Sound and Precise Analysis of Multithreaded Programs through Schedule Specialization and Execution Filters

Jingyue Wu

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2014

ABSTRACT

Sound and Precise Analysis of Multithreaded Programs through Schedule Specialization and Execution Filters

Jingyue Wu

Multithreaded programs are known to be difficult to analyze. A key reason is that they typically have an enormous number of execution interleavings, or *schedules*. Static analysis with respect to all schedules requires over-approximation, resulting in poor precision; dynamic analysis rarely covers more than a tiny fraction of all schedules, so its result may not hold for schedules not covered.

To address this challenge, we propose a novel approach called *schedule specialization* that restricts the schedules of a program to make it easier to analyze. Schedule specialization combines static and dynamic analysis. It first statically analyzes a multithreaded program with respect to a small set of schedules for precision, and then enforces these schedules at runtime for soundness of the static analysis results.

To demonstrate that this approach works, we build three systems. The first system is a specialization framework that *specializes* a program into a simpler program based on a schedule for precision. It allows stock analyses to automatically gain precision with only little modification.

The second system is PEREGRINE, a deterministic multithreading system that collects and enforces schedules on future inputs. PEREGRINE reuses a small set of schedules on many inputs, ensuring our static analysis results to be sound for a wide range of inputs. It also enforces these schedules efficiently, making schedule specialization suitable for production usage.

Although schedule specialization can make static concurrency error detection more precise, some concurrency errors such as races may still slip detection and enter production systems. To mitigate this limitation, we build LOOM, a *live-workaround* system that protects a live multithreaded program from races that slip detection. It allows developers to easily write *execution filters* to safely and efficiently work around deployed races in live multithreaded programs without restarting them.

Table of Contents

List of Figures

List of Tables

Acknowledgments

The work presented in this thesis would not have been possible without the help and support of many people.

In particular, I would like to express my greatest gratitude to my advisor, Professor Junfeng Yang, for his advice and guidance over the past five years. I would not have been able to complete this work without his support, encouragement and inspiration.

I would like to thank Professors Al Aho, Stephen Edwards, Roxana Geambasu, Martha Kim, and Simha Sethumadhavan for their valuable suggestions and useful discussion on projects I worked on and presentations I gave at conferences.

I own special thanks to Professors Stephen Edwards and Angelos Keromytis, who served on my thesis proposal committee. They provided many very constructive suggestions, which have significantly improved the thesis.

Lastly, I want to thank my colleague students Heming Cui, Gang Hu, and Yang Tang, who helped me solve lots of technical issues and read (reread) many of my papers. Without them, I would have missed far more paper deadlines than I did.

Chapter 1

Introduction

1.1 Problem

Multithreaded programs are increasingly pervasive and critical because of two technology trends. The first is the rise of multicore hardware. The speed of a single processor core is limited by fundamental physical constraints, forcing processors into multicore designs. Thus, developers must resort to parallel code for best performance on multicore processors. The second is our accelerating computational demand. Scientific computing, video and image processing, financial simulation, "big data" analytics, web search, and online social networking are all massive computations and employ various kinds of parallel programs for performance.

However, multithreaded programs are often plagued with concurrency errors [Lu *et al.*, 2008], such as races and deadlocks. Some of the worst of these bugs have killed people in the Therac 25 incidents [Leveson and Turner, 1993] and caused the 2003 Northeast blackout [Poulsen, 2004]. One study also reveals that these bugs may be exploited by attackers to violate confidentiality, integrity, and availability of critical systems [Yang *et al.*, 2011].

A key reason for these bugs is that multithreaded programs are difficult to analyze using automated tools. These programs typically have an enormous—asymptotically exponential in the total execution lengths—number of execution interleavings, or *schedules*. Neither static nor dynamic analysis can effectively analyze these many schedules. Static analysis with respect to all these schedules requires over-approximations, resulting in poor precision. For instance, RELAY [Voung *et al.*, 2007], a state-of-the-art static race detector for C programs, emits 5,022 warnings over

Linux kernel 2.6.15. Only roughly 100 of these warnings correspond to real data races. These false positives bury the true errors, and consequently make the tool less useful. Dynamic analysis can precisely analyze the schedules observed, but it rarely covers more than a tiny fraction of all possible schedules, and the next execution may well run into an unchecked schedule with errors.

1.2 Hypothesis and Approach

Our key insight is that, although a multithreaded program can have exponentially many schedules, not all these schedules are necessary for good performance. A small set often suffices, as illustrated by recent work on efficient deterministic multithreading [Olszewski *et al.*, 2009; Liu *et al.*, 2011; Aviram *et al.*, 2010; Bergan *et al.*, 2010a; Berger *et al.*, 2009; Bocchino *et al.*, 2009] and stable multithreading [Cui *et al.*, 2010; Cui *et al.*, 2011].

Based on this insight, we propose the following hypothesis: we can dramatically improve the precision of static analysis while maintaining its soundness by considering only a small set of schedules in static analysis and enforcing the program to follow the analyzed schedules at runtime. We call this approach *schedule specialization*.

Specifically, the idea of schedule specialization is two-fold. First, it statically analyzes a program with respect to a small set of schedules for precision. By focusing on a small set of schedules, we can enumerate these schedules, and analyze a program with respect to each of them. Being aware of a schedule can dramatically improve the precision of static analysis, because a schedule can usually preclude a majority of the executions. Second, schedule specialization restricts the schedules of the program for soundness of the static analysis results. Because schedule specialization analyzes only a subset of all schedules, its analysis results may not hold for program executions that deviate from these schedules. To ensure these analysis results are sound, schedule specialization enforces the program to follow the analyzed schedules at runtime.

1.3 Challenges

The idea of schedule specialization sounds appealing. However, building a real system that implements this idea faces three main challenges.

1.3.1 Collecting and Enforcing Schedules

To effectively maintain soundness, schedule specialization needs to collect highly reusable schedules, and deterministically and efficiently enforce these schedules at runtime. These requirements make both collecting and enforcing schedules challenging.

1.3.1.1 Collecting Schedules

Schedule specialization must collect highly reusable schedules for two reasons. First, the set of collected schedules needs to be small. Schedule specialization enumerates the collected schedules, and analyzes the program with respect to each one of them. Each run of the analysis takes time. To amortize the cost of the static analysis, we want to analyze as few schedules as possible. Second, the collected schedules must cover a wide range of inputs. The static analysis results produced by schedule specialization hold only when the program follows the analyzed schedules. To maximize the soundness of the analysis results, we want the analyzed schedules to cover as many inputs as possible. A finite set of schedules may not cover all inputs. To retain soundness on such inputs on uncovered inputs, we may resort to dynamic analysis (§2.2.3.1).

However, collecting highly reusable schedules is challenging. Statically computing these schedules is extremely difficult due to the undecidability of the halting problem. Dynamically collecting these schedules during program execution requires complicated computation and can incur large performance overhead.

1.3.1.2 Enforcing Schedules

Schedule specialization must deterministically and efficiently enforce schedules for two reasons. First, if schedule specialization cannot deterministically enforce a schedule, a program may deviate from the enforced schedule even if the input can be processed using this schedule. This deviation can mislead the program into executions where our analysis results do not hold, and consequently lose soundness. Second, since schedule specialization enforces a schedule for each program execution, the overhead in enforcing schedules directly contributes to the runtime overhead. If enforcing schedules is not efficient, the runtime overhead can easily offset the benefits of high precision and soundness.

However, enforcing schedules both deterministically and efficiently for general programs on commodity hardware is an open challenge because these programs may have data races. Existing

systems either enforce a deterministic order of shared memory accesses (*e.g.*, CoreDet [Bergan *et al.*, 2010a]), which can incur prohibitive overhead, or enforce only a deterministic order of synchronizations (*e.g.*, Kendo [Olszewski *et al.*, 2009]), which sacrifices determinism on programs containing data races.

1.3.2 Analyzing a Program with respect to a Schedule

How much we can improve the precision of static analysis depends on how effectively these analyses can leverage information provided by a schedule. If a static analysis fails to leverage a schedule, its precision will remain the same no matter how small the set of schedules is.

Unfortunately, analyzing a program with respect to a schedule is challenging because a static analysis tool usually involves many static analyses. One naïve method of leveraging a schedule is to manually make every analysis involved aware of the schedule. However, given so many static analyses, this method is quite labor-intensive and error-prone. It would also be fragile: if a crucial analysis is unaware of schedules, its imprecision may easily pollute other analyses. So, instead, one might need a more scalable approach that can automatically improve multiple static analyses.

1.3.3 Handling Concurrency Bugs that Slip Detection

One natural application of schedule specialization is detecting concurrency errors such as data races. For instance, we can statically detect data races with respect to a set of schedules. After fixing them, we claim the program is race-free as long as we enforce the analyzed schedules in production.

However, this approach still cannot detect all concurrency errors, because schedule specialization cannot eliminate all false negatives and false positives of a static analysis. Our static analysis results may not hold for inputs that cannot be covered by any analyzed schedules, causing false negatives. Unlike dynamic analysis which can virtually access all dynamic information, static analysis built atop schedule specialization only leverages schedules, and can emit more false positives than users' expectations.

Due to these limitations of schedule specialization, some concurrency bugs may slip detection of static analysis, and surface in production. These "deployed" concurrency errors can cause application crashes, data corruptions, and even security holes that may be exploited by attackers.

The traditional approach of fixing deployed concurrency errors is software update. However,

this approach requires application restarts, and is not suitable for server programs which require high availability. Live update systems [Arnold and Kaashoek, 2009; Chen *et al.*, 2006; Altekar *et al.*, 2005; Makris and Ryu, 2007; Neamtiu *et al.*, 2006; Neamtiu and Hicks, 2009; Subramanian *et al.*, 2009] can avoid restarts by adapting conventional patches into *hot patches* and applying them to live systems, but the reliance on conventional patches has two problems. First, conventional patches are hard to write, reason about and deploy into production because they are written in general programming languages such as C. Second, even if the patch is correct by construction, applying it to a live application can introduce new errors because the application may be running in an inconsistent state with the patch.

Two recent systems, Dimmunix [Jula *et al.*, 2008] and Gadara [Wang *et al.*, 2008], can fix deadlock bugs in legacy multithreaded programs without using conventional patches. Dimmunix extracts signatures from occurred deadlocks (or starvations) and dynamically avoids them in future executions. Gadara uses control theory to statically transform a program into a deadlock-free program.

However, deadlocks are only a minority of all concurrency bugs, empirically more than 70% of the concurrency errors are non-deadlock bugs [Lu *et al.*, 2008]. In this thesis, we denote non-deadlock bugs by "races", which include data races, atomicity violations and order violations. While Dimmunix and Gadara can control the orders of existing locks to avoid deadlocks, fixing races requires more fine-grained control over the program, *e.g.*, adding new locks to arbitrary program locations.

1.4 Overview of Contributions

To address the three main challenges (§1.3) facing schedule specialization, we build three systems which form the three main contributions of this thesis: the PEREGRINE system that collects and enforces schedules, a specialization framework that analyzes a program with respect to a schedule, and the LOOM system that bypasses races that slip detection. Figure 1.1 shows the architecture and workflow of schedule specialization. We first use PEREGRINE to collect a set of schedules from real program executions. We then enumerate these schedules and use the specialization framework to statically analyze the program with respect to each schedule. At runtime, we run the program

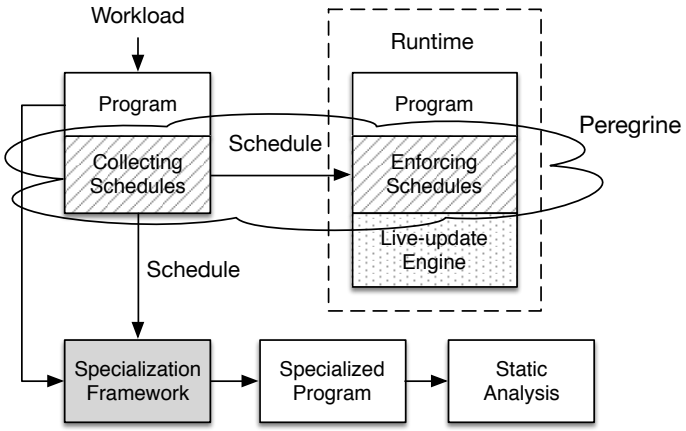


Figure 1.1: *Architecture of the schedule specialization framework.* The shaded boxes represent the PEREGRINE system we build to collect and enforce schedules; the gray box represents the specialization framework we build to analyze a program with respect to collected schedules; the dotted box represents the LOOM live-update engine we build to handle races that slip detection.

with both PEREGRINE and LOOM, PEREGRINE enforcing the analyzed schedules for soundness and LOOM handling deployed races.

Each of the following three subsections talks about a system that solves one of the challenges.

1.4.1 Peregrine

To collect and enforce schedules, we build PEREGRINE [Cui *et al.*, 2011], a deterministic multi-threading (DMT) system. To compute highly reusable schedules, PEREGRINE records schedules from past real executions, and reuses these schedules on future inputs. Recording schedules from real executions guarantees these schedules are feasible, sidestepping the halting problem. Given a recorded schedule, PEREGRINE computes what other inputs can be processed using this schedule, and reuses this schedule for them in the future.

To deterministically and efficiently enforce schedules, PEREGRINE enforces "hybrid" schedules that combine synchronization schedules and memory access schedules. The key insight of the authors of PEREGRINE is that races tend to occur only within minor portions of an execution. Therefore, we can schedule synchronizations for the race-free portions of an execution, and re-

sort to scheduling memory accesses only for the racy portions, combining both the efficiency of synchronization schedules and the determinism of memory access schedules.

1.4.2 Specialization Framework

To analyze a program with respect to a schedule, we have built a specialization framework which forms our second contribution. To avoid the need to manually modify every analysis, our specialization framework uses a set of sound and precise algorithms to *specialize* a program according to a schedule. The resultant program thus has simpler control and data flow than the original program, and can be analyzed with stock analyses for improved precision.

The specialization framework dramatically eases the process of making static analyses schedule-aware. Instead of thinking about how to leverage schedules in every static analysis, developers can simply use our specialization framework to generate a new program and then run stock analyses as is on top of the specialized program.

The framework can also dramatically improve the precision of static analysis. Our specialization algorithms specialize both the control and data flow of a program. For instance, it can detect code that is unreachable if the program follows a certain schedule, so dead-code elimination can further simplify the control flow of a program. It can also "straighten" loops according to a schedule, making more variables computable at compile time. Because the specialized program has simpler control and data flow, static analyses can automatically gain precision without being modified.

1.4.3 Loom

To handle deployed races, we build Loom, a live-update engine designed to quickly and safely bypass application races at runtime, providing a last-resort protection for multithreaded programs.

To overcome the drawbacks of conventional patches, we create a new language in which developers can write an *execution filter* that synchronizes the application source to filter out racy schedules. For instance, developers can write an execution filter that makes two code regions mutually exclusive. Using execution filters instead of conventional patches has three benefits.

- Execution filters are easier to write than conventional patches because they match developers' intents more directly. For instance, to fix an atomicity violation, developers can write an

execution filter "code region r_1 and r_2 are mutually exclusive" without worrying about low-level mutex operations.

- Execution filters are more flexible because they are temporary workarounds instead of final fixes. For instance, if a developer suspects that two code regions r_1 and r_2 contain a data race, she can temporarily apply a mutual exclusion filter before investigating the root cause and writing the final fix. LOOM also allows users to uninstall an execution filter applied to a live application. For instance, if the developer later finds making r_1 and r_2 mutually exclusive is not enough and wants to conservatively run r_1 and r_2 in single-threaded mode, she can easily revoke the old execution filter and apply a new one.

- Execution filters make safety easier to reason about. Unlike general programming languages such as C, LOOM's execution filter language allows only well-formed synchronization constraints. Therefore, LOOM need not reverse-engineer developer intents (*e.g.*what goes into a critical region) from scattered operations (*e.g.*`lock` and `unlock`).

With the help of execution filters, LOOM addresses the safety challenge using our new algorithm termed *evacuation*. Given an execution filter, this algorithm first uses static analysis to conservatively compute a set of "unsafe" program locations. A program location is unsafe if applying the execution filter when any thread is running at this location may introduce new errors. Then, the evacuation algorithm proactively evacuates threads out of unsafe locations and block them at safe program locations. After evacuating all threads, it installs the filter and then resumes these threads.

LOOM reduces the instrumentation overhead using the idea of hybrid instrumentation which combines static and dynamic instrumentation. It statically transforms an application to keep two versions: a fast original version, and a slow version which can then be updated arbitrarily by execution filters at runtime. It also inserts switches where an execution can be quickly switched between the fast and slow version. To install an execution filter at runtime, LOOM dynamically updates the slow version according to the filter, and switches the execution to the slow version to run the updated code. Realizing an execution filter usually requires very limited program change. For example, realizing a mutual exclusion filter only requires updating the entry and exit of the critical region. Therefore, the program will run in the fast original version most of the time.

1.4.4 My Roles in These Projects

All the three projects are collaborated with my colleagues. Heming Cui is the main contributor of the PEREGRINE system. My main roles in this project are:

1. building the instrumentation framework PEREGRINE uses to enforce schedules (§3.3.2);
2. building basic static analyses (*e.g.*, alias analysis) used by the slicing algorithm PEREGRINE uses to compute preconditions. Please refer to our peregrine paper [Cui *et al.*, 2011] for more details.

I am the main contributor of the specialization framework. Yang Tang was in charge of the evaluation of the specialization framework. Gang Hu speeded up the data flow specialization of the framework by parallelizing constraint solving (§4.3.2). Heming built the path slicing application (§4.4) based on the framework.

I also lead the LOOM project. My colleague Heming Cui built the LOOM controller (§5.1) that interacts with users and initiates live update sessions.

1.5 Results Summary

We evaluate the approach of schedule specialization from five aspects.

Precision. Improving the precision of static analysis is the main goal of schedule specialization. To measure precision, we built atop our specialization framework three highly precise schedule-aware analyses: alias analysis, path slicing, and race detection. We compare their precision with and without schedule specialization. Results show schedule specialization greatly improves the precision of these analyses. For example, a static race detector reports on average 69% fewer false data races (§4.5) on a schedule-specialized program than on the original program.

Soundness. Ideally, we would want our static analysis results to hold for all inputs. However, because a finite set of schedules cannot cover all inputs, we want our static analysis results to hold for as many inputs as possible. Our evaluation results show that PEREGRINE, the system we built to collect and enforce schedules, can frequently use a small number of schedules can cover a wide range of workloads. For instance, PEREGRINE can use about a hundred schedules to cover over 90% of requests in a real HTTP trace for `Apache` [Apache, 2012].

Overhead of enforcing schedules. To encourage user adoption, schedule specialization needs to have low overhead of enforcing schedules. Our evaluation results show that PEREGRINE can enforce schedules with a runtime overhead ranging from 68.7% faster to 46.6% slower than the original program on a diverse set of 18 programs, including `Apache`.

Overhead of instrumentation. Both PEREGRINE and LOOM use a hybrid instrumentation framework we built to reduce instrumentation overhead. Our evaluation results show that this framework incurs negligible instrumentation overhead and in some cases even speeds up the application. The only exception is `MySQL`, whose throughput drops by 3.76% when running on our framework.

Effectiveness of bypassing deployed races. We also evaluate how effective LOOM can bypass deployed races. The results show that LOOM can flexibly and safely fix all races we have studied; it maintains application availability when applying fixes; it can install a fix within a second even under heavy workload.

1.6 Thesis Organization

The remaining of this thesis is organized as follows. Chapter 2 formally defines schedule specialization and discusses its key properties. Chapter 3 describes how we collect and enforce schedules for soundness. Chapter 4 presents how we specialize a program with respect to a schedule for precision. Chapter 5 describes how we protect multithreaded programs against deployed races. Finally, Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Preliminaries

This chapter first defines key terms and notations we use in the thesis (§2.1), and then describes schedule specialization and its properties (§2.2).

2.1 Terms and Notations

There are many models for writing parallel programs. While we anticipate our approach will work well with many parallel programming models, this thesis focuses on multithreaded programs that use shared memory threaded programming. Specifically, we target C/C++ programs that use Pthreads, which provides APIs for the program to create threads, and manipulate threads using traditional synchronization constructs such as mutexes and semaphores. We focus on this implementation of the shared memory threaded programming model, because we believe it is the one mostly used.

Program. We implemented schedule specialization based on the LLVM compiler. We compile a program to the LLVM intermediate representation (LLVM IR) [llv, 2013], and analyze and rewrite the program in that level. LLVM IR uses a RISC-like instruction set with some key high-level information for analysis. It has three key features that ease the implementation of schedule specialization:

- **Explicit control flow graph.** LLVM makes the control flow graph (CFG) of every function explicit in the representation. A function in LLVM is a set of basic blocks. Each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction, such

as branches and return. This representation generalizes the analysis of compound C/C++ statements such as `if`, `switch`, `while` and `for`.

- **Explicit data flow representation.** LLVM represents the data flow of a program primarily using the SSA (Static Single Assignment [Cytron *et al.*, 1991]) form with an infinite virtual register set. The SSA form makes def-use chains more explicit and therefore simplifies data flow analysis. For memory locations that are not in SSA form, LLVM IR accesses them using the `load` and `store` instruction, which take a single pointer and do not perform any indexing. All pointer arithmetics (*e.g.*, computing an internal pointer of an array or a struct) are explicitly represented using the `getelementptr` instruction.

- **Type information.** LLVM IR preserves type information in the source code, such as array types and struct types, at its best effort. In contrast, some other IRs, such as the DEX IR used by Dalvik virtual machine [dal, 2013], represent composite types as byte arrays. Providing rich type information helps data flow analysis such as alias analysis. For instance, type-based alias analysis [Diwan *et al.*, 1998] can prove two pointers do not alias if their types are not compatible.

Statement. We define a *statement* as an instruction in the LLVM IR of a program. Each program has a unique *entry* statement (typically the entry of function `main` for C/C++) where the program starts executing. To identify statements, we assign each statement in a program a unique static *label* l_j.

Thread. A *thread* is a concurrent unit of execution of program statements. In the Pthread implementation, a program spawns a thread by calling `pthread_create` with a start routine. The created thread will execute the specified routine once it is created.

Dynamic statement instance. A *dynamic statement instance* $i = \langle t, l \rangle$ indicates that i is an instance of statement l executed by thread t. Given $i = \langle t, l \rangle$, we use $i.thread$ to access t, and $i.label$ to access l.

Data races. Programs that use shared memory threaded programming may have data races. The existence of data races poses a great challenge for deterministically executing multithreaded programs: even if we enforce a certain schedule, the order between two statement instances involved

in a data race can be arbitrary, causing the program to observe non-deterministic values from the shared memory location.

A *data race* is a condition where two dynamic statement instances i_1 and i_2 satisfy all the following requirements:

1. i_1 and i_2 access a common memory location;
2. at least one of the two accesses is for writing;
3. i_1 and i_2 are potentially concurrent: $i_1.thread$ and $i_2.thread$ use no explicit mechanism (*e.g.* mutexes or semaphores) to prevent i_1 and i_2 from being simultaneous.

Two dynamic statement instances in an execution are *racy* if they are involved in a data race in that execution. Two statements l_1 and l_2 are racy in an execution if an instance of l_1 and an instance of l_2 are racy in that execution. l_1 and l_2 are racy in general if l_1 and l_2 are racy in at least one execution.

Trace. A *trace* of program P, denoted by T, is a potentially infinite sequence of dynamic statement instances $i_0, i_1, \ldots$ that includes every instance executed during an execution. A trace needs to satisfy two requirements:

- **Order constraints.** If i_j was completed before the start of i_n, then $j < n$ (concurrently executed statements can be ordered either way).
- **Control-flow integrity.** Control-flow integrity [Abadi *et al.*, 2005], a program property assumed by most compilers, dictates that program execution must follow a path of the CFG of the program. For multithreaded programs, the sub-trace of each thread must follow the CFG of the thread routine executed by that thread. Note that these CFGs are inter-procedural: they include the edges from a call site to all its possible callees, and those from a function return to all possible callers. Given trace T and predicate p on a dynamic statement instance, we use $filter(T, p)$ to denote the subsequence of every dynamic instance i in T where $p(i)$ is true. The sub-trace for thread t, denoted by T_t, therefore equals $filter(T, i.thread = t)$. Control-flow integrity states that, for each pair of consecutive dynamic instances i_j^t and i_{j+1}^t in T_t, edge (i_j^t, i_{j+1}^t) is in P's CFG.

We assume *sequential consistency* for data-race-free programs, *i.e.*, the result of any execution of a data-race-free program is the same as the dynamic statement instances of all the threads were executed in some sequential order. This assumption matches the memory model of Java [Manson *et al.*, 2005] and C++ [Boehm and Adve, 2008] programs. Therefore, each execution of a data-race-free program corresponds to a trace.

Some executions of a program that contains data races may not be characterized by traces in our definition. However, because the executions PEREGRINE collects and enforces are guaranteed not to contain data races (§3.1), ignoring these executions from our consideration does not affect the soundness of our static analysis results (§2.2.3.1).

We use $run(P)$ to denote the set of all possible traces of P, including non-terminating ones. Due to the non-determinism of multithreading, a multithreaded program may have many executions on the same input. We use $run^I(P)$ to denote the set of all traces when running P on input I.

Synchronization statements. The set of synchronization statements H is a subset of all program statements that includes the following three types:

- statements that call Pthread APIs that perform synchronizations[1],
- entries and exits of thread functions (*i.e.*, functions passed to `pthread_create`) and `main` required by the implementation of our control-flow specialization algorithm (§4.2.1),
- additional function calls to resolve ambiguity when a synchronization statement of the first two types may be invoked in different calling contexts (explained in §4.2.1).

We call the first two classes *true synchronization statements* and the last class *derived synchronization statements*. Our algorithm automatically computes derived synchronization statements from true synchronization statements, and does not require user annotation.

[1]These APIs are `pthread_barrier_wait`, `pthread_cond_broadcast`, `pthread_cond_signal`, `pthread_cond_timedwait`, `pthread_cond_wait`, `pthread_create`, `pthread_exit`, `pthread_join`, `pthread_kill`, `pthread_mutex_lock`, `pthread_mutex_timedlock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`, `pthread_rwlock_rdlock`, `pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_tryrdlock`, `pthread_rwlock_trywrlock`, `pthread_rwlock_unlock`, `pthread_rwlock_wrlock`, `pthread_spin_lock`, `pthread_spin_trylock`, and `pthread_spin_unlock`.

Synchronization. A *synchronization* is a dynamic statement instance i that $i.label \in H$. If $i.label$ is a true synchronization statement, i is a *true synchronization*; otherwise, i is a *derived synchronization*. Unless otherwise specified, we use synchronizations to refer to both true and derived synchronizations.

Schedule. A *schedule*, denoted by S, is a terminating sequence of synchronizations s_0, s_1, ..., s_N. Trace T and schedule S are *compatible* if (1) T terminates and $filter(T, i.label \in H) = S$, or (2) T does not terminate and $filter(T, i.label \in H)$ is a prefix of S. Similarly, an execution and a schedule are compatible if the trace of the execution is compatible with the schedule. One schedule may be compatible with multiple traces because traces are more fine-grained than schedules. We use $run_S(P)$ to denote the set of all traces that are compatible with schedule S.

Enforcing a schedule. Some runtime systems, such as PEREGRINE, can control the environment of program execution so that P only runs a subset of $run^I(P)$ on input I. We use $run^I_{\mathscr{S}}(P)$ to denote the set of all possible traces when running P on I with the control of system $\mathscr{S}$. A system $\mathscr{S}$ *enforces* schedule S on input I if $run^I_{\mathscr{S}}(P) \subseteq run_S(P)$.

Precondition. One can hardly enforce one schedule on all possible inputs. For instance, if an input requires four threads to process, we cannot enforce a schedule of two threads on that input. Therefore, before enforcing a schedule on an input, PEREGRINE needs to check whether the input can be processed with the schedule. Ideally, PEREGRINE would compute the *weakest precondition* of an enforced schedule, denoted by $wp(P, S)$, a predicate that characterizes all inputs that P can process with schedule S enforced. In other terms, $wp(P, S)(I) \Leftrightarrow run^I(P) \cap run_S(P) \neq \emptyset$ holds for any input I. Since computing weakest preconditions is undecidable [Aho *et al.*, 1986], the precondition PEREGRINE computes, denoted by $cond(P, S)$, stays on the conservative side, *i.e.*, $cond(P, S) \Rightarrow wp(P, S)$. For simplicity, we say input I is *compatible* with schedule S if $cond(P, S)(I)$ is true. We will discuss how PEREGRINE computes preconditions (§3.2), and empirically measure how close they are to the weakest preconditions (§3.4.3).

Program analysis, soundness, and precision. A *program analysis* determines whether a certain property holds for a program. From the perspective of an error detector, a *sound* analysis reports all violations of the property of interest, but may report *false positives*, *i.e.*, violations that cannot actually occur. A *complete* analysis never reports false positives; any violation of the

property of interest reported by a complete analysis is an actual violation. However, a complete analysis may miss some actual violations. We call these missed violations *false negatives*.

