates concrete tests to confirm the bugs on real GPU hardware. Active Testing for distributed memory parallel programs works for the more general Single *Program* Multiple Data (SPMD) execution model. We could specialize Active Testing for GPU programs, modeling their execution behavior and synchronization primitives and using the static analysis techniques of GRace to reduce instrumentation overhead. Static analysis could also help UPC-Thrille reduce instrumentation overhead by pruning unnecessary instrumentation at provably race-free statements.

Chapter 10

Conclusion

10.1 Summary

Parallel programs are becoming more important as we see the increase of parallelism in hardware. Writing correct parallel programs is hard because of the additional reasoning about interference among threads. To help programmers, we have developed a methodology called Active Testing and show how to specialize it for shared memory and distributed memory parallel programs.

Active Testing is a combination of predictive dynamic analysis and testing. With efficient algorithms targeting specific concurrency bugs, we predict potential bugs with low overhead. One of the biggest disadvantages of predictive analyses is the reporting of false positives, requiring the programmer to go over reports and manually check if they are real bugs or not. In Active Testing, we have a confirmation phase, that take the predicted bugs and try to reproduce them with high probability by controlling a random scheduler towards the state required for the bug. Thus Active Testing is complete and only reports real bugs that are reproduced by the confirmation phase, removing the burden of the programmer for checking whether bug reports are real or not.

Through experiments, we have shown that Active Testing can effectively find concurrency bugs. We have found and confirmed previously known and unknown deadlocks and atomicity violations in Java programs and libraries. The runtime overhead of previous tools for confirming bugs can go up to 100×; we tame it by using precise abstractions to reduce thrashing and have seen up to 20× reduction of overhead in our experiment with reproducing deadlocks.

We have also specialized Active Testing to handle scientific applications on large scale distributed memory systems. We use several techniques to reduce the communication among threads for scalability. We structure the framework for a distributed analysis and remove the necessity for a central thread. Exploiting the structure of HPC code that use barrier synchronization, we coalesce query traffic at synchronization points. Sampling and filtering techniques are used to reduce redundant information. Our experiments show that with these techniques, Active Testing scales well to thousands of cores.

Distributed memory parallel programs often use local aliases to address global memory within a node. For these hybrid memory programs, breaking down the analysis overhead is helpful for making analyses efficient and scalable. We introduce a hybrid sampling technique and a persistent alias locality heuristic to bring down the overhead for data race detection on hybrid memory models under 50% even at 2048 cores.

To summarize, our Active Testing methodology is a precise, efficient, and scalable technique to automatically find concurrency bugs. We have used it on multi-threaded and distributed memory platforms to quickly uncover known and unknown concurrency bugs. The methodology is general enough so that it can be extended to find other types of concurrency bugs and implemented for other programming models and languages.

10.2 Discussion

We discuss future work to improve our Active Testing implementations in the areas of 1) improving performance, 2) increasing precision, and 3) additional functionality. We can increase the performance of both phase I and phase II with improved sampling and testing strategies. Handling additional synchronization primitives will increase precision of our prediction algorithms. Handling additional concurrency bugs requires writing more analyses in our framework.

We can speed up the confirmation phase of Active Testing by testing for multiple bugs in a single execution instead of re-executing the program multiple times to confirm each bug individually. This is very useful at large scale as multiple executions of a program may be too time consuming. Iterative programs may be partitioned by barrier phases and tested for distinct bugs at each partition. We could also partition an execution by groups of threads and attempt to reproduce different bugs at each group.

To increase the precision of data race detection for UPC, we can extend the analysis to handle more synchronization and data transfer operations such as strict accesses and non-blocking collectives. Currently, these operations may show up as confirmed races, even if they were intentionally used for custom synchronization. Other memory accesses that depend on these operations may show up as potential bugs that need to be confirmed in phase II, although the bugs are impossible in any execution. To handle these additional operations require us to extend our synchronization abstractions and algorithms.

We can extend our suite of analyses to find other concurrency bugs for UPC. Deadlock and atomicity violation detection for UPC requires a straightforward adaptation to the CADA framework. Other analyses, such as for detecting performance bugs are also possible. For example, we can infer repeated remote accesses that read or write the same value and use local caches to reduce unnecessary network traffic. A barrier strength reduction analysis could transform expensive full barriers to point-to-point synchronization when possible.

There are several interesting open questions related to hierarchical and function sampling. Our hierarchical approach considers two granularities: function and instruction. In order to achieve lower overhead or to improve program coverage, one can imagine decreasing the overhead from function level to some intermediate program block level. Because of the presence of deep loop nests in scientific programs we believe that sampling at these two granularities is sufficient. A theoretical question remains whether function sampling can be made more scalable. Scalability can be improved by two approaches: 1) using data structures with better scalability characteristics than Interval Skiplists; and 2) using better formalisms to reduce the number of memory references that the analysis has to track. We leave exploring these questions as future work.

An important topic for practical tools is techniques for bounding overheads and testing time to make it a usable tool in real-world development and testing environments. We could extend the data race detection implementation to provide maximum coverage on a time budget: our goal is to find the maximum number of data races with no more than a guaranteed application slowdown. Our preliminary experiences indicate that we are likely to be able to guarantee no more than 2X slowdown. To improve coverage, we plan to use and augment the concept of region stacks introduced in the formalism presented in Section 7.3. We would like to experiment with several other strategies besides exponential backoff at instruction and function level: 1) proportional sampling per unique region stack; 2) k-region context sampling (similar to k-CFA [90] and k-object-sensitive [63] analyses); and 3) proportional sampling at functions and exponential backoff at statements.

Communication Avoiding Dynamic Analysis, a standardized framework for writing analyses for distributed systems, can be beneficial in many ways. By structuring an analysis into three distinct modes, it forces the analysis writer to rethink it to be suitable for large scale distributed systems. It also provides a common library of frequently used functions that can be reused in different analyses. Another benefit of the common API is that multiple analyses could be composed to run together on an application. The total time to run all n analyses should be less than the time to run the application n times with each analysis running separately. We need to make sure that multiple analyses do not increase the memory pressure of the instrumented program in an already memory constrained environment.

Another interesting research direction is providing a domain specific language (DSL) for CADA. By describing the three modes of an analysis declaratively in a DSL with analysis primitives, it could reduce the effort of writing new scalable and efficient analyses. Although with additional language constraints, it might be difficult to write more general analyses.

Bibliography

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *18th International Symposium on Computer architecture (ISCA)*, pages 234–243. ACM, 1991.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *20th International Conference on Automated software engineering (ASE)*, pages 233–242. ACM, 2005.

[3] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Parallel and Distributed Systems: Testing and Debugging*, 2005.

[4] A. Aiken and D. Gay. Barrier inference. In *Principles of programming languages (POPL)*, pages 342–354, New York, NY, USA, 1998. ACM.

[5] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2001.

[6] M. Arnold, M. T. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, 21(1), 2011.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD'06*, pages 41–50, 2006.

[9] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2005.

[10] H.-J. Boehm and S. V. Adve. You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2):48–54, Feb. 2012.

[11] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet. Technical Report LBNL-56495, Lawrence Berkeley National Lab, October 2004.

[12] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[13] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[14] Berkeley UPC. Available at `http://upc.lbl.gov`.

[15] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[16] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. W. E. Brooks. Introduction to UPC and language specification, 1999.

[17] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.

[18] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming language design and implementation (PLDI)*, pages 258–269, New York, NY, USA, 2002. ACM.

[19] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[20] M. Christiaens and K. D. Bosschere. TRaDe, a topological approach to on-the-fly race detection in java programs. In *JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15. USENIX, 2001.

[21] T. Cormen, C. Leiserson, and R. Rivset. *Introduction to Algorithms*. The MIT Press, 1994.

[22] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Software engineering for high performance computing system applications*, SE-HPCS '05, pages 78–82, New York, NY, USA, 2005. ACM.

[23] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Workshop on Parallel and Distributed Debugging*, 1991.

[24] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE)*, pages 3–12. IEEE, 2007.

[25] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[26] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[27] A. Farzan and P. Madhusudan. Causal atomicity. In *Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 315–328. Springer, 2006.

[28] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, pages 52–65. Springer, 2008.

[29] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[30] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Computer Aided Verification (CAV)*, pages 248–262. Springer-Verlag, 2009.

[31] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[32] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Programming language design and implementation (PLDI)*. ACM, 2009.

[33] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Programming language design and implementation (PLDI)*, pages 293–303. ACM, 2008.

[34] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, 2002.

[35] Franklin: Cray XT4 (decommissioned in 2012). At `http://www.nersc.gov/users/computational-systems/retired-systems/franklin/`.

[36] P. Godefroid. Model checking for programming languages using Verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[37] R. Grosu and S. A. Smolka. Monte carlo model checking. In *11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271–286, 2005.

[38] E. N. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1992.

[39] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, 2004.

[40] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 245–264, 2000.