We illustrate these concepts using race detection. A race detector determines whether a program is free of races. A sound race detector guarantees to detect all actual races in the program, but does not guarantee all the races detected are actual races. In other words, a sound race detector does not have false negatives, but may have false positives. A complete race detector guarantees all the races detected are actual races, but does not guarantee to find all actual races. In other words, a complete race detector does not have false positives, but may have false negatives.

We measure the *precision* of an analysis by its false positive rate. Under this definition, a static analysis is typically imprecise, because it usually over-approximates the behavior of the program and reports many false positives. A dynamic analysis is typically precise, because it examines the actual behavior of the analyzed program, and any violation detected is an actual violation.

2.2 Schedule Specialization and Its Properties

Schedule specialization restricts the behavior of the program to make it easier to analyze. Without schedule specialization, a static analysis has to consider conceptually all traces in $run(P)$ for soundness. With schedule specialization, the program runs with a set of schedules enforced; thus, a static analysis needs to consider only traces in $run_S(P)$ for each enforced schedule S, potentially computing much more precise results.

To realize the idea of schedule specialization, we leverage our PEREGRINE system to compute and enforce a set of schedules, and use our specialization framework to analyze a program with respect to these schedules. The rest of this section describes the functionality of the PEREGRINE system (§2.2.1) and the specialization framework (§2.2.2), and then discusses the assumptions and non-assumptions of our system (§2.2.3).

2.2.1 Enforcing Schedules

Our PEREGRINE system first computes a finite set of schedules to enforce and analyze. We use SC to denote these schedules. PEREGRINE computes these schedules by recording the traces of the program on real representative workload and filtering synchronizations from terminating traces.

Because these schedules are computed from real executions, they are guaranteed to be *feasible*, *i.e.*, $run_S(P) \neq \emptyset$. PEREGRINE also ensures that different schedules in SC have disjoint preconditions. In other words, if $S_1, S_2 \in SC$ and $S_1 \neq S_2$, then $\neg(cond(P, S_1) \wedge cond(P, S_2))$. Therefore, an input can be processed by at most one schedule in SC.

PEREGRINE then enforces each schedule in SC on the inputs that satisfy its precondition. When input I arrives, PEREGRINE searches SC for schedule S such that $cond(P, S)(I)$, and enforces S on input I. Since SC is finite, it may not cover all the inputs. §2.2.3 will discuss how we handle inputs that are not covered.

2.2.2 Specialization Framework

With a set of schedules enforced by PEREGRINE, schedule specialization analyzes the program with respect to each enforced schedule. One method to provide schedule-aware analysis is to modify every analysis to be aware of the schedule, but this method has the drawbacks discussed in §1.4.2. To avoid modifying every analysis, our specialization framework rewrites P into a specialized program P_S for each enforced schedule S so that $run(P_S) \supseteq run_S(P)$. P_S can then be analyzed with many stock analyses for improved precision. To analyze P over a set of schedules, we can generate a specialized program for each schedule, then merge the analysis results. In addition, our framework provides a constraint-based def-use analysis on P_S that computes precise must-def-use chains on memory locations allowed only by schedule S. By querying this def-use analysis, stock or slightly modified advanced analyses such as alias analysis can then compute schedule-aware results.

Precision of the specialization framework. Ideally for full precision, our specialization framework should make $run(P_S) = run_S(P)$, so that static analysis of P_S considers only traces in $run_S(P)$. In our current framework, $run(P_S)$ may still include traces not in $run_S(P)$ because, without enforcing S with PEREGRINE, an execution of P_S may use a different schedule. Nonetheless, since $run(P_S)$ is much smaller than $run(P)$, stock analyses on P_S should still yield more precise results than on P. Our def-use analysis excludes def-use chains forbidden by the total order in S. That is, it considers traces in $run_S(P_S)$, which our specialization algorithms (§4.2) guarantee to include $run_S(P)$. The def-use results provided by our framework are thus much more precise than what a stock def-use analysis can compute on P_S (§4.5.1).

Soundness of the specialization framework. Our specialization framework guarantees that

static analysis results on P_S hold for all traces in $run_S(P)$, because (1) the results from a static analysis on P_S should hold for all traces in $run(P_S)$; and (2) our specialization algorithms guarantee $run(P_S) \supseteq run_S(P)$ which we will prove in §4.2.3. In addition, our framework guarantees that the def-use results it provides hold for $run_S(P_S)$, which includes $run_S(P)$.

2.2.3 Assumptions and Non-Assumptions

2.2.3.1 Soundness

In our current system, the soundness of static analysis results is conditioned on that one of the analyzed schedules is enforced. A finite set of schedules collected by PEREGRINE, however, may not cover all inputs. If an input cannot be processed by any of the schedules (or it can but our framework cannot determine this fact), the static analysis results may no longer hold for the execution on this input. If users want to retain soundness on such inputs, they may use dynamic analysis. For instance, if the analysis goal is to verify race freedom, they may use dynamic data race detection on such inputs. If dynamic analysis is used, the overall runtime overhead of our framework may depend on the actual workload, or how frequently it can enforce an analyzed schedule. If it can never enforce any analyzed schedule, it degenerates to pure dynamic analysis on the entire workload with potentially high overhead; if it can always enforce an analyzed schedule, it pays no dynamic analysis overhead except the overhead to enforce a schedule. Fortunately, our PEREGRINE results show that, for many programs, a small set of schedules can cover a wide range of the workloads (§3.3.2). For these programs and workloads, our framework rarely needs to use dynamic analysis.

Moreover, even if we cannot use a small set of schedules to cover a wide range of workloads, schedule specialization is still useful for many applications. These applications include all read-only applications such as error detection and post-mortem analysis which do not require soundness on the inputs not covered by analyzed schedules. They also include optimizations because we can simply run the original program on the inputs not covered.

2.2.3.2 Data Race Freedom

At the algorithmic level, we do not assume data-race-free programs because we intend to keep our framework general and applicable to not just optimizations, but other applications such as verification and error detection. In particular, even if the program has data races, we can still

enforce a schedule on it (§3.3.2). Nonetheless, if optimization is the only goal, we can easily modify our algorithms to exploit this assumption for better results. At the implementation level, our current framework leverages the LLVM compiler, which may indeed assume race freedom. This assumption may be removed by re-implementing the LLVM components we leverage.

2.2.3.3 Lattice of Sub-Schedules

Our framework does not require that a schedule collected from a trace includes all synchronizations in the trace. Instead, users can customize the schedule granularity, or which synchronizations go into the schedules, by blacklisting certain synchronization statements.

Given a schedule S, we use $(\mathscr{L}, \sqsubseteq)$ to denote a lattice of all sub-schedules of S we can get by blacklisting synchronization statements. To ensure our specialization algorithms can work correctly, sub-schedules in L need to include at least the entry and exits of `main`; otherwise, our specialization framework would generate an empty program. We denote this minimum set of synchronization statements by $H_\perp$. Therefore, $\mathscr{L}$ includes all $filter(S, i.label \in H')$ where $H_\perp \subseteq H'$ and $H' \subseteq H$.

The partial order of the lattice reflects the granularity of the sub-schedules of S: $S_1 \sqsubseteq S_2$ if S_1 is more fine-grained than S_2. In other term, $S_1 \sqsubseteq S_2$ if $S_1 = filter(S, i.label \in H_1)$, $S_2 = filter(S, i.label \in H_2)$, and $H_1 \subseteq H_2$. Under this partial order, the top of $\mathscr{L}$, denoted by $\top$, is schedule S; the bottom of $\mathscr{L}$, denoted by $\perp$, is $filter(S, i.label \in H_\perp)$ which contains only two synchronizations, the entry and one of the exits of function `main`.

Specializing programs with respect to coarse-grained sub-schedules enables users to make flexible three-way tradeoffs between precision, analysis time, and schedule reuse-rate. In general, fine-grained sub-schedules have lower reuse-rates and lead to larger specialized programs and longer analysis time, but make the analysis results more precise. One exception is that users should typically blacklist synchronizations hidden behind an abstraction boundary, such as a memory allocator interface, because these synchronizations rarely improve precision but worsen analysis time and reuse-rate. For instance, we blacklisted the lock operations in the custom memory allocator in `cholesky` (§4.3). An interesting research question is how to optimally make this three-way tradeoff, which we leave for future work.

Chapter 3

Collecting and Enforcing Schedules

As mentioned in §1.3.1, the goal of PEREGRINE is to collect reusable schedules and enforce these schedules deterministically and efficiently at runtime. Achieving this goal faces two challenges.

- *How to collect reusable schedules?* One possibility is to use static analysis to compute these schedules. However, doing so is extremely difficult for complicated programs, because the halting problem makes computing even feasible schedules undecidable. Another possibility is to compute these schedules during program execution. However, these computations may be complicated and incur large performance overhead which can discourage user adoption.

- *How to collect reusable schedules?* One possibility is to use static analysis to compute these schedules. However, predicting the behavior of the program without running it is extremly difficult. Therefore, static analysis can hardly compute schedules that are compatible with non-trivial executions.

  Another possibility is to compute these schedules during program execution. For instance, existing DMT systems typically use a deterministic scheduling algorithm to ensure that a program runs into the same schedule under the same input. However, such algorithms do not reuse schedules on different inputs: if the input slightly changes, the program may run into a complete different schedule. Therefore, simply leveraging these DMT systems into schedule specialization may result in too many schedules to enforce and analyze. To reuse schedules, one could compute whether a schedule can be used to process an input. However, this computation tends to be complicated, and performing this computation entirely at runtime

can incur large performance overhead which can discourage user adoption.

- *How to deterministically and efficiently enforce schedules?* Enforcing schedules both deterministically and efficiently for general programs on commodity hardware is an open challenge, because these programs may have data races. Existing systems that enforce schedules solve only one of the two problems. One type of systems enforces a deterministic order of shared memory accesses to resolve data races. However, scheduling all shared memory accesses can incur prohibitive overhead. For instance, CoreDet [Bergan *et al.*, 2010a] enforces memory access orders and incurs 1.2X-6X runtime overhead. The other type of systems enforces only a deterministic order of synchronizations. Although such systems can reduce runtime overhead (*e.g.*, Kendo [Olszewski *et al.*, 2009] has a low overhead of 16%), they cannot guarantee full determinism on programs containing data races.

To address the challenge of collecting schedules, PEREGRINE records schedules from past real executions and reuses these schedules on future inputs. It works as follows. At runtime, PEREGRINE maintains a persistent cache of schedules recorded from past executions. When an input arrives, PEREGRINE searches the cache for a schedule compatible with the input. If it finds one, it simply runs the program while enforcing the schedule. Otherwise, it runs the program as is while recording a new schedule from the execution, and saves the schedule into the cache for future reuse.

This approach of collecting schedules has several benefits for schedule specialization. First, the schedules recorded by PEREGRINE are guaranteed to be feasible because they are recorded from real executions. Second, by reusing a schedule on future compatible inputs, PEREGRINE shrinks the set of schedules schedule specialization needs to statically analyze, and makes soundness easier to achieve. Finally, PEREGRINE explicitly stores schedules, so schedule specialization can easily analyze them for precision. In fact, PEREGRINE also benefits from this explicitness because it analyzes a schedule to compute what inputs are compatible with it (§3.2).

One problem this approach needs to solve is how to check that an input is compatible with a schedule before executing the input under the schedule. PEREGRINE solves this problem by using an idea called *determinism-preserving slicing*. When recording a schedule, PEREGRINE records a detailed execution trace. Then, PEREGRINE slices the trace to keep only the statement instances that affect the feasibility of the schedule. Finally, PEREGRINE symbolically executes the slice

and computes the input constraints required to ensure that the execution is compatible with the schedule. The conjunction of these input constraints is the precondition of using the schedule. When an input arrives, PEREGRINE checks the input against the precondition of each schedule to determine which one to reuse.

To deterministically and efficiently enforce schedules, PEREGRINE enforces hybrid schedules: it enforces memory access schedules only in racy portions of an execution and synchronization schedules otherwise. Although the idea of hybrid schedules sounds appealing, how to efficiently enforce memory access schedules in racy portions still remains unclear. Statically instrumenting all memory accesses involved in statically-detected data races would be too expensive due to the imprecision of static race detectors.

PEREGRINE solves this problem using dynamic instrumentation. Before a program enters a racy portion, PEREGRINE dynamically updates the running program to enforce memory access orders in the racy portion. Because a dominant majority of the execution is race-free, the program runs the original code most of the time, which allows PEREGRINE to pay only the bare instrumentation overhead. Furthermore, to reduce the instrumentation overhead, we leverage a fast instrumentation framework we built for LOOM (§5.4).

In the remainder of this chapter, we first describe the architecture of PEREGRINE and illustrate the functionality of each component using an example. We then describe two ideas of PEREGRINE that are crucial to schedule specialization: computing preconditions (§3.2) and hybrid schedules (§3.3). Then, we show the evaluation results (§3.4), and related work (§3.5). Finally, we summarize this chapter (§3.6).

3.1 Overview and Example

Figure 3.1 shows the architecture of PEREGRINE. It has four main components. The *instrumentor* is an LLVM compiler plugin that prepares a program for use with PEREGRINE. It instruments synchronization operations such as `pthread_mutex_lock()`, which the recorder and replayer control at runtime. It automatically marks the `main()` arguments, data read from `read()`, `fscanf()`, and `recv()`, and values returned by `random()`-variants as inputs, so precondition computing can know what variables to compute constraints on.

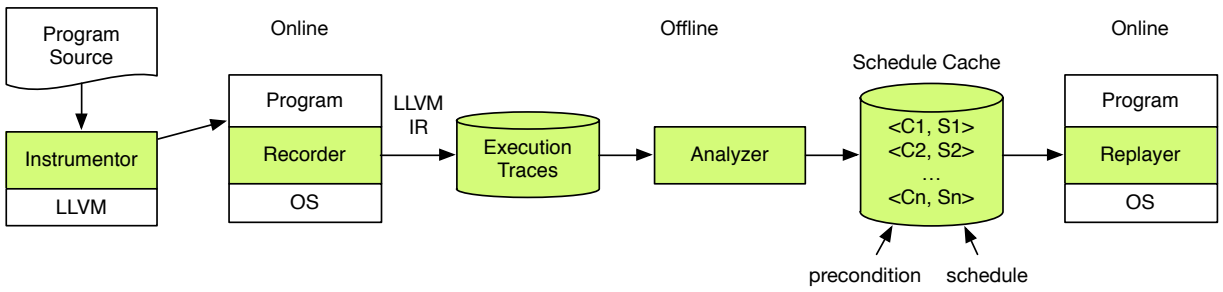


Figure 3.1: *Architecture of* PEREGRINE.

The *recorder* is similar to existing systems that deterministically record executions [Laadan *et al.*, 2010; Bhansali *et al.*, 2006; Dunlap *et al.*, 2008]. Our current recorder is implemented as an LLVM interpreter. When a program runs, the recorder saves the LLVM instructions interpreted for each thread into a central log file. The interpreter ensures dynamic statement instances across all threads are executed in a sequential order. Therefore, no data races occur in the recording phase. Given the programs we consider use a sequentially consistent concurrent memory order, every trace PEREGRINE collects corresponds to a real execution. Our recorder does not record external input data, such as data read from a file, because our analysis does not need this information. To schedule synchronization operations issued by different threads, the recorder can use a variety of DMT algorithms [Cui *et al.*, 2010].

Given the program and an execution trace, the *analyzer* is a stand-alone program that computes (1) a hybrid schedule S and (2) the precondition C required for reusing the schedule on future inputs. To compute a hybrid schedule, the analyzer first extracts a total order of synchronization operations from the execution trace. It then detects data races according to this synchronization order. Because the synchronizations in the schedule form a total order, statement instance i_1 happens before i_2 (denoted by $i_1 \prec i_2$) as long as we can find two synchronizations s_j and s_k such that $i_1 \prec s_j \wedge s_k \prec i_2 \wedge j < k$. Therefore, our data race detection algorithm is much simpler than traditional dynamic data race detection based on vector clocks. After detecting data races according to a synchronization order, PEREGRINE computes additional *execution order* constraints to deterministically order the statement instances involved the detected races. To compute the precondition of a schedule, PEREGRINE first slices the execution trace into a trace

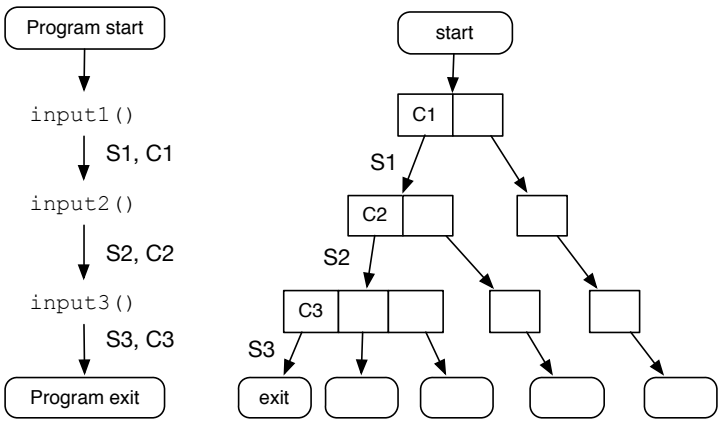


Figure 3.2: *Decision tree of* PEREGRINE*'s schedule cache.*

slice with instructions required to 1) reach all events in the schedule and 2) avoid races that did not occur in the recorded execution. For example, if a branch that is not taken in the recorded execution may introduce new races, the computation of the branch condition will be included in the slice. PEREGRINE then uses *symbolic execution* [King, 1975] to collect preconditions from the input-dependent branches in the slice. The trace slice is typically much smaller than the execution trace, so that the analyzer can compute relaxed preconditions, allowing frequent reuses of the schedule. The analyzer finally stores $\langle C, S \rangle$ into the schedule cache, which conceptually holds a set of such tuples.

Although the schedule cache is conceptually a set of $\langle C, S \rangle$ tuples, its actual structure is a decision tree because a program may incrementally read inputs from its environment. Figure 3.2 illustrates how PEREGRINE constructs the decision tree of the schedule cache. Given a $\langle C, S \rangle$ tuple, PEREGRINE breaks it down to sub-tuples $\langle C_i, S_i \rangle$ separated by `symbolic()` calls, where S_i contains the synchronization operations logged and C_i contains the constraints collected between the i^{th} and $(i+1)^{th}$ calls that read external input. It then merges the sub-tuples into the i^{th} level of the decision tree.

The *replayer* is a lightweight user-space scheduler for reusing schedules. When an input arrives, it searches the schedule cache for a $\langle C, S \rangle$ tuple such that the input satisfies the precondition C. If it finds such a tuple, it simply runs the program enforcing schedule S efficiently and deterministically. Otherwise, it forwards the input to the recorder.

3.1.1 An Example

Figure 3.3 shows our running example, a simple multithreaded program based on the real ones used in our evaluation. It first parses the command line arguments into `nthread` (line L1) and `size` (L2), then spawns `nthread` threads including the main thread (L4–L5) and processes `size/nthread` bytes of data in each thread. The thread function `worker()` allocates a local buffer (L10), reads data from a file (L11), processes the data (L12–L13), and sums the results into the shared variable `result` (L14–L16). The `main()` function may further update `result` depending on `argv[3]` (L7–L8), and finally prints out `result` (L9). This example has read-write and write-write races on `result` due to missing `pthread_join()`. This error pattern matches some of the real errors in the evaluated programs such as `PBZip2`.

Instrumentor. To run this program with PEREGRINE, we first compile it into LLVM IR and instrument it with the instrumentor. The instrumentor replaces the synchronization operations (lines L5, L14, and L16) with PEREGRINE-provided wrappers controlled by the recorder and replayer at runtime. It also inserts code to mark the contents of `argv[i]` and the data from `read()` (line L11) as input.

Recorder: execution trace. When we run the instrumented program with arguments "`2 2 0`" to spawn two threads and process two bytes of data, suppose that the recorder records the execution trace in Figure 3.4. (This figure also shows the hybrid schedule and preconditions PEREGRINE computes, explained later in this subsection.) This trace is just one possible trace depending on the scheduling algorithm the recorder uses.

Analyzer: hybrid schedule. Given the execution trace, the analyzer starts by computing a hybrid schedule. It first extracts a sync-schedule consisting of the operations tagged with (1), (2), ..., (8) in Figure 3.4. It then detects races in the trace according to this sync-schedule, and finds the race on `result` between L15 of thread t_1 and L9 of t_0. It then computes an execution order constraint to deterministically resolve this race, shown as the dotted arrow in Figure 3.4. The sync-schedule and execution order constraint together form the hybrid schedule. Although this hybrid schedule constrains the order of synchronization and the last two accesses to `result`, it can still be efficiently reused because the core computation done by `worker` can still run in parallel.

Analyzer: trace slice. The analyzer uses determinism-preserving slicing to reduce the execution

```
    int size; // total size of data
    int nthread; // total number of threads
    unsigned long result = 0;

    int main(int argc, char *argv[]) {
L1:     nthread = atoi(argv[1]);
L2:     size = atoi(argv[2]);
L3:     assert(nthread>0 && size>=nthread);
L4:     for(int i=1; i<nthread; ++i)
L5:         pthread_create(..., worker, NULL);
L6:     worker(NULL);
        // NOTE: missing pthread_join()
L7:     if(atoi(argv[3]) == 1)
L8:         result += ...; // race with line L15
L9:     printf("result = %lu\n", result); // race with line L15
        ...
    }
    void *worker(void *arg) {
L10:    char *data = malloc(size/nthread);
L11:    read(..., data, size/nthread);
L12:    for(int i=0; i<size/nthread; ++i)
L13:        data[i] = ...; // compute using data
L14:    pthread_mutex_lock(&mutex);
L15:    result += ...; // race with lines L8 and L9
L16:    pthread_mutex_unlock(&mutex);
        ...
    }
```

Figure 3.3: *Running example.* It uses the common divide-and-conquer idiom to split work among multiple threads. It contains write-write (lines L8 and L15) and read-write (lines L9 and L15) races on `result` because of missing `pthread_join()`.

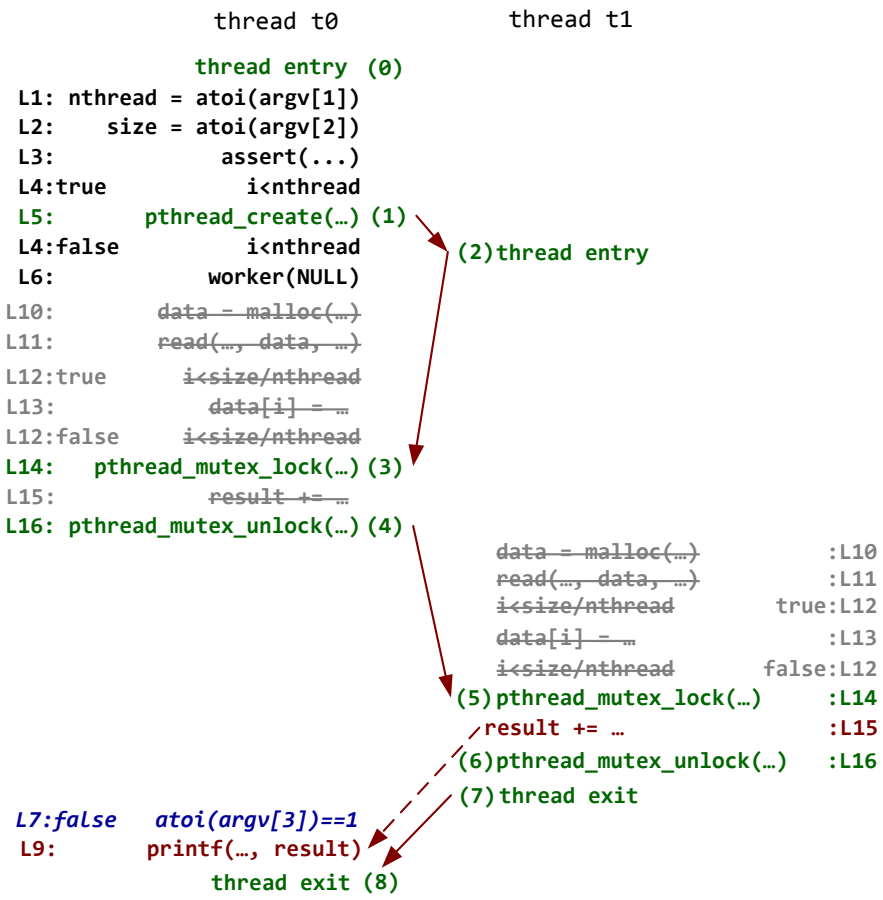


Figure 3.4: *Execution trace, hybrid schedule, and trace slice.* An execution trace of the program in Figure 3.3 on arguments "2 2 0" is shown. Each executed instruction is tagged with its static line number Li. Branch instructions are also tagged with their outcome (true or false). Synchronization operations (green), including thread entry and exit, are tagged with their relative positions in the synchronization order. They form a sync-schedule whose order constraints are shown with solid arrows. L15 of thread t_1 and L9 of thread t_0 race on `result`, and this race is deterministically resolved by enforcing an execution order constraint shown by the dotted arrow. Together, these order constraints form a hybrid schedule. Instruction L7 of t_0 (italic and blue) is included in the trace slice to avoid new races, while L6, L4:false, L4:true, L3, L2, and L1 of t_0 are included due to intra-thread dependencies. Crossed-out (gray) instructions are elided from the slice.

$$(atoi\_argv_1 = 2) \wedge (atoi\_argv_2 \geq atoi\_argv_1) \wedge (atoi\_argv_3 \neq 1)$$

Figure 3.5: *Preconditions computed from the trace slice in Figure 3.4.* Variable $atoi\_argv_i$ represents the return of `atoi(arg[i])`.

trace into a trace slice, so that it can compute relaxed preconditions. The final trace slice consists of the instructions not crossed out in Figure 3.4. The analyzer computes this trace slice using inter-thread and intra-thread steps. In the inter-thread step, it adds instructions required to avoid new races into the slice. Specifically, for t_0 it adds the false branch of L7, or L7:false, because if the true branch is taken, a new race between L8 of t_0 and L15 of t_1 occurs. It ignores branches of line L12 because alias analysis already determines that L13 of t_0 and L13 of t_1 never race.

In the intra-thread step, the analyzer adds instructions required to reach all instructions identified in the inter-thread step (L7:false of t_0 in this example) and all events in the hybrid schedule. It does so by traversing the execution trace backwards and tracking control- and data-dependencies. In this example, it removes L15, L13, L12, L11, and L10 because no instructions currently in the trace slice depend on them. It adds L6 because without this call, the execution will not reach instructions L14 and L16 of thread t_0. It adds L4:false because if the true branch is taken, the execution of t_0 will reach one more `pthread_create()`, instead of L14, `pthread_mutex_lock()`, of t_0. It adds L4:true because this branch is required to reach L5, the `pthread_create()` call. It similarly adds L3, L2, and L1 because later instructions in the trace slice depend on them.

Analyzer: preconditions. After slicing, all branches from L12 are gone. The analyzer joins the remaining branches together as the preconditions, using a version of KLEE [Cadar *et al.*, 2008] augmented with thread support [Cui *et al.*, 2010]. Specifically, the analyzer marks input data as *symbolic*, and then uses KLEE to track how this symbolic data is propagated and observed by the instructions in the trace slice. If a branch instruction inspects symbolic data and proceeds down the true branch, the analyzer adds the precondition that the symbolic data makes the branch condition true. The analyzer uses symbolic summaries [Costa *et al.*, 2007] to succinctly generalize common library functions. For instance, it considers the return of `atoi(arg)` symbolic if `arg` is symbolic.

Figure 3.5 shows the preconditions the analyzer computes from the trace slice in Figure 3.4. These preconditions illustrate two key benefits of PEREGRINE. First, they are sufficient to ensure deterministic reuses of the schedule. Second, they only loosely constrain the data size ($atoi\_argv_2$)

and do not constrain the data contents (from `read()`), allowing frequent schedule-reuses. The reason is that L10–L13 are all sliced out. One way to leverage this benefit is to populate a schedule cache with small workloads to reduce analysis time, and then reuse the schedules on large workloads.

Replayer. Suppose we run this program again on different arguments "`2 1000 8`." The replayer checks the new arguments against the preconditions in Figure 3.5 using KLEE's constraint checker, and finds that these arguments satisfy the preconditions, despite the much larger data size. It can therefore reuse the hybrid schedule in Figure 3.4 on this new input by enforcing the same order of synchronization operations and accesses to `result`.

3.2 Computing Preconditions

One challenge for computing reusable schedules is how to check whether a input is compatible with a recorded schedule before executing the input under the schedule. Otherwise, if PEREGRINE tries to enforce a schedule, for instance, of two threads on an input that requires four, the execution would not follow the schedule. This challenge turns out to be the most difficult one we must solve in building PEREGRINE. Our final solution leverages several advanced program analysis techniques, including two new ones we invent. We refer interested readers to our papers [Cui *et al.*, 2010; Cui *et al.*, 2011] for details, and only describe the high level idea here.

PEREGRINE computes preconditions of a schedule using a modified version [Cui *et al.*, 2011] of *path slicing* [Jhala and Majumdar, 2005] on the recorded execution trace to remove statement instances that do not affect the feasibility of the hybrid schedule, *i.e.*, the reachability of all synchronizations in the schedule. Our version of path slicing correctly tracks dependencies (*e.g.*shared data accesses) *across* threads. We call this slicing algorithm *determinism-preserving slicing*. Once we compute a path slice for each thread, we *symbolically* execute the slice, tracking the path constraints on input data, such as command line arguments and data read from a file or socket. We then use the conjunction of the constraints from each thread as the preconditions of the hybrid schedule.

These preconditions PEREGRINE computes have three properties.

- They do not guarantee program termination because we want to sidestep the difficult problem of statically establishing termination. This property is inherited from path slicing [Jhala and

Majumdar, 2005].

- Since computing weakest preconditions is undecidable, the preconditions PEREGRINE computes stay on the sound side: if an input satisfies these preconditions, it can be processed by the corresponding hybrid schedule (modulo termination). However, these preconditions may preclude inputs that can indeed be processed by the hybrid schedule.

- These preconditions avoid data races, which may cause an execution to diverge, preventing PEREGRINE from enforcing a given hybrid schedule. When computing preconditions, PEREGRINE detects potential races that did not occur in a recorded trace, but may occur if we reuse the hybrid schedule on different inputs. It computes preconditions sufficient to avoid these races.

Our evaluation results (§3.4.3) show that a small number of schedules can cover a wide range of workloads for half of the 18 evaluated programs. We observe that the synchronizations in these programs depend mostly on "meta" properties such as the number of processor cores, and tend to be loosely coupled with the inputs. For these programs, the coverage of a schedule can be extremely high. For instance, consider `PBZip2` which divides an input file evenly among multiple threads to compress. Two schedules are sufficient to compress any file for `PBZip2` as long as the number of worker threads remains the same. (`PBZip2` needs one schedule if the file size can be evenly divided and another otherwise.) Another data point: about a hundred schedules are sufficient to cover over 90% of requests in a real HTTP trace for `Apache` [Apache, 2012]. This *stability* [Cui *et al.*, 2010] not only makes program behaviors repeatable across inputs, but also reduces the runtime overhead of our specialization framework (Chapter 4).

3.3 Deterministically and Efficiently Enforcing Schedules

To deterministically and efficiently enforce schedules, PEREGRINE enforces "hybrid" schedules that combine synchronization schedules and memory access schedules. Our insight is that although many programs have races, the races tend to occur only within small portions of an execution, and the majority of the execution is still race-free. Intuitively, if a program is full of data races, most of them would have been caught during testing. Empirically, we analyzed the executions of seven

real programs with races, and found that, despite millions of memory accesses, only up to 10 data races were detected per execution.

3.3.1 Construct a Hybrid Schedule

PEREGRINE constructs a hybrid schedule in two steps. It first extracts a synchronization schedule from the execution trace, and includes this synchronization schedule into the hybrid schedule. Then, PEREGRINE detects data races that occurred during the recorded execution, computes execution order constraints to deterministically resolve these races, and includes these constraints into the hybrid schedule.

Detecting data races using an off-the-shelf race detector would flag too many races because it considers the original synchronization constraints of the program. Since PEREGRINE already extracts the synchronization schedule and will enforce it, PEREGRINE only needs to detect races according to this schedule. This schedule-aware race detection is significantly more precise than general race detection. Researchers observed that schedules anecdotally reduced the number of possible races greatly, in one extreme case, from more than a million to four [Park *et al.*, 2009b; Ronsse and De Bosschere, 1999].

To later enforce a hybrid schedule, PEREGRINE explicitly represents a hybrid schedule in a data structure. PEREGRINE represents the synchronization schedule simply as a total order of synchronizations. PEREGRINE gives each static instruction in the executable file an unique ID. Therefore, each synchronization is simply identified by the unique ID of the static instruction. For each detected race, PEREGRINE includes into the hybrid schedule an execution order constraint $inst_1 \rightarrow inst_2$ where $inst_1$ and $inst_2$ are the two dynamic instruction instances involved in the race. PEREGRINE identifies a dynamic instruction instance by $\langle sid, tid, nbr \rangle$ where sid refers to the unique ID of the static instruction; tid refers to the internal thread ID maintained by PEREGRINE, which always starts from zero and increments deterministically upon each `pthread_create()`; and nbr refers to the number of control-transfer instructions (branch, call, and return) locally executed within the thread from the last synchronization to $inst_1$ or $inst_2$. We must distinguish different dynamic instances of a static instruction because some of these dynamic instances may be involved in races while others are not. We do so by counting branches because if an instruction is executed twice, there must be a control-transfer between the two instances [Dunlap *et al.*, 2008]. We

count branches starting from the last synchronization because the partial schedule preceding this synchronization is already made deterministic.

3.3.2 Enforcing a Hybrid Schedule

A hybrid schedule contains a synchronization schedule and execution order constraints. PEREGRINE uses different techniques to enforce the synchronization schedule and the execution order constraints.

To enforce a synchronization schedule, PEREGRINE uses a technique called *semaphore relay* [Cui *et al.*, 2010] that orders synchronizations with per-thread semaphores. PEREGRINE statically wraps each synchronization statement. At runtime, a synchronization wrapper waits on the semaphore of the current thread. Once it is woken up, it proceeds with the actual synchronization, and then wakes up the next thread according to the synchronization order.

Efficiently enforcing a memory access schedule is more challenging, because memory accesses occur much more frequently than synchronizations. If we simply statically instrument all memory accesses, the program will run much slower. Even if we use a static race detector and only instrument memory accesses that involve in detected races, the runtime overhead can still be large.

To address this challenge, PEREGRINE uses dynamic instrumentation to enforce memory access schedules. Our insight is that, although many memory accesses can be involved in data races, when enforcing a hybrid schedule, few of them are really involved. Dynamic instrumentation allows PEREGRINE to only instrument occurred races, and leave most of the program uninstrumented.

Figure 3.6 shows the function PEREGRINE instruments the program to call before each static instruction *sid*. Given a dynamic instruction instance $\langle sid, tid, nbr \rangle$ (§3.3.1), if it is used as the target of an execution order, PEREGRINE inserts a semaphore `down()` operation before the static instruction *sid*; if the instance is used as a source, PEREGRINE inserts a semaphore `up()` operation after *sid* (Line 4–8). It also instruments the branch instructions counted in *nbr* so that when each of these branch instructions runs, a per-thread branch counter is incremented (Line 2–3). PEREGRINE activates the inserted semaphore operation for thread *tid* only when the thread's branch counter matches *nbr* (Line 7). To avoid interference and unnecessary contention when there are multiple order constraints, PEREGRINE assigns a unique semaphore to each constraint.

Although using dynamic instrumentation avoids lots of unnecessary instrumentation, running

```
1: void slot(int sid) { // sid is static instruction id
2:   if(instruction sid is branch)
3:     nbr[self()] ++; // increment per-thread branch counter
4:   // get semaphore operations for current thread at instruction sid
5:   my_actions = actions[sid][self()];
6:   for action in my_actions
7:     if nbr[self()] == action.nbr // check branch counter
8:       actions.do(); // perform up() or down()
9: }
```

Figure 3.6: *Instrumentation to enforce execution order constraints.*

dynamic instrumentation tools alone may cause large performance overhead. For example, Pin has been reported to incur 199% overhead [Luk *et al.*, 2005], and we observed 10 times slowdown on Apache with a CPU-bound workload (§5.5). To reduce this instrumentation overhead, PEREGRINE leverages a fast instrumentation framework we built for LOOM. Running our instrumentation framework alone incurs negligible overhead for most of the evaluations programs. We will detail this framework in §5.4.

3.4 Evaluation

We evaluated our PEREGRINE implementation on a diverse set of 17 programs, including `Apache`, a popular web server; `PBZip2`, a parallel compression utility; `aget`, a parallel `wget`-like utility; `pfscan`, a parallel `grep`-like utility; parallel implementations of 13 computation-intensive algorithms, 10 in SPLASH2 and 3 in PARSEC; and `racey`, a benchmark specifically designed to exercise deterministic execution and replay systems [Hill and Xu, 2009]. All SPLASH2 benchmarks were included except one that we cannot compile, one that our current prototype cannot handle due to an implementation bug, and one that does not run correctly in 64-bit environment. The chosen PARSEC benchmarks (`blackscholes`, `swaptions` and `streamcluster`) include the ones that (1) we can compile, (2) use threads, and (3) use no x86 inline assemblies. These programs were widely used in previous studies (*e.g.*, [Lu *et al.*, 2008; Xiong *et al.*, 2010; Berger *et al.*, 2009]).

Our evaluation machine was a 2.67 GHz dual-socket quad-core Intel Xeon machine with 24

Program	Race Description
`Apache`	Reference count decrement and check against 0 are not atomic, resulting in a program crash.
`PBZip2`	Variable `fifo` is used by one thread after being freed by another thread, resulting in a program crash.
`barnes`	Variable `tracktime` is read by one thread before assigned the correct value by another thread.
`fft`	`initdonetime` and `finishtime` are read by one thread before assigned the correct values by another thread.
`lu-non-contig`	Variable `rf` is read by one thread before assigned the correct value by another thread.
`streamcluster`	PARSEC has a custom barrier implementation that synchronizes using a shared integer flag `is_arrival_phase`.
`racey`	Numerous intentional races caused by multiple threads reading and writing global arrays `sig` and `m` without synchronization.

Table 3.1: *Programs used for evaluating* PEREGRINE*'s determinism.*

GB memory running Linux 2.6.35. When evaluating PEREGRINE on `Apache` and `aget`, we ran the evaluated program on this machine and the corresponding client or server on another to avoid contention between the programs. These machines were connected via 1Gbps LAN. We compiled all programs to machine code using `llvm-gcc -O2` and the LLVM compiler `llc`. We used eight worker threads for all experiments.

Unless otherwise specified, we used the following workloads in our experiments. For `Apache`, we used `ApacheBench` [apa, 2014] to repeatedly download a 100 KB webpage. For `PBZip2`, we compressed a 10 MB randomly generated text file. For `aget`, we downloaded a 77 MB file (`Linux-3.0.1.tar.bz2`). For `pfscan`, we scanned the keyword `return` from 100 randomly chosen files in GCC. For SPLASH2 and PARSEC programs, we ran workloads which typically completed in 1-100 ms.

In the remainder of this section, we focus on three questions:

- Is PEREGRINE deterministic if there are data races (§3.4.1)? Determinism is one of the strengths of PEREGRINE over the approach that enforces synchronization schedules.

- Is PEREGRINE fast (§3.4.2)? For typical multithreaded programs that have rare data races, PEREGRINE should be roughly as fast as the approach using synchronization schedules. Efficiency is one of the strengths of PEREGRINE over the approach using memory access schedules.
- Is PEREGRINE stable (§3.4.3)? That is, can it frequently reuse schedules? The higher the reuse rate, the more repeatable program behaviors become and the more PEREGRINE can amortize the cost of computing hybrid schedules.

3.4.1 Determinism

We evaluated PEREGRINE's determinism by checking whether PEREGRINE could deterministically resolve races. Table 3.1 lists the seven racy programs used in this experiment. We selected the first five because they were frequently used in previous studies [Lu *et al.*, 2006; Park *et al.*, 2009a; Lu *et al.*, 2008; Park *et al.*, 2009b] and we could reproduce their races on our evaluation machine. We selected the integer flag race in PARSEC to test whether PEREGRINE can handle ad hoc synchronization [Xiong *et al.*, 2010]. We selected `racey` to stress test PEREGRINE: each run of `racey` may have thousands of races, and if any of these races is resolved differently, `racey`'s final output changes with high probability [Hill and Xu, 2009].

For each program with races, we recorded an execution trace and computed a hybrid schedule from the trace. Table 3.2 shows for each program (1) the number of dynamic races detected according to the synchronization schedule and (2) the number of execution order constraints in the hybrid schedule. The latter can be smaller than the former because PEREGRINE can remove redundant execution constraints that are subsumed by others. In particular, PEREGRINE prunes 94% of the constraints for `racey`. For `Apache` and `streamcluster`, their races are already resolved deterministically by their synchronization schedules, so PEREGRINE adds no execution order constraints.

To verify that the hybrid schedules PEREGRINE computed are deterministic, we first manually inspected the order constraints PEREGRINE added for each program except `racey` (because it has too many races for manual verification). Our inspection results show that these constraints are sufficient to resolve the corresponding races. We then re-ran each program including `racey` 1000 times while enforcing the hybrid schedule and injecting delays; and verified that each run reused the schedule and computed equivalent results. (We determined result equivalence by checking either

Program	Races	Order Constraints
`Apache`	0	0
`PBZip2`	4	3
`barnes`	5	1
`fft`	10	4
`lu-non-contig`	10	7
`streamcluster`	0	0
`racey`	167974	9963

Table 3.2: *Hybrid schedule statistics.* Column **Races** shows the number of races detected according the corresponding synchronization schedule, and Column **Order Constraints** shows the number of execution order constraints PEREGRINE adds to the final hybrid schedule. The latter can be smaller than the former because PEREGRINE prunes subsumed execution order constraints. PEREGRINE detected no races for `Apache` and `streamcluster` because the corresponding synchronization schedules are sufficient to resolve the races deterministically; it thus adds no order constraints for these programs.

the output or whether the program crashed.)

We also compared the determinism of PEREGRINE to our previous work [Cui *et al.*, 2010] which only enforces synchronization schedules. Specifically, we reran the seven programs with races 50 times enforcing only the synchronization schedules and injecting delays, and checked whether the reuse runs computed equivalent results as the recorded run. As shown in Table 3.3, synchronization schedules are unsurprisingly deterministic for `Apache` and `streamcluster`, because no races are detected according to the corresponding synchronization schedules. However, they are not deterministic for the other five programs, illustrating one advantage of PEREGRINE over the synchronization schedule approach.

3.4.2 Enforcing Overhead

The most performance-critical component of PEREGRINE is the replayer (Figure 3.1) which enforces a schedule, because it operates within a deployed program. Figure 3.7 shows the execution times when reusing hybrid schedules; these times are normalized to the nondeterministic execution time.

Program	Deterministic?	
	synchronization schedule	hybrid schedule
Apache	✔	✔
PBZip2	✖	✔
barnes	✖	✔
fft	✖	✔
lu-non-contig	✖	✔
streamcluster	✔	✔
racey	✖	✔

Table 3.3: *Determinism of synchronization schedules v.s. hybrid schedules.*

(The next paragraph compares these times to those of synchronization schedules.) For `Apache`, we show the throughput (TPUT) and response time (RESP). All numbers reported were averaged over 500 runs. PEREGRINE has relatively high overhead on `water-nsquared` (22.6%) and `cholesky` (46.6%) because these programs do a large number of mutex operations within tight loops. Still, this overhead is lower than the reported 1.2X-6X overhead of a mem-schedule DMT system [Bergan *et al.*, 2010a]. Moreover, PEREGRINE speeds up `barnes`, `lu-non-contig`, `radix`, `water-spatial`, and `ocean` (by up to 68.7%) because PEREGRINE already enforces schedules and can safely skips some synchronization and sleep operations. For the other programs, PEREGRINE's overhead or speedup is within 15%. (Note that increasing the page or file sizes of the workload tends to reduce PEREGRINE's relative overhead because the network and disk latencies dwarf PEREGRINE's.)

For comparison, Figure 3.7 shows the normalized execution time when enforcing just the synchronization schedules. This overhead is comparable to our previous work [Cui *et al.*, 2010]. For all programs except `water-nsquared`, the overhead of enforcing hybrid schedules is only slightly larger (at most 5.4%) than that of enforcing synchronization schedules. This slight increase comes from two sources: (1) PEREGRINE has to enforce execution order constraints to resolve races deterministically for `PBZip2`, `barnes`, `fft`, and `lu-non-contig`; and (2) the instrumentation framework PEREGRINE uses also incurs overhead (§5.4). The overhead for `water-nsquared` increases by 13.4% because it calls functions more frequently than the other benchmarks, and our instrumentation framework inserts code at each function entry and return (§5.4).

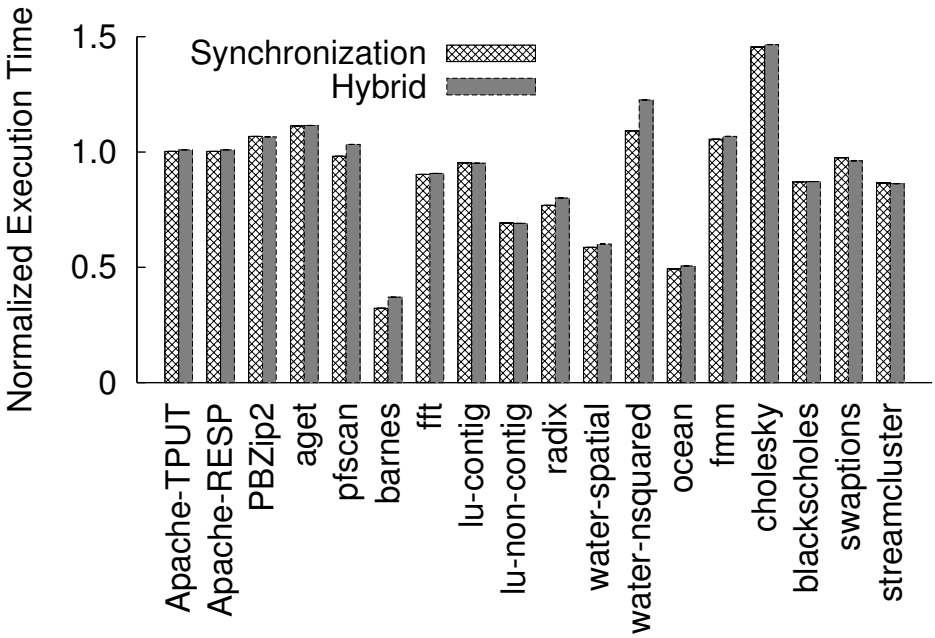


Figure 3.7: *Normalized execution time when reusing synchronization schedules v.s. hybrid schedules.* A time value greater than 1 indicates a slowdown compared to a nondeterministic execution without PEREGRINE. We did not include `racey` because it was not designed for performance benchmarking.

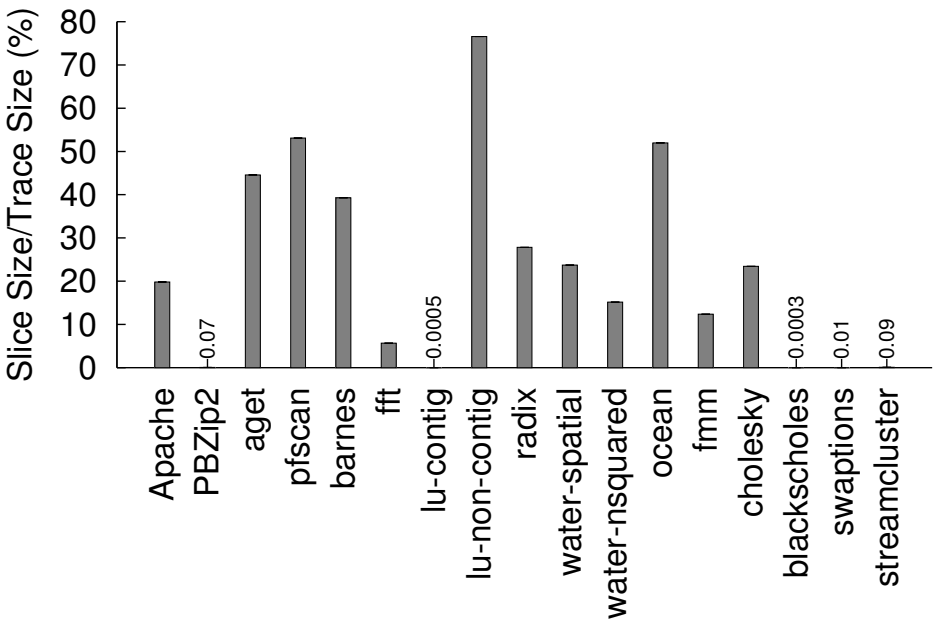


Figure 3.8: *Slicing ratio after applying determinism-preserving slicing*

3.4.3 Stability

Stability measures how frequently PEREGRINE can reuse schedules. The more frequently PEREGRINE reuses schedules, the fewer schedules schedule specialization needs to analyze.

A key factor determining PEREGRINE's schedule-reuse rates is how effectively it can slice out irrelevant instructions from the execution traces. Figure 3.8 shows the ratio of the slice size over the trace size for PEREGRINE's determinism-preserving slicing technique. The slicing technique reduces the trace size by over 50% for all programs except `pfscan`, `lu-non-contig`, and `ocean`.

Recall that PEREGRINE computes the preconditions of a schedule from the input-dependent branches in a trace slice. The fewer branches included in the slice, the more general the preconditions PEREGRINE computes tend to be. We further measured the number of such branches in the trace slices. Table 3.4 shows the results, together with a upper bound determined by the total number of input-dependent branches in the execution trace, and a lower bound determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency. For `barnes`, `fft`, `blackscholes`, `swaptions`, and

Program	UB	Slicing	LB
Apache	4,522	624	56
PBZip2	913	101	94
aget	20,826	9,514	9,491
pfscan	1,062,047	992,520	992,501
barnes	92	52	52
fft	2,266	17	17
lu-contig	2,823,379	131	128
lu-non-contig	2,962,621	2,876,364	128
radix	175,679	89,732	75
water-spatial	98,054	76,763	233
water-nsquared	89,348	76,242	1,843
ocean	2,605,185	2,361,256	400
fmm	299,816	56,532	1,642
cholesky	7,459	1,627	1,233
blackscholes	421,909	10	10
swaptions	35,584	21	21
streamcluster	20,851	42	42

Table 3.4: *Effectiveness of program analysis techniques.* **UB** shows the total number of input-dependent branches in the corresponding execution trace, an upper bound on the number included in the trace slice. **Slicing** shows the number of input-dependent branches in the slice after applying determinism-preserving slicing (§3.2). **LB** shows a lower bound on the number of input-dependent branches, determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency when computing it.

`streamcluster`, our slicing achieves the best possible reduction. For `PBZip2`, `aget`, `pfscan`, and `lu-contig`, the number of input-dependent branches in the trace slice is close to the lower bound. In the remaining programs, `Apache`, `fmm`, and `cholesky` also enjoy large reduction, while the other five programs do not.

We manually examined the preconditions PEREGRINE computed from the input-dependent branches for these programs. We categorize these programs below.

Best case: `PBZip2`, `fft`, `lu-contig`, `blackscholes`, `swaptions`, and `streamcluster`. PEREGRINE computes the weakest (*i.e.*, most relaxed) preconditions for these programs. The preconditions often allow PEREGRINE to reuse one or two schedules for each number of threads, putting no or few constraints on the data processed. Schedule-guided simplification is crucial for these programs; without simplification, the preconditions would fix the data size and contents.

Slicing limitation: `Apache` and `aget`. The preconditions PEREGRINE computes for `Apache` fix the URL length; they also constrain the page size to be within an 8 KB-aligned range if the page is not cached. The preconditions PEREGRINE computes for `aget` fix the positions of "`/`" in the URL and narrow down the file size to be within an 8 KB-aligned range. These preconditions thus unnecessarily reduce the schedule-reuse rates. Nonetheless, they can still match many different inputs, because they do not constrain the page or file contents.

Symbolic execution limitation: `barnes`. `barnes` reads in two floating point numbers from a file, and their values affect schedules. Since PEREGRINE cannot symbolically execute floating point instructions, it currently does not collect preconditions from them.

Alias limitation: `lu-non-contig`, `radix`, `water-spatial`, `water-nsquared`, `ocean`, and `cholesky`. Even with simplification, PEREGRINE's alias analysis sometimes reports may-alias for pointers accessed in different threads, causing PEREGRINE to include more instructions than necessary in the slices and compute preconditions that fix the input data. For instance, each thread in `lu-non-contig` accesses disjoint regions in a global array, but the accesses from one thread are *not* continuous, confusing PEREGRINE's alias analysis. (In contrast, each thread in `lu-contig` accesses a contiguous array partition.)

Programs that rarely reuse schedules: `pfscan` and `fmm`. For instance, `pfscan` searches a keyword in a set of files using multiple threads, and for each match, it grabs a lock to increment a

counter. A schedule computed on one set of files is unlikely to suit another.

3.5 Related Work

Deterministic execution. Existing DMT systems can enforce one schedule for the same input. However, when the input changes slightly, these systems may choose a completely different schedule. By reusing schedules, PEREGRINE mitigates input nondeterminism and makes program behaviors repeatable across inputs. This method is based on the *schedule-memoization* idea in our previous work TERN [Cui *et al.*, 2010], but PEREGRINE largely eliminates manual annotations, and provides stronger determinism guarantees than TERN. To our knowledge, no other DMT systems mitigate input nondeterminism; some actually aggravate it, potentially creating "input-heisenbugs."

PEREGRINE and other DMT systems can be complementary: PEREGRINE can use an existing DMT algorithm when it runs a program on a new input so that it may compute the same schedules at different sites; existing DMT systems can speed up their pathological cases using the schedule-relaxation idea.

Parallel functional languages automatically enjoy the benefits of determinism given its nature of referential transparency [Halstead, 1985]. However, they provide more restrictive parallel programming models than imperative languages we target in this thesis. For example, [Trinder *et al.*, 1998] introduced the `par` and `pseq` operations into Haskell as the basis of constructing parallel programs. However, these operations are difficult to use because using them requires programmers to understand operational properties of the language [Marlow *et al.*, 2011]. [Marlow *et al.*, 2011] makes writing parallel programs easier by proposing a new programming model that uses I-structures [Arvind *et al.*, 1989] for communication. However, I-structures are still more restrictive than process interaction (*e.g.* shared memory) in imperative languages. For instance, I-structures can only be written once.

Determinator [Aviram *et al.*, 2010] advocates a new, radical programming model that converts all races, including races on memory and other shared resources, into exceptions, to achieve pervasive determinism. This programming model is not designed to be backward-compatible. dOS [Bergan *et al.*, 2010b] provides similar pervasive determinism with backward compatibility, using a DMT algorithm first proposed in [Devietti *et al.*, 2009] to enforce mem-schedules. While

PEREGRINE currently focuses on multithreaded programs, the ideas in PEREGRINE can be applied to other shared resources to provide pervasive determinism. PEREGRINE's hybrid schedule idea may help reduce dOS's overhead. Grace [Berger *et al.*, 2009] makes multithreaded programs with fork-join parallelism behave like sequential programs. It detects memory access conflicts efficiently using hardware page protection. Unlike Grace, PEREGRINE aims to make general multithreaded programs, not just fork-join programs, repeatable.

Concurrent to our work, DTHREADS [Liu *et al.*, 2011] is another efficient DMT system. It tracks memory modifications using hardware page protection and provides a protocol to deterministically commit these modifications. In contrast to DTHREADS, PEREGRINE is software-only and does not rely on page protection hardware which may be expensive and suffer from false sharing; PEREGRINE records and reuses schedules; thus it can handle programs with ad hoc synchronizations [Xiong *et al.*, 2010] and make program behaviors stable.

Program analysis. Program slicing [Tip, 1995] is a general technique to prune irrelevant statements from a program or trace. Recently, systems researchers have leveraged or invented slicing techniques to block malicious input [Costa *et al.*, 2007], synthesize executions for better error diagnosis [Zamfir and Candea, 2010], infer source code paths from log messages for postmortem analysis [Yuan *et al.*, 2010], and identify critical inter-thread reads that may lead to concurrency errors [Zhang *et al.*, 2011]. Our determinism-preserving slicing technique produces a correct trace slice for multithreaded programs and supports multiple ordered targets. It thus has the potential to benefit existing systems that use slicing.

Replay and re-execution. Deterministic replay [Guo *et al.*, 2008; Geels *et al.*, 2007; Srinivasan *et al.*, 2004; Dunlap *et al.*, 2002; Konuru *et al.*, 2000; VMWare Virtual Lab Automation, ; Dunlap *et al.*, 2008; Park *et al.*, 2009b; Laadan *et al.*, 2010; Altekar and Stoica, 2009; Montesinos *et al.*, 2009] aims to replay the exact recorded executions, whereas PEREGRINE "replays" schedules on different inputs. Some recent deterministic replay systems include Scribe, which tracks page ownership to enforce deterministic memory access [Laadan *et al.*, 2010]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [Montesinos *et al.*, 2009]; PRES and ODR, which systematically search for a complete execution based on a partial one [Park *et al.*, 2009b; Altekar and Stoica, 2009]; SMP-ReVirt, which uses page protection for recording the order of conflicting memory accesses [Dunlap *et al.*, 2008]; and Respec [Lee *et al.*, 2010], which uses online

replay to keep multiple replicas of a multithreaded program in sync. Several systems [Park *et al.*, 2009b; Lee *et al.*, 2010] share the same insight as PEREGRINE: although many programs have races, these races tend to occur infrequently.

PEREGRINE can help these systems reduce CPU, disk, or network bandwidth overhead, because for inputs that hit PEREGRINE's schedule cache, these systems do not have to record a schedule.