[41] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[42] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[43] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Technical report, Berkeley, CA, USA, 2001.

[44] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[45] Hopper: Cray XE6. At `http://www.nersc.gov/users/computational-systems/hopper/`.

[46] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, 2003.

[47] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, 2009.

[48] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 327–336, New York, NY, USA, 2010. ACM.

[49] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Programming Language Design and Implementation (PLDI)*, 2009.

[50] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[51] A. Kamil and K. Yelick. Hierarchical pointer analysis for distributed programs. In *The 14th International Static Analysis Symposium (SAS 2007, Kong ens Lyngby*, 2007.

[52] B. Krammer, M. Müller, and M. Resch. Runtime checking of MPI applications with MARMOT. In *Mini-Symposium Tools Support for Parallel Programming, ParCo 2005, Malaga, Spain, September 12 - 16, 2005.*, 2005.

[53] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[54] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[55] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224, New York, NY, USA, 2012. ACM.

[56] B. Liblit and A. Aiken. Type systems for distributed data structures. In *In the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Progra mming Languages (POPL)*, pages 199–213, 2000.

[57] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *In International Static Analysis Symposium*. SpringerVerlag, 2001.

[58] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[59] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *SIGARCH Comput. Archit. News*, 34(5):37–48, 2006.

[60] S. S. Lumetta and D. E. Culler. The mantis parallel debugger. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '96, pages 118–126, New York, NY, USA, 1996. ACM.

[61] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[62] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, pages 24–33. ACM, 1991.

[63] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[64] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation (PLDI)*, 2007.

[65] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation (OSDI)*, pages 267–280, 2008.

[66] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *IEEE 31st International Conference on Software Engineering (ICSE)*, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[67] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[68] The NAS Parallel Benchmarks. Available at `http://www.nas.nasa.gov/publications/npb.html`.

[69] The UPC NAS Parallel Benchmarks. Available at `http://upc.gwu.edu/download.html`.

[70] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *8th Workshop on Runtime Verification*, 2008.

[71] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[72] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and practice of parallel programming (PPoPP)*, pages 167–178. ACM, 2003.

[73] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *International Conference on Parallel Processing (ICPP)*, ICPP '08, 2008.

[74] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Foundations of Software Engineering (FSE)*. ACM, 2008.

[75] C.-S. Park and K. Sen. Concurrent breakpoints. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 331–332, New York, NY, USA, 2012. ACM.

[76] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of the Supercomputing Conference (SC11)*, 2011.

[77] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural support for programming languages and operating systems (ASPLOS)*, pages 25–36. ACM, 2009.

[78] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[79] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Programming language design and implementation (PLDI)*, pages 14–24. ACM, 2004.

[80] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248. ACM, 2005.

[81] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, PLDI '12, 2012.

[82] M. Ronsse and K. D. Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[83] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel Thread Checker race detector. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, 2006.

[84] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[85] E. Schonberg. On-the-fly detection of access anomalies. In *Programming Language Design and Implementation (PLDI)*, volume 24, pages 285–297, 1989.

[86] K. Sen. Effective random testing of concurrent programs. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007.

[87] K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation (PLDI)*, 2008.

[88] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)*, Lecture Notes in Computer Science. Springer, 2006.

[89] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. *J. Parallel Distrib. Comput.*, 67(5):536–550, 2007.

[90] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, 1991.

[91] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *International symposium on Software testing and analysis (ISSTA)*, pages 157–168. ACM Press, 2006.

[92] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *Principles and practice of parallel programming*, PPoPP '11, pages 309–310, New York, NY, USA, 2011. ACM.

[93] C. Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. In *PARCO'07*, pages 669–676, 2007.

[94] Time of check to time of use. `http://en.wikipedia.org/wiki/TOCTTOU` Retrieved on 10/10/2012.

[95] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking MPI programs. In *Principles and practice of parallel programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[96] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[97] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[98] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*. IEEE, 2000.

[99] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Supercomputing*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[100] C. von Praun and T. R. Gross. Object race detection. In *Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[101] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *3rd Workshop on Run-time Verification*, volume 89 of *ENTCS*, 2003.

[102] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM Press, Mar. 2006.

[103] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629, 2005.

[104] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 238–248, 2008.

[105] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[106] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *7th Workshop on Parallel and Distributed Systems*, pages 1–10. ACM, 2009.

[107] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Principles and practice of parallel programming*, PPoPP '07, 2007.

[108] M. Zheng, V. T. Ravi, F. Qin, and G. 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*, PPoPP '11, pages 135–146, New York, NY, USA, 2011. ACM.