Retro [Kim *et al.*, 2010] shares some similarity with PEREGRINE because it also supports "mutated" replay. When repairing a compromised system, Retro can replay legal actions while removing malicious ones using a novel dependency graph and *predicates* to detect when changes to an object need not be propagated further. PEREGRINE's determinism-preserving slicing algorithm may be used to automatically compute these predicates, so that Retro does not have to rely on programmer annotations.

3.6 Summary

This chapter described how PEREGRINE collects and enforces schedules. PEREGRINE records schedules and maps one schedule to multiple inputs by computing its preconditions. This approach of collecting schedules provides schedule specialization a small set of highly reusable schedule to analyze.

Leveraging the insight that races are rare, PEREGRINE combines synchronization orders and memory access orders into hybrid schedules, and enforces these hybrid schedules for both determinism and efficiency. It uses static instrumentation to schedule synchronizations and uses dynamic instrumentation to enforce the order between data races.

Chapter 4

Analyzing a Program with respect to Schedules

As mentioned in §1.3.2, the goal of our specialization framework is to precisely analyze a program with respect to a schedule. Intuitively, a schedule restricts the behavior of a program and can make static analysis easier. For example, if a schedule tells us only two threads are created, schedule specialization can focus on only the two-thread scenarios and analyze them more extensively for improved precision.

Although building precise schedule-aware analysis seems promising, achieving this goal faces one key challenge: there are too many static analysis techniques ("static analyses" for short). How are we going to improve the precision of all of them? One naïve method of leveraging a schedule is to manually make every static analyses involved aware of the schedule. However, this method has two drawbacks:

First, this method would be quite labor intensive and error prone. For instance, Chord [Naik *et al.*, 2006], a static race detector, uses at least alias analysis, thread-escaping analysis, lock analysis, and call graph analysis. Modifying each of them to leverage information a schedule provides would require much manual labor. Moreover, to emit precise analysis results, static analysis algorithms tend to be complicated; thus, implementing these algorithms is prone to bugs. For instance, a system we built [Wu *et al.*, 2013] found 29 bugs in the implementations of two very popular alias analysis algorithms that are not even aware of schedules.

Second, this method would be fragile. Static analyses depend on each other. If a crucial analysis is unaware of schedules, its imprecision may easily pollute other analyses. For instance, escaping analysis requires alias analysis to determine whether a locally-constructed variable can be accessed by other threads; call graph construction also requires alias analysis to resolve function pointers. If alias analysis is unaware of schedules, it can easily render escaping analysis and call graph construction imprecise, and further affect the precision of the tools that use them.

To address this challenge, we build a framework that uses a set of sound and precise algorithms to *specialize* a program according to a schedule (§2.2.2). The resultant program thus has simpler control and data flow than the original program, and can be analyzed with stock analyses for improved precision. In addition, our framework provides a precise def-use analysis that computes *schedule-aware*, *must*-def-use results on memory locations, which can be the foundation of many other powerful analyses (§4.4).

The specialization framework dramatically eases the process of making static analyses schedule aware. Instead of thinking about how to leverage schedules in every static analysis, developers can simply use our specialization framework to generate a new program and then run stock analyses as is or with slight modification (leveraging more advanced def-use results) on top of the specialized program.

The framework can also dramatically improve the precision of static analysis. For instance, it can easily detect code that is dead with respect to a schedule, so dead-code elimination can further simplify the control flow of a program. It can also "straighten" loops according to a schedule so that each synchronization statement is control-equivalent[1] to each other, making more variables computable at compile time. Because the specialized program has simpler control and data flow, static analyses automatically gain precision without being modified.

Our specialization framework works in two steps: control-flow and data-flow specialization. One key challenge for traditional static analysis is distinguishing different instances of a static statement. To mitigate this issue, control-flow specialization rewrites the program to map each synchronization in the schedule to a unique statement. As a result, each synchronization statement

[1]Two statements l_1 and l_2 are control-equivalent if l_1 dominates l_2 and l_2 post-dominates l_1 or if l_2 dominates l_1 and l_1 post-dominates l_2. Given two control-equivalent statements, an execution runs either both of them or neither of them.

in the specialized program corresponds to a unique dynamic instance. Also, two instances of the same statement that are separated by synchronizations will be mapped to two different statements in the specialized program.

The key algorithm used in control-flow specialization is graph traversal. For every two consecutive synchronizations of a thread, the algorithm traverses the control flow graph to find all statements between the two synchronizations, and copies these statements to the control-flow specialized program. This transformation makes all synchronization statements control-equivalent to each other, and prunes out statements that are guaranteed not executed according to a schedule. §4.2.1 will describe our control-flow specialization algorithm in more detail.

After control-flow specialization, we specialize the data flow of the program. Analyzing data flow of multithreaded programs is very challenging, because data can flow across threads. For instance, data written by one thread can be read by another. Due to this challenge, traditional data flow analysis is typically very conservative about inter-thread data flow. For example, both GCC and LLVM consider synchronization statements such as `pthread_mutex_lock` to read from or write to any memory locations, and reuse traditional sequential data flow analysis techniques to analyze a multithreaded program.

Data-flow specialization addresses this challenge by leveraging the synchronization order indicated by a schedule to more precisely model inter-thread data flow. Because control-flow specialization already computes what statements can be executed between every two consecutive synchronization, we can easily compute the happens-before relation between two statements across threads by analyzing the order of their preceding and succeeding synchronizations.

Our data-flow specialization algorithm consists of a series of constrainted-based analyses leveraging the STP [Ganesh and Dill, 2007] integer constraint solver. We collect constraints on both top-level variables and variables in memory locations. For variables in memory locations, we compute def-use chains with respect to the synchronization order in the schedule, which gives much more precise results than a stock def-use analysis. §4.2.2 will describe our data-flow specialization in more detail.

Schedule specialization has broad applications. For instance, we can build precise static verifiers (*e.g.*, to verify error freedom) because our verifiers need only verify a program with respect to the schedules enforced. Stock compiler optimizations automatically become more effective on a

specialized program because it has simpler control and data flow. Schedule specialization can also benefit "read-only" analyses which do not require enforcing schedules at runtime. For instance, we can build precise error detectors that check a program against a set of common schedules to detect errors more likely to occur, while drastically reducing false positive rates. We can perform precise, automated post-mortem analysis of a failure by analyzing only the schedule recorded in a system log to trim down possible causes.

We have implemented our framework within the LLVM compiler [LLVM, 2013] and evaluated it on 17 multithreaded programs, including 2 real programs such as a popular parallel compression utility `PBZip2` [PBZIP2, 2011] and 15 widely used parallel benchmarks from SPLASH2 [SPLASH2, 2007] and PARSEC [PARSEC, 2010]. Our results show that schedule specialization greatly improves precision. We have built a schedule-aware alias analyzer, a static race detector, and a path slicer. Our specialization framework, on average, reduced may aliases by 61.9%, false race reports by 69%, and path slices by 48.7%. The improved precision also helped detect 7 unknown bugs in well-checked [Zhang *et al.*, 2011; Zhang *et al.*, 2010; Park *et al.*, 2009a; Lu *et al.*, 2008; Gao *et al.*, 2011] programs.

This chapter is organized as follows. We first present the key algorithms using an example (§5.2.1) and detailed descriptions (§4.2). We then discuss implementation issues (§4.3), describe the analyses (§4.4) we build on top of our framework, and show the evaluation results (§4.5). We finally discuss related work (§4.6) and conclude (§4.7).

4.1 An Example and Algorithm Overview

This section illustrates how our specializer operates using an example based on two programs in our evaluation benchmarks: `aget`, a parallel `wget`-like utility; and `fft`, a parallel scientific benchmark.

Figure 4.1 shows the example program. As can be seen from the code, each thread accesses a disjoint partition of `results`. Suppose we want to build a precise alias analysis to compute this fact, so that we can for example avoid wrongly flagging accesses to `results` as races for a static race detector. Unfortunately, doing so requires solving a variety of difficult problems. For instance, since `p`, the number of threads, is determined at runtime, static analysis often has to approximate these dynamic threads as one or two abstract thread instances. It may thus collapse

```
1 : int results[MAX];
2 : struct {int first; int last;} ranges[MAX];
3 : int global_id = 0;
4 : int main(int argc, char *argv[]) {
5 :   int i;
6 :   int p = atoi(argv[1]), n = atoi(argv[2]);
7 :   for (i = 0; i < p; ++i) {
8 :     ranges[i].first = n * i / p;
9 :     ranges[i].last = n * (i + 1) / p;
10:   }
11:   for (i = 0; i < p; ++i)
12:     pthread_create(&child[i], 0, worker, 0);
13:   for (i = 0; i < p; ++i)
14:     pthread_join(child[i], 0);
15:   return 0;
16: }
17: void *worker(void *arg) {
18:   pthread_mutex_lock(&global_id_lock);
19:   int my_id = global_id++;
20:   pthread_mutex_unlock(&global_id_lock);
21:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
22:     results[i] = compute(i);
23:   return 0;
24: }
```

Figure 4.1: *An example showing how schedule specialization works.* Variable `n` and `p` are two inputs that specify the size of the global array `results` and the number of worker threads, respectively. The code first partitions `results` evenly by the number of worker threads `p` and saves the range of each thread to `ranges` (lines 7–10); it then starts `p` worker threads to process the partitions (lines 11–12). Each worker enters a critical section to set its instance of `my_id` and increment `global_id` (lines 18–20), and then computes and saves the results to its partition (lines 21–22).

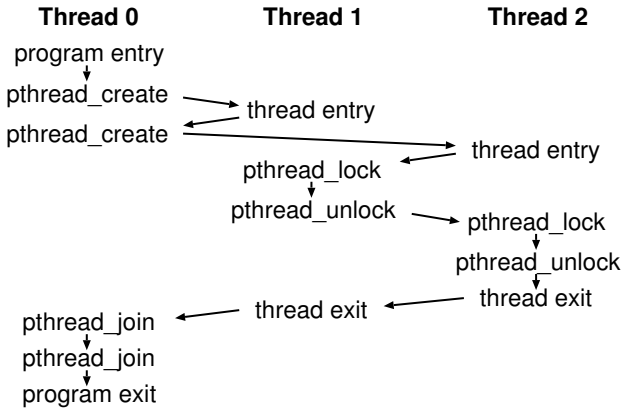


Figure 4.2: *A possible schedule of the example in Figure 4.1 when* `p` *is 2.*

distinct accesses to `results` from different threads as one access, computing imprecise alias results. Even if an analysis can (1) distinguish the accesses to `results` by different threads and (2) infer bounds on integers such as array indices using range analysis [Rugina and Rinard, 2000], it may still fail to compute that these accesses are disjoint. The reason is that the array partition each worker accesses depends on its instance of `my_id`, which further depends on the schedule (or the order in which worker threads enter the critical section at lines 18–20). In short, if a static analysis has to consider all possible schedules of this program, it would be very difficult to compute precise results.

Fortunately, these difficult problems are greatly simplified by schedule specialization. Suppose whenever `p` is 2, we always enforce the schedule shown in Figure 4.2. Since the number of threads is fixed, distinguishing them becomes easy. In addition, since the order in which threads enter the critical section at lines 18–20 is fixed, each worker thread always gets a fixed `my_id` and thus accesses a fixed partition of `results`.

To practically take advantage of these observations, our framework specializes a program with respect to a schedule. It does so in two steps. In the first step, it specializes the control flow of the program by "straightening" the program so that each synchronization in the schedule maps to a unique statement, and the synchronizations within each thread are control-equivalent. Specifically, it first constructs a super control flow graph (CFG) of the program, which includes function call and return edges. It then iterates through each pair of consecutive synchronizations (s_1, s_2) in a

```
1 : ... // same global declarations as in Figure 4.1
2 : int main(int argc, char *argv[]) {
3 :   int i;
4 :   int p = atoi(argv[1]), n = atoi(argv[2]);
5 :   for (i = 0; i < p; ++i) {
6 :     ranges[i].first = n * i / p;
7 :     ranges[i].last = n * (i + 1) / p;
8 :   }
9 :   i = 0; assume(i < p);
10:   pthread_create(&child[i], 0, worker_CLONE1, 0);
11:   ++i; assume(i < p);
12:   pthread_create(&child[i], 0, worker_CLONE2, 0);
13:   ++i; assume(i >= p);
14:   i = 0; assume(i < p);
15:   pthread_join(child[i], 0);
16:   ++i; assume(i < p);
17:   pthread_join(child[i], 0);
18:   ++i; assume(i >= p);
19:   return 0;
20: }
21: void *worker_CLONE1(void *arg) {
22:   pthread_mutex_lock(&global_id_lock);
23:   int my_id = global_id++;
24:   pthread_mutex_unlock(&global_id_lock);
25:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
26:     results[i] = compute(i);
27:   return 0;
28: }
29: void *worker_CLONE2(void *arg) {
30:   pthread_mutex_lock(&global_id_lock);
31:   int my_id = global_id++;
32:   pthread_mutex_unlock(&global_id_lock);
33:   for (int i = ranges[my_id].first; i < ranges[my_id].last; ++i)
34:     results[i] = compute(i);
35:   return 0;
36: }
```

Figure 4.3: *The resultant program after control-flow specialization.* The loops at lines 11–12 and lines 13–14 in Figure 4.1 are unrolled because they contain synchronizations in the schedule, while the other loops are not. Thread function `worker` is cloned twice, making it easy for an analysis to distinguish the two worker threads. The algorithm adds special `assume` calls to pass constraints on variables to the data-flow specialization step.

thread, and clones the statements between s_1 and s_2 in the CFG. Figure 4.3 shows the result after specializing control flow. Loops containing synchronizations such as `pthread_create` are unrolled, and thread functions are cloned, making it easy for an analysis to distinguish threads. Note that such cloning and unrolling are selectively done only to functions that may do synchronizations, so they would not explode the size of a specialized program.

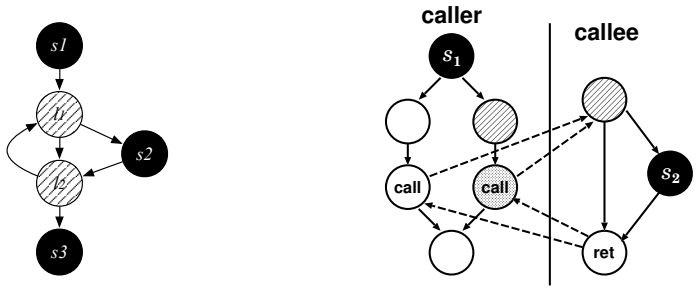
In the second step, our framework specializes the data flow through a series of analyses. For instance, it constructs an advanced def-use graph for variables and memory locations according to the schedule, and replaces variables with constant values when it can. It does these analyses by collecting constraints from the program and querying a constraint solver. For instance, from lines 9, 11, and 13 in Figure 4.3, it infers that `p` must be 2, so it replaces `p` with 2. Similarly, it computes a precise def-use chain for the accesses to `global_id`, and infers that each `my_id` instance has a constant value. Once variables are replaced with constants, we can apply techniques such as constant folding, dead code elimination, and loop unrolling to further specialize the program. Moreover, these stock techniques and our analyses can run iteratively, until the program cannot be specialized any more. Figure 4.4 shows the result of specializing data flow.

Benefits of schedule specialization. The specialized program in Figure 4.4 has much simpler control and data flow than the original program in Figure 4.1, enabling stock analyses to compute more precise results. For instance, range analysis can now compute thread-sensitive results because the `worker` function is cloned; it can also compute precise bounds for the loop index i in the `worker` clones because the indice to `ranges` are now constant, not `my_id` which can have a large range if no schedule is enforced.

Our framework guarantees that static analysis results on Figure 4.4 hold for the set of all traces with the schedule in Figure 4.2 enforced. This set is fairly large because this schedule can be enforced as long as the number of threads `p` is 2, regardless of the data size `n` or the contents of other input data. A small set of such schedules, including one for each number of threads, can practically cover all inputs because (1) most parallel programs we evaluate achieve peak performance when the number of worker threads is identical or close to the number of processor cores and (2) the number of cores a machine has is typically small (*e.g.*, smaller than 100). This example illustrates the best case of our approach: by analyzing only a small set of schedules, we enjoy both precision and soundness for practically all inputs.

```
... // same global declarations as in Figure 4.1
int main(int argc, char *argv[]) {
  int p = atoi(argv[1]), n = atoi(argv[2]);
  ranges[0].first = 0;
  ranges[0].last = n / 2;
  ranges[1].first = n / 2;
  ranges[1].last = n;
  pthread_create(&child[0], 0, worker_CLONE1, 0);
  pthread_create(&child[1], 0, worker_CLONE2, 0);
  pthread_join(child[0], 0);
  pthread_join(child[1], 0);
  return 0;
}
void *worker_CLONE1(void *arg) {
  pthread_mutex_lock(&global_id_lock);
  global_id = 1;
  pthread_mutex_unlock(&global_id_lock);
  for (int i = 0; i < ranges[0].last; ++i)
    results[i] = compute(i);
  return 0;
}
void *worker_CLONE2(void *arg) {
  pthread_mutex_lock(&global_id_lock);
  global_id = 2;
  pthread_mutex_unlock(&global_id_lock);
  for (int i = ranges[1].first; i < ranges[1].last; ++i)
    results[i] = compute(i);
  return 0;
}
```

Figure 4.4: *The resultant program after data-flow specialization.* All uses of variable `p` and `my_id` are replaced with constants, the loop at lines 5–8 in Figure 4.3 is unrolled, and some dead code is removed.



(a) *Intra-procedural ambiguity* (b) *Inter-procedural ambiguity*

Figure 4.5: *Two types of ambiguity.* The black nodes are synchronizations. The hatched nodes are the statements between s_1 and s_2 computed by the reachability analysis. The gray "call" node is a derived synchronization marked to resolve inter-procedural ambiguity.

4.2 Algorithms

This section first describes our algorithms to specialize the control flow (§4.2.1) and data flow (§4.2.2) of a program toward a schedule, and then discusses the soundness of these specialization algorithms (§4.2.3).

4.2.1 Specializing Control Flow

To ease the description of our algorithms, we define a *segment*, denoted by G, as the maximal portion of the CFG between two synchronization statements l_1 and l_2 that is free of other synchronization statements. A segment between l_1 and l_2 includes any statement such that (1) l_1 can reach this statement in the CFG without reaching any other synchronization statements and (2) this statement can reach l_2 without reaching any other synchronization statements. We denote the set of statements in G by $G.L$, and the set of control-flow edges in G by $G.E$.

To specialize the control flow, we clone the segment between every two consecutive synchronizations within a thread, and then join all cloned segments together to form the specialized program. Since we recognize entries and exits of thread functions as synchronizations, every instruction is included in at least one segment. One difficulty to find all statements between two consecutive synchronizations is *ambiguity*: there may be multiple ways from one synchronization to another. For instance, Figure 4.5a shows an ambiguous case caused by multiple paths in the CFG. The loop

body shown has two paths, only one of which contains synchronization s_2. This loop has to be transformed so that s_1 and s_2 become control-equivalent. Our algorithm to specialize control flow automatically handles this case, described later in this subsection.

Figure 4.5b shows another ambiguous case caused by multiple paths in the call graph. From s_1 to s_2, we can go through either call. To resolve inter-procedural ambiguity, we instrument calls to functions that may transitively do synchronizations, and include these *derived synchronizations* in the schedule as well. The result is that we can compute exactly one call stack for each synchronization in a schedule. Our evaluation shows that the number of derived synchronizations is small, and they incur negligible overhead (§4.5.2).

Algorithm 1 shows how we specialize the control flow of a program. We first explain the helper function `SpecializeControlFlowThread`, which takes the original program P, a schedule S, and a thread t, and outputs a subprogram P'_t specialized according to t's synchronizations in S. For every two consecutive synchronizations of t, this function does a forward and a backward depth-first search to find the segment between the two synchronizations. It then clones the segment, copying and renaming functions as necessary, and appends the segment clone to P'_t.

Function `DFS` traverses the CFG from statement l_1 to l_2 and saves the visited portion of the CFG to output parameter G_{out}. It backtracks whenever it meets a synchronization. During the traversal, it maintains the current call stack in cs so that it can compute where to return (l_{ret}). When it reaches l_2, it saves the call stack to output parameter cs_{out}, which `SpecializeControlFlowThread` will pass to `BackwardDFS` (in the same iteration) and `DFS` (in the next iteration). If there are multiple paths from l_1 to l_2, `DFS` may reach l_2 multiple times in one traversal, but each time the call stack must always be identical because call-stack ambiguity is resolved.

The results of `DFS` may include statements and CFG edges that cannot reach $s_{i+1}.label$. `BackwardDFS` prunes these statements and edges by traversing G, not the full CFG, backward from $s_{i+1}.label$ to $s_i.label$. It is similar to `DFS` except it is backward and need not output a call stack, so we omit it from Algorithm 1.

We now explain the main function of this algorithm, `SpecializeControlFlow`. It first invokes `SpecializeControlFlowThread` to compute a specialized program based on the synchronizations of each thread. It then replaces the thread function in each `pthread_create` callsite with the cloned thread function. It finally merges all the specialized programs together.

Algorithm 1: Control-Flow Specialization

Input : the original program P and a schedule S
Output: the specialized program
SpecializeControlFlow(P, S)
  foreach thread t in S **do**
    $P'_t \leftarrow$ SpecializeControlFlowThread(P, S, t)
  foreach P'_t **do**
    foreach statement l in P'_t **do**
      if l is a `pthread_create` **then**
        $ct \leftarrow$ child thread created by l
        set thread function of l to P'_{ct}
  return $\cup_t P'_t$
SpecializeControlFlowThread(P, S, t)
  $P'_t \leftarrow \emptyset$ // the specialized subprogram for thread t
  $cs \leftarrow \emptyset$ // the call stack of the current synchronization
  $S_t \leftarrow$ the sub-schedule of S for thread t
  for (s_i, s_{i+1}) in S_t **do**
    $G \leftarrow (\{s_i.label\}, \emptyset)$ // segment between s_i & s_{i+1}
    // G and the second cs below are output parameters.
    DFS(cs, $s_i.label$, $s_{i+1}.label$, cs, G)
    BackwardDFS(cs, $s_{i+1}.label$, $s_i.label$, G)
    $P'_t \leftarrow P'_t \cup$ Clone(G)
  return P'_t
DFS(cs, l_1, l_2, cs_{out}, G_{out}) // l_1 and l_2 may be the same
  if l_1 is a derived synchronization **then**
    $l \leftarrow$ entry of l_1's callee function // all possible callees of a function pointer
    TryDFS($cs + l_1$, l_1, l, l_2, cs_{out}, G_{out})
  else if l_1 is a return statement **then**
    $l_{ret} \leftarrow$ the top of cs
    $l \leftarrow l_{ret}$'s intra-procedural successor
    TryDFS(cs - l_{ret}, l_1, l, l_2, cs_{out}, G_{out})
  else
    foreach intra-procedural successor l of l_1 **do**
      TryDFS(cs, l_1, l, l_2, cs_{out}, G_{out})
TryDFS(cs, l_1, l, l_2, cs_{out}, G_{out})
  $G_{out}.E \leftarrow G_{out}.E \cup \{(l_1, l)\}$
  if $l = l_2$ **then**
    $cs_{out} \leftarrow cs$
  if $l \notin G_{out}.L$ **then**
    $G_{out}.L \leftarrow G_{out}.L \cup \{l\}$
    if l is not a synchronization **then**
      DFS(cs, l, l_2, cs_{out}, G_{out})

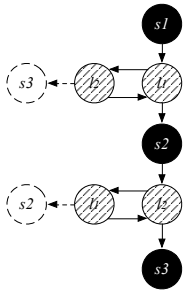


Figure 4.6: *The specialized CFG of Figure 4.5a with schedule* $\langle t, s_1\rangle\langle t, s_2\rangle\langle t, s_3\rangle$. *The dashed arrows and shapes represent removed branches and code.*

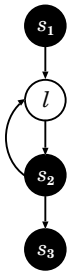


Figure 4.7: *Example illustrating the need to traverse CFG edges.*

Discussion. We note three subtleties of Algorithm 1. First, it automatically handles the ambiguity in Figure 4.5a. Suppose the schedule is $\langle t, s_1\rangle\langle t, s_2\rangle\langle t, s_3\rangle$. We do not know the loop bound because one path in the loop body calls s_2 and the other does not. However, our algorithm can still specialize the CFG into Figure 4.6. Since control-flow specialization removes dead code according to a schedule, we use dashed arrows and shapes to denote removed branches and code. This feature is critical in some cases. For example, if s_2 is `pthread_create`, this feature clones the thread functions, providing thread-sensitivity.

Second, `DFS` traverses into the body of a function only when the call to this function is a derived synchronization, *i.e.*, this function may transitively do synchronization. Thus, the G_{out} it computes includes only CFG portions from such functions, and is not a full segment. The effect is that only these CFG portions are cloned, possibly multiple times, and each clone is *unique* to a segment in the resultant program; other functions are copied verbatim to the resultant program. By doing so, we capture the information from the schedule without exploding the size of a specialized program.

Lastly, to compute a segment, we cannot simply compute only the statements in the segment and copy all edges from the original CFG. To illustrate, consider Figure 4.7. Suppose the schedule is $s_1 s_2 s_3$. The segment between s_1 and s_2 should include statements s_1, s_2, and l, but exclude the CFG edge from s_2 to l.

Instruction type	**LLVM instruction**	**STP constraint**	**Note**
binary	$a = b$ op c	$a = b$ op c	"op" is a binary operator
integer comparison	a = icmp pred, b, c	$a = (b$ pred $c)$	"pred" is an integer comparison predicate such as $<$ and $\neq$
pointer arithmetic	a = gep b, i_1, ..., i_k	$a = b + i_1 \times$ sizeof(*b) + the offset of i_k-th field of i_{k-1}-th field of ... of i_2-th field of b	"gep" calculates the location of a specific field in an array/structure
select instruction	a = select b, v_1, v_2	$b = 0 \rightarrow a = v_1$ $b = 1 \rightarrow a = v_2$	If $b = 0$, $a = v_1$; otherwise $a = v_2$
Φ instruction	$a = \Phi(v_1, \ldots, v_k)$	$a = v_1$ or ... or $a = v_k$	a may have any incoming value
call and return	$a = \text{func}(b_1, \ldots, b_k)$ func$(f_1, \ldots, f_k)$ {return r;}	$b_1 = f_1, \ldots, b_k = f_k$ $a = r$	Capture the equalities between the actual and formal parameters/return values

Table 4.1: Collecting constraints from LLVM instructions on virtual registers.

4.2.2 Specializing Data Flow

Given a control-flow-specialized program, we specialize its data flow using a series of constraint-based analyses leveraging the STP [Ganesh and Dill, 2007] integer constraint solver. We collect constraints on two types of program constructs: the LLVM virtual registers and memory locations. LLVM virtual registers are already in static-single-assignment (SSA) form, so collecting constraints on them is relatively straightforward. Table 4.1 lists how we collect constraints for some LLVM instructions; the remaining instructions are similar and omitted for space. We handle loops by exploiting LLVM's loop structure: most of the loops in our evaluated programs are canonicalized by LLVM so that we can easily collect range constraints on the loop indices.

Our algorithm to collect constraints on memory locations mimics classic def-use analysis. We treat an LLVM load instruction as a "use" and an LLVM *load or store* instruction as a "def". Treating loads as defs shortens the def-use chains and reduces analysis time. Our algorithm collects two forms of constraints on memory locations: (1) the value loaded by an LLVM load instruction is the same as the value stored by a store instruction and (2) the value loaded by a load is the same as the value loaded by another load.

Collecting precise def-use constraints on memory locations is challenging for multithreaded programs. Fortunately, schedule specialization enables two key refinements over classic def-use analysis so that our algorithm can collect more precise constraints. First, we compute def-use chains spanning across threads because multiple threads may indeed access the same shared memory location. Without schedule specialization, it would be hopeless to track precise inter-thread def-use chains because the defs and uses may pair up in numerous ways depending on the schedules.

Second, we compute *must*-def-use, instead of may-def-use, results. Our insight is that by fixing the schedule, we often fix the def-use chains of key shared data, such as the `global_id` in Figure 4.1. It is thus most cost-effective to focus on these must-def-use chains because (1) they essentially capture the information (and hence the precision; see results in §4.5.1) from the schedule and (2) there are typically much fewer must-def-use constraints than may-def-use constraints, so the constraint-solving cost is low.

To practically implement these refinements, we compute def-use chains only on the instructions unique to each segment. In addition, we look for defs of a use only from its *inter-thread dominators* which always execute before the use once the schedule is enforced. These dominators are dominators

on the CFG augmented with edges representing the total order of synchronizations in a schedule. We explain our algorithm as well as these refinements in the next few paragraphs.

Algorithm 2 shows the pseudo code to collect def-use constraints for memory locations given a schedule. It operates on a control-flow specialized program with each synchronization in the schedule mapped to a unique instruction. At the top level, the algorithm is similar to classic def-use analysis. Given a use, it computes all defs and adds an equality constraints between the use and each def. It references several symbols defined on the instructions and segments (§4.2.1) in this program:

- $value(l)$: the value loaded if l is a load instruction, or the value stored if l is a store;
- $loc(l)$: the memory location accessed by instruction l;
- $segment(l)$: the unique (explained in the next paragraph) segment containing instruction l, or $None$;
- $thread(G)$: the thread containing segment G;
- $begin(G)$: the synchronization at the beginning of segment G;
- $end(G)$: the synchronization at the end of segment G;
- $reach(l_1, l_2)$: the set of instructions on all *simple paths* from l_1 to l_2 in the CFG. A simple path has no repeated instructions.

Each segment has some unique instructions cloned by Algorithm 1; $segment(l)$ returns the containing segment for these instructions, or $None$ otherwise. We define a *partial order* over these instructions as follows: l_1 *happens-before* l_2, or $l_1 \prec l_2$, if (1) both l_1 and l_2 are synchronizations, and l_1 comes earlier than l_2 in the schedule; (2) $segment(l_1) = segment(l_2)$, and l_1 comes earlier in the segment; or (3) $\exists l_3$, s.t. $l_1 \prec l_3$ and $l_3 \prec l_2$. We say $G_1 \prec G_2$ if $end(G_1) \prec begin(G_2)$. Two instructions (segments) are concurrent if there is no happens-before between them.

`CaptureConstraints` takes a use u, calls `PotentialDefs` to get a set of potential defs, and, for each *live* def not killed, adds an equality constraint between the value used and the live def.

`PotentialDefs` illustrates how we implement the refinements enabled by schedule specialization. It processes a use u only if $segment(u)$ is not $None$, *i.e.*, u is unique to a segment. It searches for defs only in the instructions unique to each segment. To account for shared data access, it searches each thread t_d for a def of u. If t_d is the thread containing u, we search backwards from u for the latest dominator in CFG that accesses the same location. Otherwise, we search backwards

Algorithm 2: Capture Constraints on Memory Locations

```
CaptureConstraints(u)
    foreach d ∈ PotentialDefs(u) do
        if not MayBeKilled(d, u) then
            AddConstraint("value(u) = value(d)")
PotentialDefs(u)
    defs ← ∅
    G_u ← segment(u)
    p ← loc(u)
    if G_u ≠ None then
        foreach thread t_d do
            if thread(G_u) = t_d then
                defs ← defs ∪ LatestDef(u, p)
            else
                G_d ← latest segment of t_d s.t. G_d ≺ G_u
                defs ← defs ∪ LatestDef(end(G_d), p)
    return defs
LatestDef(l, p)                          // returns the latest intra-thread definition
    d ← l
    repeat
        d ← ImmediateDominator(d);
    until d = None or MustAlias(p, loc(d))
    if d ≠ None then return {d}
    else return ∅
MayBeKilled(d, u)
    G_d ← segment(d)
    G_u ← segment(u)
    if G_d = G_u then
        return MayStore(reach(d, u), u)
    foreach segment G s.t. begin(G) ≺ end(G_u) and begin(G_d) ≺ end(G) and G ≠ G_d and
    G ≠ G_u do
        if MayStore(G, u) then return true
    return MayStore(reach(d, end(G_d)), u) or MayStore(reach(begin(G_u), u), u)
MayStore(L, u)
    foreach store l ∈ L do
        if MayAlias(loc(l), loc(u)) then return true
    return false
```

from the latest synchronization s in t_d that happens-before u, and locate the latest dominator of s in CFG that accesses the same location. This dominator of s is essentially an inter-thread dominator of u. The result of `PotentialDefs` contains at most one def from each thread. In addition, it contains at most one store and possibly many loads (because we treat loads as defs).

Function `MayBeKilled` checks whether def d is killed by any instruction that may store to $loc(u)$ and execute after d and before u. It considers all instructions, not just the unique ones, for soundness. If d and u are in the same segment, `MayBeKilled` simply checks the instructions between d and u in the CFG. Otherwise, it checks instructions in any segment that may (1) begin no later than the end of u's segment G_u and (2) end no earlier than the begin of d's segment G_d. Note that these segments include not only the ones concurrent to G_u or G_d, but also any segment G s.t. $G_d \prec G \prec G_u$.

The above algorithm to collect constraints on memory locations needs alias analysis to determine whether two pointers must or may alias. We answer these queries again using STP. The question "whether v_1 and v_2 may alias" is rephrased as "whether $v_1 = v_2$ is satisfiable," and "whether v_1 and v_2 must alias" is rephrased as "whether $v_1 = v_2$ is provable." Since constraint solving may take time, we also ported `bddbddb` [Whaley and Lam, 2004], a faster but less precise alias analysis to our framework, by writing an LLVM front end to collect program facts into the format `bddbddb` expects. We always query `bddbddb` first before STP.

Once constraints are captured, we perform a series of analyses based on these constraints. One analysis infers which LLVM virtual register in the specialized program has constant values (recall that these registers are in SSA form). It first queries STP for an assignment of value v_j to each variable a_j. For each a_j, it then adds $a_j = v_j$ to the current constraints, and checks whether STP can prove the new constraints. If so, it replaces a_j with v_j, such as replacing variable `p` in Figure 4.1 with constant 2. Once variables are replaced with constants, we perform stock LLVM analyses and transformations, such as constant folding, control flow simplification, and loop unrolling, which automatically gain precision when performed on a specialized program. These analyses may further simplify a program, so we iteratively run data-flow specialization and these stock analyses until the specialized program converges.

4.2.3 Soundness of Specialization Algorithms

As mentioned in §2.2.2, our specialization framework should guarantee that $run(P_S) \supseteq run_S(P)$ so that static analysis results on P_S hold for all traces in $run_S(P)$. In this subsection, we prove that both control-flow and data-flow specialization preserve all traces in $run_S(P)$, *i.e.*, every execution of P that is compatible with S is a valid execution in both control-flow and data-flow specialized programs.

4.2.3.1 Soundness of Control-Flow Specialization

To prove the soundness of control-flow specialization, we show that our control-flow specialization algorithm maps each trace in the original program to a trace in the control-flow specialized program, and that this new trace satisfies both order constraints and control-flow integrity.

Our algorithm maps each dynamic statement instance to a unique instance in the new trace. Given an instance i between two synchronizations s_j and s_{j+1}, the corresponding new instance is constructed as follows:

- Function `SpecializeControlFlow` creates a dedicated thread function for each thread in the original trace, and links it off the `pthread_create` that spawns the thread function. Therefore, $i.thread$ is naturally mapped to the thread that invokes the cloned thread function.
- Since the original trace satisfies control-flow integrity and $i.label$ is on at least one path from s_j to s_{j+1}, the segment between s_j and s_{j+1} contains $i.label$. Therefore, `SpecializeControFlowThread` clones all the statements in the segment including $i.label$, and maps $i.label$ to an instance in the specialized program.

To construct the new trace, we concatenate the new instances in the same order as their originals, ensuring that the new trace still satisfies the order constraints (§2.1). We denote the cloned instance of i by $clone(i)$. Because we preserve the order between instances, for each instance pair i_j and i_k where $j < k$, $clone(i_j)$ still appears before $clone(i_k)$ in the new trace.

The new trace satisfies control-flow integrity because our algorithm clones all the control-flow edges in each segment. Given two consecutive instances i_1 and i_2 from the same thread, edge $(i_1.label, i_2.label)$ is guaranteed to be a CFG edge per control-flow integrity. Therefore, when cloning the segment that contains i_1 and i_2, function `SpecializeControlFlowThread` clones edge

$(i_1.label, i_2.label)$ to $(clone(i_1).label, clone(i_2).label)$. The existence of these cloned edges guarantees that the new trace follows a CFG path in the specialized program.

4.2.3.2 Soundness of Data-Flow Specialization

As described in §4.2.2, our data-flow specialization consists of a schedule-aware data flow analysis and a set of stock transformations to further simplify the specialized program. To prove the soundness of data-flow specialization, we need to show that (1) the results of our schedule-aware data flow analysis are sound with respect to the analyzed schedule, and (2) the simplification preserves all executions that are compatible with the analyzed schedule.

Our schedule-aware data flow analysis constructs a def-use chain from d to u only when it can prove the value loaded/stored by d is definitely loaded by u in all executions of the control-flow specialized program that are compatible with the schedule. This property is guaranteed by function `PotentialDefs` and `MayBeKilled`. Given use u, `PotentialDefs` treats memory access d as a potential def only if $d \prec u$ and $loc(d)$ and $loc(u)$ must alias. Function `MayBeKilled` further screens the potential defs by checking whether another statement that may overwrite the same memory location can be executed between d and u.

When performing stock transformations to further simplify a program, we need to guarantee that the simplification preserves all executions that are compatible with the analyzed schedule. To meet this requirement, we cannot run arbitrary optimization passes in this step. For instance, dead store elimination may lose important traces for the purpose of data race detection. Dead store elimination typically assumes data race freedom. If the value stored by an instruction is not later used by that thread, dead store elimination may eliminate that store even though this store may race with a concurrent memory access. As a result, our race detector may falsely conclude that the program is free of data races.

By default, we only run a selective set of transformations that do not remove memory accesses that may be executed, such as constant folding, control flow simplification, and loop unrolling. Running these transformations is sound for the purposes of our three applications (§4.4): alias analysis, race detection, and path slicing. Users may customize the transformations they want to run. For instance, they can refine stock compiler optimizations to remove the assumption of data race freedom, and use them to simplify the program for detecting data races. Also, if users want

to use schedule specialization for better optimization, they may assume data race freedom and run more aggressive transformations, such as SROA (Scalar Replacement of Aggregates) and dead code elimination.

4.3 Implementation

We implement our framework within LLVM [LLVM, 2013]. The specialization algorithms described in the previous section are implemented as an LLVM pass that specializes the LLVM bitcode representation of an original program into a new bitcode program. These passes are of 11,754 lines of C/C++ code, with 2,586 lines for control-flow specialization and 9,168 lines for data-flow specialization. The remainder of this section discusses several implementation issues.

4.3.1 Constructing Call Graph

For clarity, Algorithm 1 assumes a simple call graph where each function call has a unique callee and unique return address. However, the programs we analyze do use function pointers and exceptions, invalidating this assumption. To resolve a call to a function pointer, we query our alias analysis and call *TryDFS* on each possible callee. To handle exceptions, we pair up the LLVM instruction that unwinds the stack (`UnwindInst`) with the call instruction that sets an exception handler (`InvokeInst`).

4.3.2 Optimizing Constraint Solving

Data-flow specialization (§4.2.2) frequently queries STP, and these queries can be quite expensive. We thus create four optimizations to speed up constraint solving. First, when identifying which variables are constants, we avoid querying STP for each variable. Specifically, when we query STP to prove that a variable is constant, we remember the value assignment STP returns even if STP cannot prove the query. If a variable has two different values in these assignments, then this variable cannot be constant, so we do not need to query STP whether this variable is constant. This optimization alone speeds up our framework by over 10 times.

Second, we cache extensively. Recall that data-flow specialization runs Algorithm 2 and other analyses iteratively. Within each iteration, we cache all alias results to avoid redundantly querying

STP. Across iterations, we cache "must" and "must-not" alias results and the def-use results because each iteration only adds more constraints and does not invalidate these precise results.

Third, we use a union-find algorithm to speed up equality constraints solving. The def-use constraints we collect are all equality constraints. When the number of STP variables involved in these constraints increases, the STP queries tend to run slower. To reduce the number of STP variables, we put program variables that must be equal in one equivalent class represented by a union-find data structure, and assign only one STP variable to each class.

Lastly, we parallelize constraint solving. The STP queries issued by our framework are mostly independent. Specifically, each run of Algorithm 2 on a use is completely independent of another. In addition, checking whether one variable is constant is often independent of checking another. We have built a parallel version of STP that speeds up our framework by roughly 3x when running on a quad-core machine.

4.3.3 Manual Annotations

Our framework supports three types of annotations to increase precision and reduce analysis time. First, users can provide a blacklist of synchronization statements to customize which synchronizations go into the schedules (§2.2.3.3). Second, users can write regular assertions to enable our framework to collect more precise constraints. For instance, they can write "`assert(lower_bound <= index && index <= upper_bound)`" to inform our framework the range constraints on `index`. Third, users can provide function summaries to reduce analysis time. Some functions in the evaluated programs are quite complex and contain many load and store instructions, causing our framework to collect a large set of constraints. However, these functions often have simple shared memory access patterns which users can easily summarize to speed up constraint solving. These summaries can be quite coarse-grained, such as "function `foo` writes an unconstrained value to variable `x`." Users write summaries by writing fake functions, which our framework analyzes instead of the original functions. By supporting annotations in forms of assertions and fake functions, we avoid the need to support an annotation language.

Of the 17 programs evaluated, we annotated only 4 programs. We blacklisted the lock operations in the custom memory allocator in `cholesky`. We annotated two loops in `raytrace` with assertions because they are not in LLVM-canonical form. (Currently our framework handles only canonical

loops.) We summarized five functions: `image_segment` and `image_extract_helper` in `ferret`, `TraverseBVH_with_StandardMesh` and `TraverseBVH_with_DirectMesh` in `raytrace`, and `LogLikelihood` in `bodytrack`. These summaries range from 8 to 18 lines.

4.4 Applications

To demonstrate the precision of our framework, we build three powerful analyses. The first is a precise schedule-aware alias analyzer, built on top of our def-use analysis (§4.2.2). We chose to build an alias analyzer because alias analysis is crucial to many program analyses and optimizations. Our analyzer provides a standard MayAlias(p, q) query interface, making it effortless to switch existing analyses to our our alias analyzer. Under the hood, MayAlias first queries a coarse-grained, existing alias analysis; if this coarse-grained analysis returns true, MayAlias then queries our def-use analysis for precise answers. The existing alias analysis we used is the C version[2] of `bddbddb` ported to LLVM.

The second tool is a precise static race detector that detects races that may occur only when a schedule is enforced. The detection logic appears identical to classic static race detectors (such as Chord [Naik *et al.*, 2006] and RacerX [Engler and Ashcraft, 2003]) based on the lockset algorithm: two memory accesses are a race if (1) they may alias, (2) at least one of the accesses is a store, and (3) they are concurrent. There are two key differences. First, ours uses schedule-aware alias analysis. Second, ours detects concurrent accesses with respect to a total synchronization order (§4.2.2), more stringent than the execution order constraints dictated by the synchronizations. These two differences enable our detector to emit no or extremely few false positives (§4.5.1). Multiple races flagged on a specialized program may map to the same race in the original program because control-flow specialization clones statements. Our race detector emits only one report in this case.

The third tool is a thread-sensitive path slicer. Intuitively, given a program path, *path slicing* removes from the path the statements irrelevant to reach a given target statement [Jhala and

[2]The alias analysis we use makes several assumptions about the C programs it analyzes for soundness; these assumptions are described in [Avots *et al.*, 2005].

Majumdar, 2005]. The algorithm for doing so tracks control- and data-dependencies between statements, and removes statements that the target does not control- or data-depend upon. As previous work describes, path slicing can be applied to many problems, such as post-mortem analysis [Jhala and Majumdar, 2005], counter-example generation [Jhala and Majumdar, 2005], and malicious-input filtering [Costa *et al.*, 2007]. Our path slicer improves upon existing path slicing work [Jhala and Majumdar, 2005] by tracking dependencies across threads. Dependencies may arise across threads due to data or control flow. For example, if thread t_0 stores to pointer p, and t_1 then loads from q which may alias p, then the load in t_1 data-depends on the store in t_0. Similarly, if different branches of a branch statement in t_0 may cause t_1 to load different values from a shared variable, then the load in t_1 control-depends on the branch statement in t_0. Our path slicer correctly tracks these dependencies by leveraging our precise alias analyzer. Another feature of our path slicer is that it can slice toward multiple targets.

4.5 Evaluation

We evaluated our schedule specialization framework on a diverse set of 17 programs, including `PBZip2` 1.1.5, a parallel compression utility; `aget` 0.4.1, a parallel `wget`-like utility; parallel implementations of 15 computation-intensive algorithms, 8 in SPLASH2 and 7 in PARSEC. We excluded 4 programs from SPLASH2 and 6 from PARSEC because we cannot compile them down to LLVM bitcode code, they use OpenMP [Board, 2008] instead of Pthreads [The Open Group and the IEEE, 2008], data-flow specialization runs out of time on them, the compiled code does not run correctly in 64-bit environment, or our current prototype cannot handle them due to an implementation bug. All of the programs were widely used in previous studies [Zhang *et al.*, 2011; Zhang *et al.*, 2010; Park *et al.*, 2009a; Lu *et al.*, 2008; Gao *et al.*, 2011].

We generated schedules for the programs evaluated using the following workloads: for SPLASH2 programs, we used the default arguments except that we ran them with four threads. These programs finish within 100 ms. For `PBZip2`, we compressed a 10 MB randomly generated text file. For `aget`, we downloaded a file of 1 MB from the internet. The machine is a 2.8GHz Intel 12-core machine with 64 GB memory running 64-bit Linux 2.6.38. Running a program on one workload and one setting may still generate multiple schedules. For the precision evaluation, we pick the one

that appears most frequently. Unless otherwise specified, we ran all programs using four threads. We compiled all programs using Clang and LLVM 2.9 with the default optimization level (often `-O2` or `-O3`) of each program.

In the remainder of this section, we first focus on two evaluation questions: (1) how much more precise the analyses become with schedule specialization (§4.5.1); and (2) what the overhead of our framework is (§4.5.2). We then present the bugs we detected with the precision provided by schedule specialization (§4.5.3).

4.5.1 Precision

We measured how our framework can improve the precision on the three analyses we built, the alias analyzer, the race detector, and the path slicer (§4.4). Our methodology is to apply these analyses on an original program, the control-flow specialized program, and the data-flow specialized program, and then compare the results.

Alias analysis precision. We quantified alias analysis precision by measuring *alias percentage*, the percentage of alias queries that return "may", instead of "must" and "must-not", responses because the latter two responses are precise. We selected the two pointers in each query from different threads to stress-test our alias analyzer because these pointers may appear to point to the same global array or the "same" thread-private heap or stack location, but are actually not aliases because most programs we evaluated access distinct memory locations in different threads. The total number of such queries can be large for programs that do many memory accesses, so we sampled some percentage of the queries to bound the total query time. The sampling ratio for `aget`, `PBZip2` and `fft` is $\frac{1}{50}$; for `water-spatial`, $\frac{1}{5000}$; and for `barnes`, `water-nsquared`, and `ocean`, $\frac{1}{50000}$. For all other programs, we processed all queries.

Figure 4.8 shows the alias precision results. For 12 programs (`aget`, `PBZip2`, `fft`, `lu-contig`, `radix`, `blackscholes`, `swaptions`, `streamcluster`, `canneal`, `bodytrack`, `ferret`, and `raytrace`), control-flow and data-flow specialization combined greatly improved precision. For instance, they reduced the alias percentage down to below 0.1% for `aget`. For 7 (`PBZip2`, `fft`, `swaptions`, `streamcluster`, `canneal`, `bodytrack`, and `raytrace`) of these 12 programs, control-flow specialization improved the alias analysis precision significantly. For instance, it reduced the alias percentage of `PBZip2` from 28.04% to 8.30%, a 70.4% reduction. The reason is that these programs allocate a

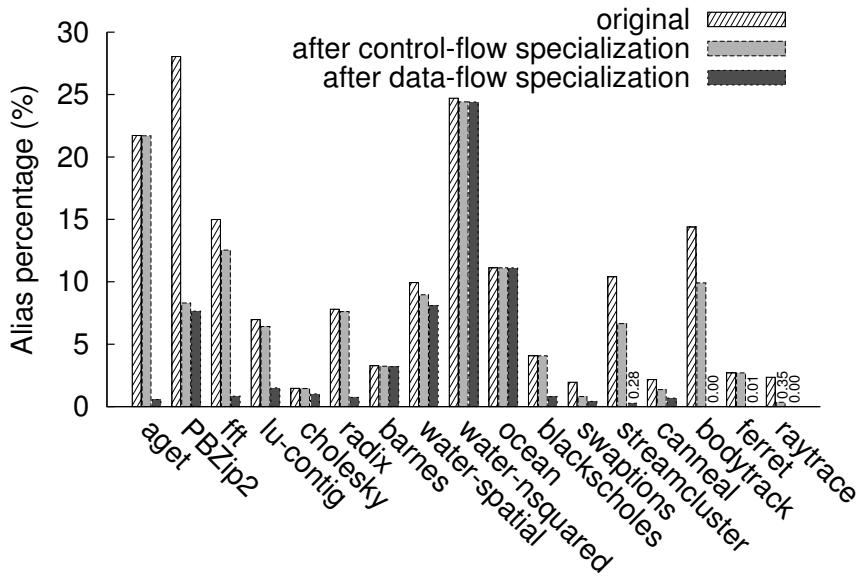


Figure 4.8: *Precision of the schedule-aware alias analysis.* Y axis represents the percentage of alias queries that return "may" responses. Lower bars mean more precise results. Each cluster in the figure corresponds to the results from one program. The three columns in a cluster represent the alias percentage when applying our analysis on the original program, the control-flow specialized program, and the data-flow specialized program.

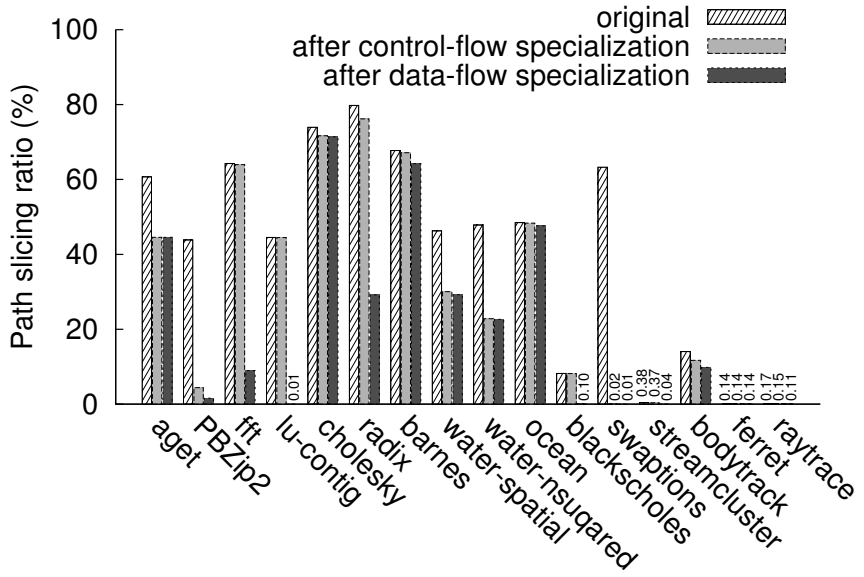


Figure 4.9: *Path slicing ratio on the original program, after control-flow specialization, and after data-flow specialization.*

fair amount of thread-private heap or stack data. Since the control-flow specialization clones thread functions and thus automatically provides thread sensitivity, our alias analyzer distinguishes accesses from different threads, achieving high precision. Although control-flow specialization alone did not significantly improve precision for the other 6 programs (`aget`, `PBZip2`, `lu-contig`, `radix`, `blackscholes`, and `ferret`) of the 12 programs, it created opportunities for data-flow specialization to improve precision. For the remaining 5 programs (`cholesky`, `barnes`, `water-spatial`, `water-nsquared`, and `ocean`), the reduction was not significant. This small reduction could be caused by the imprecision in our analyzer (*e.g.*, it is not path-sensitive), or real bugs in the original programs (§4.5.3). The mean reduction over all programs is 61.9%.

Race detection precision. We report the precision of our race detector here, and describe the detected bugs in §4.5.3. To evaluate its precision, we compared the number of false positives it emitted to a baseline race detector we built. This baseline detector uses the same definition of concurrent accesses (§4.4), except that it queries our `bddbddb` port instead of our alias analysis because our precise alias results can only be computed assuming a given schedule will be enforced. Since the total number of concurrent memory access pairs may be large, we used sampling for some

benchmarks. The sampling ratio was $\frac{1}{50}$ for `PBZip2`, `fft`, `barnes`, `water-spatial`, and `ocean`; and $\frac{1}{5000}$ for `water-nsquared`. For all other programs, we checked all concurrent access pairs.

We counted the number of false positives for our detector, denoted by FP_{ours}, by inspecting its reports. Since the number of reports for `barnes`, `water-spatial`, `water-nsquared`, and `ocean` is large, we inspected a random selection of 20 reports, and found that they were all false positives. We thus conservatively treated all reports from these programs as false positives. We counted the number of false positives for the baseline detector, denoted by $FP_{baseline}$, by computing $R_{baseline} - R_{ours} + FP_{ours}$ where R is the number of reports. This formula works because our detector is more precise than the baseline detector, and a report flagged only by the baseline detector must be a false positive.

Table 4.2 shows the false positive comparison results for all 17 programs. For 10 of them, the precision of framework enabled our detector to reduce the number of false positives to 0. The reduction for `cholesky` and `radix` is also large. For `lu-contig`, `barnes`, `water-nsquared`, `water-spatial`, and `ocean`, our race detector reported slightly fewer races than the baseline; some of these results are expected based on our alias precision results. The mean reduction over all programs is 69.0%.

Table 4.2 also shows the number of true but benign races detected. (We present the harmful races in §4.5.3.) Our detector found 1 race on variable `bwritten` in `aget` and 32 races on variable `AllDone`, `NumBlocks`, and `OutputBuffer` in `PBZip2`. Without the precision of our framework, users may have to inspect hundreds of reports before finding the true races.

Path slicer precision. To quantify the precision of our path slicer, we measured *slicing ratio*, the percentage of statements that remain in an execution trace. Figure 4.9 compares the precision of path slicing on the original programs and the specialized programs.[3] Control-flow specialization largely reduced the slicing ratio for `PBZip2`, `aget`, `swaptions`, `water-spatial`, and `water-nsquared`. For instance, it reduced the slicing ratio for `PBZip2` from 43.84% to 4.40%, a 89.97% reduction. Data-flow specialization further reduced the ratio for `PBZip2`, `swaptions`, `blackscholes`, `streamcluster`, `fft`, `lu-contig` and `radix`. For instance, it reduced the slicing ratio for `fft` from 64.25% to 8.95%, a 86.08% reduction. The ratio reduction for `cholesky`, `barnes`, and `ocean` was relatively small because our alias analysis sometimes reports may-alias for pointers

[3]We obtained the slicing results from an earlier prototype of our framework.

Program	FP_{ours}	$FP_{baseline}$	*Races*
`aget`	0	72	1
`PBZip2`	0	125	32
`fft`	0	96	0
`lu-contig`	18	18	0
`cholesky`	7	31	0
`radix`	14	53	0
`barnes` *	369	370	n/a
`water-spatial` *	1799	2447	n/a
`water-nsquared` *	333	354	n/a
`ocean` *	292	331	n/a
`blackscholes`	0	3	0
`swaptions`	0	165	0
`streamcluster`	0	4	0
`canneal`	0	21	0
`bodytrack`	0	4	0
`ferret`	0	6	0
`raytrace`	0	215	0

Table 4.2: *Precision of the race detector.* FP_{ours} and $FP_{baseline}$ show the number of false positives for our detector and the baseline, respectively. *Races* show the true races detected. The four starred programs have a larger number of reports, so we conservatively treated most reports as false positives.

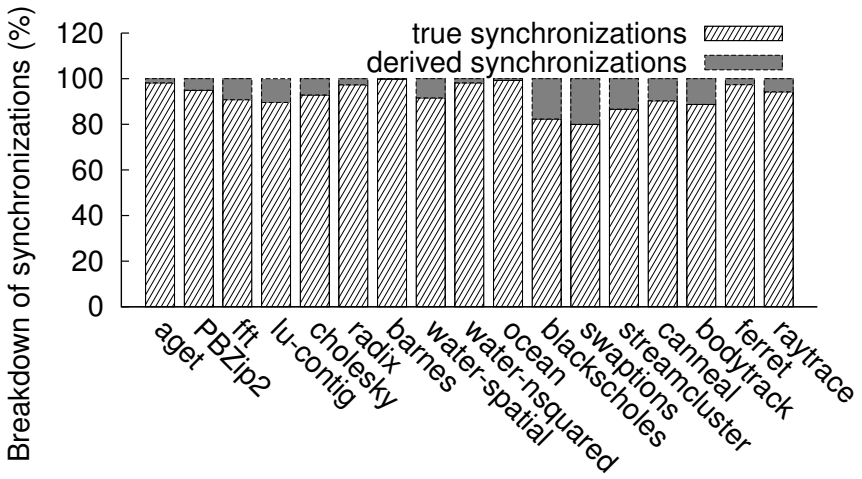


Figure 4.10: *True vs. derived synchronizations in schedules.*

accessed in different threads, causing more instructions than necessary to be included in the slices. The mean reduction over all programs is 48.7%.

4.5.2 Overhead

Table 4.3 shows the time specializing the control and data flow for each program. Control-flow specialization is much faster than data-flow specialization because it does not require expensive constraint-based analysis. Data-flow specialization typically finishes within minutes or hours. This time is correlated with the size of the schedule, the number of constraints collected, and the number of constraint-solving queries.

We also measured the number of derived synchronizations, *i.e.*, the calls to functions may transitively do synchronizations to resolve call-stack ambiguity (§4.2.1). If this number is large, the overhead to record schedules and specialize programs may increase. Figure 4.10 shows that, for every program evaluated, the majority of the synchronizations in the schedule are still true synchronizations. Therefore, including derived synchronizations in the schedules does not incur significant overhead.

Program	LOC	Sched	Cons	Use	CF (s)	DF (s)
aget	866	219	2667	720	1.4	1551.7
PBZip2	9869	158	1382	480	3.1	973.1
fft	877	98	1122	277	1.5	113.8
lu-contig	904	48	824	229	1.4	65.0
cholesky	3962	42	1550	1302	2.3	1967.9
radix	919	112	1016	330	1.5	42.5
barnes	2234	5555	1605	19280	5.4	1968.1
water-spatial	1958	1037	14572	5008	2.8	313.4
water-nsquared	1620	4768	48023	14530	4.1	468.8
ocean	2958	5709	66253	18580	6.8	1795.0
blackscholes	1264	51	482	130	1.2	38.3
swaptions	1094	15	215	105	1.3	9.7
streamcluster	1765	90	840	216	1.5	834.6
canneal	2794	31	535	311	5.4	96.9
bodytrack	7696	381	998	240	1.7	20.9
ferret	10765	153	439	118	1.0	35.3
raytrace	13226	87	13468	417	1.4	5397.8

Table 4.3: *Specialization time.* **LOC** shows the lines of code in each program. The LOC of PBZip2 includes the bzip2 compression library. We show the time spent in specializing control flow (**CF**) and data flow (**DF**). Since specialization time are affected by the schedule, constraints, and queries, we also show the schedule length (**Sched**), the number of constraints (**Cons**), and the number of uses in the def-use analysis (**Use**).

Program	Position	Bug	Effect
`aget`	Download.c: 87-95	`rbuf` may not have the byte sequence `\r\n\r\n`. `aget` needs an extra bound check.	race, program crash, or large data corruption
`aget`	Download.c: 98-99	The bound check should be `td->offset + dr - i > foffset`, not `dr - i > foffset`. The size passed to `pwrite` should be `foffset - td->offset`, not `foffset - i`.	race, or program crash
`aget`	Download.c: 99,101,113,115	`aget` should check the return value of `pwrite`.	race, program crash, or small data corruption
`aget`	Download.c:111	`aget` should check the return value of `recv`.	race, program crash, or small data corruption
`radix`	radix.C:148	`radix` should check variable `radix` against its upper bound 4096.	race, program crash.
`radix`	radix.C:159	`radix` should check `num_keys` against its upper bound 262144.	race, or program crash
`fft`	fft.C:162	`fft` should check `log2_line_size` against its upper bound 4.	program crash (before a race may occur)

Table 4.4: *Bugs found.*

4.5.3 Bugs Found

The precision of our framework helped detect 7 previously unknown bugs in the evaluated programs. This result is particularly interesting considering that the evaluated programs have been well checked in previous work [Zhang *et al.*, 2011; Zhang *et al.*, 2010; Park *et al.*, 2009a; Lu *et al.*, 2008; Gao *et al.*, 2011].

These bugs were typically detected as follows. Our analyses flagged two memory accesses from different threads as potential aliases or races, even though they should not be. We initially thought these "false positives" were due to the imprecision of our analyses and inspected them. Specifically, we queried STP for a solution to make the two pointers accessed identical.

Surprisingly, many of these "false positives" turned out to be real bugs. A common cause is that an input variable is used as an array index without being checked against the upper or lower bounds of the array or the partition of the array assigned to a thread. Such off-bound accesses may indeed cause different threads to race, and our analyses thus flagged them. We detected 7 such bugs in `aget`, `radix`, and `fft`, which are shown in Table 4.4. These bugs may cause races, program crashes, or, worse, file data corruption. We manually verified these effects by running the buggy benchmarks on the bug-inducing inputs generated by STP. (The results presented in the previous two subsections are from the patched programs because we do not want these bugs to pollute our evaluation.)

4.6 Related Work

Slicing. Slicing techniques can remove irrelevant statements or instructions. Program slicing [Weiser, 1979] does so on programs, dynamic slicing [Agrawal and Horgan, 1990; Zhang and Gupta, 2004] on dynamic execution traces, and path slicing [Jhala and Majumdar, 2005] on (potentially infeasible) paths. Precondition slicing [Costa *et al.*, 2007] is similar to path slicing with improved precision by incorporating dynamic information into the analysis.

Our technique to specialize a program toward a schedule differs from these slicing techniques because it takes as input both a program and a schedule, and outputs a specialized program that can actually run. Moreover, our technique does not merely remove statements; instead, it may transform a program by cloning statements when specializing the control flow of the program,

replacing variables with constants when specializing the data flow, *etc.*

Program specialization. Program specialization can specialize a program according to various goals [Reps and Turnidge, 1996; Nirkhe and Pugh, 1992; Jørgensen, 1992; Consel and Danvy, 1993; Glück and Jørgensen, 1995]. For instance, it can specialize according to common inputs [Consel and Danvy, 1993; Nirkhe and Pugh, 1992]. The specialized programs can then be better optimized. Unlike previous work, our framework specializes a program toward a set of schedules, thus allowing stock analyses and optimizations to run on the specialized programs. To the best of our knowledge, we are the first to specialize a program toward schedules.

Concolic testing. The idea of schedule specialization is loosely analogous to concolic testing [Sen *et al.*, 2005; Larson and Austin, 2003; Godefroid *et al.*, 2005]. Concolic testing systematically explores possible paths of a program. It first runs a program on a concrete input, and computes a conjunction of symbolic constraints along the path of this execution as the "precondition" of this path; it then negates this precondition and computes another concrete input that leads the program to a different path. Schedule specialization is similar in that we collect schedules from "concrete" runs and compute the preconditions of these schedules. However, the goals of these two ideas are opposite: concolic testing aims to explore as many paths as possible, and schedule specialization aims to limit the number of possible schedules. Also, given schedules are typically much more coarse-grained than program paths, preconditions of program paths would be much more stronger than the preconditions of schedules, causing low reuse-rates.

Data race detection. Data race detectors are broadly classified into two classes: static and dynamic. Dynamic data race detectors are in general classified into three types: happens-before based, lockset-based, and hybrid. These three types generally follow the definition of data races described in §2.1, but differ from each other in the way they compute whether two statement instances are concurrent. Happens-before-based data race detectors [Dinning and Schonberg, 1990; Ronsse and De Bosschere, 1999; Schonberg, 1989] determine whether two statement instances are concurrent based on Lamport's happens-before relation [Lamport, 1978]. Lockset-based data race detectors [Savage *et al.*, 1997; von Praun and Gross, 2003] treat two statement instances not protected by the same mutex as concurrent. The lockset-based approach can detect more data races than the happens-before approach, but may report false positives. Hybrid data race detectors [O'Callahan and Choi, 2003; Yu *et al.*, 2005] combine these two approaches and overcome

the disadvantages of either.

Static data race detectors use either model checking [Henzinger *et al.*, 2004; Qadeer and Wu, 2004] or static versions of the lockset algorithm [Loginov *et al.*, 2001; Engler and Ashcraft, 2003; Naik *et al.*, 2006]. Model-checking-based race detectors try to simulate each program path and detect data races along each path. Therefore, it is relatively precise but has only been applied to small programs due to path explosion. Static data race detectors based on the lockset algorithm reduce the analysis time by abstraction. It conceptually simulates the lockset algorithm on abstract threads, instructions, and locks. Therefore, these detectors suffer from imprecision, and can generate many false positives.

The data race detector we have built based on schedule specialization (§4.4) differs from both dynamic and static data race detection. Our approach covers more executions than dynamic data race detection, because we consider not just one execution but all executions that are compatible with a certain schedule. Compared to static data race detection, we analyze a program with respect to a schedule instead of all possible schedules. One key challenge for static race detectors is how to compute whether two statements can run concurrently. By focusing on the synchronization order that the target schedule specifies, we can easily identify statements that run concurrently.

4.7 Summary

This chapter presented our specialization framework. It specializes a program into a simpler program based on a schedule, so that the resultant program can be analyzed with stock analyses. It provides a precise schedule-aware def-use analysis, enabling many powerful applications. Our results show that our framework can drastically improve the precision of alias analysis, path slicing, and race detection.

In our future work, we plan to leverage the precision provided by our framework to build precise error detectors, post-mortem analyzers, verifiers, and optimizers for multithreaded programs. For example, we can apply the idea of schedule specialization to precisely detect commutativity races [Dimitrov *et al.*, 2014]. In addition, we believe a similar specialization approach can improve analysis precision for sequential programs, too. In general, static analysis over all possible executions may be imprecise. To improve precision, we may perform static analysis over only a small

set of executions (*e.g.*, the most common executions) and, if necessary, resort to dynamic analyses for the other executions. We believe this direction will face many interesting precision, soundness, and overhead tradeoffs, which we will investigate.

Chapter 5

Bypassing Races in Live Applications with Execution Filters

Deployed multithreaded applications contain many races because these applications are difficult to write, test, and debug. These races include data races, atomicity violations, and order violations [Lu *et al.*, 2008]. They can cause application crashes and data corruptions. Worse, the number of "deployed races" may drastically increase due to the rise of multicore and the immaturity of race detectors.

Many static and dynamic race detectors (*e.g.*, [Musuvathi *et al.*, 2008; Savage *et al.*, 1997; Yu *et al.*, 2005; Lu *et al.*, 2007; Lu *et al.*, 2006; Erickson *et al.*, 2010; Veeraraghavan *et al.*, 2011; Wester *et al.*, 2013]) have been proposed to detect races. However, none of them is guaranteed to detect all races with a reasonable precision. Static race detectors tend to report many false positives and bury true errors in false reports. Dynamic race detectors only cover a very limited number of executions and can have many false negatives.

Schedule specialization advances the state of the art of race detection by balancing the tradeoff between soundness and precision. However, it still cannot detect all races for two reasons. First, the soundness of our static analysis results is conditioned on that one of the analyzed schedules is enforced. Although schedule specialization analyzes highly reusable schedules, a finite set of schedules may not cover all inputs for complicated real-world programs. If an input cannot be processed by any of the analyzed schedules, it may still trigger races even if our static detection claims

the program is race-free for the schedules it observed. Second, although schedule specialization can dramatically reduce false reports from a static race detector, the detector may still emit more false positives than dynamic race detectors because schedule specialization does not leverage full dynamic information. As a result, users may filter out some error reports which may correspond to true errors [Voung *et al.*, 2007; Engler and Ashcraft, 2003].

A conventional solution to fixing deployed races is software update, but this method requires application restarts, and is at odds with high availability demand. Live update systems [Arnold and Kaashoek, 2009; Chen *et al.*, 2006; Altekar *et al.*, 2005; Makris and Ryu, 2007; Neamtiu *et al.*, 2006; Neamtiu and Hicks, 2009; Subramanian *et al.*, 2009] can avoid restarts by adapting conventional patches into *hot patches* and applying them to live systems, but the reliance on conventional patches has two problems.

First, due to the complexity of multithreaded applications, race-fix patches can be *unsafe* and introduce new errors [Lu *et al.*, 2008]. Safety is crucial to encourage user adoption, yet automatically ensuring safety is difficult because conventional patches are created from general, difficult-to-analyze languages. Thus, previous work [Neamtiu *et al.*, 2006; Neamtiu and Hicks, 2009] had to resort to extensive programmer annotations.

Second, creating a releasable patch from a correct diagnosis can still take time. This delay leaves buggy applications unprotected, compromising reliability and potentially security. This delay can be quite large: we analyzed the Bugzilla records of nine real races and found that this delay can be days, months, or even years. Table 5.1 shows the detailed results.

Many factors contribute to this delay. At a minimum level, a conventional patch has to go through code review, testing, and other mandatory software development steps before being released, and these steps are all time-consuming. Moreover, though a race may be fixed in many ways (*e.g.*, lock-free flags, fine-grained locks, and coarse-grained locks), developers are often forced to strive for an efficient option. For instance, two of the bugs we analyzed caused long discussions of more than 30 messages, yet both can be fixed by adding a single critical section. Performance pressure is perhaps why many races were *not* fixed by adding locks [Lu *et al.*, 2008].

This chapter presents Loom, a "live-workaround" system designed to quickly protect applications against races until correct conventional patches are available and the applications can be restarted. It reflects our belief that the true power of live update is its ability to provide immediate

Race ID	Report	Diagnosis	Fix	Release	Delay (days)
Apache-25520	12/15/03	12/18/03	01/17/04	03/19/04	30
Apache-21287	07/02/03	N/A	12/18/03	03/19/04	169
Apache-46215	11/14/08	N/A	N/A	N/A	N/A
MySQL-169	03/19/03	N/A	03/24/03	06/20/03	5
MySQL-644	06/12/03	N/A	N/A	05/30/04	353
MySQL-791	07/04/03	07/04/03	07/14/03	07/22/03	10
Mozilla-73761	03/28/01	03/28/01	04/09/01	05/07/01	12
Mozilla-201134	04/07/03	04/07/03	04/16/03	01/08/04	9
Mozilla-133773	03/27/02	03/27/02	12/01/09	01/21/10	2816

Table 5.1: *Long delays in race fixing.* We studied the delays in the fix process of nine real races; some of the races were extensively studied [Lu *et al.*, 2006; Park *et al.*, 2009a; Lu *et al.*, 2008; Park *et al.*, 2009b; Altekar and Stoica, 2009]. We identify each race by "$Application-\langle Bugzilla\ \#\rangle$." Column **Report** indicates when the race was reported, **Diagnosis** when a developer confirmed the root cause of the race, **Fix** when the final fix was posted, and **Release** when the version of application containing the fix was publicly released. We collected all dates by examining the Bugzilla record of each race. An N/A means that we could not derive the date. Column **Delay** indicates the days between diagnosis and fix, which range from a few days to a month to a few years. For all but two races, the bug reports from the application users contained correct and precise diagnoses. Mozilla-201134 and Mozilla-133773 caused long discussions of more than 30 messages, though both can be fixed by adding a critical region.

workarounds. To use LOOM, developers first compile their application with LOOM. At runtime, to workaround a race, an application developer writes an *execution filter* that synchronizes the application source to filter out racy thread interleavings. This filter is kept separate from the source. Application users can then download the filter and, for immediate protection, install it to their application without a restart.

LOOM decouples execution filters from application source to achieve safety and flexibility. Execution filters are safe because LOOM's execution filter language allows only well formed synchronization constraints. For instance, "code regions r_1 and r_2 are mutually exclusive." This declarative language is simpler to analyze than a general programing language such as C because LOOM need not reverse-engineer developer intents (*e.g.*, what goes into a critical region) from scattered operations (*e.g.*, `lock()` and `unlock()`).

As temporary workarounds, execution filters are more flexible than conventional patches. One main benefit is that developers can make better performance and reliability tradeoffs during race fixing. For instance, to make two code regions r_1 and r_2 mutually exclusive when they access the same memory object, developers can use critical regions larger than necessary; they can make r_1 and r_2 always mutually exclusive even when accessing different objects; or in extreme cases, they can run r_1 and r_2 in single-threaded mode. This flexibility enables quick workarounds; it can benefit even the applications that do not need live update.

We believe the execution filter idea and the LOOM system as described are worthwhile contributions. To the best of our knowledge, LOOM is the first live-workaround system designed for races. Our additional technical contributions include the techniques we created to address the following two challenges.

A key safety challenge LOOM faces is that even if an execution filter is safe by construction, installing it to a live application can still introduce errors because the application state may be inconsistent with the filter. For instance, if a thread is running inside a code region that an execution filter is trying to protect, a "double-unlock" error could occur. Thus, LOOM must (1) check for inconsistent states and (2) install the filter only in consistent ones. Moreover, LOOM must make the two steps atomic, despite the concurrently running application threads and multiple points of updates. This problem cannot be solved by a common safety heuristic called *function quiescence* [Gilmore and Walton, 1997; k42, ; Baumann *et al.*, 2005; Neamtiu *et al.*, 2006]. We

thus create a new algorithm termed *evacuation* to solve this problem by proactively quiescing an arbitrary set of code regions given at runtime. We believe this algorithm can also benefit other live update systems.

A key performance challenge Loom faces is to maintain negligible performance overhead during an application's normal operations to encourage adoption. The main runtime overhead comes from the engine used to live-update an application binary. Although Loom can use general-purpose binary instrumentation tools such as Pin, the overhead of these tools (up to 199% [Luk *et al.*, 2005] and 1065.39% in our experiments) makes them less suitable as options for Loom. We thus use a fast *hybrid instrumentation* engine. It statically transforms an application to include a "hot backup", which can then be updated arbitrarily by execution filters at runtime. Besides helping reducing Loom's runtime overhead, this instrumentation framework also reduces the overhead of scheduling memory accesses in Peregrine (§3.3.2).

We implemented Loom on Linux. It runs in user space and requires no modifications to the applications or the OS, simplifying deployment. It does not rely on non-portable OS features (*e.g.*, `SIGSTOP` to pause applications, which is not supported properly on Windows). Loom's static transformation is a plugin to the LLVM compiler [LLVM, 2013], requiring no changes to the compiler either.

We evaluated Loom on nine real races from a diverse set of six applications: two server applications, MySQL and Apache; one desktop application PBZip2 (a parallel compression tool); and implementations of three scientific algorithms in SPLASH2 [SPLASH2, 2007]. Our results show that

1. Loom is effective. It can flexibly and safely fix all races we have studied. It does not degrade application availability when installing execution filters. Its evacuation algorithm can install a fix within a second even under heavy workload, whereas a live update approach using function quiescence cannot install the fix in an hour, the time limit of our experiment.
2. Loom is fast. Loom has negligible performance overhead and in some cases even speeds up the applications. The one exception is MySQL. Running MySQL with Loom alone increases response time by 4.11% and degrades throughput by 3.76%.
3. Loom is scalable. Experiments on a 48-core machine show that Loom scales well as the number of application threads increases.

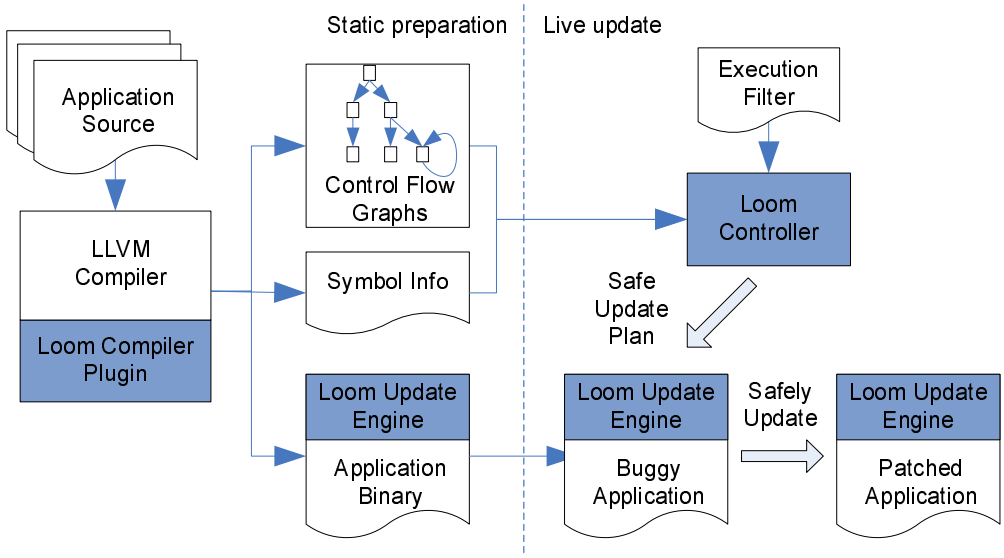


Figure 5.1: Loom *overview.* Its components are shaded.

4. Loom is easy to use. Execution filters are concise, safe, and flexible (able to fix all races studied, often in more than one way).

This chapter is organized as follows. We first give an overview of Loom (§5.1). We then describe Loom's execution filter language (§5.2), the evacuation algorithm (§5.3), and the hybrid instrumentation engine (§5.4). We then present our experimental results (§5.5). We finally discuss related work (§5.6) and conclude (§5.7).

5.1 Overview

Figure 5.1 presents an overview of Loom. To use Loom for live update, developers first statically transform their applications with Loom's compiler plugin. This plugin injects a copy of Loom's update engine into the application binary; it also collects the application's control flow graphs (CFG) and symbol information on behalf of the live update engine.

Loom's compiler plugin runs within the LLVM compiler [LLVM, 2013]. We choose LLVM for its compatibility with GCC and its easy-to-analyze intermediate representation (IR). However, Loom's algorithms are general and can be ported to other compilers such as GCC. Indeed, for clarity we will present all our algorithms at the source level (instead of the LLVM IR level).

To fix a race, application developers write an execution filter in Loom's filter language and distribute the filter to application users. A user can then install the filter to immediately protect their application by running

```
% loomctl add <pid> <filter-file>
```

Here `loomctl` is a user-space program called the Loom controller that interacts with users and initiates live update sessions, `pid` denotes the process ID of a buggy application instance, and `filter-file` is a file containing the execution filter. Under the hood, this controller compiles the execution filter down to a safe update plan using the CFGs and symbol information collected by the compiler plugin. This update plan includes three parts: (1) synchronization operations to enforce the constraints described in the filter and where, in the application, to add the operations; (2) safety preconditions that must hold for installing the filter; and (3) sanity checking code to detect potential errors in the filter itself. The controller sends the update plan to the update engine running as a thread inside the application's address space, which then monitors the runtime states of the application and carries out the update plan only when all the safety preconditions are satisfied.

If Loom detects a problem with a filter through one of its sanity checks, it can automatically remove the problematic filter. It again waits for all the safety preconditions to hold before removing the filter.

Users can also remove a filter manually, if for example, the race that the filter intends to fix turns out to be benign. They do so by running

```
% loomctl ls <pid>
% loomctl remove <pid> <filter-id>
```

The first command "`loomctl ls`" returns a list of installed filter IDs within process `pid`. The second command "`loomctl remove`" removes filter `filter-id` from process `pid`.

Users can replace an installed filter with a new filter, if for example the new filter fixes the same race but has less performance overhead. Users do so by running

```
% loomctl replace <pid> <old-id> <new-file>
```

where `old-id` is the ID of the installed filter, and `new-file` is a file containing the new filter. Loom ensures that the removal of the old filter and the installation of the new filter are atomic, so that the application is always protected from the given race.

5.1.1 Usage Scenarios

LOOM enables users to explicitly describe their synchronization intents and orchestrate thread interleavings of live applications accordingly. Using this mechanism, we envision a variety of strategies users can use to fix races.

Live update. At the most basic level, users can translate some conventional patches into execution filters, and use LOOM to install them to live applications.

Temporary workaround. Before a permanent fix (*i.e.*a correct source patch) is out, users can create an execution filter as a crude, temporary fix to a race, to provide immediate protection to highly critical applications.

Preventive fix. When a *potential* race is reported (*e.g.*, by automated race detection tools or users of the application), users can immediately install a filter to prevent the race suspect. Later, when developers deem this report false or benign, users can simply remove the filter.

Cooperative fix. Users can share filters with each other. This strategy enjoys the same benefits as other cooperative protection schemes [Costa *et al.*, 2005; Perkins *et al.*, 2009; Sidiroglou *et al.*, 2009; Jula *et al.*, 2008]. One advantage of LOOM over some of these systems is that it automatically verifies filter safety, thus potentially reducing the need to trust other users.

Site-specific fix. Different sites have different workloads. An execution filter too expensive for one site may be fine for another. The flexibility of execution filters allows each site to choose what specific filters to install.

Fix without live update. For applications that do not need live update, users can still use LOOM to create quick workarounds, improving reliability.

Besides fixing races, LOOM can be used for the opposite: demonstrating a race by forcing a racy thread interleaving. Compared to previous race diagnosis tools that handle a fixed set of race patterns [Park *et al.*, 2009a; Sen, 2008; Park and Sen, 2008; Joshi *et al.*, 2009], LOOM's advantage is to allow developers to construct potentially complex "concurrency" testcases.

Although LOOM can also avoid deadlocks by avoiding deadlock-inducing thread interleavings, it is less suitable for this purpose than existing tools (*e.g.*, Dimmunix [Jula *et al.*, 2008]). To avoid races, LOOM's update engine can add synchronizations to arbitrary program locations. This engine is overkill for avoiding deadlocks: intercepting lock operations (*e.g.*, via `LD_PRELOAD`) is often

enough.

5.1.2 Limitations

Loom is explicitly designed to work around (broadly defined) races because they are some of the most difficult bugs to fix and this focus simplifies Loom's execution filter language and safety analysis. Loom is not intended for other classes of errors. Nonetheless, we believe the idea of high-level and easy-to-verify fixes can be generalized to many other classes of errors.

Loom does not attempt to fix *occurred* races. That is, if a race has caused bad effects (*e.g.*, corrupted data), Loom does not attempt to reverse the effects (*e.g.*, recover the data). It is conceivable to allow developers to provide a general function that Loom runs to recover occurred races before installing a filter. Although this feature is simple to implement, it makes safety analysis infeasible. We thus rejected this feature.

Safety in Loom terms means that an execution filter and its installation/removal processes introduce no new correctness errors to the application. However, similar to other safe error recovery [Qin *et al.*, 2007] or avoidance [Jula *et al.*, 2008; Wang *et al.*, 2008] tools, Loom runs with the application and perturbs timing, thus it may expose some existing application races because it makes some thread interleavings more likely to occur. Moreover, execution filters synchronize code, and may introduce deadlocks and performance problems. Loom can recover from filter-introduced deadlocks (§5.2.3) using timeouts, but currently does not deal with performance problems.

At an implementation level, Loom currently supports a fixed set of synchronization constraint types. Although adding new types of constraints is easy, we have found the existing constraint types sufficient to fix all races evaluated. Another issue is that Loom uses debugging symbol information in its analysis, which can be inaccurate due to compiler optimization. This inaccuracy has not been a problem for the races in our evaluation because Loom keeps an unoptimized version of each basic block for live update (§5.4).

5.2 Execution Filter Language

Loom's execution filter language allows developers to explicitly declare their synchronization intents on code. This declarative approach has several benefits. First, it frees developers from the

low-level details of synchronization, increasing race fixing productivity. Second, it also simplifies Loom's safety analysis because Loom does not have to reverse-engineer developer intents (*e.g.*, what goes into a critical section) from low-level synchronization operations (*e.g.*, scattered `lock()` and `unlock()`), which can be difficult and error-prone. Lastly, Loom can easily insert error-checking code for safety when it compiles a filter down to low-level synchronization operations.

5.2.1 Example Races and Execution Filters

In this section, we present two real races and the execution filters to fix them to demonstrate Loom's execution filter language and its flexibility.

The first race is in MySQL (Bugzilla # 791), which causes the MySQL on-disk transaction log to miss records. Figure 5.2 shows the race. The code on the left (function `new_file()`) rotates MySQL's transaction log file by closing the current log file and opening a new one; it is called when the transaction log has to be flushed. The code on the right is used by MySQL to append a record to the transaction log. It uses *double-checked locking* and writes to the log only when the log is open. The race occurs if the racy `is_open()` (T2, line 3) catches a closed log when thread T1 is between the `close()` (T1, line 5) and the `open()` (T1, line 6).

Although a straightforward fix to the race exists, performance demands likely forced developers to give up the fix and choose a more complex one instead. The straightforward fix should just remove the racy check (T2, line 3). Unfortunately, this fix creates unnecessary overhead if MySQL is configured to skip logging for speed; this overhead can increase MySQL's response time by more than 10% as observed in our experiments. The concern to this overhead likely forced MySQL developers to use a more involved fix, which adds a new flag field to MySQL's transaction log and modifies the `close()` function to distinguish a regular `close()` call and one for reopening the log.

In contrast, Loom allows developers to create temporary workarounds with flexible performance and reliability tradeoffs. These temporary fixes can protect the application until developers create a correct and efficient fix at the source level. Figure 5.3 shows several execution filters that can fix this race. Execution filter 1 in the figure is the most conservative fix: it makes the code region between T1, line 5 and T1, line 6 atomic against all code regions, so that when a thread executes this region, all other threads must pause. We call such a synchronization constraint *unilateral exclusion*

```
1: // log.cc. thread T1
2: void MYSQL_LOG::new_file(){
3:    lock(&LOCK_log);
4:    ...
5:    close(); // log is closed
6:    open(...);
7:    ...
8:    unlock(&LOCK_log);
9: }
```

```
1: // sql_insert.cc. thread T2
2: // [race] may return false
3: if (mysql_bin_log.is_open()){
4:    lock(&LOCK_log);
5:    if (mysql_bin_log.is_open()){
6:       ... // write to log
7:    }
8:    unlock(&LOCK_log);
9: }
```

Figure 5.2: *A real MySQL race, slightly modified for clarity.*

```
// Execution filter 1: unilateral exclusion
{log.cc:5, log.cc:6} <> *

// Execution filter 2: mutual exclusion of code
{log.cc:5, log.cc:6} <> MYSQL_LOG::is_open

// Execution filter 3: mutual exclusion of code and data
{log.cc:5 (this), log.cc:6 (this)} <> MYSQL_LOG::is_open(this)
```

Figure 5.3: *Execution filters for the MySQL race in Figure 5.2.*

```
// pbzip2.cpp. thread T1
1: main() {
2:   for(i=0;i<numCPU;i++)
3:     pthread_create(...,
4:       decompress, fifo);
5:   queueDelete(fifo);
6: }
```

```
// pbzip2.cpp. thread T2
7 : void *decompress(void *q){
8 :   queue *fifo = (queue *)q;
9 :   ...
10:   pthread_mutex_lock(fifo->mut);
11:   ...
12: }
```

Figure 5.4: *A real PBZip2 race, simplified for clarity.*

```
pbzip2.cpp:10 {numCPU} > pbzip2.cpp:5
```

Figure 5.5: *Execution filter for the PBZip2 race in Figure 5.4.*

in contrast to mutual exclusion that requires participating threads agree on the same lock.[1] Here operator "`<>`" expresses mutual exclusion constraints, its first operand "{`log.cc:5, log.cc:6`}" specifies a code region to protect, and its second operand "`*`" represents all code regions. This "expensive" fix incurs only 0.48% overhead (§5.5.1) because the log rotation code rarely executes.

Execution filter 2 reduces overhead by refining the "`*`" operand to a specific code region, function `MYSQL_LOG::is_open()`. This filter makes the two code regions mutually exclusive, regardless of what memory locations they access. Execution filter 3 further improves performance by specifying the memory location accessed by each code region.

The second race causes PBZip2 to crash due to a use-after-free error. Figure 5.4 shows the race. The crash occurs when `fifo` is dereferenced (line 10) after it is freed (line 5). The reason is that the `main()` thread does not wait for the `decompress()` threads to finish. To fix this race, developers can use the filter in Figure 5.5, which constrains line 10 to run for `numCPU` times before line 5.

Constructs	**Syntax**
Event (short as e)	$file : line$
	$file : line\,(expr)$
	$e\{n\}$, n is # of occurrence
Region (short as r)	$\{e_1, ..., e_i; e_{i+1}, ..., e_n\}$
	$func\,(args)$
Execution Order	$e_1 > e_2 > ... > e_n$
Mutual Exclusion	$r_1 <> r_2 <> ... <> r_n$
Unilateral Exclusion	$r <> *$

Table 5.2: *Execution filter language summary.*

5.2.2 Syntax and Semantics

Table 5.2 summarizes the main syntax and semantics of Loom's execution filter language. This language allows developers to express synchronization constraints on *events* and *regions*. An event in the simplest form is "$file : line$," which represents a dynamic instance of a static program statement, identified by file name and line number. An event can have an additional "$(expr)$" component and an "$\{n\}$" component, where $expr$ and n refer to valid expressions with no function calls or dereferences. The $expr$ expression distinguishes different dynamic instances of program statements and Loom synchronizes the events only with matching $expr$ values. The n expression specifies the number of occurrences of an event and is used in execution order constraints. A *region* represents a dynamic instance of a static code region, identified by a set of entry and exit events or an application function. A region representing a function call can have an additional "$(args)$" component to distinguish different calls to the same function.

Loom currently supports three types of synchronization constraints (the bottom three rows in Table 5.2). Although adding new constraint types is easy (e.g., synchronization barriers that enforce all threads to reach a rendezvous point before continuing), we have found existing ones enough to fix all races evaluated. An execution order constraint as shown in the table makes event e_1 happen before e_2, e_2 before e_3, and so forth. A mutual exclusion constraint as shown makes every pair of code regions r_i and r_j mutually exclusive with each other. A unilateral exclusion

[1]Note that unilateral exclusion differs (subtly) from single-threaded execution: unilateral exclusion allows no context switches.

constraint conceptually makes the execution of a code region single-threaded.

5.2.3 Language Implementation

LOOM implements the execution filter language using locks and semaphores. Given an execution order constraint $e_i > e_{i+1}$, LOOM inserts a semaphore `up()` operation at e_i and a `down()` operation at e_{i+1}. LOOM implements a mutual exclusion constraint by inserting `lock()` at region entries and `unlock()` at region exits. LOOM implements a unilateral exclusion constraint reusing the evacuation mechanism (§5.3), which can pause threads at safe locations and resume them later.

LOOM creates the needed locks and semaphores on demand. The first time a lock or semaphore is referenced by one of the inserted synchronization operations, LOOM creates this synchronization object based on the ID of the filter, the ID of the constraint, and the value of *expr* if present. It initializes a lock to an unlocked state and a semaphore to 0. It then inserts this object into a hash table for future references. To limit the size of this table, LOOM garbage-collects these synchronization objects. Freeing a synchronization object is safe as long as it is unlocked (for locks) or has a counter of 0 (for semaphores). If this object is referenced later, LOOM simply re-creates it. The default size of this table is 256 and LOOM never needed to garbage-collect synchronization objects in our experiments.

The `up()` and `down()` operations LOOM inserts behave slightly differently than standard semaphore operations when n, the number of occurrences, is specified. Given $e1\{n_1\} > e2\{n_2\}$, `up()` conceptually increases the semaphore counter by $\frac{1}{n_1}$ and `down()` decreases it by $\frac{1}{n_2}$. Our implementation uses integers instead of floats. LOOM stores the value of n the first time the corresponding event runs and ignores future changes of n.

LOOM computes the values of *expr* and n using debugging symbol information. We currently allow *expr* and n in an event to be the following expressions: `a` (constant or primitive variable), `a+b`, `&a`, `&a[i]`, `&a->f`, or any recursive combinations of these expressions. For safety, we do not allow function calls or dereferences. These expressions are sufficient for writing the execution filters in our evaluation. The performance overhead of including debugging symbol information is negligible, because hybrid instrumentation only generates debugging information on the hot backup (§5.4).

We implemented this feature using the DWARF library and the `parse_exp_1()` function in GDB. Specifically, we use `parse_exp_1()` to parse the *expr* or n component into an expression

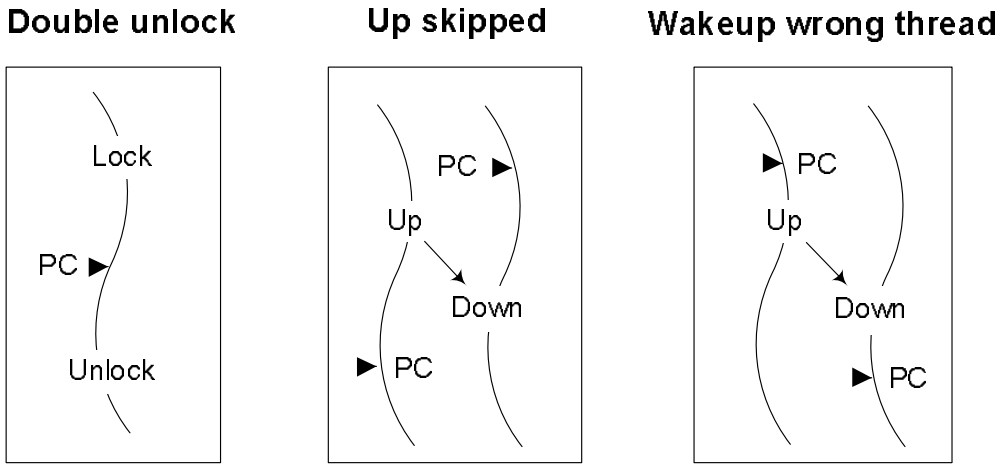


Figure 5.6: *Unsafe program states for installing filters.*

tree, then compile this tree into low level instructions by querying the DWARF library. Note this compilation step is done inside the LOOM controller, so that the live update engine does not have to pay this overhead.

LOOM implements three mechanisms for safety. First, by keying synchronization objects based on filter and constraint IDs, it uses a disjoint set of synchronization objects for different execution filters and constraints, avoiding interference among them. Second, LOOM inserts additional checking code when it generates the update plan. For example, given a code region c in a mutual exclusion constraint, LOOM checks for errors such as c's `unlock()` releasing a lock not acquired by c's `lock()`. Lastly, LOOM checks for filter-induced deadlocks to guard against buggy filters. If a buggy filter introduces a deadlock, one of its synchronization operations must be involved in the wait cycle. LOOM detects such deadlocks using timeouts, and automatically removes the offending filter.

5.3 Avoiding Unsafe Application States

Figure 5.6 shows three unsafe scenarios LOOM must handle. For a mutual exclusion constraint that turns code regions into critical sections, LOOM must ensure that no thread is executing within the code regions when installing the filter to avoid "double-unlock" errors. Similarly, for an execution order constraint $e_1 > e_2$, LOOM must ensure either of the following two conditions when installing the filter: (1) both e_1 and e_2 have occurred or (2) neither has occurred; otherwise the `up()` LOOM

```
1:  // database worker thread
2:  void handle_client(int fd) {
3:    for(;;) {
4:      struct client_req req;
5:      int ret = recv(fd, &req, ...);
6:      if(ret <= 0) break;
7:      open_table(req.table_id);
8:      ... // do real work
9:      close_table(req.table_id);
10:   }
11: }
12: void open_table(int table_id) {
13:     // fix: acquire table lock
14:     ... // actual code to open table
15: }
16: void close_table(int table_id) {
17:     ... // actual code to close table
18:     // fix: release table lock
19: }
```

Figure 5.7: *A contrived race.*

inserts at e_1 may get skipped or wake up a wrong thread.

Note that a naïve approach is to simply ignore an `unlock()` if the corresponding lock is already unlocked, but this approach does not work with execution order constraints. Moreover, it mixes unsafe program states with buggy filters, and may reject correct filters simply because it tries to install the filters at unsafe program states.

A common safety heuristic called *function quiescence* [Gilmore and Walton, 1997; Baumann *et al.*, 2005; Neamtiu *et al.*, 2006] cannot address this unsafe state problem. This technique updates a function only when no stack frame of this function is active in any call stack of the application. Unfortunately, though this technique can ensure safety for many live updates, it is insufficient for execution filters because their synchronization constraints may affect multiple functions.

We demonstrate this point using a race example. Figure 5.7 shows the worker thread code of a contrived database. Function `handle_client()` is the main thread function. It takes a client socket as input and repeatedly processes requests from the socket. For each request, function `handle_client()` opens the corresponding database table by calling `open_table()`, serves the request, and closes the table by calling `close_table()`. The race in Figure 5.7 occurs when multiple clients concurrently access the same table.

To fix this race, an execution filter can add a lock acquisition at line 13 in `open_table()` and a lock release at line 18 in `close_table()`. To safely install this filter, however, the quiescence of `open_table()` and `close_table()` is not enough, because a thread may still be running at line 8 and cause a double-unlock error. An alternative fix is to add the lock acquisition and release in function `handle_client()`, but this function hardly quiesces because of the busy loop (line 3-10) and the blocking call `recv()`.

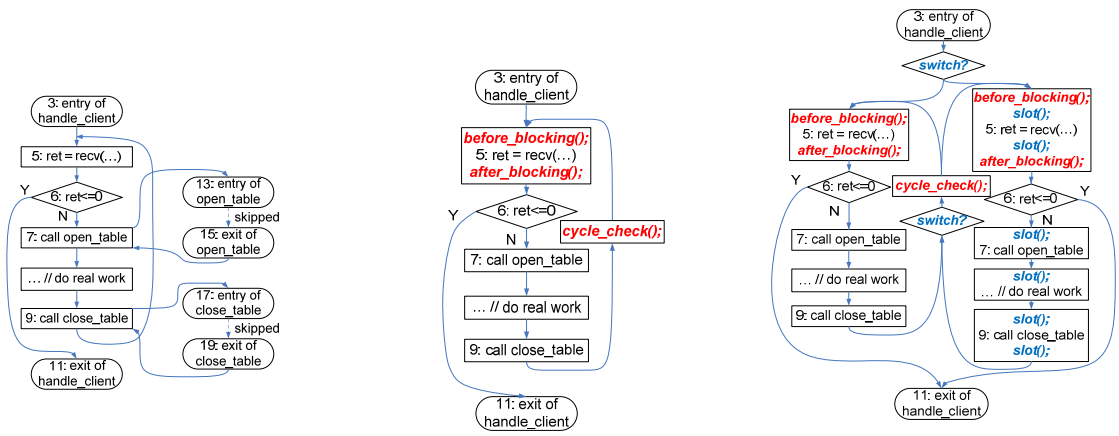
LOOM solves the unsafe state program using an algorithm termed *evacuation* that can proactively quiesce arbitrary code regions. From a high level, this algorithm takes a filter and computes a set of unsafe program locations that may interfere with the filter. It does so conservatively to avoid marking an unsafe location as safe. Then, it "evacuates" threads out of the unsafe locations and blocks them at safe program locations. After that, it installs the filter and resumes the threads.

5.3.1 Computing Unsafe Program Locations

The first step of our evacuation algorithm is to compute the unsafe program locations for the given execution filter. LOOM uses slightly different methods to compute the unsafe program locations for mutual exclusion and for execution order constraints. To compute unsafe program locations for mutual exclusion constraints, LOOM performs a static reachability analysis on the *interprocedural control flow graph (ICFG)* $\mathcal{G}$ of an application. An ICFG connects each function's control flow graphs by following function calls and returns. Figure 5.8a shows the ICFG for the code in Figure 5.7. We say statement s_1 *reaches* s_2 on G or $reachable(G, s_1, s_2)$ if there is a path from s_1 to s_2 on ICFG G. For example, the statement at line 13 reaches the statement at line 8 in Figure 5.7.

Given an execution filter f with mutual exclusion constraint $r_1 <> r_2 <> ... <> r_n$, the set of unsafe program locations $unsafe(f)$ includes any statement s potentially inside one of the regions. Specifically, $unsafe(f)$ is the set of statements s such that $reachable(\mathcal{G} \backslash r_i.entries, s, r_i.exits)$ for $i \in [1, n]$, where $r_i.entries$ are the entry statements to region r_i and $r_i.exits$ are the exit statements.

LOOM computes unsafe program locations for an execution order constraint by first deriving code regions from the constraint, then reusing the method for mutual exclusion to compute unsafe program locations. Specifically, given a filter f with an execution order constraint $e_1 > e_2 > ... > e_n$, LOOM first computes the set TF of all thread functions (including the main function) that may execute any e_i. It then identifies the entry s_j of each thread function $tf_j \in TF$. It finally computes



(a) *ICFG of the code in Figure 5.7.* (b) *After update-checks inserted.* (c) *After basic blocks cloned.*

Figure 5.8: *Static transformations that* Loom *does for safe and fast live update.* Subfigure 5.8a shows the ICFG of the code in Figure 5.7; 5.8b shows the resulting CFG of function `handle_client()` after the instrumentation to control application threads (§5.3); 5.8c shows the final CFG of function `handle_client()` after basic block cloning (§5.4).

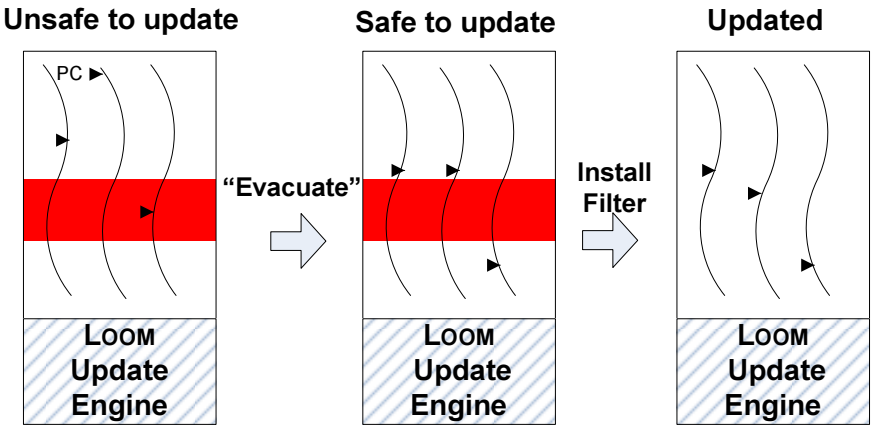


Figure 5.9: *Evacuation.* Curved lines represent application threads, solid triangles (in black) represents the threads' program counters (PC), and solid stripes (in red) represents an unsafe code region.

$unsafe(f)$ as the set of statements between s_j and e_i, denoted by $\{s_j; e_i\}$.

We compute the set $\{s_j; e_i\}$ using a backward reachability analysis. Given the ICFG $\mathcal{G}$ of the program, the set $\{s_j\}$, and the set $\{e_i\}$, our backward reachability analysis first computes $\mathcal{G}^T$, the inversion of $\mathcal{G}$; it then includes s_j, e_i, and every statements that e_i can reach on $\mathcal{G}^T$ in the set $\{s_j; e_i\}$. We only do backward search from e_i, without doing forward search from s_j, because the set s_j of thread entries dominates all statements that these threads may execute.

Our algorithm to compute unsafe locations for execution order filters is safe because it waits until all existing threads to execute beyond e_i. That is, no existing thread will ever run any statements in $\{s_j; e_i\}$. (Threads created after the filter is installed can still run statements in $\{s_j; e_i\}$.)

However, this algorithm is conservative because it may compute a larger-than-necessary set of unsafe locations. That is, to avoid the two unsafe scenarios for order constraints ("up skipped" and "wakeup wrong thread" in Figure 5.6), it may be unnecessary to require the program counters of every threads to be outside the set. Being conservative may cause a delay when installing a filter. Fortunately, for server applications, which need live-update most, such scenarios rarely occur because most order violations and their fixes only involve with worker threads that run for a short period of time.

Server applications sometimes use thread pools, creating problems for our algorithm. Specifi-

cally, during initialization these servers create a fixed set of threads, which loop forever until the application exits to process request. If the loop body contains an event of an execution order filter, the entire loop body will be marked unsafe, and Loom would have to wait until the application exits to install the filter. Fortunately, we can annotate the request processing function as the conceptual thread function, so that our algorithm does not compute a overly large set of unsafe program locations. In our experiments, we did not (and need not) annotate any request processing function.

5.3.2 Controlling Application Threads

Loom needs to control application threads to pause and resume them. It does so using a read-write lock called the update lock. To live update an application, Loom grabs this lock in write mode, performs the update, and releases this lock. To control application threads, Loom's compiler plugin instruments the application so that the application threads hold this lock in read mode in normal operation and check for an update once in a while by releasing and re-grabbing this lock.

Loom carefully places update-checks inside an application to reduce the overhead and ensure a timely update. Figure 5.8b shows the placement of these update-checks. Loom needs no update-checks inside straight-line code with no blocking calls because such code can complete quickly. Loom places one update-check for each cycle in the control flow graph, including loops and recursive function call chains, so that an application thread cycling in one of these cycles can check for an update at least once each iteration. Currently Loom instruments the backedge of a loop and an arbitrary function entry in a recursive function cycle. Loom does not instrument every function entry because doing so is costly.

Loom also instruments an application to release the update lock before a blocking call and re-grab it after the call, so that an application thread blocking on the call does not delay an update. For the example in Figure 5.7, Loom can perform the update despite some threads blocking in `recv()`. Loom instruments only the "leaf-level" blocking calls. That is, if `foo()` calls `bar()` and `bar()` is blocking, Loom instruments the calls to `bar()`, but not the calls to `foo()`. Currently Loom conservatively considers calls to external functions (*i.e.*, functions without source), except Math library functions, as blocking to save user annotation effort.

```
// inserted at CFG cycle
void cycle_check(stmt_id) {
  if(wait[stmt_id]) {
    read_unlock(&update);
    while(wait[stmt_id]);
    read_lock(&update);
  }
}
```

```
// inserted before blocking call
void before_blocking(callsite_id) {
  atomic_inc(&counter[callsite_id]);
  read_unlock(&update);
}
// inserted after blocking call
void after_blocking(callsite_id) {
  read_lock(&update);
  atomic_dec(&counter[callsite_id]);
}
```

Figure 5.10: *Instrumentation to pause application threads.*

5.3.3 Pausing at Safe Program Locations

Besides the update lock, Loom uses additional synchronization variables to ensure that application threads pause at safe locations. Loom assigns a wait flag for each backedge of a loop and the chosen function entry of a recursive call cycle. To enable/disable pausing at a safe/unsafe location, Loom sets/clears the corresponding flag. The instrumentation code for each CFG cycle (left of Figure 5.10) checks for an update only when the corresponding wait flag is set. These wait flags allow application threads at unsafe program locations to run until they reach safe program locations, effectively evacuating the unsafe program locations.

Note that the statement "`if(wait[stmt_id])`" in Figure 5.10 greatly improves Loom's performance. With this statement, application threads need not always release and re-grab the update lock which can be costly, and hardware cache and branch prediction can effectively hide the overhead of checking these flags. This technique speeds up Loom significantly (§5.5) because wait flags are almost always 0 with read accesses.

Loom cannot use the wait-flag technique to skip a blocking function call because doing so changes the application semantics. Instead, Loom assigns a counter to each blocking callsite to track how many threads are at the callsites (right of Figure 5.10). Loom uses a counter instead of a binary flag because multiple threads may be doing the same call.

Now that Loom's instrumentation is in place, Figure 5.11 shows Loom's evacuation method which runs within Loom's live update engine. This method first sets the wait flags for safe

```
volatile int wait[NBACKEDGE] = {0};
volatile int counter[NCALLSITE] = {0};
rwlock_t update;
void evacuate() {
  for each B in safe backedges
    wait[B] = 1; // turn on wait flags
retry:
  write_lock(&update); // pause app threads
  for each C in unsafe callsites
    if(counter[C]) { // threads paused at unsafe callsites
      write_unlock(&update);
      goto retry;
    }
  ... // update
  for each B in safe backedges
    wait[B] = 0; // turn off wait flags
  write_unlock(&update); // resume app threads
}
```

Figure 5.11: *Pseudo code of the evacuation algorithm.*

backedges. It then grabs the update lock in write mode, which pauses all application threads. It then examines the counters of unsafe callsites and if any counter is positive, it releases the update lock and retries, so that the thread blocked at unsafe callsites can wake up and advance to safe locations. Next, it updates the application (§5.4), clears the wait flags, and releases the update lock.

Our evacuation algorithm may cause the program to deadlock if (1) the execution filter itself cause the program to deadlock, or (2) the static analysis used to compute unsafe program locations is too conservative and includes more statements than necessary. In an extreme case, if Loom treats all program locations as unsafe, it can never evacuate any thread outside the unsafe region. Due to these limitations, we set an upper-bound (100 in our experiments) on the number of retries in our evacuation algorithm. If the number of retries reaches this limit, we hand over the control to the user of Loom, who can revoke this execution filter and investigate the issue further. In our experiments, we did not encounter any of this case.

5.3.4 Correctness Discussion

In program analysis terms, our reachability analysis (§5.3.1) is interprocedural and flow-sensitive. We use a crude pointer analysis to discover thread functions, thread join sites, and function pointer targets. We could have refined our analysis to improve precision, but we find it sufficient to compute unsafe locations for all evaluated races because (1) our analysis is sound and never marks an unsafe location safe and (2) execution filters are quite small and slight imprecision does not matter. In the worst case, if our analysis turns out too imprecise for some filters, the flexibility of Loom allows developers to easily adjust their filters to pass the safety analysis.

Our reachability analysis gives correct results despite compiler reordering. In order to pause application threads at safe locations, our reachability analysis returns only the set of unsafe backedges and external callsites. These locations are instrumented by Loom; this instrumentation acts as barriers and prevents compilers from reordering instructions across them.

The synchronization between the instrumentation in Figure 5.10 and the evacuation algorithm in Figure 5.11 is correct under two conditions: (1) read and write to wait flags are atomic and (2) the operations to the update lock contain correct memory barriers that prevent hardware reordering. Currently we implement wait flags using aligned integers; our update lock operations use atomic operations similar to the Linux kernel's `rw_spinlock`. Thus, our evacuation algorithm works correctly on X86 and AMD64 which do not reorder instructions across atomic instructions. We expect our algorithm to work on other commodity hardware that also provides this guarantee. To cope with more relaxed hardware (*e.g.*, Alpha), we can augment these operations with full barriers.

5.4 Hybrid Instrumentation

Most previous live update systems update binaries by compiling updated functions and redirecting old functions to the new function binaries using a table or jump instructions. This approach requires source patches to generate the updates, thus it has the limitations of unsafe updating and untimeliness. Moreover, this approach pays the overhead of position independent code (PIC) because application functions must be compiled as PIC for live update. It also suffers the aforementioned

```
void slot(int stmt_id) {
  op_list = operations[stmt_id];
  foreach op in op_list
    do op;
}
```

Figure 5.12: *Slot function.*

function quiescence problem.[2]

Another alternative is to use general-purpose binary instrumentation tools such as vx32 [Ford and Cox, 2008], Pin [Luk *et al.*, 2005] and DynamoRIO [Bruening, 2004], but they tend to incur significant runtime overhead just to run their frameworks alone. For example, Pin has been reported to incur 199% overhead [Luk *et al.*, 2005], and we observed 10 times slowdown on Apache with a CPU-bound workload (§5.5).

Loom's hybrid instrumentation framework reduces runtime overhead by combining static and dynamic instrumentation. This framework statically transforms an application's binary to anticipate dynamic updates. The static transformation pre-pads, before each program location, a *slot function* which interprets the updates to this program location at runtime. Figure 5.12 shows the pseudo code of this function. It iterates though a list of synchronization operations assigned to the current statement and performs each. To update a program location at runtime, Loom simply modifies the corresponding operation list.

Inserting the slot function at every statement incurs high runtime overhead and hinders compiler optimization. Loom solves this problem using a basic block cloning idea [Liblit *et al.*, 2003]. Loom keeps two versions of each basic block in the application binary, an originally compiled version that is optimized, and a hot backup that is unoptimized and padded for live update. To update a basic block at runtime, Loom simply updates the backup and switches the execution to the backup by flipping a switch flag.

Loom instruments only function entries and loop backedges to check the switch flags because doing so for each basic block is expensive. Similar to the wait flags in (§5.3), the switch flags are

[2]The function quiescence problem can be addressed by transforming loop bodies into functions [Neamtiu *et al.*, 2006; Neamtiu and Hicks, 2009] but only if the CFGs are reducible [Hecht and Ullman, 1974].

Race ID	Description
MySQL-791	Calls to `close()` and `open()` to flush log file are not atomic. Figure 5.2 shows the code.
MySQL-169	Table update and log write in `mysql_delete()` are not atomic.
MySQL-644	Calls to `prepare()` and `optimize()` in `mysql_select()` are not atomic.
Apache-21287	Reference count decrement and checking are not atomic.
Apache-25520	Threads write to same log buffer concurrently, resulting in corrupted logs or crashes.
PBZip2	Variable `fifo` is used in one thread after being freed by another. Figure 5.4 shows the code.
SPLASH2-`fft`	Variable `finishtime` and `initdonetime` are read before assigned the correct values.
SPLASH2-`lu`	Variable `rf` is read before assigned the correct value.
SPLASH2-`barnes`	Variable `tracktime` is read before assigned the correct value.

Table 5.3: *All races used in evaluation.* We identify races in MySQL and Apache as "⟨*application name*⟩ – ⟨*Bugzilla #*⟩", the only race in PBZip2 "*PBZip2*", and races in SPLASH2 "SPLASH2 – ⟨*benchmark name*⟩".

almost always 0, so that hardware cache and branch predication can effectively hide the overhead of checking them. This technique makes live-update-ready applications run as fast as the original application during normal operations (§5.5). Figure 5.8c shows the final results after all Loom transformations.

Note that the accesses to switch flags are correctly protected by the update lock. An application checks the switch flag when holding the update lock in read mode, and the update engine sets the switch flag when holding the update lock in write mode.

5.5 Evaluation

We implemented Loom in Linux. It consists of 4,852 lines of C++ code, with 1,888 lines for the LLVM compiler plugin, 2,349 lines for the live-update engine, and 615 lines for the controller.

We evaluated Loom on nine real races from a diverse set of applications, ranging from two server applications MySQL [mys, 2014] and Apache [Apache, 2012], to one desktop application PBZip2 [PBZIP2, 2011], to three scientific applications `fft`, `lu`, and `barnes` in SPLASH2 [SPLASH2, 2007].[3] Table 5.3 lists all nine races. Our race selection criteria is simple: (1) they are extensively

[3]We include applications that do not need live update for two reasons. First, Loom can provide quick workarounds

used in previous studies [Lu *et al.*, 2006; Park *et al.*, 2009a; Park *et al.*, 2009b] and (2) the application can be compiled by LLVM and the race can be reproduced on our main evaluation machine, a 2.66 GHz Intel quad-core machine with 4 GB memory running 32-bit Linux 2.6.24.

We used the following workloads in our experiments. For MySQL, we used SysBench [sys, 2004] (advanced transaction workload), which randomly selects, updates, deletes, and inserts database records. For Apache, we used ApacheBench [apa, 2014], which repeatedly downloads a webpage. Both benchmarks are multithreaded and used by the server developers. We made both SysBench and ApacheBench CPU bound by fitting the database or web contents within memory; we also ran both the client and the server on the same machine, to avoid masking Loom's overhead with the network overhead. Unless otherwise specified, we ran 16 worker threads for MySQL and Apache because they performed best with 8-16 threads. We ran four worker threads for PBZip2 and SPLASH2 applications because they are CPU-intensive and our evaluation machine has four cores.

We measured throughput (TPUT) and response time (RESP) for server applications and overall execution time for other applications. We report Loom's relative overhead, the smaller the better. We compiled the applications down to x86 instructions using `llvm-gcc -O2` and LLVM's bitcode compiler `llc`. For all the performance numbers reported, we repeated the experiment 50 times and take the average.

We focus our evaluation on five dimensions:

1. Overhead. Does Loom incur low overhead?
2. Scalability. Does Loom scale well as the number of application threads increases?
3. Reliability. Can Loom be used to fix the races listed in Table 5.3? What are the performance and reliability tradeoffs of execution filters?
4. Availability. Does Loom severely degrade application availability when execution filters are installed?
5. Timeliness. Can Loom install fixes in a timely way?

5.5.1 Overhead

Figure 5.13 shows the performance overhead of Loom during the normal operations of the applications. The base line of the comparison is the run time of the original program compiled using

for these applications as well. Second, we use them to measure Loom's overhead and scalability.

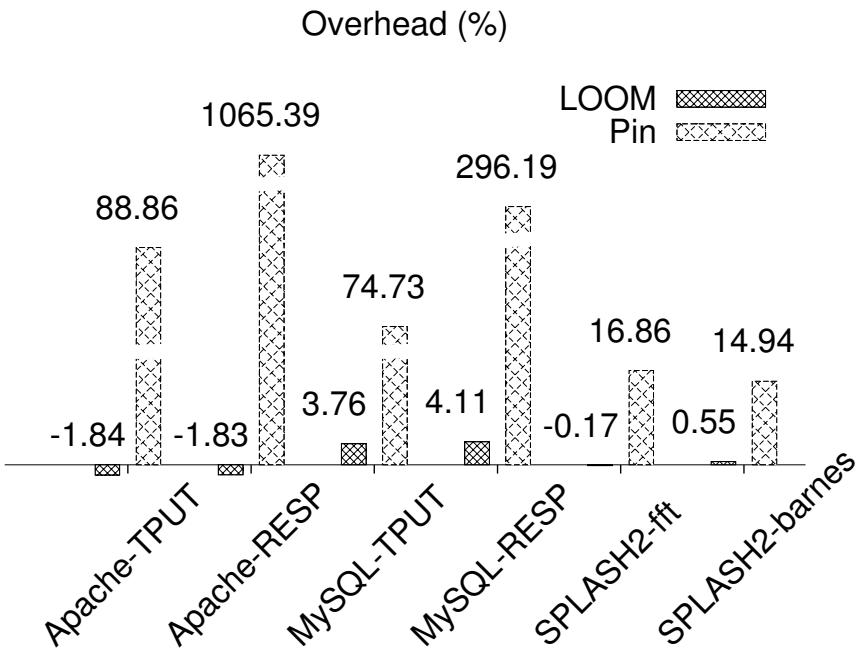


Figure 5.13: Loom*'s relative overhead during normal operation.* Smaller numbers are better. We show Pin's overhead for reference. Some Pin bars are broken.

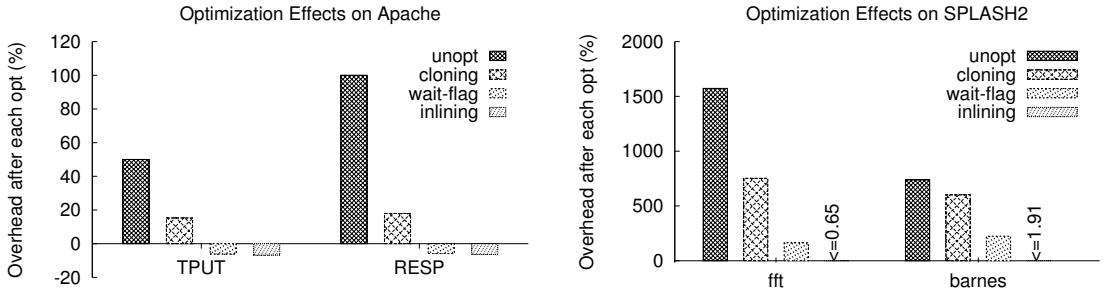


Figure 5.14: *Effects of* LOOM*'s optimizations.* Label **unopt** represents the versions with no optimizations; **cloning** represents the version with basic block cloning (§5.4); **wait-flag** represents the version with statement "`if(wait[stmt_id])`" added (§5.3.2); and **inlining** indicates the version with all LOOM instrumentation inlined into the applications.

`llvm-gcc -O2`. We also show the overhead of bare Pin for reference. LOOM incurs little overhead for Apache and SPLASH2 benchmarks. It increases MySQL's response time by 4.11% and degrades its throughput by 3.76%. In contrast, Pin incurs higher overhead for all applications evaluated, especially for Apache and MySQL.

We also evaluated how the optimizations we do reduce LOOM's overhead. Figure 5.14 shows the effects of these optimizations. Both cloning and wait-flag are very effective at reducing overhead. Cloning reduces LOOM's response-time overhead on Apache from 100% to 17%. It also reduces LOOM's overhead on `fft` from 15 times to 8 times. Wait-flag actually makes Apache run faster than the original version. Inlining does not help the servers much, but it does help for SPLASH2 applications.

5.5.2 Scalability

LOOM synchronizes with application threads via a read-write lock. Thus, one concern is, can LOOM scale well as the number of application threads increases? To evaluate LOOM's scalability, we ran Apache and MySQL with LOOM on a 48-core machine with four 1.9 GHz 12-core AMD CPUs and 64 GB memory running 64-bit Linux 2.6.24. In each experiment, we pinned the benchmark to one CPU and the server to the other three to avoid unnecessary CPU contention between them.

Figure 5.15 shows LOOM's relative overhead vs. the number of application threads for Apache and MySQL. LOOM scales well with the number of threads. Its relative overhead varies only slightly.

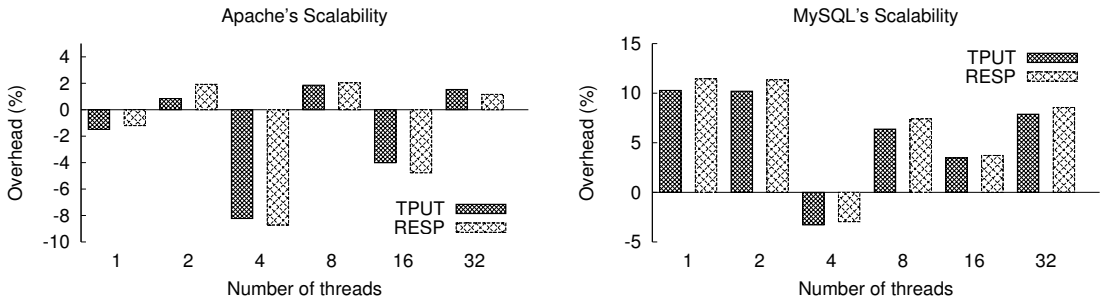


Figure 5.15: LOOM*'s relative overhead vs. the number of application threads.*

Even with 32 server threads, the overhead for Apache is less than 3%, and the overhead for MySQL is less than 12%.

Our initial MySQL overhead was around 16%. We analyzed the execution counts of the LOOM-inserted functions and immediately identified two update-check sites (`cycle_check()` calls) that executed exceedingly many times. These update-check sites are in MySQL functions `ptr_compare_1` and

`Field_varstring::val_str`. The first function compares two strings, and the second copies one string to another. Each function has a loop with a few statements and no function calls. Such tight loops cause higher overhead for LOOM, but rarely need to be updated. We thus disabled the update-check sites in these two functions, which reduced the overhead of MySQL down to 12%. This optimization can be easily automated using static or dynamic analysis, which we leave for future work.

5.5.3 Reliability

LOOM can be used to fix all races evaluated. (We verified this result by manually inspecting the application binary.) Table 5.4 shows the statistics for the execution filters that fix atomicity errors. Table 5.5 shows the statistics for the execution filters that fix order errors.

In all cases, we can fix the race using multiple execution filters, demonstrating the flexibility of LOOM. (The filters for MySQL-791 are shown in Figure 5.3.) We only show the statistics of one execution filter of each constraint type; other filters of the same type are similar. Our results show that the filters are fairly small, 3.79 events on average and no more than 16 events, demonstrating

Race ID	Mutual			Unilateral		
	Events	**TPUT**	**RESP**	**Events**	**TPUT**	**RESP**
MySQL-169	2	0.14%	0.15%	1	3.28%	3.37%
MySQL-644	4	0.22%	0.20%	4	32.58%	48.34%
MySQL-791	4	0.23%	0.32%	2	0.33%	0.48%
Apache-21287	16	-0.02%	-0.03%	2	54.03%	118.16%
Apache-25520	1	0.52%	0.55%	1	86.04%	637.03%

Table 5.4: *Execution filter stats for atomicity errors.* Column Events counts the number of events in each filter.

Race ID	**Events**	**Overhead**
PBZip2	6	1.26%
SPLASH2-`fft`	6	0.08%
SPLASH2-`lu`	2	1.68%
SPLASH2-`barnes`	2	1.99%

Table 5.5: *Execution filter stats for order errors.*

the ease of use of Loom. Most filters incur only a small overhead on top of Loom. Unilateral filters tend to be slightly smaller than mutual exclusion filters, but they can be expensive sometimes. They incur little overhead for two of the MySQL bugs because the code regions protected by the filters rarely run.

These different reliability and performance overheads present an interesting tradeoff to developers. For example, users can choose to install a unilateral filter for immediate protection, then atomically replace it with a faster mutual exclusion filter. Moreover, a user can choose an "expensive" filter as long as their workload is compatible with the filter.

5.5.4 Availability

We show that Loom can improve server availability by comparing Loom to the restart-based software update approach. We restarted a server by running its startup script under `/etc/init.d`. We chose two races, MySQL-791 and Apache-25520, and measured how software updates (conventional or with Loom) might degrade performance. Note this comparison favors conventional updates be-

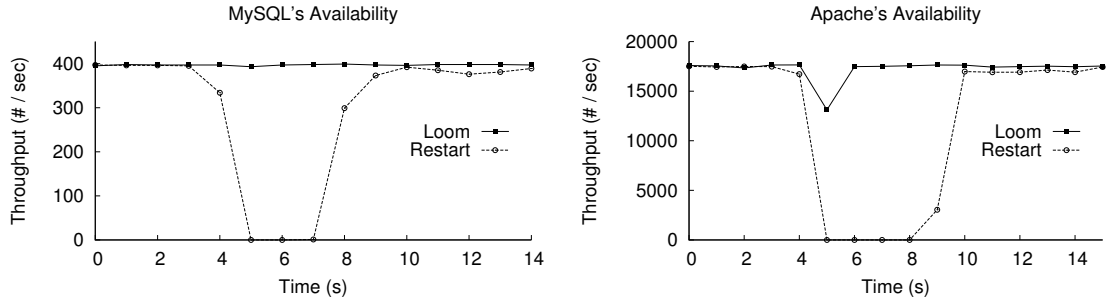


Figure 5.16: *Throughput degradation for fixing races with* Loom *vs. with conventional software update.*

cause we only compare the installation of the fix, but Loom also makes it quick to develop fixes. Figure 5.16 shows the comparison result. Using the restart approach, Apache is unavailable for 4 seconds, and MySQL is unavailable for 2 seconds. Moreover, the restarts also cause Apache and MySQL to lose their internal cache, leading to a ramp-up period after the restart. In contrast, installing an filter using Loom (at second 5) does not degrade throughput for MySQL and only degrades throughput slightly for Apache.

5.5.5 Timeliness

The more timely Loom installs a filter, the quicker the application is protected from the corresponding race. This timeliness is critical for server applications because malicious clients may exploit a known race and launch attacks. In this subsection, we compare how timely Loom's evacuation algorithm installs an aggressive filter vs. an approach that passively waits for function quiescence. We chose Apache-25520 as the benchmark race. We wrote a simple mutual exclusion filter that fixes the race by making function `ap_buffered_log_writer` a critical region. We then measured the latency from the moment Loom receives a filter to the moment the filter is installed. We simulated a function quiescence approach by running Loom without making any `wait_flag` false, so that a thread can pause wherever we insert update-checks. We used the same SysBench and ApacheBench workload. Our results show that Loom can install the filter within 368 ms. It spends majority of the time waiting for threads to evacuate. In contrast, an approach based on function quiescence fails to install the filter in an hour, our experiment's time limit.

5.6 Related Work

Live update Loom differs from previous live update systems [Neamtiu *et al.*, 2006; Neamtiu and Hicks, 2009; Arnold and Kaashoek, 2009; Chen *et al.*, 2006; Subramanian *et al.*, 2009; Altekar *et al.*, 2005; Makris and Ryu, 2007] in that it is explicitly designed for developers to quickly develop temporary workarounds to races. Moreover, it can automatically ensure the safety of the workarounds. In contrast, previous work focuses only on live update after a source patch is available, thus it does not address the automatic-safety and flexibility problems Loom addresses.

The live update system closest to Loom is STUMP [Neamtiu and Hicks, 2009], which can live-update multithreaded applications written in C. Its prior version Ginseng [Neamtiu *et al.*, 2006] works with single-threaded C applications. Both STUMP and Ginseng have been shown to be able to apply arbitrary source patches and update applications across major releases. Unlike Loom, both STUMP and Ginseng require source modifications and rely on extensive user annotations for safety because the safety of arbitrary live updates has been proven undecidable [Gupta *et al.*, 1996].

A number of live update systems can update kernels without reboots [Arnold and Kaashoek, 2009; Chen *et al.*, 2006; Makris and Ryu, 2007]. The most recent one, Ksplice [Arnold and Kaashoek, 2009], constructs live updates from object code, and does not require developer efforts to adapt existing source patches. Unlike Loom, Ksplice uses function quiescence for safety, and is thus prone to the unsafe state problem discussed in §5.3. Another kernel live update system, DynAMOS [Makris and Ryu, 2007], requires users to manually construct multiple versions of a function to update non-quiescent functions. This technique is different from basic block cloning (§5.4): the former is manual and for safety, whereas the later is automatic and for speed.

Error workaround and recovery We compare Loom to recent error workaround and recovery tools. ClearView [Perkins *et al.*, 2009], ASSURE [Sidiroglou *et al.*, 2009], and Failure-oblivious computing can increase application availability by letting them continue despite errors. Compared to Loom, these systems are unsafe, and do not directly deal with races. Rx [Qin *et al.*, 2007] can safely recover from runtime faults using application checkpoints and environment modifications, but it does not fix errors because the same error can re-appear. Vigilante [Costa *et al.*, 2005] enables hosts to collaboratively contain worms using self-verifiable alerts. By automatically ensuring filter safety, Loom shares similar benefits.

Two recent systems, Dimmunix [Jula *et al.*, 2008] and Gadara [Wang *et al.*, 2008], can fix dead-

locks in legacy multithreaded programs. Dimmunix extracts signatures from occurred deadlocks (or starvations) and dynamically avoids them in future executions. Gadara uses control theory to statically transform a program into a deadlock-free program. Both systems have been shown to work on real, large applications. They may possibly be adapted to fix races, albeit at a coarser granularity because these systems control only lock operations.

Kivati [Chew and Lie, 2010] automatically detects and prevents atomicity violations for production systems. It reduces performance overhead by cleverly using hardware watch points, but the limited number of watch points on commodity hardware means that Kivati cannot prevent all atomicity violations. Nor does Kivati prevent execution order violations. Loom can be used to workaround these errors missed by Kivati.

Program instrumentation frameworks Previous work [Engler *et al.*, 2000; Necula *et al.*, 2002; LLVM, 2013] can instrument programs with low runtime overhead, but instrumentation has to be done at compile time. Translation-based dynamic instrumentation frameworks [Luk *et al.*, 2005; Bruening, 2004; Ford and Cox, 2008] can update programs at runtime but incur high overhead. In particular, vx32 [Ford and Cox, 2008] is a novel user-level sandbox that reduces overhead using segmentation hardware; it can be used as an efficient dynamic binary translator. Jump-based instrumentation frameworks [Hunt and Brubacher, 1998; Schulz *et al.*, 2005] have low overhead but automatically ensuring safety for them can be difficult due to low-level issues such as position-dependent code, short instructions, and locations of basic blocks.

One advantage of these instrumentation frameworks over ours is that our framework requires CFGs and symbol information to be distributed to user machines, thus it risks leaking proprietary code information. However, this risk is not a concern for open-source software. Moreover, our framework only mildly increases this risk because CFGs can often be reconstructed from binaries, and companies such as Microsoft already share symbol information [mic, 2014].

The advantage of our framework is that it combines static and dynamic instrumentation, thus allowing arbitrary dynamic updates issued by execution filters with negligible runtime overhead. Our framework borrows basic block cloning from previous work by [Liblit *et al.*, 2003], but their framework is static only. This idea has also been used in other systems (*e.g.*, LIFT [Qin *et al.*, 2006]).

Other related work Our work was inspired by many observations made by [Lu *et al.*, 2008].

Aspect-oriented programming (AOP) allows developers to "weave" in synchronizations into code [Kiczales *et al.*, 1997; Lohmann *et al.*, 2009]. LOOM's execution filter language shares some similarity to AOP, and can be made more expressive by incorporating more aspects. However, to the best of our knowledge, no existing AOP systems were designed to support race fixing at runtime. We view the large body of race detection and diagnosis work (*e.g.*, [Musuvathi *et al.*, 2008; Park *et al.*, 2009a; Savage *et al.*, 1997; Yu *et al.*, 2005; Sen, 2008; Lu *et al.*, 2007; Lu *et al.*, 2006]) as complimentary to our work and LOOM can be used to fix errors detected and isolated by these tools.

5.7 Summary

This chapter presented LOOM, a live-workaround system designed to quickly and safely fix application races at runtime. Its flexible language allows developers to write concise execution filters to declare their synchronization intents on code. Its evacuation algorithm automatically ensures the safety of execution filters and their installation/removal processes. It uses hybrid instrumentation to reduce its performance overhead during the normal operations of applications. We evaluated LOOM on nine real races from a diverse set of applications. Our results show that LOOM is fast, scalable, and easy to use. It can safely fix all evaluated races in a timely manner, thereby increasing application availability.

LOOM demonstrates that live-workaround systems can increase application availability with little performance overhead. In our future work, we plan to extend this idea to other classes of errors (*e.g.*, security vulnerabilities).

Chapter 6

Conclusions

This thesis presented a novel approach called schedule specialization. It statically analyzes a multithreaded program with respect to a small set of schedules and enforces them at runtime. Using this idea, we dramatically improved the precision of static analysis on multithreaded programs while maintaining the soundness of our static analysis results.

We built three systems to realize this vision. The first system is a specialization framework that precisely analyzes a program with respect to a small set of schedules. It specializes a program with respect to a schedule so stock analyses can automatically gain precision from these schedules. This ability of automatically benefiting stock analyses allows us to build three highly precise schedule-aware analyses.

The second system is our Peregrine system that enforces analyzed schedules at runtime for soundness. It computes preconditions of recorded schedules so one schedule can be reused to process multiple inputs. It uses the idea of hybrid schedule to deterministically and efficiently enforces schedules with the presence of data races.

The third system is Loom, a live-update engine designed to quickly and safely bypass application races at runtime. It provides a last-resort protection for multithreaded programs against races that slip detection. Its flexible language allows developers to declare their synchronization intents on code; its evacuation algorithm guarantee the safety of execution filters; its hybrid instrumentation framework significantly reduces the runtime overhead of Loom.

Future work. There are two major lines of future work. Along the line of schedule specialization,

we believe the general idea of restricting the behavior of a program to make it easier to analyze can be applied beyond schedules. We can restrict other factors of programs' execution environment to benefit other static analyses. Along the line of LOOM, we plan to extend LOOM's interface and implementation to fix more types of errors (such as deadlocks and memory errors) in live applications. Also, we plan to generalize the hybrid instrumentation framework to support more complicated instrumentation, and build more applications on it.

Bibliography

[Abadi *et al.*, 2005] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS '05)*, pages 340–353, 2005.

[Agrawal and Horgan, 1990] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, June 1990.

[Aho *et al.*, 1986] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Altekar and Stoica, 2009] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, October 2009.

[Altekar *et al.*, 2005] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[apa, 2014] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`, 2014.

[Apache, 2012] Apache web server. `http://www.apache.org`, 2012.

[Arnold and Kaashoek, 2009] Jeff Arnold and Frans M. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 2009 ACM European Conference on Computer Systems (EUROSYS '09)*, pages 187–198, April 2009.

[Arvind *et al.*, 1989] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[Aviram *et al.*, 2010] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[Avots *et al.*, 2005] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 332–341, May 2005.

[Baumann *et al.*, 2005] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 32–32, 2005.

[Bergan *et al.*, 2010a] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, March 2010.

[Bergan *et al.*, 2010b] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[Berger *et al.*, 2009] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, October 2009.

[Bhansali *et al.*, 2006] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihocka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 154–163, June 2006.

[Board, 2008] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[Bocchino *et al.*, 2009] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, October 2009.

[Boehm and Adve, 2008] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. pages 68–78, 2008.

[Bruening, 2004] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004. Supervisor-Amarasinghe, Saman.

[Cadar *et al.*, 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, December 2008.

[Chen *et al.*, 2006] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006.

[Chew and Lie, 2010] Lee Chew and David Lie. Kivati: fast detection and prevention of atomicity violations. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 307–320, 2010.

[Consel and Danvy, 1993] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 493–501, 1993.

[Costa *et al.*, 2005] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In

Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05), pages 133–147, 2005.

[Costa *et al.*, 2007] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, October 2007.

[Cui *et al.*, 2010] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[Cui *et al.*, 2011] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, October 2011.

[Cytron *et al.*, 1991] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment from and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[dal, 2013] `http://source.android.com/tech/dalvik/`, 2013.

[Devietti *et al.*, 2009] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, March 2009.

[Dimitrov *et al.*, 2014] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. Commutativity race detection. In *Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI '14)*, pages 305–315, 2014.

[Dinning and Schonberg, 1990] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd Symposium on Principles and Practice of Parallel Programming (PPOPP '90)*, pages 1–10, March 1990.

[Diwan *et al.*, 1998] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, 1998.

[Dunlap *et al.*, 2002] George Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrat, and Peter Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, December 2002.

[Dunlap *et al.*, 2008] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, March 2008.

[Engler and Ashcraft, 2003] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, October 2003.

[Engler *et al.*, 2000] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, September 2000.

[Erickson *et al.*, 2010] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[Ford and Cox, 2008] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, pages 293–306, 2008.

[Ganesh and Dill, 2007] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference On Computer Aided Verification (CAV '07)*, pages 519–531, 2007.

[Gao *et al.*, 2011] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: towards manifesting hidden concurrency typestate bugs. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 239–250, March 2011.

[Geels *et al.*, 2007] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, April 2007.

[Gilmore and Walton, 1997] S. Gilmore and C. Walton. Dynamic ML without dynamic types. Technical report, Lab. for the Foundations of Computer Science, University of Edinburgh, 1997.

[Glück and Jørgensen, 1995] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 259–278, 1995.

[Godefroid *et al.*, 2005] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[Guo *et al.*, 2008] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, December 2008.

[Gupta *et al.*, 1996] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[Halstead, 1985] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hecht and Ullman, 1974] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[Henzinger *et al.*, 2004] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, 2004.

[Hill and Xu, 2009] Mark D. Hill and Min Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`, 2009.

[Hunt and Brubacher, 1998] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[Jhala and Majumdar, 2005] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 38–47, June 2005.

[Jørgensen, 1992] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL '92)*, pages 258–268, 1992.

[Joshi *et al.*, 2009] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 110–120, June 2009.

[Jula *et al.*, 2008] Horatiu Jula, Daniel Tralamazza, Zamfir Cristian, and Candea George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, December 2008.

[k42,] The K42 Project. `http://www.research.ibm.com/K42/`.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[Kim *et al.*, 2010] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–9, October 2010.

[King, 1975] James C. King. A new approach to program testing. In *Proceedings of the 1975 International Conference on Reliable Software*, pages 228–233, June 1975.

[Konuru *et al.*, 2000] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[Laadan *et al.*, 2010] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

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

[Larson and Austin, 2003] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium (Security 2003)*, August 2003.

[Lee *et al.*, 2010] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, March 2010.

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

[Liblit *et al.*, 2003] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, 2003.

[Liu *et al.*, 2011] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, October 2011.

[llv, 2013] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`, 2013.

[LLVM, 2013] The LLVM compiler framework. `http://llvm.org`, 2013.

[Loginov *et al.*, 2001] Alexey Loginov, Vivek Sarkar, Jong deok Choi, Jong deok Choi, Alexey Logthor, and I Vivek Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.

[Lohmann *et al.*, 2009] Daniel Lohmann, Wanja Hofer, Wolfgang Schrder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, 2009.

[Lu *et al.*, 2006] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, October 2006.

[Lu *et al.*, 2007] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[Lu *et al.*, 2008] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, March 2008.

[Luk *et al.*, 2005] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, 2005.

[Makris and Ryu, 2007] K. Makris and K.D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, page 340, 2007.

[Manson *et al.*, 2005] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, pages 378–391, 2005.

[Marlow *et al.*, 2011] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 71–82, 2011.

[mic, 2014] Download windows symbol packages. `http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx`, 2014.

[Montesinos *et al.*, 2009] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, March 2009.

[Musuvathi *et al.*, 2008] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, December 2008.

[mys, 2014] MySQL Database. `http://www.mysql.com/`, 2014.

[Naik *et al.*, 2006] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, pages 308–319, 2006.

[Neamtiu and Hicks, 2009] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 13–24, June 2009.

[Neamtiu *et al.*, 2006] Iulian Neamtiu, Michael Hicks, Garath Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, pages 72–83, June 2006.

[Necula *et al.*, 2002] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction*, March 2002.

[Nirkhe and Pugh, 1992] Vivek Nirkhe and William Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL '92)*, pages 269–280, 1992.

[O'Callahan and Choi, 2003] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. pages 167–178, June 2003.

[Olszewski *et al.*, 2009] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, March 2009.

[Park and Sen, 2008] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, November 2008.

[Park *et al.*, 2009a] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, March 2009.

[Park *et al.*, 2009b] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, October 2009.

[PARSEC, 2010] The Princeton application repository for shared-memory computers (PARSEC). `http://parsec.cs.princeton.edu/`, 2010.

[PBZIP2, 2011] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`, 2011.

[Perkins *et al.*, 2009] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, 2009.

[Poulsen, 2004] Kevin Poulsen. Software bug contributed to blackout. `http://www.securityfocus.com/news/8016`, February 2004.

[Qadeer and Wu, 2004] S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, June 2004.

[Qin *et al.*, 2006] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[Qin *et al.*, 2007] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3):7, 2007.

[Reps and Turnidge, 1996] Thomas Reps and Todd Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1101, pages 409–429. Springer-Verlag, 1996.

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

[Rugina and Rinard, 2000] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 182–195, June 2000.

[Savage *et al.*, 1997] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, November 1997.

[Schonberg, 1989] D. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, 1989.

[Schulz *et al.*, 2005] Martin Schulz, Dong Ahn, Andrew Bernat, Bronis R. de Supinski, Steven Y. Ko, Gregory Lee, and Barry Rountree. Scalable dynamic binary instrumentation for blue gene/l. *SIGARCH Comput. Archit. News*, 33(5):9–14, 2005.

[Sen *et al.*, 2005] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, September 2005.

[Sen, 2008] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[Sidiroglou *et al.*, 2009] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: automatic software self-healing using rescue points. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 37–48, 2009.

[SPLASH2, 2007] Stanford parallel applications for shared memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`, 2007.

[Srinivasan *et al.*, 2004] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[Subramanian *et al.*, 2009] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI '09)*, pages 1–12, 2009.

[sys, 2004] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`, 2004.

[The Open Group and the IEEE, 2008] The Open Group and the IEEE. POSIX.1-2008. `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2008.

[Tip, 1995] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages 3(3)*, pages 121–189, 1995.

[Trinder *et al.*, 1998] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[Veeraraghavan *et al.*, 2011] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 369–384, 2011.

[VMWare Virtual Lab Automation,] `http://www.vmware.com/solutions/vla/`.

[von Praun and Gross, 2003] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 115–128, 2003.

[Voung *et al.*, 2007] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE '07)*, pages 205–214, 2007.

[Wang *et al.*, 2008] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings*

of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08), pages 281–294, December 2008.

[Weiser, 1979] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, 1979.

[Wester *et al.*, 2013] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 27–38, March 2013.

[Whaley and Lam, 2004] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 131–144, June 2004.

[Wu *et al.*, 2013] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proceedings of the Ninth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE '13)*, August 2013.

[Xiong *et al.*, 2010] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[Yang *et al.*, 2011] Junfeng Yang, Ang Cui, John Gallagher, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. Technical Report CUCS-028-11, Columbia University, 2011.

[Yu *et al.*, 2005] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, October 2005.

[Yuan *et al.*, 2010] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Fifteenth Inter-*

national Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10), pages 143–154, March 2010.

[Zamfir and Candea, 2010] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 2010 ACM European Conference on Computer Systems (EUROSYS '10)*, pages 321–334, April 2010.

[Zhang and Gupta, 2004] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 94–106, 2004.

[Zhang *et al.*, 2010] Wei Zhang, Chong Sun, and Shan Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, March 2010.

[Zhang *et al.*, 2011] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, March 2011.